



# smxUSBH™ User's Guide

USB Host Stack

Version 3.00  
December 18, 2020

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	3
<b>3. smxUSBH Library and Demos</b>	<b>7</b>
3.1 smxUSBH Configuration	7
3.2 Building the Library	14
3.3 Building and Running the Demos	14
3.4 Initialization	16
<b>4. USB Class Drivers</b>	<b>17</b>
4.1 Class Driver API Overview	17
4.2 Class Driver Interface (USBDI)	17
4.3 Audio	21
4.4 Communication (CDC)	37
4.5 Human Interface (HID)	78
4.6 Mass Storage	82
4.7 Printer	86
4.8 Video	89
4.9 Writing a New Class Driver	101
4.10 Device Plugin/Remove Event Callback	102
4.11 Stack Event Callback	102
<b>5. Host Controller Drivers</b>	<b>104</b>
5.1 Host Controller Driver Interface	104
5.2 Interrupt Handler (ISR)	106
5.3 Root Hub	106
5.4 EHCI	107
5.5 OHCI	107
5.6 UHCI	108
5.7 ISP116x	108
5.8 ISP1362	108
5.9 ISP176x	108
5.10 Blackfin	109
5.11 CF522xx	109
5.12 LM3Sxxxx	109
5.13 MAX3421	109
5.14 Renesas	110

5.15 Synopsys .....	110
5.16 uPD720150 .....	110
5.17 Writing a New Host Controller Driver.....	111
<b>6. Hardware Porting Notes .....</b>	<b>113</b>
6.1 uport.h.....	113
6.2 uport.c .....	113
6.3 DMA Transfer.....	115
<b>Appendix A. Porting smxUSBH to Another OS .....</b>	<b>116</b>
A.1 uheap.h and uheap.c .....	116
A.2 Non-Multitasking Support .....	116
A.3 Task Priority.....	117
<b>Appendix B. Memory Usage and Performance Summary .....</b>	<b>118</b>
B.1 Size.....	118
B.2 Performance .....	120
<b>Appendix C. Tested Host Controllers and Devices .....</b>	<b>123</b>
C.1 Host Controllers .....	123
C.2 Audio Devices .....	123
C.3 CDC ACM (Modem) Devices.....	124
C.4 HID Devices.....	124
C.5 Mass Storage Devices .....	124
C.6 Printer Devices .....	125
C.7 FTDI Serial Devices.....	125
C.8 Windows Mobile 5 Serial Devices .....	126
C.9 USB to Ethernet Devices.....	126
C.10 Ralink RT250x WiFi Dongles.....	126
C.11 Ralink RT2870 WiFi Dongles.....	126
C.12 Ralink RT3070 WiFi Dongles.....	126
C.13 Ralink RT3572 WiFi Dongles.....	126
C.14 Ralink RT5370 WiFi Dongles.....	126
C.15 MediaTek MT7601 WiFi Dongles .....	126
C.16 MediaTek MT7612 WiFi Dongles .....	127
C.17 Sierra Wireless Dongles .....	127
C.18 Video Cameras .....	127
C.19 CDC ECM/NCM (Ethernet) Devices .....	127
<b>Appendix D. Testing .....</b>	<b>128</b>
<b>Appendix E. Specification Reference .....</b>	<b>129</b>
E.1 USB Specifications.....	129
E.2 Host Controller Specifications.....	129
E.3 PCI Specification.....	129
E.4 Audio Devices Specifications.....	129
E.5 Communication Devices Specifications .....	129
E.6 HID Specifications .....	129
E.7 Mass Storage Specifications.....	130
E.8 Printer Specifications.....	130
E.9 Video Class Specifications .....	130
E.10 Referenced Documents.....	130
<b>Appendix F. Glossary .....</b>	<b>131</b>

© Copyright 2004-2020

Micro Digital Associates, Inc.  
2900 Bristol Street, #G204  
Costa Mesa, CA 92626  
(714) 437-7333  
support@smxrtos.com  
www.smxrtos.com

All rights reserved.

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



# 1. Overview

smxUSBH™ is a full-featured USB host stack for the SMX® RTOS. It offers a clean, modular design that allows embedded system developers to easily add USB host capabilities. Low-cost USB devices such as a keyboard or a USB flash drive can then be integrated with the system. smxUSBH is written in C and can be easily ported to another RTOS (see Appendix A. Porting smxUSBH to Another OS).

The reader should be familiar with the USB 2.0 specification, “Universal Serial Bus Specification, Revision 2.0”. All USB specifications can be found at <http://www.usb.org/>.

smxUSBH has four layers:

1. **Class Driver Layer:** Provides support for different USB class drivers, such as mouse, keyboard and mass storage drivers.
2. **USB Driver Layer (USB D):** Provides the common USB Device framework. See chapters 5, 8 and 9 in the USB 2.0 specification for details. This is also referred to as the *core*.
3. **Host Controller Driver (HCD) Layer:** Provides a driver for the Host Controller and also contains Root Hub support. This layer supports devices based on the OHCI/UHCI/EHCI, NXP ISP host controllers, and those on many embedded processors, such as ARM and ColdFire. See chapters 5, 8, and 10 of the USB 2.0 specification for details.
4. **Porting Layer:** Provides functions related to the hardware, operating system and compiler. Detailed in chapter 6. Hardware Porting Notes.

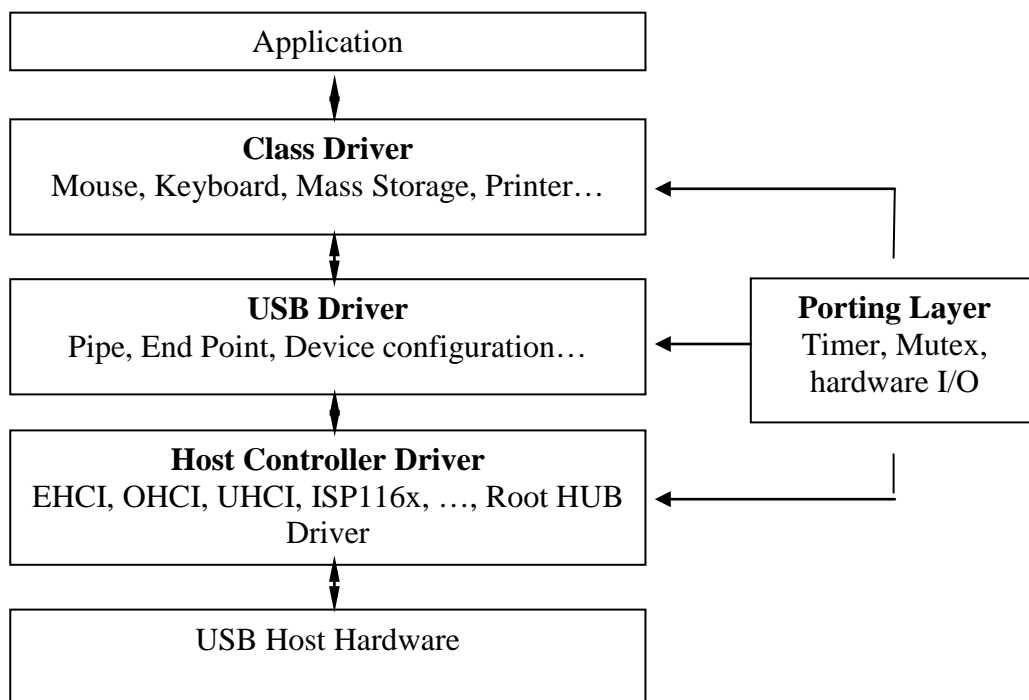


Figure1 smxUSBH Structure

v2.30 adds the capability to support multiple host controllers of different types.  
v2.40 adds the capability to support multiple host controllers of the same (and different) types.  
v2.50 changes the porting layer to smxBASE.  
v3.00 has many global renames to shorten names so the code is more compact and easy to read.  
It should drop in, in place of recent v2 releases, except code written by the user such as host controller drivers and class drivers, which will require the same renames. See DOC\smxusbh.txt.

## 2. Files

Like other SMX RTOS products, all source code for smxUSBH is stored in its own directory, named "XUSBH", under the main SMX directory. Below is a summary of the directory structure.

### 2.1 Directory Structure

SMX

APP

DEMO	usbhdemo.c (for SMX)
NORTOS	Build directory for standalone (non-SMX) releases. Has demo too.
XUSBH	Configuration and porting files.
XX.YY	Build directory for SMX releases
Audio	Audio class driver
CCID	SmartCard CCID class driver
CDC	Communication Device Class driver (serial, ACM (modem), Ethernet, etc.)
CTempl	Class driver template
EHCI	EHCI host controller driver
HCD	Host Controller Drivers (other than EHCI/OHCI/UHCI)
HID	Human Interface Device class drivers (keyboard, mouse, generic HID)
Hub	Hub driver
MassStor	Mass Storage class driver
OHCI	OHCI host controller driver
Printer	Printer class driver
UHCI	UHCI host controller driver
USB	USB Driver
Video	Video class driver

## 2.2 Files

### 2.2.1 Main Files

FILE	DESCRIPTION
smxusbh.h	smxUSBH API header file. Use in application files.
ucfg.h	smxUSBH configuration file. Allows enabling/disabling main components of smxUSBH.
uinit.c,h	Initialization of the USB host stack, including the hub, selected host controller, and selected class drivers.
uintern.h	Main internal header file. Included by smxUSBH 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
uport.c,h	Porting functions. OS and compiler porting is based on smxBase.
uheap.c,h	Memory allocation/free related functions. Includes support for specifying the alignment of the allocated memory.

### 2.2.3 USB Driver

FILE	DESCRIPTION
udrvinit.h	USB driver initialization function prototypes.
ubase.c,h	Utility functions for the USB host stack. Includes linked list functions and macros for handling USB pipes.
udesc.c,h	Utility functions for USB device configuration.
udriver.c,h	USB host stack protocol functions. Handles events associated with discovering and removing USB devices, and requests data.
udevice.h	Definitions and macros for the device framework.
ustruct.h	Structure definitions for stack and class drivers.

### 2.2.4 Hub Driver

FILE	DESCRIPTION
uhubinit.h	USB hub initialization function prototypes.
uhubbase.c,h	USB root hub functions.
uhubdrv.c,h	USB external hub class driver.
uhubthrd.c,h	Contains task that polls for hub status change and enumerates the new device.

## 2.2.5 Host Controller Driver

FILE	DESCRIPTION
uehcinit.c,h	EHCI Controller initialization function.
uehcdrv.c,h	EHCI related functions. Includes Device Configuration, Resource Management, and Data Transfers
uehcthrd.c,h	Thread for EHCI interrupt handler. Processes the completed asynchronous or periodic list.
uohcinit.c,h	OHCI Controller initialization function.
uohcdrv.c,h	OHCI related functions. Includes Device Configuration, Resource Management, and Data Transfers
uohcthrd.c,h	Thread for OHCI interrupt handler. Processes the completed transfer descriptor list.
uuhcinit.c,h	UHCI Controller initialization function.
uuhcdrv.c,h	UHCI related functions. Includes Device Configuration, Resource Management, and Data Transfers
uuhcreg.h	UHCI register definitions.
uuhcthrd.c,h	Thread for UHCI interrupt handler. Processes the completed frame list.
uisp116x.c,h	NXP ISP116x host controller driver.
uisp1362.c,h	NXP ISP1362 host controller driver.
uisp176x.c,h u176xdrv.c,h	NXP ISP176x host controller driver.
u720150.c,h	NEC uPD720150 host controller driver.
upd720150.c,h	NEC uPD720150 low-level register access routines. May need to be built into the BSP code instead of the USBH library.
ublkfn.c,h	Analog Devices Blackfin USB host controller driver.
ucf522xx.c,h	Freescale Coldfire 52223/52211/52259 host controller driver.
ulm3s.c,h	TI LM3Sxxxx host controller driver.
umax3421.c,h	Maxim MAX3421 host controller driver.
urenesas.c,h	Renesas USB host controller driver.
usyndwc.c,h	Synopsys DWC2 host controller driver.
uhctempl.c,h	Host Controller Driver template.

## 2.2.6 Audio Class Device Driver

FILE	DESCRIPTION
uaudio.c,h	USB Audio Class support.



## 2.2.7 Communication Device Class (CDC) Device Driver

FILE	DESCRIPTION
ucdcacm.c,h	USB CDC ACM (modem) support.
uftdi232.c,h	FTDI FT232B/LC/R Serial Converter support.
upl2303.c,h	Prolific 2303HXD Serial Converter support.
userial.c,h	USB serial support.
uwceser.c,h	Windows CE non-CDC/ACM USB serial support.
usierra.c,h	Sierra Wireless Dongle support.
uk4510.c,h	Huawei K4510 3G Wireless Dongle support.
uobid.c,h	FEIG OBID-iscan RFID Reader support.
urt250x.c,h	Ralink RT250x WiFi dongle support.
urt2870.c,h	Ralink RT2870 WiFi dongle support.
urt3070.c,h	Ralink RT3070 WiFi dongle support.
urt3572.c,h	Ralink RT3572 WiFi dongle support.
urt5370.c,h	Ralink RT5370 WiFi dongle support.
urt5572.c,h	Ralink RT5572 WiFi dongle support.
umt7601.c,h	MediaTek MT7601 WiFi dongle support.
umt7612.c,h	MediaTek MT7612 WiFi dongle support.
ucdcnet.c, h	USB CDC ECM/NCM (ethernet driver)
unet.c,h unetchip.h unetchip_asix.c	USB to Ethernet Adapter support. ASIX88772 chip support

## 2.2.8 Human Interface Device (HID) Class Device Driver

FILE	DESCRIPTION
ukbd.c,h	USB keyboard support.
umouse.c,h	USB mouse support.
uhid.c,h	USB generic HID support.

## 2.2.9 Mass Storage Class Device Driver

FILE	DESCRIPTION
umsinit.c,h	Initialization function that registers the mass storage device with USBSD.
umscmd.c,h	Build mass storage request for QIC, UFI, and ATAPI protocol devices.
umscmdt.c,h	Send mass storage request for QIC, UFI, and ATAPI protocol devices.
umsdrv.c,h	Mass storage device functions (USBDDI).
umsintf.c,h	File IO interface and implementation.
umsscnd.c,h	Send mass storage request for SCSI protocol devices.

### 2.2.10 Printer Class Device Driver

FILE	DESCRIPTION
uprinter.c,h	USB printer support.
uuf8000.c,h	Panasonic UF 8000 USB facsimile transceiver support

### 2.2.11 Video Class Device Driver

FILE	DESCRIPTION
uvideo.c,h	USB video camera support.
uv20k13.c,h	Videology 20K13 camera support.

### 2.2.12 SmartCard CCID Class Driver

FILE	DESCRIPTION
uccid.c,h	SmartCard CCID support.

### 2.2.13 Class Device Driver Template

FILE	DESCRIPTION
uctempl.c,h	USB class driver template.

## **3. smxUSBH Library and Demos**

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

### **3.1 smxUSBH Configuration**

#### **3.1.1 ucfg.h**

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

#### **Operating System Selection**

Operating system selection uses the smxBase configuration in bcfg.h.

#### **Controller Selection**

##### **SU\_EHCI**

Set to “1” or more to indicate the maximum number of EHCI host controllers to support in your system. Set to “0” to exclude support. su\_EHCIGetHCNum() can be called to get the number of detected and enabled EHCI controllers.

##### **SU\_OHCI**

Set to “1” or more to indicate the maximum number of OHCI host controllers to support in your system. Set to “0” to exclude support. su\_OHCIGetHCNum() can be called to get the number of detected and enabled OHCI controllers.

##### **SU\_UHCI**

Set to “1” or more to indicate the maximum number of UHCI host controllers to support in your system. Set to “0” to exclude support. su\_UHCIGetHCNum() can be called to get the number of detected and enabled UHCI controllers.

##### **SU\_ISP116X**

Set to “1” or more to indicate the maximum number of NXP ISP116x host controllers to support in your system. Set to “0” to exclude support.

##### **SU\_ISP1362**

Set to “1” or more to indicate the maximum number of NXP ISP1362 host controllers to support in your system. Set to “0” to exclude support.

##### **SU\_ISP176X**

Set to “1” or more to indicate the maximum number of NXP ISP176x host controllers to support in your system. Set to “0” to exclude support.

## **SU\_LM3S**

Set to “1” or more to indicate the maximum number of TI LM3Sxxxx host controllers to support in your system. Set to “0” to exclude support.

## **SU\_SYNOPSISYS**

Set to “1” or more to indicate the maximum number of Synopsys DWC2 host controllers to support in your system. Set to “0” to exclude support.

## **SU\_AT91OTG**

Set to “1” to include support for a Microchip AT91 OTG USB host controller. Set to “0” to exclude support.

## **SU\_BLACKFIN**

Set to “1” to include support for an Analog Devices Blackfin USB host controller. Set to “0” to exclude support.

## **SU\_CF522XX**

Set to “1” to include support for a Freescale MCF52211/MCF52223/MCF52259 host controller. Set to “0” to exclude support.

## **SU\_MAX3421**

Set to “1” to include support for a Maxim MAX3421 host controller. Set to “0” to exclude support.

## **SU\_PD720150**

Set to “1” to include support for a NEC uPD720150 host controller. Set to “0” to exclude support.

## **SU\_RENESAS**

Set to “1” to include support for a Renesas USB host controller. Set to “0” to exclude support.

Note: Starting with smxUSBH v2.40, multiple host controller drivers of the same (or different) type may be enabled simultaneously. (v2.30 only supported enabling multiple controllers of different types.) For example, smxUSBH supports use of multiple OHCI and one EHCI or even multiple OHCI and multiple EHCIs. It supports shared interrupts used by different host controllers, whether they are the same type or not.

## **Class Driver Selection**

### **SU\_AUDIO**

Set to “1” or more to indicate the maximum number of Audio devices. Set to “0” to exclude support.

**SU\_CCID**

Set to “1” or more to indicate the maximum number of SmartCard CCID devices. Set to “0” to exclude support.

**SU\_CDCACM**

Set to “1” or more to indicate the maximum number of CDC ACM (modem) devices. Set to “0” to exclude support.

**SU\_CDCNET**

Set to “1” or more to indicate the maximum number of CDC ECM/NCM devices. Set to “0” to exclude support. You also need a TCP/IP stack, such as smxNS.

**SU\_FTDI232**

Set to “1” or more to indicate the maximum number of FTDI FT232B/LC/R Serial Converter devices. Set to “0” to exclude support. This is not a standard USB class driver.

**SU\_HID**

Set to “1” or more to indicate the maximum number of generic HID devices. Set to “0” to exclude support.

**SU\_HUB**

Set to “1” or more to indicate the maximum number of external hub devices. Set to “0” to exclude support. External hubs are NOT supported for low-end host controllers such as MAX3421.

**SU\_K4510**

Set to “1” or more to indicate the maximum number of Huawei K4510 3G Wireless Dongle devices. Set to “0” to exclude support. This is not a standard USB class driver.

**SU\_KBD**

Set to “1” or more to indicate the maximum number of keyboard devices. Set to “0” to exclude support.

**SU\_MOUSE**

Set to “1” or more to indicate the maximum number of mouse devices. Set to “0” to exclude support.

**SU\_MSTOR**

Set to “1” or more to indicate the maximum number of Mass Storage class devices. Set to “0” to exclude support.

**SU\_MT7601**

Set to “1” or more to indicate the maximum number of MediaTek MT7601 WiFi dongles. Set to “0” to exclude support.

## **SU\_MT7612**

Set to “1” or more to indicate the maximum number of MediaTek MT7612 WiFi dongles.  
Set to “0” to exclude support.

## **SU\_NET**

Set to “1” or more to indicate the maximum number of USB to Ethernet converter devices. Set to “0” to exclude support. This is not a standard USB class driver. You may need to add some code to support a specific USB to Ethernet converter chip. You also need a TCP/IP stack, such as smxNS.

## **SU\_NOVATEL**

Set to “1” or more to indicate the maximum number of Novatel Wireless Dongle devices.  
Set to “0” to exclude support.

## **SU\_OBID**

Set to “1” or more to indicate the maximum number of FEIG OBID-iscan RFID Reader devices. Set to “0” to exclude support.

## **SU\_PL2303**

Set to “1” or more to indicate the maximum number of Prolific 2303HXD-based serial converter devices. Set to “0” to exclude support.

## **SU\_PRINTER**

Set to “1” or more to indicate the maximum number of printer devices. Set to “0” to exclude support.

## **SU\_RT250X**

Set to “1” or more to indicate the maximum number of Ralink RT250x WiFi dongles. Set to “0” to exclude support.

## **SU\_RT2870**

Set to “1” or more to indicate the maximum number of Ralink RT2870 WiFi dongles.  
Set to “0” to exclude support.

## **SU\_RT3070**

Set to “1” or more to indicate the maximum number of Ralink RT3070 WiFi dongles. Set to “0” to exclude support.

## **SU\_RT3572**

Set to “1” or more to indicate the maximum number of Ralink RT3572 WiFi dongles. Set to “0” to exclude support.

## **SU\_RT5370**

Set to “1” or more to indicate the maximum number of Ralink RT5370 WiFi dongles. Set to “0” to exclude support.

## **SU\_RT5572**

Set to “1” or more to indicate the maximum number of Ralink RT5572 WiFi dongles. Set to “0” to exclude support.

## **SU\_SERIAL**

Set to “1” or more to indicate the maximum number of serial devices. Set to “0” to exclude support. This is not a standard USB class driver.

## **SU\_SIERRA**

Set to “1” or more to indicate the maximum number of Sierra Wireless Dongle devices. Set to “0” to exclude support. This is not a standard USB class driver.

## **SU\_UF8000**

Set to “1” or more to indicate the maximum number of Panasonic UF-8000 Facsimile transceiver devices. Set to “0” to exclude support. This is not a standard USB class driver.

## **SU\_V20K13**

Set to “1” or more to indicate the maximum number of Videology V20K13 camera devices. Set to “0” to exclude support. This is not a standard USB class driver.

## **SU\_VIDEO**

Set to “1” or more to indicate the maximum number of Video camera devices. Set to “0” to exclude support.

## **SU\_WCESERIAL**

Set to “1” or more to indicate the maximum number of Windows Mobile non-CDC/ACM serial devices. Set to “0” to exclude support. This is not a standard USB class driver.

## **SU\_CTEMPL**

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

## **On the Go support**

### **SU\_OTG**

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

## **Debug Settings**

### **SU\_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

### **SU\_MS\_DEBUG**

Set to “1” to provide additional mass storage debug information.

## **Miscellaneous Settings**

### **SU\_HUB\_RESET\_TWICE**

Set to “1” to reset the hub twice. Set to “0” for once. Some devices may need to reset the hub port twice before they can be enumerated. You don’t need to enable this feature in most cases. Contact MDI if you have any question about this.

### **SU\_DEV\_ADDR\_INCREASE**

Set to “1” to always increase the device address. Set to “0” to re-use any smaller available address first.

### **SU\_ENUMERATION\_RETRY**

Number of times to retry for the enumeration GetDesc request.

### **SU\_SAFETY\_CHECKS**

Set to “1” to enable special run time checking operations. Set to “0” to disable. These extra checks add extra run-time overhead so this should be disabled normally and only enabled to help diagnose a problem.

### **SU\_NEED\_NC\_MEMORY**

Set to “1” if the host controller requires non-cacheable memory for data transfers. Set to “0” if the host controller does not require it. Currently, of the controllers supported by smxUSBH, only OHCI, UHCI, and EHCI require non-cacheable memory. But other controller drivers may need this flag set to 1, if they use DMA to transfer data and the memory is in a cacheable area, such as the Synopsys DWC2 controller.



## **SU\_USE\_C\_HEAP**

Set to “1” if your system has C library heap function (malloc()/free()) support. Set to “0” to enable the internal heap functions within smxUSBH. smx users may set this to 1, since it translates malloc()/free() calls to smx heap routines.

## **SU\_HEAPSIZE**

Sets the size of internal heap memory in smxUSBH. Heap size is the sum of USBD heap size, host controller heap size, and class driver heap size. The smxUSBH internal heap should be mainly used for allocating non-cacheable memory.

## **SU\_COPY\_DATA**

Set to “1” if your system’s CPU memory space is different than the USB Host Controller’s memory space. This will increase the memory requirement.

## **SU\_SPLIT\_TRANSFER**

Set to “1” if your host controller limits the size of a data buffer of a mass storage device. This splits the request into smaller ones.

## **SU\_SPLIT\_TRANSFER\_SIZE**

The maximum data transfer size (KB) per request for a mass storage device when **SU\_SPLIT\_TRANSFER** is set.

## **SU\_HEAPSIZE**

smxUSBH will calculate the heap size based on the host controller and class driver enabled.

### **3.1.2 uport.h**

General interrupt-related hardware porting interface is defined by smxBase. Please see smxBase User’s Guide for details. The following are the smxUSBH-specific macros.

## **SU\_HC\_TASK\_STACK, SU\_HUB\_TASK\_STACK**

Stack sizes of the USB host stack tasks. There are two tasks for multitasking environment. One is host controller interrupt processing task; another is the hub task for the enumeration of devices.

**SU\_EHCI\_BASE, SU\_OHCI\_BASE, SU\_UHCI\_BASE,  
SU\_ISP116X\_BASE, SU\_ISP1362\_BASE, SU\_ISP176X\_BASE,  
SU\_SYNOPSISYS\_BASE, SU\_BLACKFIN\_BASE, SU\_RENESAS\_BASE**

**SU\_OHCI\_IRQ, SU\_UHCI\_IRQ, SU\_EHCI\_IRQ,  
SU\_ISP116X\_IRQ, SU\_ISP1362\_IRQ, SU\_ISP176X\_IRQ, SU\_MAX3421\_IRQ,  
SU\_SYNOPSISYS\_IRQ, SU\_BLACKFIN\_IRQ, SU\_RENESAS\_IRQ, SU\_LM3S\_IRQ,  
SU\_PD720150\_IRQ**

Set to the base address and IRQ number for your host controller. These do not need to be set for x86 PCI systems because this information is retrieved from the PCI BIOS. The exception is SU\_OHCI\_BASE for x86 real mode. In that case set it to a low memory address in the range 0xC8000 to 0xF0000 and see if it works for your system. Run our OHCIBASE utility from DOS to see what addresses will work, if any. See 5.5.1 OHCI for x86 Real Mode for more information.

### **SU\_NC\_MEM\_START**

Set the non-cacheable memory start address. Separate non-cacheable memory is needed if SU\_NEED\_NC\_MEMORY is set to 1 and you enabled data cache on your system.

**SU\_FREESCALE\_EHCI\_FS\_LS, SU\_FREESCALE\_EHCI\_ULPI,  
SU\_FREESCALE\_EHCI\_UTMI**

Set Freescale EHCI-compatible host controller transceiver types. Check the processor's manual to determine which one should be enabled.

### **SU\_USB\_INT\_DISPATCHER**

Set to 1 if the USB host controller's interrupt needs a special dispatcher. Currently, of the processors we have supported, this is only needed to be 1 for NXP LPC17xx and LPC24XX.

## **3.2 Building the Library**

After configuring ucfg.h (see 3.1.1 ucfg.h), build the library with the makefile or project file supplied in the build directory (e.g. IAR.ARM). It is built like other SMX module libraries, as documented in the SMX Quick Start (see E.10 Referenced Documents). If a makefile is provided, run the mak.bat file to invoke it. Run mak.bat without arguments for syntax help.

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

For non-SMX releases, a simple demo is provided in \SMX\APP\NORTOS\usbhdemo.c

For SMX releases, the demos are stored in \SMX\APP\DEMO. They are:

usbhdemo.c	main USB Host demo
fsdemo.c	demonstrates smxFS USB disk driver

The demos are integrated with the smx Protosystem. They are enabled just like all other SMX module demos, as documented in the SMX Quick Start (E.10 Referenced Documents). For makefile builds, simply uncomment the macros **subh** in pro.mak and **subhdm** or **sfsdm** in demodefs.mki.

Each class driver demo is enabled when that class is enabled in ucfg.h (e.g. **SU\_MOUSE**). 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.

The following is a summary of what each class driver demo does:

**Mouse** This demo shows the mouse movement and/or click events. The display format is:

*L M R X:xx Y:xx W:U/D*

If left button is pressed then L will be displayed. M is displayed for middle button and R for right button. X and Y indicate movement of the mouse and W is the wheel of the mouse, Up or Down.

**Keyboard** This demo shows which key is pressed. For example if the user presses the U key it will display “U” on the terminal or screen. Some shift keys will also be displayed such as *Alt, Ctrl, Shift, Num Lock, Caps Lock*.

**HID** This demo shows events from a generic HID device such as a Joystick or Multimedia keyboard. The information includes the event’s type, for example, key or relative, the event’s code, for example, *Key 0* or *Mouse Left button*, the event value, for example 1 or pressed and 0 for released.

**Mass Storage** This demo reads all the sectors of the flash disk that is plugged in. It shows the sector number it is reading, for example: *Testing Reading... 12345*

**Net** This demo calls Portinit() for smxNS, after the USB to Ethernet adapter is configured by the host stack. smxNS or another TCP/IP stack does the remaining work for the data transfer.

**Printer** This demo sends data to the printer to print information such as “*This is a USB printer*”. Because different printers use different print control languages, the demo only supports HP LaserJet or HP Desktop 3500 printers. For other printers, you may need to encode the print data yourself.

**Serial** This demo sends data to a serial device and then reads back the data, comparing the result to see if it matches. It assumes the serial device is in the loop back test mode and sends back any received data. This demo shows the data length it has written, for example: “*WrittenData: 12345*”

**CDC ACM** This demo sends AT commands to the CDC ACM device. Normally it is a USB modem and it displays the response for the AT command to the terminal or screen. After that it tries to dial a number, logs on to an Internet Service Provider (ISP), and then hangs up. You need to configure the ISP’s phone number and username/password by setting the macros DIAL\_PHONE\_NUM, DIAL\_USER\_NAME and DIAL\_PASSWORD at the top of usbhdemo.c.

TEST\_READ\_POLL is a flag to use a separate task to poll the incoming Bulk data. Polling APIs such as su\_CDCACMReadPoll () should be used if you need to implement full duplex communication for CDC ACM devices.

**CDC NET** There is no special demo for CDC ECM/NCM driver. smxNS demo has code to init and config the interface, and the normal smxNS demo can be used to test the function of it, such as telnet server or DHCP server.

**FTDI232** This demo receives data from the USB to serial converter and then sends that data back. You can connect the converter to a PC serial port and use a terminal emulator program (e.g. TeraTerm) to input a message, and then you will see the echo returned by this demo.

**Audio** This demo records about 10 seconds voice data from a USB microphone and then plays it back through a USB speaker. (Typically a headset is used which has both.) When the demo shows “Testing Recording”, talk into the microphone. Then the demo shows “Testing Playback”, playing the voice just recorded.

**PL2303** This demo receives data from the USB to serial converter and then sends that data back. You can connect the converter to a PC serial port and use a terminal emulator program to input a message. You will see the message echoed by this demo.

**WiFi Dongle** Refer to the smxWiFi User’s Guide for more information about the smxWiFi demo.

**Video** This demo capture data from the video camera. If we don’t have a display driver for the board, the demo will not display the image in this demo. We also do not have JPEG decoder so the demo will not display MJPEG format video data on the display even if we have display driver.

**V20K13** This demo receives data from the V20K13 camera we don’t have the display driver here so we will not display the image in this demo, just show the number of byte we get.

### 3.4 Initialization

smxUSBH is automatically initialized by an SMX application, if SMXUSBH is defined by the application makefile or project file. This is done by smxusbh\_init(), which is called by smx\_modules\_init(). For non-SMX applications, call su\_Initialize() from your initialization code.

Note that the hardware initialization requires delays. These are done with a polling loop (see sb\_OS\_WAIT\_USEC\_POLL() and sb\_OS\_WAIT\_MSEC\_POLL()). For SMX these macros map onto BSP delay loops. These are implemented using the constant **BSP\_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. USB Class Drivers**

### **4.1 Class Driver API Overview**

This section describes the general class driver structure and class driver API for each device that smxUSBH supports.

The overall structure of the application interface to a device depends upon the type of device. For a mouse or a keyboard, the interface is rather simple. The application simply registers a callback function, and then receives information on mouse movement or key presses when the callback function is called. The application can also check to see if a mouse or keyboard has been connected by using a specific status function.

For mass storage devices or printers, the application must ensure that the device is connected before attempting to use the device. Ideally a task should be created that monitors the connection state of the device. Alternatively, the device event callback function can be used. Once a status call indicates that the device is connected, the application can start making calls to set up and use the device. If a device returns an error while it is being used, the connection status should be rechecked to make sure that the device was not unplugged.

**Note:** For most smxUSBH callback functions, the application should not make any blocked function calls because this may block either USB host controller or class driver. smxUSBH may not work properly if a callback function blocks.

The following sections provide detailed descriptions of the class driver APIs.

### **4.2 Class Driver Interface (USBDI)**

This section describes important interfaces used in a class driver.

#### **4.2.1 Register the Class Driver with the USB Driver**

A Class Driver must first be registered with the USB Driver (USBD). When the USB Driver detects that a new device has been plugged in, it searches through a list of registered devices to find an appropriate driver. To register the class driver with the USB Driver, perform the following steps:

1. Declare a global structure of type SU\_CLS\_INFO. The structure is declared as follows in the file ustruct.h:

```
typedef struct SU_CLS_INFO
{
    SU_LIST drvList;
    const char *devName;
    SU_CHK_DEV *devArray;
    SB_OS_MUTEX_HANDLE mutexDrv;
    void *(*devConn)(SU_DEV_INFO_PTR pDevInfo, uint intf,
                    const SU_CHK_DEV *id);
    void (*devClose)(SU_DEV_INFO_PTR pDevInfo, void *);
} SU_CLS_INFO;
```

You only need to fill in the following members:

#### ***devName***

This is the driver's name, such as "usb-mouse". The name has no special meaning so you can name it anything you like.

#### ***devConn***

USB D calls this function when it detects a new device and finds that this class driver can handle that new device. You can indicate which kind of device this class driver can handle in deviceArray, documented below.

When this function is called, the device has already been configured by the USB driver through the device's default pipe. This function can perform additional initialization. For example, this function may retrieve the manufacturer, product, and serial number information for the device, retrieve an additional endpoint and/or class/vendor specific descriptor according to device specific requirements, and allocate data transfer buffers which will be used for the device.

This function should also allocate a device specific driver information structure and return it as void \*. This device specific (class) driver information structure can be considered as a handler for this device and it should maintain appropriate device state information to allow for the proper operation of the device.

#### ***devClose***

USB D calls this function when it detects that this device has been removed from the USB system. This function typically performs clean up operations, such as removing pending requests and freeing allocated data buffers and internal data structures. The second parameter of this function is the device specific driver information structure returned by deviceConn function.

#### ***deviceArray***

This is a list of devices you want this class driver to handle. Each item in this array is a structure of type SU\_CHK\_DEV. This structure is defined as:

```
typedef struct SU_CHK_DEV
{
    u16 checkMask;
    u16 vendor;
    u16 product;
    u8 devClass;
    u8 devSubClass;
    u8 devProtocol;
    u8 intfClass;
    u8 intfSubClass;
    u8 intfProtocol;
} SU_CHK_DEV;
```

### ***checkMask***

This bitmap indicates which members of SU\_CHK\_DEV structure need to match the values reported by a device in order for this device driver to be associated with the device. The default setting is for the device class, device subclass, and device protocol to match. Mask definitions are in the file udriver.h.

### ***vendor***

Vendor ID to be checked by the driver. 0 indicates don't care.

### ***product***

Product ID to be checked by the driver. 0 indicates don't care.

### ***devClass***

Device Class to be checked by the driver. 0 indicates don't care.

### ***devSubClass***

Device Sub Class to be checked by the driver. 0 indicates don't care.

### ***devProtocol***

Device Protocol to be checked by the driver. 0 indicates don't care.

### ***intfClass***

Interface Class to be checked by the driver. 0 indicates don't care.

### ***intfSubClass***

Interface Sub Class to be checked by the driver. 0 indicates don't care.

### ***intfClass***

Interface Class to be checked by the driver. 0 indicates don't care.

### ***intfProtocol***

Interface Protocol to be checked by the driver. 0 indicates don't care.

2. Call function `su_AddDrvInfoToList()` to register your driver with USBDB.

## 4.2.2 Transfer of Data between the Class Driver and the USB Driver

### Transfer data via the Default Control Pipe:

Call the function `su_SendCtrlReq()` to transfer data through the Default Control Pipe. You may use this method to set up a device specific feature. The request's parameter is passed in the function's parameter list. Check your device specification and standard USB request values to find out which value you should use.

### Transfer data via another pipe:

Call the function `su_SendReq()` to transfer data via another pipe. To cancel the request, call the function `su_UnlinkReq()`.

Before you call `su_SendReq()`, you should set up a `SU_REQ_INFO` structure and initialize appropriate members within this structure. These may include:

#### *pDev*

The `SU_DEV_INFO` structure pointer. Usually this is passed by the USBDB when calling the `deviceConn` function and you should keep it in your own driver information structure.

#### *pipe*

A bitmap indicating the transfer direction, transfer type and endpoint number for this request.

#### *pTransferBuf*

The data buffer for the request.

#### *transferBufLen*

The data buffer's length.

#### *completeFunc*

The callback function used when the request is done. You may use it to inform your driver.

#### *pContext*

Your own class driver information structure pointer.

This function return 0 for success.

In order to call `su_UnlinkReq()`, just pass the pointer to `SU_REQ_INFO` structure you passed to `su_SendReq()`;



## 4.3 Audio

### 4.3.1 API

The application level interface is defined in **uaudio.h**. Normally, this kind of device is a USB headset, speaker or microphone. The interface includes:

```
BOOLEAN su_AudioInserted(uint iID);  
void su_AudioRegDevEvtCallback(SU_PDEVEVTCBFUNC func);
```

#### Record

```
int su_AudioRecGetChannelNum(uint iID);  
int su_AudioRecGetChannelInfo(uint iID, uint iChannel, SU_AUDIO_CHAN_INFO *pInfo);  
int su_AudioRecGetCurChannel(uint iID, uint *piChannel);  
int su_AudioRecGetCurVolume(uint iID, uint iChannel, uint *piVolume, uint *piMax, uint *piMin,  
uint *piRes);  
  
int su_AudioRecGetCurMute(uint iID, uint iChannel, uint *piMute);  
int su_AudioRecSelectChannel(uint iID, uint iChannel);  
int su_AudioRecSetVolume(uint iID, uint iChannel, uint iVolume);  
int su_AudioRecSetMute(uint iID, uint iChannel, uint iMute);  
int su_AudioRecGetFormatNum(uint iID);  
int su_AudioRecGetFormat(uint iID, uint iIndex, SU_AUDIO_FORMAT_INFO *pFormat);  
int su_AudioRecSetFormat(uint iID, uint wFormat, u32 dwSamRate, uint iBits, uint Channel);  
int su_AudioRecOpen(uint iID);  
int su_AudioRecClose(uint iID);  
int su_AudioRecRead(uint iID, u8 *pData, uint iLen);
```

#### Playback

```
int su_AudioPlaybackGetChannelNum(uint iID);  
int su_AudioPlaybackGetChannelInfo(uint iID, uint iChannel, SU_AUDIO_CHAN_INFO *pInfo);  
int su_AudioPlaybackGetCurVolume(uint iID, uint iChannel, uint *piVolume, uint *piMax,  
uint *piMin, uint *piRes);  
  
int su_AudioPlaybackGetCurMute(uint iID, uint iChannel, uint *piMute);  
int su_AudioPlaybackSetVolume(uint iID, uint iChannel, uint iVolume);  
int su_AudioPlaybackSetMute(uint iID, uint iChannel, uint iMute);  
int su_AudioPlaybackGetFormatNum(uint iID);  
int su_AudioPlaybackGetFormat(uint iID, uint iIndex, SU_AUDIO_FORMAT_INFO *pFormat);  
int su_AudioPlaybackSetFormat(uint iID, uint wFormat, u32 dwSamRate, uint iBits, uint Channel);  
int su_AudioPlaybackOpen(uint iID);  
int su_AudioPlaybackClose(uint iID);  
int su_AudioPlaybackWrite(uint iID, u8 *pData, uint iLen);
```

#### MIDI

```
BOOLEAN su_MIDIInserted(uint iID);  
int su_MIDIPackEvent(u8 *pData, SU_AUDIO_MIDI_EVENT *pEvent);  
int su_MIDIUnpackEvent(u8 *pData, SU_AUDIO_MIDI_EVENT *pEvent);  
int su_MIDIRead(uint iID, u8 *pData, uint iLen, uint iTimeout);  
int su_MIDIWrite(uint iID, u8 *pData, uint iLen);  
void su_MIDISetDataReadyNotify(MIDIDATAREADYFUNC handler);  
int su_MIDIStartReadTask(uint iID);
```

int **su\_MIDIStopReadTask**(uint iID);  
BOOLEAN **su\_MIDIReadTaskStarted**(uint iID)  
int **su\_MIDIOutCableNum**(uint iID)

int **su\_AudioRecGetChannelNum** (uint iID)

**Summary** Get the number of recording channels.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.  
**If the USB audio device is a USB speaker then this function will return 0. You can use this function to decide if this device supports recording function.**

**Parameters** iID Index of the device. Should be less than SU\_AUDIO.

**Returns** > 0 Number of audio recording channels.  
== 0 This device has no recording channel.  
< 0 An error occurred.

int **su\_AudioRecGetChannelInfo**(uint iID, uint iChannel, SU\_AUDIO\_CHAN\_INFO \*pInfo)

**Summary** Get a recording channel's information based on the index.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver. Call **su\_AudioRecGetChannelNum()** to get the total number of recording channels and then retrieve the channel information one by one.

**Parameters** iID Index of the device. Should be less than SU\_AUDIO.  
iChannel Index of the channel, from 1 to **su\_AudioRecGetChannelNum()**.  
pInfo The structure pointer for the channel  
The information for the channel is defined as:

```
typedef struct
{
    uint iID;
    uint iFeature;
    char szName[32];
}SU_AUDIO_CHAN_INFO;
```

iID is the internal ID of this channel  
iFeature is the bitmap of this channel. Valid bits include:  
SU\_AUDIO\_MUTE\_PRESENT  
SU\_AUDIO\_VOLUME\_PRESENT  
SU\_AUDIO\_BASS\_PRESENT

SU\_AUDIO\_MID\_PRESENT  
SU\_AUDIO\_TREBLE\_PRESENT  
SU\_AUDIO\_GRAPHIC\_EQUALIZER\_PRESENT  
SU\_AUDIO\_AUTOMATIC\_GAIN\_PRESENT  
SU\_AUDIO\_DELAY\_PRESENT  
SU\_AUDIO\_BASS\_BOOST\_PRESENT  
SU\_AUDIO\_LOUDNESS\_PRESENT

szName is the name of this channel

**Returns** == 0 Function executed successfully.  
< 0 An error occurred.

int **su\_AudioRecGetCurChannel**(uint IID, uint \*piChannel)

**Summary** Get the currently selected recording channel.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters** IID Index of the device. Should be less than SU\_AUDIO.  
piChannel Index of the currently selected recording channel

**Returns** == 0 Function executed successfully.  
< 0 An error occurred.

int **su\_AudioRecGetCurVolume**(uint IID, uint iChannel, uint \*piVolume, uint \*piMax, uint \*piMin, uint \*piRes)

**Summary** Get a recording channel's volume information.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters** IID Index of the device.  
iChannel Index of the channel for which you want to get the volume information, from 1 to **su\_AudioRecGetChannelNum()**.  
piVolume Pointer to current volume  
piMax Pointer to maximum volume  
piMin Pointer to minimum volume  
piRes Pointer to volume resolution

**Returns** == 0 Function executed successfully.  
< 0 An error occurred.

int **su\_AudioRecGetCurMute**(uint iID, uint iChannel, uint \*piMute)

**Summary** Get a recording channel's mute status.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters**

iID	Index of the device.
iChannel	Index of the channel for the mute status request, from 1 to <b>su_AudioRecGetChannelNum()</b> .
piMute	Pointer to mute status

**Returns**

== 0	Function executed successfully.
< 0	An error occurred.

int **su\_AudioRecSelectChannel**(uint iID, uint iChannel)

**Summary** Select a recording channel.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters**

iID	Index of the device.
iChannel	Index of the channel for the recording selection.

**Returns**

== 0	Function executed successfully.
< 0	An error occurred.

int **su\_AudioRecSetVolume**(uint iID, uint iChannel, uint iVolume)

**Summary** Set the recording volume.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters**

iID	Index of the device.
iChannel	Index of the channel for the volume setting.
iVolume	The volume value.

**Returns**

== 0	Function executed successfully.
< 0	An error occurred.

int **su\_AudioRecSetMute**(uint iID, uint iChannel, uint iMute)

**Summary** Set the recording mute status.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters** iID Index of the device.  
iChannel Index of the channel for the mute status setting.  
iMute Mute or not. 1 for mute and 0 for unmute

**Returns** == 0 Function executed successfully.  
< 0 An error occurred.

int **su\_AudioRecGetFormatNum**(uint iID)

**Summary** Get the supported recording format number.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters** iID Index of the device.

**Returns** >= 0 Supported format number.  
< 0 An error occurred.

int **su\_AudioRecGetFormat**(uint iID, uint iIndex, SU\_AUDIO\_FORMAT\_INFO \*pFormat)

**Summary** Set the detailed supported recording format

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters** iID Index of the device.  
iIndex Index of the format. It should be less than **su\_AudioRecGetFormatNum()**.  
pFormat Pointer to the format information structure. The structure is defined as:

```

typedef struct
{
    u16 wFormatTag;
    u8 bChannels;
    u8 bBits;
    uint iSamRateNum;
    u32 SamRate[SU_AUDIO_MAX_FREQ];
}SU_AUDIO_FORMAT_INFO;

```

wFormatTag is the data format:

```

SU_AUDIO_FORMAT_PCM
SU_AUDIO_FORMAT_PCM8
SU_AUDIO_FORMAT_IEEE_FLOAT
SU_AUDIO_FORMAT_ALAW
SU_AUDIO_FORMAT_MULAW

```

bChannels is the channel number of the data, 1 or 2.

bBits is the number of bits per sample, 8 or 16

iSamRateNum is the supported sample rate number

SamRate is the array of supported sample rate values, for example, 22050, 44100 or 48000

**Returns**      == 0    Function executed successfully.  
                 < 0    An error occurred.

int **su\_AudioRecSetFormat**(uint iID, uint wFormat, u32 dwSamRate, uint iBits, uint iChannel)

**Summary**      Set the recording data format.

**Details**      This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver. Call this function at least one time before calling **su\_AudioRecOpen**(). The format must be supported by the device. Call **su\_AudioRecGetFormat**() to get all the supported formats.

**Parameters**    iID                Index of the device.  
                 wFormat        Data format macro. For example SU\_AUDIO\_FORMAT\_PCM.  
                 dwSamRate    Sample rate. For example, 44100.  
                 iBits            Number of bits per sample. For example, 16.  
                 iChannel        The channel number to sample. For example 1.

**Returns**      == 0    Function executed successfully.  
                 < 0    Format is not supported or an error occurred.

int **su\_AudioRecOpen**(uint IID)

**Summary** Open and start the recording channel for recording.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver. Call **su\_AudioRecSetFormat()** at least once before calling this function.

**Parameters** IID Index of the device.

**Returns** == 0 Function executed successfully.  
< 0 Could not open the device.

int **su\_AudioRecClose**(uint IID)

**Summary** Close and stop the recording channel.

**Details** This function must be called **after** you call **su\_AudioRecOpen()** and you want to stop the recording.

**Parameters** IID Index of the device.

**Returns** == 0 Function executed successfully.  
< 0 Could not close the device.

int **su\_AudioRecRead**(uint IID, u8 \*pData, uint iLen)

**Summary** Get the recorded data.

**Details** This function must be called **after** calling **su\_AudioRecOpen()** and **before** calling **AudioRecClose()**.

**Parameters** IID Index of the device.  
pData Buffer pointer to the recorded data.  
iLen Length of the data buffer.

**Returns** >= 0 Length of data recorded.  
< 0 An error occurred.

int **su\_AudioPlaybackGetChannelNum** (uint iID)

**Summary** Get the playback channel number of the audio device.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.  
**If the USB audio device is a USB Microphone then this function will return 0. Use this function to decide if this device supports playback function.**

**Parameters** iID Index of the device. Should be less than SU\_AUDIO.

**Returns** > 0 Channel number for audio playback.  
== 0 This device has no playback channel.  
< 0 An error occurred.

int **su\_AudioPlaybackGetChannelInfo**(uint iID, uint iChannel, SU\_AUDIO\_CHAN\_INFO \*pInfo)

**Summary** Get a playback channel's information based on the index.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver. Call **su\_AudioPlaybackGetChannelNum()** to get the total number of recording channels and then retrieve the channel information one by one.

**Parameters** iID Index of the device. Should be less than SU\_AUDIO.  
iChannel Index of the channel, from 1 to **su\_AudioPlaybackGetChannelNum()**.  
pInfo The structure pointer for the channel

The information of the channel is defined as:

```
typedef struct
{
    uint iID;
    uint iFeature;
    char szName[32];
}SU_AUDIO_CHAN_INFO;
```

iID is the internal ID of this channel

iFeature is the bitmap of this channel.. Valid bits include:

```
SU_AUDIO_MUTE_PRESENT
SU_AUDIO_VOLUME_PRESENT
SU_AUDIO_BASS_PRESENT
SU_AUDIO_MID_PRESENT
SU_AUDIO_TREBLE_PRESENT
```



SU\_AUDIO\_GRAPHIC\_EQUALIZER\_PRESENT  
SU\_AUDIO\_AUTOMATIC\_GAIN\_PRESENT  
SU\_AUDIO\_DELAY\_PRESENT  
SU\_AUDIO\_BASS\_BOOST\_PRESENT  
SU\_AUDIO\_LOUDNESS\_PRESENT

szName is the name of this channel

**Returns**     > 0     Audio playback channel number.  
              == 0    This device has no playback channel.  
              < 0     An error occurred.

int **su\_AudioPlaybackGetCurVolume**(uint iID, uint iChannel, uint \*piVolume, uint \*piMax, uint \*piMin, uint \*piRes)

**Summary**     Get current playback volume information.

**Details**     This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters**   iID             Index of the device.  
                  iChannel       Index of the channel to get the volume information  
                  piVolume       Pointer to current volume  
                  piMax          Pointer to maximum volume  
                  piMin          Pointer to minimum volume  
                  piRes          Pointer to volume resolution

**Returns**     == 0     Function executed successfully.  
              < 0     An error occurred.

int **su\_AudioPlaybackGetCurMute**(uint iID, uint iChannel, uint \*piMute)

**Summary**     Get current playback mute status.

**Details**     This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters**   iID             Index of the device.  
                  iChannel       Index of the channel to get the mute status.  
                  piMute         Pointer to current mute status.

**Returns**     == 0     Function executed successfully.  
              < 0     An error occurred.

int **su\_AudioPlaybackSetVolume**(uint iID, uint iChannel, uint iVolume)

**Summary** Set the playback volume.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters**

iID	Index of the device.
iChannel	Index of the channel to set the volume.
iVolume	Volume value.

**Returns**

== 0	Function executed successfully.
< 0	An error occurred.

int **su\_AudioPlaybackSetMute**(uint iID, uint iChannel, uint iMute)

**Summary** Set the playback mute status.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters**

iID	Index of the device.
iChannel	Index of the channel to set the mute status.
iMute	Mute or not.

**Returns**

== 0	Function executed successfully.
< 0	An error occurred.

int **su\_AudioPlaybackGetFormatNum**(uint iID)

**Summary** Get the supported playback format number.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters**

iID	Index of the device.
-----	----------------------

**Returns**

>= 0	Supported format number.
< 0	An error occurred.

int **su\_AudioPlaybackGetFormat**(uint IID, uint iIndex, SU\_AUDIO\_FORMAT\_INFO \*pFormat)

**Summary** Set the detailed supported playback format.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver.

**Parameters**

iID	Index of the device.
iIndex	Index of the format. It should be less than <b>su_AudioPlaybackGetFormatNum()</b> .
pFormat	Pointer of the format information structure. The structure is defined as:

```
typedef struct
{
    u16 wFormatTag;
    u8 bChannels;
    u8 bBits;
    uint iSamRateNum;
    u32 SamRate[SU_AUDIO_MAX_FREQ];
}SU_AUDIO_FORMAT_INFO;
```

wFormatTag is the data format which is one of the following macro:

```
SU_AUDIO_FORMAT_PCM
SU_AUDIO_FORMAT_PCM8
SU_AUDIO_FORMAT_IEEE_FLOAT
SU_AUDIO_FORMAT_ALAW
SU_AUDIO_FORMAT_MULAW
```

bChannels is the channel number of the data, 1 or 2.

bBits is the number of bits per sample, 8 or 16

iSamRateNum is the supported sample rate number

SamRate is the array of supported sample rate values, for example, 22050, 44100 or 48000

**Returns**

= 0	Function executed successfully.
< 0	An error occurred.

int **su\_AudioPlaybackSetFormat**(uint IID, uint wFormat, u32 dwSamRate, uint iBits, uint iChannel)

**Summary** Set the playback data format.

**Details** This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver. Call this function at least one time before calling **su\_AudioPlaybackOpen()**. The format must be supported by the device. Call **su\_AudioPlaybackGetFormat()** to get all the supported formats.

**Parameters**    iID            Index of the device.  
                  wFormat        Data format macro such as SU\_AUDIO\_FORMAT\_PCM.  
                  dwSamRate    Sample rate.  
                  iBits          Number of bits per sample.  
                  iChannel      Number of channels for each sample.

**Returns**        == 0    Function executed successfully.  
                  < 0    Format is not supported or an error occurred.

int **su\_AudioPlaybackOpen**(uint iID)

**Summary**        Open and start the playback channel for recording.

**Details**        This function must be called **after** the USB Audio device has been connected to the USB and configured by the device driver. Call **su\_AudioPlaybackSetFormat()** at least once before you call this function.

**Parameters**    iID        Index of the device.

**Returns**        == 0    Function executed successfully.  
                  < 0    Could not open the device.

int **su\_AudioPlaybackClose**(uint iID)

**Summary**        Close and stop the playback channel.

**Details**        Call **su\_AudioPlaybackOpen()** before calling this function.

**Parameters**    iID        Index of the device.

**Returns**        == 0    Function executed successfully.  
                  < 0    Could not close the device.

int **su\_AudioPlaybackWrite**(uint iID, u8 \*pData, uint iLen)

**Summary**        Send the playback data

**Details**        Call **su\_AudioPlaybackOpen()** before calling this function. Call this function **before** calling **AudioPlaybackClose()**.

**Parameters**    iID        Index of the device.  
                  pData      Pointer to playback data buffer.  
                  iLen      Length of the data buffer.

**Returns**      $\geq 0$    Length of data sent to the audio device's internal buffer.  
               $< 0$      An error occurred.

BOOLEAN **su\_MIDIInserted**(uint iID)

**Summary**     Check if a USB MIDI device is inserted.

**Details**     Before any MIDI operations, this function must be called to assure a USB MIDI device is connected.

**Parameters**  iID     Index of the device.

**Returns**     TRUE            A USB MIDI device has been connected to the USB and initialized by the device driver.  
              FALSE          No USB MIDI device has been connected.

int **su\_MIDIPackEvent**(u8 \*pData, SU\_AUDIO\_MIDI\_EVENT \*pEvent)

**Summary**     Pack a MIDI event structure into a memory space

**Details**     MIDI devices accept data in a format of MIDI event defined in the "Universal Serial Bus Device Class Definition for MIDI device" document. Please refer to Figure 8: 32-bit USB-MIDI Event Packet for details. When sending data to a MIDI device, call this function to convert to this format.

**Parameters**  \*pData        The memory where a MIDI event structure will be packed to.  
              \*pEvent      The MIDI event structure used to pack to memory. The structure is defined as:

```
typedef struct
{
    u8 iCN;               /* Cable Number */
    u8 iCIN;             /* Code Index Number */
    u8 iMIDI_0;          /* MIDI data byte 0 */
    u8 iMIDI_1;          /* MIDI data byte 1 */
    u8 iMIDI_2;          /* MIDI data byte 2 */
}SU_MIDI_EVENT;
```

**Returns**      $\geq 0$    Data have been packed successfully.  
               $< 0$      An error occurred.

int **su\_MIDIUnPackEvent**(u8 \*pData, SU\_AUDIO\_MIDI\_EVENT \*pEvent)

**Summary**     Unpack a MIDI event from memory to an event structure.

**Details**     MIDI devices send data in a format of MIDI event defined in the “Universal Serial Bus Device Class Definition for MIDI device” document. Please refer to Figure 8: 32-bit USB-MIDI Event Packet for details. When receiving data from a MIDI device call this function to convert to a MIDI event structure.

**Parameters**   \*pData           The memory pointer to data to convert into a MIDI event.  
                  \*pEvent         The MIDI event structure where data is unpacked. The structure is defined as:

```
typedef struct
{
    u8 iCN;                 /* Cable Number */
    u8 iCIN;                /* Code Index Number */
    u8 iMIDI_0;             /* MIDI data byte 0 */
    u8 iMIDI_1;             /* MIDI data byte 1 */
    u8 iMIDI_2;             /* MIDI data byte 2 */
}SU_MIDI_EVENT;
```

**Returns**     >= 0     Data have been unpacked successfully.  
              < 0     An error occurred.

int **su\_MIDIRead**(uint iID, u8 \*pData, uint iLen, uint iTimeout)

**Summary**     Read data from the MIDI device.

**Details**     This function tries to read some data from the MIDI device. If there is no data within timeout milliseconds, it returns 0. Note: this function should not be called when the polling read task has been started.

**Parameters**   iID                Index of the device. Should be less than SU\_AUDIO.  
                  pData            Pointer to the data buffer.  
                  iLen             Length of the data buffer.  
                  iTimeout         The timeout value for this operation, in milliseconds.

**Returns**     > 0     Length of the data read from the device.  
              = 0     An error occurred or there is no data.

int **su\_MIDIWrite**(uint iID, u8 \*pData, uint iLen)

**Summary** Sends data to the MIDI device.

**Details** This function can be called once a MIDI device has been connected to the USB and initialized by the device driver. It tries to write some data to that device.

**Parameters** iID Index of the device. Should be less than SU\_AUDIO  
pData Pointer to the data buffer.  
iLen Length of the data buffer.

**Returns** > 0 Length of the data written to the device.  
= 0 An error occurred.

int **su\_MIDIOutCableNum**(uint iID)

**Summary** Get the cable number associated with the MIDI device BULK OUT endpoint.

**Details** Cable number is used to assemble the MIDI event packet when the host needs to send data to the MIDI device. Please refer to Figure 8: 32-bit USB-MIDI Event Packet in “Universal Serial Bus Device Class Definition for MIDI devices” document.

**Parameters** iID Index of the device. Should be less than SU\_AUDIO

**Returns** > =0 The cable number  
< 0 An error occurred.

void **su\_MIDISetDataReadyNotify**(MIDIDATAREADYFUNC handler)

**Summary** Register a callback function to handle the event that incoming data is ready.

**Details** This function can be called any time after smxUSBH has started.  
The callback function is defined as:  
typedef void (\*MIDIDATAREADYFUNC)(uint iID, u8 \*pData, uint iLen);  
You should only call this when you choose to use the polling task to poll for incoming data. Note: The callback function will be called from the polling task. To avoid a long delay in the next request scheduling, the application should not put long data processing code in the callback function; it should be placed in another task.

**Parameters** handler Callback function pointer.

**Returns** None

int **su\_MIDIStartReadTask**(uint iID)

**Summary** Start the read polling task.

**Details** This function will start a task to keep polling the incoming data. You should not call `su_MIDIRead()` to get the data in your application after the task is started.

**Parameters** iID Index of the device. Should be less than `SU_AUDIO`.

**Returns** 0 Task started.  
-1 Failed to start the task.

int **su\_MIDIStopReadTask**(uint iID)

**Summary** Stop the read polling task.

**Details** This function will stop the read polling task.

**Parameters** iID Index of the device. Should be less than `SU_AUDIO`.

**Returns** 0 Task stopped.  
-1 Task has not been started yet.

BOOLEAN **su\_MIDIReadTaskStarted** (uint iID)

**Summary** Check if the read polling task is started.

**Details** This function will indicate whether the polling task has been started or not.

**Parameters** iID Index of the device. Should be less than `SU_AUDIO`.

**Returns** TRUE Task started.  
FALSE Task has not been started yet.

### 4.3.2 Audio Device Limitations

There are some limitations for this audio class device driver:

1. It does not support selecting the playback channel.
2. It does not support multiple microphones or speakers in the same device.
3. It is not tested for compressed data format recording and playback, such as MPEG or AC-3.
4. It does not support Processing Units such as Dolby Prologic Processing Unit.
5. It does not support multiple MIDI stream interfaces in the same devices.



6. It does not support MIDI element units.
7. It does not support MIDI dedicated transfer bulk endpoints.

## 4.4 Communication (CDC)

### 4.4.1 ACM (Modem)

#### 4.4.1.1 API

The application level interface is defined in **ucdcacm.h**. Normally, this kind of device is used for a modem or ISDN adapter. The interface includes:

```

int          su_CDCACMOpen (uint iID)
int          su_CDCACMClose (uint iID)
BOOLEAN     su_CDCACMInserted (uint iID)
void        su_CDCACMRegDevEvtCallback(SU_PDEVEVTCBFUNC func)

int          su_CDCACMRead (uint iID, u8 * pData, int len, uint timeout)
int          su_CDCACMReadStartPoll(uint iID, uint size)
int          su_CDCACMReadStopPoll(uint iID)
int          su_CDCACMReadPoll(uint iID, u8 * pData, uint timeout)
int          su_CDCACMWrite (uint iID, u8 * pData, int len)
int          su_CDCACMGetLineState(int port, uint *piState);
int          su_CDCACMGetLineCoding(int port, u32 *pdwDTERate, u8 *pbParityType,
                                     u8 *pbDataBits, u8 *pbStopBits);

int          su_CDCACMSetLineState(int port, uint iState);
int          su_CDCACMSendLineBreak(uint iID, u16 ms);
int          su_CDCACMSetLineCoding(uint iID, u32 dwBaudrate, u8 iParity, u8 iDataBits, u8 iStopBit);

int          su_CDCACMSetCommFeature(uint iID, u16 wSelector, u16 wStatus);
int          su_CDCACMGetCommFeature(uint iID, u16 wSelector, u16 *pwStatus);

void        su_CDCACMRegisterStateChangeNotify(PHOSTCDCACMFUNC handler);

```

int **su\_CDCACMOpen** (uint iID)

**Summary**     Open the CDC ACM device.

**Details**     This function must be called **after** the USB CDC ACM device has been connected to the USB and configured by the device driver and **before** any CDCACMRead/CDCACMWrite functions. It sets the line coding information such as baud rate and number of data bits. It also sets the DCD and DTR signals of the CDC ACM device.

**Parameters**   iID     Index of the device. Should be less than SU\_CDCACM.

**Returns**       == 0     The device has been opened and the line coding and line state have also been set up.  
                   < 0     An error occurred.

int **su\_CDCACMClose** (uint iID)

**Summary** Close the CDC ACM device.

**Details** This function must be called **after** you finish any CDCACMRead/CDCACMWrite function calls. It clears the DCD and DTR signals of the serial port.

**Parameters** iID Index of the device. Should be less than SU\_CDCACM.

**Returns** == 0 The device has been closed.  
< 0 An error occurred.

BOOLEAN **su\_CDCACMInserted** (uint iID)

**Summary** Returns status indicating if a USB CDC ACM device has been connected.

**Details** This function can be called at any time to determine if a CDC ACM device has been connected to the USB port.

**Parameters** iID Index of the device. Should be less than SU\_CDCACM.

**Returns** TRUE A USB CDC ACM device has been connected to the USB and initialized by the device driver.  
FALSE No USB CDC ACM device has been connected.

int **su\_CDCACMRead** (uint iID, u8 \* pData, int len, uint timeout)

**Summary** Reads data from the serial device.

**Details** This function tries to read some data from the serial port. If there is no data within timeout milliseconds, it returns 0. This function will remove the BULK IN request after the timeout. If you need to keep polling the incoming data in a loop or task, please use the following polling functions.

**Parameters** iID Index of the device. Should be less than SU\_CDCACM.  
pData Pointer to the data buffer.  
len Length of the data buffer.  
timeout The timeout value for this operation, in milliseconds.

**Returns** > 0 Length of the data read from the device.  
= 0 An error occurred or there is no data.

int **su\_CDCACMReadPoll** (uint iID, u8 \* pData, uint timeout)

**Summary** Reads data from the serial device.

**Details** This function tries to read some data from the serial port. If there is no data within timeout milliseconds, it returns -1. This function will resubmit the BULK IN request after it gets some incoming data automatically.

**Parameters**

iID	Index of the device. Should be less than SU_CDCACM.
pData	Pointer to the data buffer. Size should be at least the same size when you call <b>su_CDCACMReadStartPoll</b> ().
timeout	The timeout value for this operation, in milliseconds.

**Returns**

> 0	Length of the data read from the device.
<= 0	An error occurred or there is no data.

int **su\_CDCACMReadStartPoll** (uint iID, int size)

**Summary** Start polling incoming data from the serial device.

**Details** This function will submit an IN request to the BULK IN endpoint. It will also allocate an internal buffer specified by size to hold the incoming data.

**Parameters**

iID	Index of the device. Should be less than SU_CDCACM.
size	Maximum size of the data buffer which the serial device may send.

**Returns**

< 0	Failed to send request.
= 0	Request submitted.

int **su\_CDCACMReadStopPoll** (uint iID)

**Summary** Stop the polling request from the serial device.

**Details** This function will unlink the IN request from the BULK IN endpoint. It will also free the internal buffer allocated.

**Parameters**

iID	Index of the device. Should be less than SU_CDCACM.
-----	---

**Returns**

< 0	Failed to unlink request.
= 0	Request removed.

int **su\_CDCACMWrite** (uint iID, u8 \* pData, int len)

**Summary** Sends data to the CDC ACM device.

**Details** This function can be called once a CDC ACM device has been connected to the USB and configured by the device driver. It tries to write some data to that device.

**Parameters** iID Index of the device. Should be less than SU\_CDCACM  
pData Pointer to the data buffer.  
len Length of the data buffer.

**Returns** > 0 Length of the data written to the device.  
= 0 An error occurred.

int **su\_CDCACMGetLineState** (uint iID, uint \*piState)

**Summary** Get the current Line State.

**Details** This function can be called once a CDC ACM device has been connected to the USB and configured by the device driver, or after getting a line state change notification. It returns the current Line State bitmap. The bitmap is compatible with USB CDC Class Specification. Values include:

SU\_CDCACM\_LINE\_IN\_DCD  
SU\_CDCACM\_LINE\_IN\_DSR  
SU\_CDCACM\_LINE\_IN\_BRK  
SU\_CDCACM\_LINE\_IN\_RI  
SU\_CDCACM\_LINE\_IN\_FRAMING  
SU\_CDCACM\_LINE\_IN\_PARITY  
SU\_CDCACM\_LINE\_IN\_OVERRUN

**Parameters** iID Index of the device. Should be less than SU\_CDCACM.  
piState Pointer to the line state bitmap.

**Returns** 1 Got the line state bitmap  
= 0 An error occurred or no device is connected.

int **su\_CDCACMGetLineCoding** (uint iID, u32 \*pdwDTERate, u8 \*pbParityType, u8 \*pbDataBits, u8 \*pbStopBits)

**Summary** Get the current Line Coding information.

**Details** This function can be called once a CDC ACM device has been connected to the USB and configured by the device driver. It returns the current device configuration, such as baud rate and number of data bits. **The default setting for the line coding is 115200 N 8 1.**

**Parameters**

iID	Index of the device. Should be less than SU_CDCACM.
pdwDTERate	Pointer to the baud rate value, if the baud rate is 115200 then it will return 115200.
pbParityType	Pointer to the parity value. Valid values include: SU_CDCACM_PARITY_NONE SU_CDCACM_PARITY_ODD SU_CDCACM_PARITY_EVEN SU_CDCACM_PARITY_MARK SU_CDCACM_PARITY_SPACE
pbDataBits	Pointer to the number of data bits value. May be 7 or 8.
pbStopBits	Pointer to the number of stop bits value. Valid values include: SU_CDCACM_STOP_BITS_1 SU_CDCACM_STOP_BITS_1_5 SU_CDCACM_STOP_BITS_2

**Returns**

1	Got the line coding value.
= 0	An error occurred or the device is not connected.

int **su\_CDCACMSetLineState** (uint iID, uint \*piState)

**Summary** Change the current Line State.

**Details** This function can be called once a CDC ACM device has been connected to the USB and configured by the device driver. This function sets the current Line State bitmap. This bitmap is compatible with USB CDC Class Specification. Valid values include:  
SU\_CDCACM\_LINE\_OUT\_RTS  
SU\_CDCACM\_LINE\_OUT\_DTR

**Parameters**

iID	Index of the device. Should be less than SU_CDCACM.
piState	Pointer to the line state bitmap.

**Returns**

1	Got the line state bitmap
= 0	An error occurred or no device is connected.

int **su\_CDCACMSendLineBreak** (uint iID, uint ms)

**Summary** Send an RS-232 style break.

**Details** This function can be called once a CDC ACM device has been connected to the USB and configured by the device driver. Some CDC ACM devices may not support this feature.

**Parameters** iID Index of the device. Should be less than SU\_CDCACM.  
ms Duration of the break.

**Returns** 1 Break has been sent out.  
= 0 An error occurred. Device does not support this feature or no device is connected.

int **su\_CDCACMSetLineCoding**(uint iID, u32 dwBaudrate, u8 iParity, u8 iDataBits, u8 iStopBit);

**Summary** Set the Line Coding information.

**Details** This function can be called once a CDC ACM device has been connected to the USB and configured by the device driver. It set the device configuration, such as baud rate and number of data bits.

**Parameters** iID Index of the device. Should be less than SU\_CDCACM.  
dwDTERate Baud rate value, for example, 115200.  
bParityType Parity value. Valid values include:  
SU\_CDCACM\_PARITY\_NONE  
SU\_CDCACM\_PARITY\_ODD  
SU\_CDCACM\_PARITY\_EVEN  
SU\_CDCACM\_PARITY\_MARK  
SU\_CDCACM\_PARITY\_SPACE  
bDataBits Data bits value. May be 7 or 8.  
bStopBits Stop bits value. Valid values include:  
SU\_CDCACM\_STOP\_BITS\_1  
SU\_CDCACM\_STOP\_BITS\_1\_5  
SU\_CDCACM\_STOP\_BITS\_2

**Returns** 1 Set the line coding value.  
= 0 An error occurred or the device is not connected.

int **su\_CDCACMGetCommFeature** (uint iID, u16 wSelector, u16 \*pwStatus)

**Summary** Return the current settings for the selected communication feature.

**Details** This function can be called once a CDC ACM device has been connected to the USB and configured by the device driver. Some CDC ACM devices may not support this feature. Valid selectors include:

SU\_CDCACM\_FEATURE\_ABSTRACT\_STATE  
SU\_CDCACM\_FEATURE\_COUNTRY\_SETTING

**Parameters**

iID	Index of the device. Should be less than SU_CDCACM.
wSelector	Feature selector.
pwStatus	Pointer to the returned status.

**Returns**

1	Got the feature.
= 0	An error occurred, device does not support this feature, or no device is connected.

int **su\_CDCACMSetCommFeature** (uint iID, u16 wSelector, u16 wStatus)

**Summary** Change the current settings for the communication feature as selected.

**Details** This function can be called once a CDC ACM device has been connected to the USB and configured by the device driver. Some CDC ACM devices may not support this feature. Valid selectors include:

SU\_CDCACM\_FEATURE\_ABSTRACT\_STATE  
SU\_CDCACM\_FEATURE\_COUNTRY\_SETTING

**Parameters**

iID	Index of the device. Should be less than SU_CDCACM.
wSelector	Feature selector.
wStatus	The new status.

**Returns**

1	Feature has been set.
= 0	An error occurred, device does not support this feature, or no device is connected.

void **su\_CDCACMRegisterStateChangeNotify**(PHOSTCDCACMFUNC handler)

**Summary** Register a callback notification function for device state changes.

**Details** This function is for INT IN transfers not BULK IN. BULK IN should be polled by the application using `su_CDCACMRead()`. INT IN must be polled by the host controller driver because only it can handle the interval properly, not the application. This function can be called any time `smxUSBH` has started.

The callback function is defined as:

```
typedef void (* PHOSTCDCACMFUNC)(uint IID, uint wNotifyType, u16 wParam);
```

Values for `wNotifyType` include:

```
SU_CDCACM_CONNECT_CHANGED  
SU_CDCACM_LINE_STATE_CHANGED  
SU_CDCACM_RESPONSE_READY
```

If the type is `SU_CDCACM_CONNECT_CHANGED`, then `wParam = 1` means connected; 0 means disconnected.

If the type is `SU_CDCACM_LINE_STATE_CHANGED`, then `wParam` contains the current line state.

`wParam` is not used for `SU_CDCACM_RESPONSE_READY`.

**Parameters**

<code>IID</code>	Index of the device. Should be less than <code>SU_CDCACM</code> .
<code>wNotifyType</code>	The notification type. Some device may not support <code>SU_CDCACM_CONNECT_CHANGED</code> type notification
<code>wParam</code>	Parameter for the notification.

**Returns** None

## 4.4.2 Serial (Windows Mobile 5 Device, CDC/ACM-like)

### 4.4.2.1 API

There are two ways to get the incoming data: Keep calling `su_SerialRead()` in the application, or call `su_SerialStartPollingTask()/su_SerialStopPollingTask()` to use the built-in polling task and use the data ready callback function.

**Please see section 4.4.2.2 Limitations for important information about this driver.**

Micro Digital provides separate USB to serial adapter drivers for FTDI FT232 and Prolific 2303HXD chips. If using one of these, select it in `ucfg.h`, and see the corresponding section below: 4.4.3 Serial Converter FTDI FT232B or 4.4.4 Serial Converter Prolific 2303HXD.



The application level interface is defined in **userial.h**. This interface includes:

```
int      su_SerialOpen (uint IID)
int      su_SerialClose (uint IID)
BOOLEAN su_SerialInserted (uint IID)
void     su_SerialRegDevEvtCallback(SU_PDEVEVTCBFUNC func)
int      su_SerialRead (uint IID, u8 * pData, int len, uint iTimeout)
int      su_SerialWrite (uint IID, u8 * pData, int len)
int      su_SerialGetLineState(int port, uint *piState);
int      su_SerialGetLineCoding(int port, u32 *pdwDTERate, u8 *pbParityType, u8 *pbDataBits,
                                u8 *pbStopBits);

int      su_SerialSendLineBreak(uint IID, u16 ms);
int      su_SerialSetLineCoding(uint IID, u32 dwBaudrate, u8 iParity, u8 iDataBits, u8 iStopBit);
void     su_SerialLineStateChangeNotify(PHOSTSERIALFUNC handler);
void     su_SerialDataReadyNotify(PSERIALDATAREADYFUNC handler);
int      su_SerialStartPollingTask(uint IID);
int      su_SerialStopPollingTask(uint IID);
BOOLEAN su_SerialPollingTaskStarted(uint IID);
```

int **su\_SerialOpen** (uint IID)

**Summary**     Open the serial port.

**Details**     This function must be called **after** the USB serial device has been connected to the USB and configured by the device driver and **before** any SerialRead/SerialWrite functions. It sets the line coding information such as baud rate and number of data bits. It also sets the DCD and DTR signals of the serial port.

**Parameters**   IID     Index of the serial device. Should be less than SU\_SERIAL.

**Returns**       == 0     The device has been opened and the line coding and line state have also been set up.  
                  < 0     An error occurred.

int **su\_SerialClose** (uint IID)

**Summary**     Close the serial port.

**Details**     This function must be called **after** finishing any SerialRead/SerialWrite function calls. It clears the DCD and DTR signals of the serial port.

**Parameters**   IID     Index of the serial device. Should be less than SU\_SERIAL.

**Returns**       == 0     The device has been closed.  
                  < 0     An error occurred.

BOOLEAN **su\_SerialInserted** (uint iID)

**Summary** Returns status indicating if a USB serial device has been connected.

**Details** This function can be called at any time to determine if a supported serial device has been connected to the USB.

**Parameters** iID Index of the serial device. Should be less than SU\_SERIAL.

**Returns** TRUE A supported USB serial device has been connected to the USB and initialized by the device driver.  
FALSE No USB serial device has been connected.

int **su\_SerialRead** (uint iID, u8 \* pData, int len, uint iTimeout)

**Summary** Reads data from the serial device.

**Details** This function tries to read some data from the serial port. If there is no data within iTimeout milliseconds, this function returns 0.

**You should not call this function if you use the built-in polling task.**

**Parameters** iID Index of the serial device. Should be less than SU\_SERIAL.  
pData Pointer to the data buffer.  
len Length of the data buffer.  
iTimeout Timeout value (millisecond) for the read operation.

**Returns** > 0 Length of the data read from the serial device.  
= 0 An error occurred or there is no data during the timeout period.

int **su\_SerialWrite** (uint iID, u8 \* pData, int len)

**Summary** Sends data to the serial device.

**Details** This function can be called once a serial device has been connected to the USB and configured by the device driver. It tries to write some data to the serial device.

**Parameters** iID Index of the serial device. Should be less than SU\_SERIAL  
pData Pointer to the data buffer.  
len Length of the data buffer.

**Returns** > 0 Length of the data written to the serial device.  
= 0 An error occurred.

int **su\_SerialGetLineState** (uint iID, uint \*piState)

**Summary** Get the current Line State.

**Details** This function can be called after a serial device has been connected to the USB and configured by the device driver, or it can be called after a line state change notification. It returns the current Line State bitmap. This bitmap is compatible with USB CDC Class Specification. Values include:

SU\_SERIAL\_LINE\_IN\_DCD  
SU\_SERIAL\_LINE\_IN\_DSR  
SU\_SERIAL\_LINE\_IN\_BRK  
SU\_SERIAL\_LINE\_IN\_RI  
SU\_SERIAL\_LINE\_IN\_FRAMING  
SU\_SERIAL\_LINE\_IN\_PARITY  
SU\_SERIAL\_LINE\_IN\_OVERRUN

**Parameters** iID Index of the serial device. Should be less than SU\_SERIAL.  
piState Pointer to the line state bitmap.

**Returns** 1 Got the line state bitmap  
= 0 An error occurred or no serial device is connected.

int **su\_SerialGetLineCoding** (uint iID, u32 \*pdwDTERate, u8 \*pbParityType,  
u8 \*pbDataBits, u8 \*pbStopBits)

**Summary** Get the current Line Coding information.

**Details** This function can be called after a serial device has been connected to the USB and configured by the device driver. It returns the current serial device configuration such as baud rate and number of data bits. **The default setting for the line coding is 115200 N 8 1.**

**Parameters** iID Index of the serial device. Should be less than SU\_SERIAL.  
pdwDTERate Pointer to the baud rate value. For example, 115200.  
pbParityType Pointer to the parity value. Valid values include:  
SU\_SERIAL\_PARITY\_NONE  
SU\_SERIAL\_PARITY\_ODD  
SU\_SERIAL\_PARITY\_EVEN  
SU\_SERIAL\_PARITY\_MARK  
SU\_SERIAL\_PARITY\_SPACE  
pbDataBits Pointer to the number of data bits value. May be 7 or 8.  
pbStopBits Pointer to the number of stop bits value. Valid values include:  
SU\_SERIAL\_STOP\_BITS\_1

SU\_SERIAL\_STOP\_BITS\_1\_5  
SU\_SERIAL\_STOP\_BITS\_2

**Returns** 1 Got the line coding value.  
= 0 An error occurred or the serial device is not connected.

int **su\_SerialSendLineBreak**(uint iID, u16 ms)

**Summary** Send RS232 line break signal.

**Details** This function can be called after a serial device has been connected to the USB and configured by the device driver. It will trigger the Line Break signal on the actual RS232 bus.

**Parameters** iID Index of the serial device. Should be less than SU\_SERIAL.  
ms Line break signal duration in millisecond.

**Returns** 0 Sent the signal.  
-1 An error occurred or the serial device is not connected.

int **su\_SerialSetLineCoding**(uint iID, u32 dwBaudrate, u8 iParity, u8 iDataBits, u8 iStopBit)

**Summary** Set the Line Coding information.

**Details** This function can be called after a serial device has been connected to the USB and configured by the device driver. It sets serial device configuration such as baud rate and number of data bits.

**Parameters** iID Index of the serial device. Should be less than SU\_SERIAL.  
dwDTERate Baud rate value. For example, 115200.  
bParityType Parity value. Valid values include:  
SU\_SERIAL\_PARITY\_NONE  
SU\_SERIAL\_PARITY\_ODD  
SU\_SERIAL\_PARITY\_EVEN  
SU\_SERIAL\_PARITY\_MARK  
SU\_SERIAL\_PARITY\_SPACE  
bDataBits Data bits value. May be 7 or 8.  
bStopBits Stop bits value. Valid values include:  
SU\_SERIAL\_STOP\_BITS\_1  
SU\_SERIAL\_STOP\_BITS\_1\_5  
SU\_SERIAL\_STOP\_BITS\_2

**Returns** 0 Set the line coding.  
-1 An error occurred or the serial device is not connected.

void **su\_SerialLineStateChangeNotify**(PHOSTSERIALFUNC handler)

**Summary** Register a callback notification function for line state changes.

**Details** This function can be called any time after smxUSBH has started.  
The callback function is defined as:  
typedef void (\* PHOSTSERIALFUNC)(uint iID, u16 wNewLineStyle);

**Parameters** iID Index of the device. Should be less than SU\_SERIAL.  
wNewLineStyle New Line State.

**Returns** None

void **su\_SerialDataReadyNotify**(PSERIALDATAREADYFUNC handler)

**Summary** Register a callback function to handle the event that incoming data is ready.

**Details** This function can be called any time after smxUSBH has started.  
The callback function is defined as:  
typedef void (\* PSERIALDATAREADYFUNC)(uint iID, u8 \*pData, uint iLen);  
You should only call this when you choose to use the built-in polling task to poll the incoming data.

**Parameters** handler Callback function pointer.

**Returns** None

int **su\_SerialStartPollingTask** (uint iID)

**Summary** Start the built-in polling task.

**Details** This function will start a built-in task to keep polling the incoming data. Only call this function after you call su\_SerialOpen(). You can not call su\_SerialRead() to get the data in your application after the task is started.

**Parameters** iID Index of the device. Should be less than SU\_SERIAL.

**Returns** 0 Task started.  
-1 Failed to start the task.

int **su\_SerialStopPollingTask** (uint iID)

**Summary** Stop the built-in polling task.

**Details** This function will stop the built-in task. Only call this function after you call su\_SerialOpen() and before su\_SerialClose().

**Parameters** iID Index of the device. Should be less than SU\_SERIAL.

**Returns** 0 Task stopped.  
-1 Task has not been started yet.

BOOLEAN **su\_SerialPollingTaskStarted** (uint iID)

**Summary** Check if the built-in polling task is started.

**Details** This function will indicate whether the built-in polling task has been started or not.

**Parameters** iID Index of the device. Should be less than SU\_SERIAL.

**Returns** TRUE Task started.  
FALSE The task is not started yet.

#### 4.4.2.2 Limitations

*Universal Serial Bus Device Class Definition for Communication Devices* ([www.usb.org/developers/defined\\_class](http://www.usb.org/developers/defined_class)) defines a set of communication devices such as modems and LAN cards. It does not define a USB serial device but it can be used to implement one, which Micro Digital has done and which has been done in Windows.

The USB Serial device may be a USB to RS232 adapter that connects to another RS232 port through a serial cable. It can also be an embedded device that uses the USB port to communicate with a USB host. In this case, there is no RS232 link. In our testing, Windows Mobile 5 devices use this kind of device to connect to a USB host. For the smxUSBH serial class driver the connection type is not relevant.

A USB Serial device that follows the Communication Devices specification can be supported by Windows 2000, XP, Vista, and 7/8/10 without any special driver. Only an .inf file is needed. However, there are many products in the market that require a special driver to work under Windows, which do not follow the existing well-defined specification. They define their own proprietary interface. smxUSBH serial follows the USB specification so **it cannot automatically support serial devices that need a special driver for Windows**. It can support any serial device that has the following features:

1. Its Interface Class number is 2 and/or 10, which means it is a CDC Communication class device or CDC Data class device.
  2. It has three endpoints (one BULK IN, one BULK OUT, and one INT IN for notification) in one or two interfaces:
    - a. CDC COMM interface contains only INT IN endpoint, and CDC Data interface contains BULK IN and BULK OUT endpoints.
    - b. CDC COMM interface contains all three endpoints.
  3. It uses the Abstract Control Mode request SET\_LINE\_CODING, GET\_LINE\_CODING, and SET\_CONTROL\_LINE\_STATE to set/get the serial device's DTE rate, parity, data bits, stop bits, and number-of-character bits. See the CDC specification v1.1 table 4 for the details.
  4. It uses the Abstract Control Mode Notification SERIAL\_STATE to notify the host through the INT IN endpoint that the RS232 Control line state has changed.
  5. The LINE\_CODING data format is compatible with CDC specification v1.1 table 50.
  6. The Control Line State data format is compatible with CDC specification v1.1 table 51
- For serial adapters and other types of serial devices that don't follow the CDC specification, additional code must be written in the smxUSBH serial driver, which could take some effort. Source code that may be provided on the CD that comes with the adapters (e.g. Linux driver) will show what is necessary to communicate with the adapter. If there is no source code, use a USB packet analyzer to study the details of messages sent while in use by a Windows application to see what is required. This could take a lot of time.

## 4.4.3 Serial Converter FTDI FT232B/LC/R

### 4.4.3.1 API

The application level interface is defined in **uftdi232.h**. This interface includes:

```
int          su_FTDIOpen(uint iID)
int          su_FTDClose (uint iID)
BOOLEAN     su_FTDIInserted (uint iID)
void        su_FTDIRegDevEvtCallback(SU_PDEVEVTCBFUNC func)
int          su_FTDIReadData (uint iID, u8 * pData, int len)
int          su_FTDIWriteData (uint iID, u8 * pData, int len)
int          su_FTDISetModemCtrl(uint iID, uint Data)
int          su_FTDISetFlowCtrl(uint iID, uint Data)
int          su_FTDISetLineCoding(uint iID, u32 dwDTERate, u16 wbParityType, u16 wDataBits,
                                                                    u16 wStopBits);

int          su_FTDIGetModemStatus(uint iID);
int          su_FTDIGetStatus(uint iID, u8 *pModemStatus, u8 *pLineStatus);
int          su_FTDISetEventChar(uint iID, u8 cData);
int          su_FTDISetErrorChar(uint iID, u8 cData);
int          su_FTDISetLatencyTimer(uint iID, uint iMS);
int          su_FTDIGetLatencyTimer(uint iID);
```

int **su\_FTDIOpen** (uint iID)

**Summary**     Open the FTDI serial converter port.

**Details**     This function must be called **after** the FTDI serial converter has been connected to the USB and configured by the device driver and **before** any FTDIRead/FTDIWrite functions. It sets the line coding information such as baud rate and number of data bits. It also sets the RTS and DTR signals of the serial port. The default flow control setting is to hardware signal CTS\_RTS. The default baud rate is 115200. The default line coding is 8 data bits, none parity and 1 stop bit.

**Parameters**   iID     Index of the serial device. Should be less than SU\_FTDI232.

**Returns**        == 0     The device has been opened and the line coding and line state have also been set up.  
                  < 0     An error occurred.

int **su\_FTDClose** (uint iID)

**Summary**     Close the FTDI serial converter port.

**Details**     This function must be called **after** finishing any FTDIRead/FTDIWrite function calls. It clears the DSR and DTR signals of the serial converter port.



**Parameters**    iID      Index of the serial device. Should be less than SU\_FTDI.

**Returns**        == 0    The device has been closed.  
                 < 0    An error occurred.

BOOLEAN **su\_FTDIInserted** (uint iID)

**Summary**        Returns status indicating if a FTDI serial converter device has been connected.

**Details**         This function can be called at any time to determine if a supported serial converter device has been connected to the USB.

**Parameters**    iID                    Index of the serial converter device. Should be less than SU\_FTDI232.

**Returns**        TRUE                  A supported FTDI serial converter device has been connected to the USB and initialized by the device driver.  
                 FALSE                No FTDI serial converter device has been connected.

int **su\_FTDIReadData** (uint iID, u8 \* pData, int len)

**Summary**        Reads data from the serial converter device.

**Details**         This function tries to read some data from the serial port. If there is no data within the latency time (default time is 40 ms), this function returns 0.

**Parameters**    iID      Index of the serial converter device. Should be less than SU\_FTDI232.  
                 pData    Pointer to the data buffer.  
                 len      Length of the data buffer.

**Returns**        > 0      Length of the data read from the serial converter device.  
                 = 0      An error occurred or there is no data.

int **su\_FTDIWriteData** (uint iID, u8 \* pData, int len)

**Summary**        Sends data to the serial converter device.

**Details**         This function can be called once a serial converter device has been connected to the USB and configured by the device driver. It tries to write some data to the serial converter device.

**Parameters**    iID      Index of the serial converter device. Should be less than SU\_FTDI232

pData Pointer to the data buffer.  
len Length of the data buffer.

**Returns** > 0 Length of the data written to the serial converter device.  
= 0 An error occurred.

int **su\_FTDISetModemCtrl** (uint iID, uint Data)

**Summary** Set the DTR and RTS signal.

**Details** This function can be called once a serial converter device has been connected to the USB and configured by the device driver and after calling su\_FTDIOpen(). su\_FTDIOpen() sets DTR and RTS high, by default. Call this function to override the defaults.

**Parameters** iID Index of the serial converter device. Should be less than SU\_FTDI232  
Data DTR and RTS settings. SU\_FTDI232\_DTR will set DTR to high and SU\_FTDI232\_RTS will set RTS to high. You can also OR those two macros together to setting both DTR and RTS to high.

**Returns** == 0 DTR and RTS has been set.  
-1 An error occurred.

int **su\_FTDISetFlowCtrl** (uint iID, uint Data)

**Summary** Set the Flow Control of the FTDI serial converter.

**Details** This function can be called once a serial converter device has been connected to the USB and configured by the device driver and after calling su\_FTDIOpen(). su\_FTDIOpen() sets the default Flow Control to SU\_FTDI232\_FLOWCTRL\_RTSCTS. Call this function to override the Flow Control default.

**Parameters** iID Index of the serial converter device. Should be less than SU\_FTDI232  
Data Flow Control settings. Valid settings include:  
SU\_FTDI232\_FLOWCTRL\_NONE  
SU\_FTDI232\_FLOWCTRL\_RTSCTS  
SU\_FTDI232\_FLOWCTRL\_DTRDSR  
SU\_FTDI232\_FLOWCTRL\_XONXOFF

**Returns** == 0 Flow Control has been set.  
-1 An error occurred.

int **su\_FTDISetLineCoding** (uint iID, u32 dwDTERate, u16 wParityType,  
u16 wDataBits, u16 wStopBits)

**Summary** Set the Baudrate, Parity type, Data bits and Stop bits of the FTDI serial converter.

**Details** This function can be called once a serial converter device has been connected to the USB and configured by the device driver and after calling su\_FTDIOpen(). su\_FTDIOpen() sets default line values. Call this function to override the line coding defaults.

**Parameters**

iID	Index of the serial converter device. Should be less than SU_FTDI232
dwDTERate	Baud rate. Valid settings include: 115200, 57600, 38400, 19200, 9600, 4800, 2400, 1200 and 300
wParityType	Parity type of the serial converter, valid settings include: SU_FTDI232_PARITY_NONE SU_FTDI232_PARITY_ODD SU_FTDI232_PARITY_EVEN SU_FTDI232_PARITY_MARK SU_FTDI232_PARITY_SPACE
wDataBits	7 or 8 bits
wStopBits	Stop bits, valid settings include: SU_FTDI232_STOP_BITS_1 SU_FTDI232_STOP_BITS_15 SU_FTDI232_STOP_BITS_2

**Returns**

== 0	Line coding has been set.
-1	An error occurred.

int **su\_FTDIGetModemStatus** (uint iID)

**Summary** Get the current Modem status register.

**Details** This function can be called once a serial converter device has been connected to the USB and configured by the device driver. Returns the current modem status register value.

**Parameters**

iID	Index of the serial converter device. Should be less than SU_FTDI232
-----	--

**Returns**

> 0	Mode status register which may include the following bitmap. SU_FTDI232_CTS SU_FTDI232_DSR SU_FTDI232_RI SU_FTDI232_RLSD
-1	An error occurred or the serial converter was removed.

int **su\_FTDIGetStatus** (uint iID, u8 \*pModemStatus, u8 \*pLineStatus)

**Summary** Get the current Modem and Line status register.

**Details** FTDI serial converter returns the modem and line status register after a call to `su_FTDIReadData()`. This function returns the last value of modem and line status register returned by `su_FTDIReadData()`.

**Parameters**

iID	Index of the serial converter device. Should be less than SU_FTDI232
pModemStatus	Pointer to the current Modem Status Register. It contains the following bitmap. SU_FTDI232_CTS SU_FTDI232_DSR SU_FTDI232_RI SU_FTDI232_RLSD
pLineStatus	Pointer to the current Line Status Register. It contains the following bitmap. SU_FTDI232_DR SU_FTDI232_OE SU_FTDI232_PE SU_FTDI232_FE SU_FTDI232_BI SU_FTDI232_THRE SU_FTDI232_TEMT SU_FTDI232_FIFOE

**Returns**

== 0	Got the status.
-1	An error occurred or the serial converter was removed.

int **su\_FTDISetEventChar** (uint iID, u8 cData)

**Summary** Set the event character.

**Details** This function can be called once a serial converter device has been connected to the USB and configured by the device driver, to change the default setting for event character.

**Parameters**

iID	Index of the serial converter device. Should be less than SU_FTDI232
cData	The new event character.

**Returns**

== 0	The new event character has been set.
-1	An error occurred or the serial converter was removed.

int **su\_FTDISetErrorChar** (uint iID, u8 cData)

**Summary** Set the error character.

**Details** This function can be called once a serial converter device has been connected to the USB and configured by the device driver, to change the default setting for error character.

**Parameters** iID Index of the serial converter device. Should be less than SU\_FTDI232  
cData The new error character.

**Returns** == 0 The new error character has been set.  
-1 An error occurred or the serial converter was removed.

int **su\_FTDISetLatencyTimer** (uint iID, uint iMS)

**Summary** Set the new latency timer.

**Details** This function can be called once a serial converter device has been connected to the USB and configured by the device driver, to change the default setting for latency timer. The latency timer is the timer used to time out read requests. The default time is 40 ms. If there is no data within the latency period, the read request will return zero length data.

**Parameters** iID Index of the serial converter device. Should be less than SU\_FTDI232  
iMS New latency timer in milliseconds.

**Returns** == 0 The new latency timer has been set.  
-1 An error occurred or the serial converter was removed.

int **su\_FTDIGetLatencyTimer** (uint iID)

**Summary** Get the current latency timer.

**Details** This function can be called once a serial converter device has been connected to the USB and configured by the device driver, to retrieve the current latency timer value.

**Parameters** iID Index of the serial converter device. Should be less than SU\_FTDI232

**Returns** > 0 The current latency timer.  
-1 An error occurred or the serial converter was removed.

## 4.4.4 Serial Converter Prolific 2303HXD

### 4.4.4.1 API

The application level interface is defined in **upl2303.h**. This interface includes:

```
int          su_PL2303Open(uint iID)
int          su_PL2303Close (uint iID)
BOOLEAN     su_PL2303Inserted (uint iID)
void        su_PL2303RegDevEvtCallback (SU_PDEVEVTCBFUNC func) int
            su_PL2303ReadData (uint iID, u8 * pData, int len)
int         su_PL2303WriteData (uint iID, u8 * pData, int len)
int         su_PL2303SetModemCtrl(uint iID, uint Data)
int         su_PL2303SetFlowCtrl(uint iID, uint Data)
int         su_PL2303SetLineCoding(uint iID, u32 dwDTERate, u16 wbParityType, u16 wDataBits,
            u16 wStopBits);
int         su_PL2303LineStateChangeNotify(PHOSTPL2303FUNC handler)
```

int **su\_PL2303Open** (uint iID)

**Summary** Open the Prolific 2303 serial converter port.

**Details** This function must be called **after** the Prolific serial converter has been connected to the USB and configured by the device driver and **before** any PL2303Read/PL2303Write functions. It sets the line coding information such as baud rate and number of data bits. It also sets the RTS and DTR signals of the serial port. The default flow control setting is to none. The default baud rate is 115200. The default line coding is 8 data bits, none parity and 1 stop bit.

**Parameters** iID Index of the serial device. Should be less than SU\_PL2303.

**Returns** == 0 The device has been opened and the line coding and line state have also been set up.  
< 0 An error occurred

int **su\_PL2303Close** (uint iID)

**Summary** Close the Prolific 2303 serial converter port.

**Details** This function must be called **after** finishing any PL2303Read/PL2303Write function calls. It clears the DSR and DTR signals of the serial converter port.

**Parameters** iID Index of the serial device. Should be less than SU\_PL2303.

**Returns** == 0 The device has been closed.  
< 0 An error occurred.

BOOLEAN **su\_PL2303Inserted** (uint iID)

**Summary** Returns status indicating if a Prolific 2303 serial converter device has been connected.

**Details** This function can be called at any time to determine if a supported serial converter device has been connected to the USB.

**Parameters** iID Index of the serial converter device. Should be less than SU\_PL2303.

**Returns** TRUE A supported Prolific serial converter device has been connected to the USB and initialized by the device driver.  
FALSE No Prolific serial converter device has been connected.

int **su\_PL2303ReadData** (uint iID, u8 \* pData, uint len, uint timeout)

**Summary** Reads data from the serial converter device.

**Details** This function tries to read some data from the serial port. If there is no data within timeout, this function returns 0.

**Parameters** iID Index of the serial converter device. Should be less than SU\_PL2303.  
pData Pointer to the data buffer.  
len Length of the data buffer.  
timeout Timeout period (in milliseconds) of the read operation.

**Returns** > 0 Length of the data read from the serial converter device.  
= 0 An error occurred or there is no data.

int **su\_PL2303WriteData** (uint iID, u8 \* pData, uint len)

**Summary** Sends data to the serial converter device.

**Details** This function can be called once a serial converter device has been connected to the USB and configured by the device driver. It tries to write some data to the serial converter device.

**Parameters** iID Index of the serial converter device. Should be less than SU\_PL2303  
pData Pointer to the data buffer.  
len Length of the data buffer.

**Returns**     > 0     Length of the data written to the serial converter device.  
              = 0     An error occurred.

int **su\_PL2303SetModemCtrl** (uint iID, uint Data)

**Summary**     Set the DTR and RTS signal.

**Details**     This function can be called once a serial converter device has been connected to the USB and configured by the device driver and after calling su\_PL2303Open(). su\_PL2303Open() sets DTR and RTS high, by default. Call this function to override the defaults.

**Parameters**   iID     Index of the serial converter device. Should be less than SU\_PL2303  
              Data     DTR and RTS settings. SU\_PL2303\_LINE\_OUT\_DTR will set only DTR to high and SU\_PL2303\_LINE\_OUT\_RTS will set only RTS to high. You can also OR those two macros together to setting both DTR and RTS to high.

**Returns**     == 0     DTR and RTS has been set.  
              -1     An error occurred.

int **su\_PL2303SetFlowCtrl** (uint iID, uint Data)

**Summary**     Set the Flow Control of the Prolific serial converter.

**Details**     This function can be called once a serial converter device has been connected to the USB and configured by the device driver and after calling su\_PL2303Open(). su\_ProlificOpen() sets the default Flow Control to none. Call this function to override the default.

**Parameters**   iID     Index of the serial converter device. Should be less than SU\_PL2303  
              Data     Flow Control settings. Valid settings include:  
                      SU\_PL2303\_FLOWCTRL\_NONE  
                      SU\_PL2303\_FLOWCTRL\_RTSCTS

**Returns**     == 0     Flow Control has been set.  
              -1     An error occurred.



int **su\_PL2303SetLineCoding** (uint iID, u32 dwDTERate, u16 wParityType,  
u16 wDataBits, u16 wStopBits)

**Summary** Set the Baudrate, Parity type, Data bits and Stop bits of the Prolific serial converter.

**Details** This function can be called once a serial converter device has been connected to the USB and configured by the device driver and after calling su\_PL2303Open(). su\_PL2303Open() sets default line coding values. Call this function to override the defaults.

**Parameters**

iID	Index of the serial converter device. Should be less than SU_PL2303
dwDTERate	Baud rate. Valid settings include: 115200, 57600, 38400, 19200, 9600, 4800, 2400 and 1200
wParityType	Parity type of the serial converter, valid settings include: SU_PL2303_PARITY_NONE SU_PL23032_PARITY_ODD SU_PL2303_PARITY_EVEN SU_PL2303_PARITY_MARK SU_PL2303_PARITY_SPACE
wDataBits	7 or 8 bits
wStopBits	Stop bits, valid settings include: SU_PL2303_STOP_BITS_1 SU_PL2303_STOP_BITS_1_5 SU_PL2303_STOP_BITS_2

**Returns**

== 0	Line coding has been set.
-1	An error occurred.

void **su\_PL2303LineStateChangeNotify** (PHOSTPL2303FUNC handler)

**Summary** Set the line state change notification callback function.

**Details** This function sets up the callback function when the line status is changed. The callback function is defined as:

```
typedef void (* PHOSTPL2303FUNC)(uint iID, u16 wNewLineState);
```

This callback function will be called when any of the following line states is changed:

```
SU_PL2303_LINE_IN_DCD  
SU_PL2303_LINE_IN_DSR  
SU_PL2303_LINE_IN_BRK  
SU_PL2303_LINE_IN_RI
```

SU\_PL2303\_LINE\_IN\_FRAMING  
SU\_PL2303\_LINE\_IN\_PARITY  
SU\_PL2303\_LINE\_IN\_OVERRUN  
SU\_PL2303\_LINE\_IN\_CTS

For example, if the remote side opened the serial port, a new line state is detected with the value 0x81 which means DCD and CTS are high. After the remote side closes the serial port connection, a new line state 0x0 is detected.

**Parameters** handler      Callback function pointer.

**Returns**      None.

#### 4.4.5 USB to Ethernet Adapter

This driver is not the CDC ECM/NCM class driver. You may need to add a special USB to Ethernet chipset driver if that chipset is not supported by smxUSBH yet.

##### 4.4.5.1 API

The application level interface is defined in **unet.h**. This interface includes:

```
BOOLEAN su_NetInserted(uint iID);  
int su_NetOpen(uint iID);  
int su_NetClose(uint iID);  
void su_NetRegDevEvtCallback(SU_PDEVEVTCBFUNC func)  
int su_NetWriteData(uint iID, u8 *pData, int len);  
int su_NetGetNodeID(int port, u8 *pData);  
void su_NetRegisterPortNotify(int port, PNETFUNC handler);  
void su_NetCheckData(uint iID);
```

BOOLEAN **su\_NetInserted** (uint iID)

**Summary**      Returns status indicating if a USB to Ethernet device has been connected.

**Details**      This function can be called at any time to determine if a supported USB to Ethernet device has been connected to the USB.

**Parameters** iID      Index of the serial device. Should be less than SU\_NET.

**Returns**      TRUE      A supported USB to Ethernet device has been connected to the USB and initialized by the device driver.  
              FALSE      No USB to Ethernet device has been connected.

int **su\_NetOpen** (uint iID)

**Summary** Open a USB to Ethernet device for further use.

**Details** This function must be called **after** the USB to Ethernet adapter has been connected to the USB and configured by the device driver and **before** any NetWrite functions. It will initialize the adapter and create a separate task to receive the incoming data packets sent to this adapter.

**Parameters** iID Index of the serial device. Should be less than SU\_NET.

**Returns** 0 This USB to Ethernet device has been opened successfully.  
< 0 An error occurred when trying to open this device.

int **su\_NetClose** (uint iID)

**Summary** Close the opened USB to Ethernet device.

**Details** This function must be called **after** the USB to Ethernet adapter has been removed from the USB host do not call any NetWrite functions after this function call.

**Parameters** iID Index of the serial device. Should be less than SU\_NET.

**Returns** 0 This USB to Ethernet device has been closed successfully.  
< 0 An error occurred when try to close this device.

int **su\_NetWriteData** (uint iID, u8 \*pData, int len)

**Summary** Send Ethernet packet data to the USB to Ethernet device.

**Details** This function must be called **after** **su\_NetOpen()**.

**Parameters** iID Index of the serial device. Should be less than SU\_NET.  
pData Ethernet packet data buffer pointer for sending.  
len Length of the Ethernet data buffer.

**Returns** 0 The data has been successfully sent.  
< 0 An error occurred when sending data.

int **su\_NetGetNodeID** (uint iID, u8 \*pMACAddr)

**Summary** Get the Node ID (MAC address) of this USB over Ethernet adapter.

**Details** Call this function to get the MAC address of this adapter.

**Parameters** iID Index of the serial device. Should be less than SU\_NET.  
pMACAddr The byte array of the Ethernet address. The size of this buffer should be at least 6 bytes.

**Returns** 0 Got the address successfully.  
< 0 An error occurred when getting the address.

void **su\_NetRegisterPortNotify** (uint iID, PNETFUNC handler)

**Summary** Register the callback function for receiving Ethernet packet data.

**Details** The callback function is called when the driver receives an Ethernet data packet. The received data should be passed onward to the TCP/IP stack.

**Parameters** iID Index of the serial device. Should be less than SU\_NET.  
handler The callback function pointer.

**Returns** None.

void **su\_NetCheckData**(uint iID)

**Summary** Polls incoming data for non-multitasking (NORTOS) case.

**Details** Call this function to get incoming data, only for non-multitasking (NORTOS). This driver has a built-in task to poll the incoming data so there is no need to call this function if you have a multitasking environment.

**Parameters** iID Index of the serial device. Should be less than SU\_NET.

**Returns** None

#### 4.4.5.2 Supporting Other USB to Ethernet Chips

Different USB to Ethernet chip vendors may use different vendor-specific requests to set up their chip. They may use different USB headers to wrap Ethernet data packets, so the smxUSBH USB to Ethernet driver provides an easy way to support other chips if needed. It is only necessary to

implement a Net Chip Operation interface and add it to the USB to Ethernet framework. That interface is defined in unetchip.h.

```
typedef struct
{
    int (*Init) (void);
    int (*Release) (void);
    int (*AddPacketHeader) (SU_DEV_INFO *pDev, SU_REQ_INFO *pReqInfo, u8 *pData, uint len);
    int (*GetOnePacket) (SU_DEV_INFO_T *pDev, SU_REQ_INFO *pReqInfo, u8 *pData, uint len);
    int (*GetEthernetID) (u8 *pMACAddr);
    SU_CLS_CHK * (*GetCheckDevID) (uint *piSize);
} SU_NET_CHIP_OPER_T;
```

Init() is called when opening this device for data transfer. Some vendor-specific requests for the device may be needed to set internal registers of the adapter or start the adapter.

Release() is called when data transfer is finished.

AddPacketHeader() is called when sending an Ethernet packet to the USB adapter. Some additional header may be needed to generate a USB packet.

GetOnePacket() is called when the stack gets some data from this adapter. Analysis of the header of the USB packet may be required and additional send requests may be needed to receive the remaining data.

GetEthernetID() is called by the stack to retrieve the Ethernet address of this adapter. This may require a vendor specific request.

GetCheckDevID() is called by the stack to get the Check Device information. This information is used by the stack to match the device through the Vendor ID and Product ID. Adapters from different vendors may use the same chip, so more entries can be added to this array to support them.

## 4.4.6 WCESerial (Windows Mobile 5 Device, non-CDC/ACM)

### 4.4.6.1 API

The application level interface is defined in **uwceser.h**. This interface includes:

```
int          su_WCESerialOpen (uint iID)
int          su_WCESerialClose (uint iID)
BOOLEAN     su_WCESerialInserted (uint iID)
void        su_WCERegDevEvtCallback(SU_PDEVEVTCBFUNC func)
int         su_WCESerialRead (uint iID, u8 * pData, int len)
int         su_WCESerialWrite (uint iID, u8 * pData, int len)
```

int **su\_WCESerialOpen** (uint iID)

**Summary**     Open the serial port.

**Details**     This function must be called **after** the USB serial device has been connected to the USB and configured by the device driver and **before** any SerialRead/SerialWrite functions. It sets the line coding information such as baud rate and number of data bits. It also sets the DCD and DTR signals of the serial port.

**Parameters**   iID     Index of the serial device. Should be less than SU\_WCESERIAL.

**Returns**       == 0     The device has been opened and the line coding and line state have also been set up.  
                  < 0     An error occurred.

int **su\_WCESerialClose** (uint iID)

**Summary**     Close the serial port.

**Details**     This function must be called **after** finishing SerialRead/SerialWrite function calls. It clears the DCD and DTR signals of the serial port.

**Parameters**   iID     Index of the serial device. Should be less than SU\_WCESERIAL.

**Returns**       == 0     The device has been closed.  
                  < 0     An error occurred.

BOOLEAN **su\_WCESerialInserted** (uint iID)

**Summary** Returns status indicating if this serial device has been connected.

**Details** This function can be called at any time to determine if a supported serial device has been connected to the USB.

**Parameters** iID Index of the serial device. Should be less than SU\_WCESERIAL.

**Returns** TRUE A supported USB serial device has been connected to the USB and initialized by the device driver.

FALSE No USB serial device has been connected.

int **su\_WCESerialRead** (uint iID, u8 \* pData, int len)

**Summary** Reads data from the serial device.

**Details** This function tries to read some data from the serial port. If there is no data within 1 second, this function returns 0.

**Parameters** iID Index of the serial device. Should be less than SU\_WCESERIAL.

pData Pointer to the data buffer.

len Length of the data buffer.

**Returns** > 0 Length of the data read from the serial device.

= 0 An error occurred or there is no data.

int **su\_WCESerialWrite** (uint iID, u8 \* pData, int len)

**Summary** Sends data to the serial device.

**Details** This function can be called once a serial device has been connected to the USB and configured by the device driver. It tries to write some data to the serial device.

**Parameters** iID Index of the serial device. Should be less than SU\_WCESERIAL

pData Pointer to the data buffer.

len Length of the data buffer.

**Returns** > 0 Length of the data written to the serial device.

= 0 An error occurred.

## 4.4.7 Sierra Wireless Dongle

### 4.4.7.1 API

The application level interface is defined in **usierra.h**. This interface includes:

```
int          su_SierraOpen (uint iID)
int          su_SierraClose (uint iID)
BOOLEAN     su_SierraInserted (uint iID)
void        su_SierraRegDevEvtCallback(SU_PDEVEVTCTBFUNC func)
int         su_SierraRead (uint iID, uint iPort, u8 * pData, int len, int timeout)
int         su_SierraWrite (uint iID, uint iPort, u8 * pData, int len, int timeout)
void        su_SierraLineStateChangeNotify (SU_PSIERRAFUNC handler);
```

int **su\_SierraOpen** (uint iID)

**Summary**     Open the Sierra Wireless port.

**Details**     This function must be called **after** the USB Sierra Wireless device has been connected to the USB and configured by the device driver and **before** any SerialRead/SerialWrite functions. It sets the DCD and DTR signals of the device.

**Parameters**   iID     Index of the device. Should be less than SU\_SIERRA.

**Returns**       == 0     The device has been opened and DCD and DT have also been set up.  
                  < 0     An error occurred.

int **su\_SierraClose** (uint iID)

**Summary**     Close the Sierra Wireless port.

**Details**     This function must be called **after** finishing SierraRead/SierraWrite function calls. It clears the DCD and DTR signals of the Sierra Wireless port.

**Parameters**   iID     Index of the device. Should be less than SU\_SIERRA.

**Returns**       == 0     The device has been closed.  
                  < 0     An error occurred.



BOOLEAN **su\_SierraInserted** (uint iID)

**Summary** Returns status indicating if the Sierra Wireless device has been connected.

**Details** This function can be called at any time to determine if a supported Sierra Wireless device has been connected to the USB.

**Parameters** iID Index of the device. Should be less than SU\_SIERRA.

**Returns** TRUE A supported USB Sierra Wireless device has been connected to the USB and initialized by the device driver.  
FALSE No USB Sierra Wireless device has been connected.

int **su\_SierraRead** (uint iID, uint iPort, u8 \* pData, int len, int timeout)

**Summary** Reads data from the Sierra device port.

**Details** This function tries to read some data from the Sierra Wireless port. If there is no data within timeout milliseconds, this function returns 0. A Sierra Wireless dongle may have one or multiple ports. Port 0 may be used for data transfer and other ports may be used as control channels.

**Parameters** iID Index of the device. Should be less than SU\_SIERRA.  
iPort Index of the port.  
pData Pointer to the data buffer.  
len Length of the data buffer.  
timeout Timeout for the operation, in milliseconds.

**Returns** > 0 Length of the data read from the port.  
= 0 An error occurred or there is no data.

int **su\_SierraWrite** (uint iID, uint iPort, u8 \* pData, int len, int timeout)

**Summary** Sends data to the Sierra device port.

**Details** This function can be called once a Sierra Wireless device has been connected to the USB and configured by the device driver. It tries to write some data to the device. A Sierra Wireless dongle may have one or multiple ports. Port 0 may be used for data transfer and other ports may be used as control channels.

**Parameters** iID Index of the device. Should be less than SU\_SIERRA.  
iPort Index of the port.

pData      Pointer to the data buffer.  
 len        Length of the data buffer.  
 timeout    Timeout for the operation, in milliseconds.

**Returns**    > 0      Length of the data written to the serial device.  
               = 0      An error occurred.

void **su\_SierraLineStateChangeNotify** (SU\_PSIERRAFUNC handler)

**Summary**     Register a callback notification function for device line state changes.

**Details**      This function can be called any time the smxUSBH has started.  
 The callback function is defined as:  
 typedef void (\*SU\_PSIERRAFUNC)( uint iID, u16 wNewLineState);  
 Valid values for wNewLineState includes:

SU\_SIERRA\_LINE\_IN\_DCD  
 SU\_SIERRA\_LINE\_IN\_DSR  
 SU\_SIERRA\_LINE\_IN\_BRK  
 SU\_SIERRA\_LINE\_IN\_RI  
 SU\_SIERRA\_LINE\_IN\_FRAMING  
 SU\_SIERRA\_LINE\_IN\_PARITY  
 SU\_SIERRA\_LINE\_IN\_OVERRUN

**Parameters**    iID                      Index of the device. Should be less than SU\_SIERRA.  
                   wNewLineState    Value for the new line state.

**Returns**        None

## 4.4.8 Huawei K4510 3G Wireless Dongle

### 4.4.8.1 API

The application level interface is defined in **uk4510.h**. This interface includes:

int            **su\_K4510Open** (uint iID)  
 int            **su\_K4510Close** (uint iID)  
 BOOLEAN      **su\_K4510Inserted** (uint iID)  
 void          **su\_K4510RegDevEvtCallback**(SU\_PDEVEVTCBFUNC func)  
 BOOLEAN      **su\_K4510Connected** (uint iID)  
 int            **su\_K4510CheckNetworkState** (uint iID)  
 BOOLEAN      **su\_K4510GetNetworkName** (uint iID, char \*pName)

```

int      su_K4510SendATCmd(uint iID, u8 * pData, int len, int timeout);
int      su_K4510SendATCmdAndWait(uint iID, u8 * pData, int len);

int      su_K4510Write(uint iID, u8 * pData, int len, int timeout);
int      su_K4510Read(uint iID, u8 *pData, int len, int timeout);
int      su_K4510PollRead(uint iID, u8 *pData, int len, int timeout);

```

int **su\_K4510Open** (uint iID)

**Summary**     Open the Huawei K4510 3G Wireless port.

**Details**     This function must be called **after** the USB Huawei 3G Wireless device has been connected to the USB and configured by the device driver.

**Parameters**   iID     Index of the device. Should be less than SU\_K4510.

**Returns**       == 0     The device has been opened.  
                  < 0     An error occurred.

int **su\_K4510Close** (uint iID)

**Summary**     Close the Huawei K4510 3G Wireless port.

**Details**     This function must be called **after** finishing Huawei K4510 3G wireless communication function.

**Parameters**   iID     Index of the device. Should be less than SU\_K4510.

**Returns**       == 0     The device has been closed.  
                  < 0     An error occurred.

BOOLEAN **su\_K4510Inserted** (uint iID)

**Summary**     Returns status indicating if the Huawei 3G Wireless device has been inserted to the USB port.

**Details**     This function can be called at any time to determine if a supported Huawei K4510 3G Wireless device is present on the USB port.

**Parameters**   iID             Index of the device. Should be less than SU\_K4510.

**Returns**       TRUE            A supported Huawei K4510 3G Wireless device has been inserted to the USB and initialized by the device driver.  
                  FALSE          No USB Huawei K4510 3G Wireless device has been inserted.

int **su\_K4510CheckNetworkState** (uint iID)

**Summary** Check the Huawei 3G Wireless dongle network register state.

**Details** Keep calling this function to check the network register state of the Huawei K4510 3G wireless dongle. This dongle internally automatic search the available 3G network and try to register to it. It may take more than one minute for the dongle to register to a network. A roaming network may need more time.

**Parameters** iID Index of the device. Should be less than SU\_K4510.

**Returns** One of the following states:  
**SU\_K4510\_NS\_REG\_NOT**  
The dongle has not registered to a network yet  
**SU\_K4510\_NS\_REG\_LOCAL**  
The dongle has registered to a local network.  
**SU\_K4510\_NS\_SEARCHING**  
The dongle is searching the network.  
**SU\_K4510\_NS\_REG\_REJECTED**  
The dongle's attempt to register to a network has been rejected.  
**SU\_K4510\_NS\_REG\_UNKNOWN**  
The network state is unknown  
**SU\_K4510\_NS\_REG\_ROAMING**  
The dongle has registered to a roaming network.

BOOLEAN **su\_K4510Connected** (uint iID)

**Summary** Returns status indicating if the Huawei 3G Wireless device has connected to a carrier's network.

**Details** This function can be called at any time to determine if a supported Huawei K4510 3G Wireless device has connected to a carrier's network.

**Parameters** iID Index of the device. Should be less than SU\_K4510.

**Returns** TRUE The Huawei K4510 3G Wireless device has connected to a carrier's network. `su_K4510CheckNetworkState()` should return `SU_K4510_NS_REG_LOCAL` or `SU_K4510_NS_REG_ROAMING`.  
FALSE The USB Huawei K4510 3G Wireless device has not been inserted or it is still trying to find and connect to a carrier's network.

BOOLEAN **su\_K4510GetNetworkName** (uint iID, char \*pName)

**Summary** Return to the carrier's network name.

**Details** After **su\_K4510Connected()** returns TRUE, you can call this function return the name of the carrier's network, such as "AT&T".

**Parameters** iID Index of the device. Should be less than SU\_K4510.  
pName Pointer to the buffer for the network name.

**Returns** TRUE Got the network name.  
FALSE An error occurred.

int **su\_K4510SendATCmdAndWait**(uint iID, u8 \* pData, int count)

**Summary** Send AT command to the Huawei K4510 3G wireless device port.

**Details** This function will send an AT command to the device and check if gets OK response.

**Parameters** iID Index of the device. Should be less than SU\_K4510.  
pData Pointer to AT command string.  
count Length of the AT command string.

**Returns** -1 error happened.  
= 0 OK response received.

int **su\_K4510SendATCmd**(uint iID, u8 \* pData, int count, int timeout)

**Summary** Send AT command to the Huawei K4510 3G wireless device port.

**Details** This function will send an AT command to the device. This API will not check the response.

**Parameters** iID Index of the device. Should be less than SU\_K4510.  
pData Pointer to AT command string.  
count Length of the AT command string;  
timeout Timeout for the operation, in milliseconds.

**Returns** > 0 command sent out.  
= 0 error happened.

int **su\_K4510PollRead**(uint iID, u8 \*pData, int len, int timeout)

**Summary** Polls the incoming data

**Details** This function will send a request to poll incoming data. The difference between this function and su\_K4510Read() is it will not unlink the request if it times out. Normally you need to keep calling this function in a separate task's main loop function to get the incoming data.

**Parameters**

iID	Index of the device. Should be less than SU_K4510.
pData	Pointer for the incoming data you want to get.
len	length of the data buffer
timeout	Timeout to retrieve the data, in milliseconds.

**Returns**

>= 0	data received.
-1	error happened.

int **su\_K4510Read** (uint iID, u8 \* pData, int len, int timeout)

**Summary** Reads data from the Huawei K4510 3G wireless device port.

**Details** This function tries to read some data from the Huawei K4510 3G Wireless dongle. Only call this function after su\_K4510CheckNetworkState() returns SU\_K4510\_NS\_REG\_LOCAL or SU\_K4510\_NS\_REG\_ROAMING. If there is no data within timeout milliseconds, this function returns 0.

**Parameters**

iID	Index of the device. Should be less than SU_K4510.
pData	Pointer to the data buffer.
len	Length of the data buffer.
timeout	Timeout for the operation, in milliseconds.

**Returns**

> 0	Length of the data read from the port.
= 0	An error occurred or there is no data.

int **su\_K4510Write** (uint iID, u8 \* pData, int len, int timeout)

**Summary** Sends data to the Huawei K4510 3G wireless device port.

**Details** Only call this function after su\_K4510CheckNetworkState() returns SU\_K4510\_NS\_REG\_LOCAL or SU\_K4510\_NS\_REG\_ROAMING. It tries to send some data to the device.

**Parameters**

iID	Index of the device. Should be less than SU_K4510.
pData	Pointer to the data buffer.
len	Length of the data buffer.
timeout	Timeout for the operation, in milliseconds.

**Returns**

> 0	Length of the data written to the wireless device.
= 0	An error occurred.

## 4.4.9 CDC ECM/NCM (Ethernet)

### 4.4.9.1 API

The application level interface is defined in **ucdcnet.h**. Normally, this kind of device is used for Ethernet (IEEE 802.3) and similar devices. The interface includes:

```

int      su_CDCNetOpen (uint iID)
int      su_CDCNetClose (uint iID)
BOOLEAN su_CDCNetInserted (uint iID)
void     su_CDCNetRegDevEvtCallback(SU_PDEVEVTCBFUNC func)

int      su_CDCNetWrite (uint iID, u8 * pData[], u16 len[], uint packets)
void     su_CDCNetRegisterPortNotify(uint iID, PCDCNETFUNC handler);
int      su_CDCNetIsAggregationSupported(uint iID);

```

BOOLEAN **su\_CDCNetInserted** (uint iID)

**Summary** Returns status indicating if a USB CDC ECM/NCM device has been connected and is ready to be used (reported by the device according the CDC spec).

**Details** This function can be called at any time to determine if a CDC ECM/NCM device has been connected to the USB port and is ready to be used

**Parameters**

iID	Index of the device. Should be less than SU_CDCNET.
-----	---

**Returns**

TRUE	A USB CDC ECM/NCM device has been connected to the USB and is ready to be used.
FALSE	No USB CDC ECM/NCM device has been connected.

int **su\_CDCNetOpen** (uint iID)

**Summary** Open the CDC Net device.

**Details** This function must be called **after** the USB CDC Net device has been connected to the USB and configured by the device driver and **before** any CDCNetWrite functions. It sets Ethernet filter

**Parameters** iID Index of the device. Should be less than SU\_CDCNET.

**Returns** == 0 The device has been opened and the line coding and line state have also been set up.  
 < 0 An error occurred.

int **su\_CDCNetClose** (uint iID)

**Summary** Close the CDC Net device.

**Details** This function must be called **after** you finish any CDCNetWrite function calls. It clears the DCD and DTR signals of the serial port.

**Parameters** iID Index of the device. Should be less than SU\_CDCNET.

**Returns** == 0 The device has been closed.  
 < 0 An error occurred.

int **su\_CDCNetWrite** (uint iID, u8 \* pBuf[] u16 len[], uint packets)

**Summary** Send Ethernet packet data to the CDC ECM/NCM device.

**Details** This driver will encode the packet according to the device is ECM or NCM. Network stack can pass multiple packets to the API. If the device is using ECM, packets will be sent one by one in multiple USB packets. If the device is using NCM, an aggregation packet will be generated to send them at once. The maximum transmit packet size is define within the driver as NET\_TRANSMIT\_BUFFER\_SIZE

**Parameters** iID Index of the device. Should be less than SU\_CDCNET.  
 pData Pointer array of the transmit data buffer.  
 len Length array of the transmit data buffer.  
 packets Size of the pointer and size array.

**Returns** > 0 Length of the data transmitted to the device.  
 <= 0 An error occurred.



int **su\_CDCNetIsAggregationSupported** (uint IID)

**Summary** Check if device support packets aggregation.

**Details** NCM supports packets aggregation but ECM does not. CDC NET driver supports both ECM and NCM. Use this API to check if the attached device is using ECM or NCM.

**Parameters** IID Index of the device. Should be less than SU\_CDCNET.

**Returns** > 1 Aggregation is supported (NCM).  
0 Aggregation is not supported (ECM).

void **CDCNetRegisterPortNotify**(uint IID, PCDCNETFUNC handler)

**Summary** Register callback function for incoming ethernet packets.

**Details** CDC Net class driver has a built-in read polling function to get incoming packets from the device. After the driver gets the incoming Ethernet packets, it will call this callback function to pass the data to the network stack. If the received packet is an aggregate packet, the callback function will be called multiple times for each packet in the aggregated packet.

**Parameters** IID Index of the device. Should be less than SU\_CDCNET.  
handler Callback function.

**Returns** None

## 4.5 Human Interface (HID)

smxUSBH not only supports the general HID class, but also includes simplified keyboard and mouse drivers.

### 4.5.1 Keyboard

#### 4.5.1.1 API

The application level interface is defined in **ukbd.h**. This interface includes:

```
BOOLEAN  su_KbdInserted(void)
void      su_KbdRegDevEvtCallback(SU_PDEVEVTCBFUNC func)
void      su_KbdSetCallback(PKBDFUNC handler)
```

```
typedef void (* PKBDFUNC)(unsigned long key);
```

This is a callback function that is called by the keyboard device driver. This function is called when any key event occurs, such as when a key is pressed or released. The application should implement this function to process the key scan code information that is passed as appropriate for the application.

**BOOLEAN su\_KbdInserted** (void)

**Summary** Returns status indicating if a USB keyboard has been connected.

**Details** You can call this function at any time to find out if a supported keyboard has been connected to the USB.

**Parameters** none

**Returns** TRUE A supported USB keyboard has been connected to the USB and initialized by the device driver.  
FALSE No USB keyboard is connected.

**void su\_KbdSetCallback** (PKBDFUNC handler)

**Summary** Registers a keyboard event callback handler with the device driver.

**Details** This function can be called at any time. This function is called for any key event, once a keyboard is connected to the USB and the device is configured by the device driver.

**Parameters** handler Callback function pointer.

**Returns** none

## 4.5.2 Mouse

### 4.5.2.1 API

The application level interface is defined in **umouse.h**. This interface includes:

```
BOOLEAN  su_MouseInserted(void)
void      su_MouseRegDevEvtCallback(SU_PDEVEVTCBFUNC func)
void      su_MouseSetCallback(PMOUSEFUNC handler)
```

```
typedef void (* PMOUSEFUNC)(SU_MOUSE_MSG * mouseMsg);
```

This is a callback function. It is called when any mouse event occurs, such as when the right mouse button is pressed. The application should implement this callback function and pass the mouse event to the part of the application requiring mouse events. A mouse event is defined as one of the following:

```
typedef struct
{
    u8 button;
    s8 x;
    s8 y;
    s8 wheel;
} SU_MOUSE_MSG;
```

1. *button* is a bitmap encoding the button state. When a bit is set to '1', it indicates that the corresponding button is pressed. The following button definitions are supported:
  - SU\_MOUSE\_BTN\_LEFT
  - SU\_MOUSE\_BTN\_RIGHT
  - SU\_MOUSE\_BTN\_MIDDLE
  - SU\_MOUSE\_BTN\_SIDE
  - SU\_MOUSE\_BTN\_EXTRA
2. *x* indicates 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
3. *y* indicates 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.
4. *wheel* indicates the direction of mouse wheel rotation. 1 indicates that the wheel is rolling up, and -1 indicates that the wheel is rolling down.

BOOLEAN **su\_MouseInserted** (void)

**Summary** Returns status indicating if a USB mouse has been connected.

**Details** Call this function at any time to find out if a supported mouse has been connected to the USB.

**Parameters** none

**Returns** TRUE A supported USB mouse has been connected to the USB and initialized by the device driver.  
FALSE No USB mouse is connected.

void **su\_MouseSetCallback** (PMOUSEFUNC handler)

**Summary** Registers a mouse event callback handler with the device driver.

**Details** This function can be called at any time. Once the USB mouse is connected and the device is configured by the device driver, this function will be called whenever a mouse event occurs.

**Parameters** handler Callback function pointer.

**Returns** none

## 4.5.3 Generic HID

### 4.5.3.1 API

The application level interface is defined in **uhid.h**. This interface includes:

```
BOOLEAN su_HIDInserted(void)
void su_HIDRegDevEvtCallback(SU_PDEVEVTCBFUNC func)
void su_HIDSetCallback(PHIDFUNC handler)
```

```
typedef void (* PHIDFUNC)(SU_HID_FIELD_INFO *pFieldInfo, SU_HID_USAGE_INFO *pUsageInfo,
                                                                    s32 data);
```

Callback function type, which is called when any HID event occurs, for example a right mouse button press. The application should implement this callback function and pass the HID event to the part of the application requiring HID events. An HID Usage Information structure is defined as:

```
typedef struct
{
    uint hid;
    u16 code;
    u8 type;
}SU_HID_USAGE_INFO;
```

*hid* is the internal HID ID for this event

*code* indicates which HID usage issued this event, for example, mouse left key or joystick trigger button. For the complete usage codes, check the macros defined in `uhid.h`

*type* is the type of this HID event. Valid types include:

```
SU_HID_TYPE_RST
SU_HID_TYPE_KEY
SU_HID_TYPE_REL
SU_HID_TYPE_ABS
SU_HID_TYPE_MSC
SU_HID_TYPE_LED
SU_HID_TYPE_SOUND
SU_HID_TYPE_MAX
```

An HID Field Information structure is defined as:

```
typedef struct
{
    uint physical;
    uint logical;
    uint application;
    uint flags;
    s32 logicalMin;
    s32 logicalMax;
    s32 physicalMin;
    s32 physicalMax;
    uint unitExponent;
    uint unit;
}SU_HID_FIELD_INFO;
```

This structure is used to provide additional information of the event data, such as minimum and maximum values of the data.

**BOOLEAN** `su_HIDInserted` (void)

**Summary** Returns status indicating if a USB HID has been connected.

**Details** You can call this function at any time to find out if an HID has been connected to the USB.

**Parameters** none

<b>Returns</b>	TRUE	A supported USB HID has been connected to the USB and initialized by the device driver.
	FALSE	No USB HID is connected.

void **su\_HIDSetCallback** (PHIDFUNC handler)

**Summary** Registers an HID event callback handler with the device driver.

**Details** This function can be called at any time. Once the USB HID is connected and the device is configured by the device driver, this function is called whenever an HID event occurs.

**Parameters** handler          Callback function pointer.

**Returns** none

## 4.6 Mass Storage

### 4.6.1 API

The application level interface is defined in **umsintf.h**. This interface includes:

```

int          su_MStorIO(uint iID, uint iLUN, u8* pRAMAddr, u32 nStartSector, u16 nSectors,
                                BOOLEAN Reading)
BOOLEAN     su_MStorMediaInserted(uint iID, uint iLUN)
BOOLEAN     su_MStorMediaRemoved(uint iID, uint iLUN)
void        su_MStorRegDevEvtCallback(SU_PDEVEVTCBFUNC func)
u32         su_MStorSectorNum(uint iID, uint iLUN)
u32         su_MStorSectorSize(uint iID, uint iLUN)
BOOLEAN     su_MStorMediaChanged(uint iID, uint iLUN, int nClearStatus);
BOOLEAN     su_MStorMediaProtected(uint iID, uint iLUN);
BOOLEAN     su_MStorTestReady(uint iID, uint iLUN);
int         su_MStorMaxLUN(uint iID);

```

int **su\_MStorIO** (uint iID, uint iLUN, u8\* pRAMAddr, u32 nStartSector, u16 nSectors, BOOLEAN Reading)

**Summary** Write or read data to/from the USB Mass Storage Device.

**Details** Call this function any time after `su_Initialize()` has been called. If a USB Mass Storage Class device is not connected, this function will return an error. This function will not return until the data transfer has completed.

**Parameters**

iID	Index of the mass storage device, from 0 to SU_MSTOR-1.
iLUN	The logical unit number of this mass storage device. Use 0 for this parameter if it does not support multiple LUNs.
pRAMAddr	Pointer to the transfer data buffer in RAM.
nStartSector	The initial sector that will be transferred.
nSectors	Number of Sectors to transfer to or from the Mass Storage Device.
Reading	TRUE for a read operation, FALSE for write.

**Returns**

SU_STORAGE_E_OK	The data transfer completed OK.
SU_MS_E_PROTECTED	Attempted write to a write protected USB drive.
	Otherwise an error occurred during the transfer.

**See Also** su\_MStorSectorSize()

BOOLEAN **su\_MStorMediaInserted** (uint iID, uint iLUN)

**Summary** Determine if a USB drive is inserted and correctly configured by the Mass Storage Device driver.

**Details** This function can be called any time after su\_Initialize() has been called. It checks if a recognized USB Mass Storage Device has been connected and all the necessary parameters have been retrieved.

**Parameters**

iID	Index of the mass storage device. Should be less than SU_MSTOR-1.
iLUN	The logical unit number of this mass storage device. Use 0 for this parameter if it does not support multiple LUNs.

**Returns**

TRUE	A USB Mass Storage Device has been recognized.
FALSE	No Mass Storage device is connected or the device driver does not support this device.

**See Also** su\_MStorMediaRemoved()

BOOLEAN **su\_MStorMediaRemoved** (uint iID, uint iLUN)

**Summary** Determine if a connected USB drive has been removed.

**Details** This function can be called any time after su\_Initialize() has been called.

**Parameters**

iID	Index of the mass storage device. Should be less than SU_MSTOR-1.
iLUN	The logical unit number of this mass storage device. Use 0 for this parameter if it does not support multiple LUNs.

**Returns** TRUE A USB Mass Storage Device has been removed.  
FALSE A recognized Mass Storage device is connected.

**See Also** su\_MStorMediaInserted()

u32 **su\_MStorSectorNum** (uint IID, uint iLUN)

**Summary** Gets the total size (in sectors) for the USB Mass Storage device.

**Details** This function can be called at any time after su\_Initialize() has been called. This function will return the total size of the Mass Storage device, in sectors. If no Mass Storage device is connected, this function returns 0.

**Parameters** IID Index of the mass storage device. Should be less than SU\_MSTOR-1.  
iLUN The logical unit number of this mass storage device. Use 0 for this parameter if it does not support multiple LUNs.

**Returns** 0 No Mass Storage device is connected.  
Otherwise returns the total size in blocks.

u32 **su\_MStorSectorSize** (uint IID, uint iLUN)

**Summary** Gets the sector size (in bytes) for the USB Mass Storage device.

**Details** This function can be called at any time after su\_Initialize() has been called. This function will return the sector size for the Mass Storage device. If no Mass Storage device is connected, this function returns 0.

**Parameters** IID Index of the mass storage device. Should be less than SU\_MSTOR-1.  
iLUN The logical unit number of this mass storage device. Use 0 for this parameter if it does not support multiple LUNs.

**Returns** 0 No Mass Storage device is connected.  
Otherwise returns the sector size in bytes, for example 512.

BOOLEAN **su\_MStorMediaChanged** (uint IID, uint iLUN, int nClearStatus)

**Summary** Determine if the USB drive has been changed.

**Details** This function can be called at any time after su\_Initialize() has been called. Normally file system will call this function to determine if it need to remount



(unmount and then mount) the disk volume, This function does not detect if the reinserted device is the same one as the just removed one.

**Parameters**    iID                    Index of the mass storage device. Should be less than SU\_MSTOR-1.  
                  iLUN                    The logical unit number of this mass storage device. Use 0 for this parameter if it does not support multiple LUNs.  
                  nClearStatus    If smxUSBH should clear the internal status after this function returns.

**Returns**            TRUE if Mass Storage device has been changed.  
                  FALSE the Mass Storage device did not change.

BOOLEAN **su\_MStorMediaProtected**(uint iID, uint iLUN)

**Summary**            Determine if the connected USB drive is write protected.

**Details**             Call this function any time after su\_MStorMediaInserted() has returned 1.

**Parameters**    iID                    Index of the mass storage device. Should be less than SU\_MSTOR-1.  
                  iLUN                    The logical unit number of this mass storage device. Use 0 for this parameter if it does not support multiple LUNs.

**Returns**            TRUE                    The Mass Storage device is write protected.  
                  FALSE                    The Mass Storage device is NOT write protected.

BOOLEAN **su\_MStorTestReady**(uint iID, uint iLUN)

**Summary**            Check if the media in this device is ready to be used.

**Details**             This function sends the SCSI Test Unit Ready command to the device. You can use it to check if the media of this device is ready to be used. Some flash disks also use this command to flush internal buffers. Windows uses this command to poll the status of removable disks, such as an SD card reader.

**Parameters**    iID                    Index of the mass storage device. Should be less than SU\_MSTOR-1.  
                  iLUN                    The logical unit number of this mass storage device. Use 0 for this parameter if it does not support multiple LUNs.

**Returns**            TRUE                    The Mass Storage device is ready to be used, or this command is not supported.  
                  FALSE                    The Mass Storage device has no media in it.

int **su\_MStorMaxLUN**(uint iID)

**Summary** Get the maximum LUN of the USB device.

**Details** Call this function to get the maximum logical unit number supported by this USB device. Most USB flash disks only support one LUN, but some card readers support many LUNs requiring a non-zero iLUN parameter to be used by other API calls.

**Parameters** iID Index of the mass storage device. Should be less than SU\_MSTOR-1.

**Returns** > 0 The total LUN number this device supports.  
= 0 if the device is not connected to the USB.

## 4.6.2 Support for USB CD-ROM and Floppy

For USB CD-ROM or floppy support, set SU\_MSTOR\_FULL\_SUPPORT to 1. This configuration parameter defaults to 0, in which case only Bulk Only Transport protocol and Reduced Block Command (RBC) set are supported. Almost all flash disk devices (thumb drives) use that protocol. One known exception is the Prolific pl2518 chipset. It uses the SFI8070i protocol.

## 4.7 Printer

### 4.7.1 API

The application level interface is defined in **uprinter.h**. This interface includes:

```
int          su_PrnID (uint iID, u8 * pData, int len)
BOOLEAN     su_PrnInserted (uint iID)
void        su_PrnRegDevEvtCallback(SU_PDEVEVTCBFUNC func)
int         su_PrnRead (uint iID, u8 * pData, int len)
int         su_PrnReset (uint iID)
u8          su_PrnStatus (uint iID)
int         su_PrnWrite (uint iID, u8 * pData, int len)
```

int **su\_PrnID** (uint iID, u8 \* pData, int len)

**Summary** Returns the DeviceID string.

**Details** This function can be called once a printer has been connected to the USB and configured by the device driver. It returns the DeviceID string compatible with the IEEE 1284.4 bi-directional interface. For syntax and formatting information, see the IEEE 1284.4 specification.

**Parameters**    iID     Index of the printer device. Should be less than SU\_PRINTER-1.  
                  pData    Pointer to the data buffer.  
                  len     Length of the data buffer.

**Returns**        > 0     Length of the data read from the printer.  
                  < 0     An error occurred.

BOOLEAN **su\_PrnInserted** (uint iID)

**Summary**        Returns status indicating if a USB printer has been connected

**Details**        This function can be called at any time to determine if a supported printer has been connected to the USB.

**Parameters**    iID                Index of the printer device. Should be less than SU\_PRINTER-1.

**Returns**        TRUE              A supported USB printer has been connected to the USB and initialized by the device driver.  
                  FALSE             No USB printer has been connected.

int **su\_PrnRead** (uint iID, u8 \* pData, int len)

**Summary**        Reads data from the printer.

**Details**        This function can be called if the printer supports bi-directional mode.

**Parameters**    iID     Index of the printer device. Should be less than SU\_PRINTER-1.  
                  pData    Pointer to the data buffer.  
                  len     Length of the data buffer.

**Returns**        > 0     Length of the data read from the printer  
                  = 0     An error occurred.

int **su\_PrnReset** (uint iID)

**Summary**        Resets the printer.

**Details**        This function can be called once a printer has been connected to the USB and configured by the device driver.

**Parameters**    iID     Index of the printer device. Should be less than SU\_PRINTER-1.

**Returns**     0       The command has been sent to the printer successfully.  
              < 0      An error occurred.

u8 **su\_PrnStatus** (uint iID)

**Summary**     Returns the status of the USB printer.

**Details**     This function returns the printer status. The following three bit flags are supported.  
              SU\_PRN\_STATUS\_PAPEREMPTY  
              SU\_PRN\_STATUS\_SELECTED  
              SU\_PRN\_STATUS\_NOERROR

**Parameters** iID     Index of the printer device. Should be less than SU\_PRINTER-1.

**Returns**     Printer status

int **su\_PrnWrite** (uint iID, u8 \* pData, int len)

**Summary**     Sends data to the printer.

**Details**     This function can be called once a printer has been connected to the USB and configured by the device driver.

**Parameters** iID     Index of the printer device. Should be less than SU\_PRINTER-1.  
              pData    Pointer to the data buffer.  
              len     Length of the data buffer.

**Returns**     > 0     The amount of data sent to the printer.  
              = 0     An error occurred.

## 4.8 Video

The video class driver currently only supports video cameras. Output and transport operations are not supported. Limitations:

1. Video camera still image capture function methods 2 and 3 are currently not supported.
2. MPEG streaming video format is not currently supported.

### 4.8.1 API

The application level interface is defined in **uvideo.h**. This interface includes:

```
int su_VideoCameraGetStillImageFormatNum(uint iID);
int su_VideoCameraGetStillImageFormat(uint iID, uint iIndex,
                                     SU_VIDEO_STILL_IMAGE_FORMAT *pFormat);
int su_VideoCameraGetCaptureFormatNum(uint iID);
int su_VideoCameraGetCaptureFormat(uint iID, uint iIndex,
                                   SU_VIDEO_CAPTURE_FORMAT *pFormat);
int su_VideoCameraGetCurrent(uint iID, SU_VIDEO_SETTINGS *pCurrent);
int su_VideoCameraGetDefault(uint iID, SU_VIDEO_SETTINGS *pDefault);
int su_VideoCameraGetMin(uint iID, SU_VIDEO_MIN_MAX *pMin);
int su_VideoCameraGetMax(uint iID, SU_VIDEO_MIN_MAX *pMax);
int su_VideoCameraGetInfo(uint iID, SU_VIDEO_INFO *pInfo);
int su_VideoCameraSetCaptureFrame(uint iID, uint iFormat, u16 wWidth, u16 wHeight,
                                  u32 dwFrameInterval);
int su_VideoCameraSetStillImageFormat(uint iID, uint iFormat, u16 wWidth, u16 wHeight);
int su_VideoCameraSetCurrent(uint iID, SU_VIDEO_SETTINGS *pNewCurrent);
int su_VideoCameraGetCaptureSize(uint iID, u32 *pMaxVideoFrameSize, u32 *pMaxPayloadBufferSize);

int su_VideoCameraOpen(uint iID);
int su_VideoCameraClose(uint iID);
int su_VideoCameraCapture(uint iID, u8 *pData, uint iLen, BOOLEAN *pbIncludeNewFrame,
                          uint *piNewFrameOffset);

BOOLEAN su_VideoInserted(uint iID);
void su_VideoRegDevEvtCallback(SU_PDEVEVTCBFUNC func)
```

int **su\_VideoCameraGetStillImageFormatNum** (uint iID)

**Summary** Returns video camera still image number of formats.

**Details** This function can be called once the video camera has been connected to the USB and configured by the device driver. It returns the total number of still image formats.

**Parameters** iID Index of the video device. Should be less than SU\_VIDEO.

**Returns** >= 0 Number of still image formats.  
< 0 An error occurred.

int **su\_VideoCameraGetStillImageFormat** (uint iID, uint iIndex,  
SU\_VIDEO\_STILL\_IMAGE\_FORMAT \*pFormat)

**Summary** Returns video camera still image format detailed information.

**Details** This function can be called once the video camera has been connected to the USB and configured by the device driver. It returns the detailed video camera still image format information such as width, height and color primaries.

**Parameters** iID Index of the format, should be less than the format number returned by su\_VideoCameraGetStillImageFormatNum().

pFormat Pointer to the detailed format information.

SU\_VIDEO\_STILL\_IMAGE\_FORMAT is defined as:

typedef struct

```
{  
    u16 wWidth[SU_VIDEO_MAX_STILL_IMAGE_PATTERN];  
    u16 wHeight[SU_VIDEO_MAX_STILL_IMAGE_PATTERN];  
    u8 bColorPrimaries;  
    u8 bTransferCharacteristics;  
    u8 bMatrixCoefficients;  
    uint iPatternNum;  
    uint iFormat;  
}
```

}SU\_VIDEO\_STILL\_IMAGE\_FORMAT;

**wWidth[]** is the array of the width of the still image. Real array number is iPatternNum.

**wHeight[]** is the array of the height of the still image. Real array number is iPatternNum.

**bColorPrimaries** is the color primaries. For details see the USB Video Class Device spec.

**bTransferCharacteristics** is the transfer characteristics. For details see the USB Video Class Device spec.

**bMatrixCoefficients** is the matrix coefficients, for details check USB Video Class Device spec.

**iPatternNum** is the total pattern number (the array size of the wWidth[] and wHeight[]).

**iFormat** is the format of the still image. Is is one of the following:

SU\_VIDEO\_FORMAT\_UNCOMPRESSED

SU\_VIDEO\_FORMAT\_MJPEG

**Returns** 0 Gott the format information.

< 0 An error occurred.

int **su\_VideoCameraGetCaptureFormatNum** (uint iID)

**Summary** Returns video camera number of capture formats.

**Details** This function can be called once the video camera has been connected to the USB and configured by the device driver. It returns the total number of capture formats.

**Parameters** iID Index of the video device. Should be less than SU\_VIDEO.

**Returns**  $\geq 0$  Capture format number.  
 $< 0$  An error occurred.

int **su\_VideoCameraGetCaptureFormat** (uint iID, uint iIndex,  
SU\_VIDEO\_CAPTURE\_FORMAT \*pFormat)

**Summary** Returns video camera capture format detailed information.

**Details** This function can be called once the video camera has been connected to the USB and configured by the device driver. It returns the detailed video camera capture format information such as frame and bits per pixel.

**Parameters** iID Index of the format, should be less than the format number returned by su\_VideoCameraGetCaptureFormatNum().  
pFormat Pointer to the detailed format information.  
SU\_VIDEO\_CAPTURE\_FORMAT is defined as  
typedef struct  
{  
    uint iFormat;  
    uint iYUVFormat;  
    u8 bBitsPerPixel;  
    u8 bDefaultFrameIndex;  
    SU\_VIDEO\_CAPTURE\_FRAME Frame[SU\_VIDEO\_MAX\_FRAME];  
    uint bNumFrame;  
}SU\_VIDEO\_CAPTURE\_FORMAT;  
**iFormat** is the format of the still image. One of the following:  
SU\_VIDEO\_FORMAT\_UNCOMPRESSED  
SU\_VIDEO\_FORMAT\_MJPEG  
SU\_VIDEO\_FORMAT\_MPEG2TS  
**iYUVFormat** is one of the following YUV format if iFormat is  
SU\_VIDEO\_FORMAT\_UNCOMPRESSED:  
SU\_VIDEO\_FORMAT\_UNCOMPRESSED\_UNKNOWN  
SU\_VIDEO\_FORMAT\_UNCOMPRESSED\_YUV422  
SU\_VIDEO\_FORMAT\_UNCOMPRESSED\_YUV420

**bBitsPerPixel** is the bit number of each pixel.

**bDefaultFrameIndex** is the default frame index if you don't call `su_VideoCameraSetCaptureFrame()` to change it.

**Frame[]** is the array of the support frame information for this format. The size of the array is **bNumFrame**. The frame information is defined as:

```
typedef struct
{
    u16 wWidth;
    u16 wHeight;
    u32 dwFrameInterval[SU_VIDEO_MAX_FRAME_INTERVAL];
    uint iNumFrameInterval;
}SU_VIDEO_CAPTURE_FRAME;
```

**wWidth** is the width of the video capture frame.

**wHeight** is the height of the video capture frame.

**dwFrameInterval[]** is the array of the frame interval. Units are 1/10 microsecond. For example, 333333 is 33.3333 millisecond which means 30 fps

**iNumFrameInterval** is the size of frame interval array.

**bNumFrame** is the size of frame array `Frame[]`

**Returns**        0        Got the format information.  
                 < 0        An error occurred.

`int su_VideoCameraGetCurrent (uint iID, SU_VIDEO_SETTINGS *pCurrent)`

**Summary**        Returns video camera current setting information.

**Details**        This function can be called once the video camera has been connected to the USB and configured by the device driver. It returns the current setting information, which includes the camera controls, such as `AutoExposureMode` and `Focus`. It also includes processing unit controls, such as `Brightness` and `Contrast`.

**Parameters**    `iID`        Index of the video device. Should be less than `SU_VIDEO`.

`pCurrent`    Pointer to the current settings.  
                 Video Camera settings are defined as:

```
typedef struct
{
    u8 bDevPowerMode;
    /* Camera Control */
    u8 bScanningMode;
    u8 bAutoExposureMode;
    u8 bAutoExposurePriority;
    u32 dwExposureTimeAbsolute;
    u8 bExposureTimeRelative;
    u16 wFocusAbsolute;
```



```

u8 bFocusRelative;
u8 bFocusRelativeSpeed;
u8 bFocusAuto;
u16 wIrisAbsolute;
u8 bIrisRelative;
u16 wObjectiveFocalLength;
s8 bZoom;
u8 bDigitalZoom;
u8 bZoomSpeed;
s32 dwPanAbsolute;
s32 dwTiltAbsolute;
s8 bPanRelative;
u8 bPanSpeed;
s8 bTiltRelative;
u8 bTiltSpeed;
u16 wRollAbsolute;
u8 bRollRelative;
u8 bRollRelativeSpeed;
u8 bPrivacy;
/* Select Unit Control */
u8 bSelector;
/* Processing Unit Control */
u16 wBacklightCompensation;
s16 wBrightness;
u16 wContrast;
u16 wGain;
u8 bPowerLineFrequency;
s16 wHue;
u8 bHueAuto;
u16 wSaturation;
u16 wSharpness;
u16 wGamma;
u16 wWhiteBalanceTemperature;
u8 bWhiteBalanceTemperatureAuto;
u16 wWhiteBalanceBlue;
u16 wWhiteBalanceRed;
u8 bWhiteBalanceComponentAuto;
u16 wMultiplierStep;
u16 wMultiplierLimit;
u8 bVideoStandard;
u8 bAnalogVideoLockStatus;
u16 wSyncDelay;
}SU_VIDEO_SETTINGS;

```

**Returns**     0     Get the current settings.  
                  < 0    An error occurred.

int **su\_VideoCameraGetDefault** (uint iID, SU\_VIDEO\_SETTINGS \*pCurrent)

**Summary** Returns video camera default setting information.

**Details** This function can be called once the video camera has been connected to the USB and configured by the device driver. It returns the default setting information, which includes the camera controls, such as AutoExposureMode and Focus. It also includes processing unit controls, such as Brightness and Contrast.

**Parameters** iID Index of the video device. Should be less than SU\_VIDEO.  
pDefault Pointer to the default settings.

**Returns** 0 Got default settings.  
< 0 An error occurred.

int **su\_VideoCameraGetMin** (uint iID, SU\_VIDEO\_MIN\_MAX \*pMin)

**Summary** Returns video camera minimum values of the setting information.

**Details** This function can be called once the video camera has been connected to the USB and configured by the device driver. It returns the minimum values of the setting information, which includes the camera control, such as ExposureTimeAbsolute and FocusAbsolute. It also includes processing unit controls, such as Brightness and Contrast.

**Parameters** iID Index of the video device. Should be less than SU\_VIDEO.  
pMin Pointer to the minimum values.

minimum value structure is defined as  
typedef struct  
{  
/\* Camera Control \*/  
u32 dwExposureTimeAbsolute;  
u16 wFocusAbsolute;  
u8 bFocusRelative;  
u8 bFocusRelativeSpeed;  
u16 wIrisAbsolute;  
u16 wObjectiveFocalLength;  
s8 bZoom;  
u8 bDigitalZoom;  
u8 bZoomSpeed;  
s32 dwPanAbsolute;  
s32 dwTiltAbsolute;  
s8 bPanRelative;  
u8 bPanSpeed;  
s8 bTiltRelative;

```

    u8 bTiltSpeed;
    u16 wRollAbsolute;
    u8 bRollRelative;
    u8 bRollRelativeSpeed;
    /* Processing Unit Control */
    u16 wBacklightCompensation;
    s16 wBrightness;
    u16 wContrast;
    u16 wGain;
    s16 wHue;
    u16 wSaturation;
    u16 wSharpness;
    u16 wGamma;
    u16 wWhiteBalanceTemperature;
    u16 wWhiteBalanceBlue;
    u16 wWhiteBalanceRed;
    u16 wMultiplierStep;
    u16 wMultiplierLimit;
    u16 wSyncDelay;
}SU_VIDEO_MIN_MAX;

```

**Returns**     0     Got the minimum values of settings.  
               < 0    An error occurred.

int **su\_VideoCameraGetMax**(uint iID, SU\_VIDEO\_MIN\_MAX \*pMax)

**Summary**     Returns video camera maximum values of the setting information.

**Details**     This function can be called once the video camera has been connected to the USB and configured by the device driver. It returns the maximum values of the setting information, which includes the camera controls, such as ExposureTimeAbsolute and FocusAbsolute. It also includes processing unit controls, such as Brightness and Contrast.

**Parameters**   iID     Index of the video device. Should be less than SU\_VIDEO.  
                   pMax    Pointer to the maximum values.

**Returns**     0     Got the maximum values of settings.  
               < 0    An error occurred.

int **su\_VideoCameraGetInfo**(uint iID, SU\_VIDEO\_INFO \*pInfo)

**Summary** Returns video camera capabilities and status of the specified control.

**Details** This function can be called once the video camera has been connected to the USB and configured by the device driver. It returns the capabilities and status of the specified control, which includes the camera controls, such as ExposureTimeMode and FocusAuto. It also includes processing unit controls, such as Brightness and Contrast.

**Parameters** iID Index of the video device. Should be less than SU\_VIDEO.

pInfo Pointer to the information. The information is defined as:

```
typedef struct
{
    u8 bDevPowerMode;
    u8 bRequestErrorCode;
    /* Camera Control */
    u8 bScanningMode;
    u8 bAutoExposureMode;
    u8 bAutoExposurePriority;
    u8 dwExposureTimeAbsolute;
    u8 bExposureTimeRelative;
    u8 wFocusAbsolute;
    u8 bFocusRelative;
    u8 bFocusRelativeSpeed;
    u8 bFocusAuto;
    u8 wIrisAbsolute;
    u8 bIrisRelative;
    u8 wObjectiveFocalLength;
    u8 bZoom;
    u8 bDigitalZoom;
    u8 bZoomSpeed;
    u8 dwPanAbsolute;
    u8 dwTiltAbsolute;
    u8 bPanRelative;
    u8 bPanSpeed;
    u8 bTiltRelative;
    u8 bTiltSpeed;
    u8 wRollAbsolute;
    u8 bRollRelative;
    u8 bRollRelativeSpeed;
    u8 bPrivacy;
    /* Select Unit Control */
    u8 bSelector;
    /* Processing Unit Control */
    u8 wBacklightCompensation;
```

```

    u8 wBrightness;
    u8 wContrast;
    u8 wGain;
    u8 bPowerLineFrequency;
    u8 wHue;
    u8 bHueAuto;
    u8 wSaturation;
    u8 wSharpness;
    u8 wGamma;
    u8 wWhiteBalanceTemperature;
    u8 bWhiteBalanceTemperatureAuto;
    u8 wWhiteBalanceBlue;
    u8 wWhiteBalanceRed;
    u8 bWhiteBalanceComponentAuto;
    u8 wMultiplierStep;
    u8 wMultiplierLimit;
    u8 bVideoStandard;
    u8 bAnalogVideoLockStatus;
    u8 wSyncDelay;
    u8 ProbeCommit;
}SU_VIDEO_INFO;

```

For each field of this structure, 0 means this setting is not supported.

**Returns**     0     Get the information settings.  
               < 0    An error occurred.

```
int su_VideoCameraSetCaptureFrame(uint iID, uint iFormat, u16 wWidth, u16 wHeight, u32
dwFrameInterval)
```

**Summary**     Set the desired video camera capture format and frame settings.

**Details**     This function can be called once the video camera has been connected to the USB and configured by the device driver. Most video camera support multiple format, such as uncompressed (YUV 422 or YUV 420) or MJPEG. For each format, it may also support multiple frame size (such as 640x480, 320x240, etc) and frame interval (such as 333333 for 20 frame per second or 1000000 for 10 fps). Video camera also has default format and frame that you can use without calling this function but if you don't want to use the default settings. You can use this function to change the capture format and frame settings.

You may need to call su\_VideoCameraGetCaptureFormatNum() and su\_VideoCameraGetCaptureFormat() to query the format and frame information this video camera supported and only pass the supported format, width, height and frame interval to this function.

**Parameters**

iID	Index of the video device. Should be less than SU_VIDEO.
iFormat	One of the following format. SU_VIDEO_FORMAT_UNCOMPRESSED SU_VIDEO_FORMAT_MJPEG
wWidth	The width of the desired frame
wHeight	The height of the desired frame
dwFrameInterval	The interval of each frame. The unit is 1/10 microsecond so 333333 mean 33 frame per second.

**Returns**

0	Format and frame is set.
< 0	video camera is not connected or the settings is invalid.

int **su\_VideoCameraSetCurrent**(uint iID, SU\_VIDEO\_SETTINGS \*pNewCurrent)

**Summary** Change the current settings of the video camera control.

**Details** If you need to change any of the video camera control, such as the camera's controls or the processing unit controls, you need to call this function. You need to call su\_VideoCameraGetCurrent() first to get the current one and modify the desired value and call this function to let it take effect. Video class driver will only set request to the changed control.

**Parameters**

iID	Index of the video device. Should be less than SU_VIDEO.
pNewCurrent	Pointer to the new camera control settings.

**Returns**

0	Video camera control settings are updated.
< 0	Video camera is not connected.

int **su\_VideoCameraGetCaptureSize**(uint iID, u32 \*pMaxVideoFrameSize, u32 \*pMaxPayloadBufferSize)

**Summary** Get the size of the video capture-related buffers.

**Details** The application needs to allocate enough memory buffers to perform the video capture smoothly. This function will provide the maximum buffer size that is needed according to the settings for format and frame size. You may need to call su\_VideoCameraSetCaptureFrame() first if you don't want to use the default settings.

Because it is unlikely the video camera output data format (either uncompressed YUV or compressed MJPEG) can be used directly by the LCD display frame buffer (most are RGB format), the application may require at least two captured frame

buffers. One should be used to store the captured whole frame data so application can convert them to the desired LCD display format. Another buffer can be used to capture the incoming video streaming data.

**Parameters** `iID` Index of the video device. Should be less than `SU_VIDEO`.  
`pMaxVideoFrameSize` Pointer to the maximum video frame buffer size. For uncompressed format, this value is the same for each frame and it is  $(\text{bBitsPerPixel}/8) * \text{wWidth} * \text{wHeight}$ . For compressed formats, the value may be much larger than the actual data size because some video cameras may use the uncompressed video frame size for the compressed format. The actual data length for each frame may vary. See the information for function `su_VideoCameraCapture()` about how to determine the data length.  
`pMaxPayloadBufferSize` The pointer to the maximum payload buffer size for each packet. Normally you need to pass this size when you call `su_VideoCameraCapture()` to make sure the video class driver will not drop any data because of the buffer size.

**Returns** `0` Got the video capture buffer size.  
`< 0` Video camera is not connected.

int `su_VideoCameraOpen` (uint `iID`)

**Summary** Start to capture video streaming data.

**Details** Call this API to start video streaming.

**Parameters** `iID` Index of the video device. Should be less than `SU_VIDEO`.

**Returns** `0` Video streaming started.  
`< 0` Video camera is not connected.

int `su_VideoCameraCapture`(uint `iID` u8 \*`pData`, uint `iLen`, BOOLEAN \*`pbIncludeNewFrame`, uint \*`piNewFrameOffset`)

**Summary** Retrieve some video streaming data.

**Details** After the application calls `su_VideoCameraOpen()`, it will still need to keep calling `su_VideoCameraCapture()` to retrieve the received video streaming data. Normally the application needs to call it in a high priority task to keep up with the streaming

data. Check the the example code in `usbhdemo.c` for details about how to capture video streaming data

**Parameters**

- `iID` Index of the video device. Should be less than `SU_VIDEO`.
- `pData` Pointer to the captured video streaming data
- `iLen` Size of the capture buffer. Should be `MaxPayloadBufferSize` or multiple of `MaxPayloadBufferSize`
- `pbIncludeNewFrame` Boolean value if the return data buffer contains the start of a new frame. If the returned value is `TRUE` and `iNewFrameOffset` is less than the return value of this function, then the returned data buffer contains the data for two frames. The data from `pData` to `iNewFrameOffset` is the last part of the previous frame and the data from `iNewFrameOffset` to `iLen` is the start of the next frame.
- `piNewFrameOffset` Pointer to the offset of the new frame within the returned data buffer if `bIncludeNewFrame` is `TRUE`. Do not use the value if `bIncludeNewFrame` is `FALSE`.

**Returns**

- `>= 0` Data length in the buffer.
- `< 0` Video camera is not connected or an error occurred during the capture.

int **su\_VideoCameraClose** (uint iID)

**Summary** Stop capturing video streaming data.

**Details** Call this API to stop the video streaming.

**Parameters** `iID` Index of the video device. Should be less than `SU_VIDEO`.

**Returns**

- `0` Video streaming stopped.
- `< 0` Video camera is not connected.

int **su\_VideoInserted** (uint iID)

**Summary** Check if there is any video device connected.

**Details** Call this API to check is any video class device is available to use.

**Parameters** `iID` Index of the video device. Should be less than `SU_VIDEO`.

**Returns**

- `TRUE` Video class device is connected.
- `< 0` No video class device.



## 4.9 Writing a New Class Driver

**Note: Writing a new USB class driver is difficult and may require assistance from Micro Digital. The following information may not cover every detail about implementing a new class driver. Please discuss this with Micro Digital before you decide to do it.**

Micro Digital provides a class driver template in the CTempl subdirectory. This class driver template is a full-featured, working driver that shows how to send/receive control, bulk, and interrupt requests. smxUSBD has a function driver template which can work with this class driver for demonstration.

The class driver requires the interface class, subclass, and protocol to be 0xFF, and there are four endpoints not including the default control endpoint. These four endpoints are for BULK IN, BULK OUT, INT IN, and INT OUT.

### 4.9.1 Send Vendor-Specific Control Request

Two vendor-specific requests are defined as:

```
SU_REQ_SET_DATA
SU_REQ_GET_DATA
```

su\_CTemplSendCtrl() sends a vendor-specific request to send 8 bytes of data to the device.  
su\_CTemplSendCtrl() sends a vendor-specific request to get 8 bytes of data from the device.

The data sent and received can be compared to verify that the transfer is correct. Refer to the code for the details.

### 4.9.2 Send and Receive BULK Requests

su\_CTemplSendBulk() sends bulk data to the device.  
su\_CTemplRecvBulk() receives bulk data from the device.

Refer to the code for the details of how to send and receive bulk requests.

### 4.9.3 Send INT Request

su\_CTemplSendINT() sends INT data to the device.  
su\_CTemplRecvINT() receives INT data from the device.

Normally for the INT IN request, the host controller will resubmit so the device can interrupt the host at any time. Therefore the class driver does not need to call the su\_CTemplRecvINT() function to poll it. The INT callback functions will be called when the host gets the data from the device. However, if there is a requirement to support polling, handle the INT IN request in the class driver instead of the host driver. In order to do that, within the INT IN call back function, you need to set pReqInfo->status to SU\_REQ\_STAT\_KILLED so the host controller driver will not automatically resubmit the request.

## 4.10 Device Plugin/Remove Event Callback

In smxUSBH version 2.31, a new callback function was added to allow the application to be notified when a USB device is plugged in or removed. Each class driver has its own notification function.

Use the following function to register a PlugIn / Remove callback function

```
void su_XXXRegDevEvtCallback(SU_PDEVEVTCBFUNC func);
```

Check the class driver header file to find out the exactly function name for each class driver.

Following is a sample callback function

```
static void smx_msc_device_event(uint iID, BOOLEAN bInserted)
{
    if(bInserted)
    {
        printf("mass storage device plugged in\r\n");
    }
    else
    {
        printf("mass storage device remove\r\n");
    }
}
```

## 4.11 Stack Event Callback

In smxUSBH version 2.51, a new callback function was added to allow the application to be notified when smxUSBH needs to report some events. In version 2.54, the data parameter was added to this callback function

Use the following function to register the callback function:

```
void su_RegStackCallback(SU_PSTACKEVTCBFUNC pCallback);.
```

```
typedef void (* SU_PSTACKEVTCBFUNC)(uint iFlag, u32 data);
```

iFlag will be one of the following values:

**SU\_CB\_DEVICE\_INSERTED**

When the hub (root or external) detects there is a device on the hub port, the callback function will be called with this flag. At that time, the device is not enumerated yet, so the stack does not know if it can support it or not. Data is (HubAddr<<16|HubPort).

**SU\_CB\_DEVICE\_REMOVED**

When the hub (root or external) detects that a device on the hub port is removed, the callback function will be called with this flag. Data is (HubAddr<<16|HubPort).

**SU\_CB\_HUB\_INSERTED**

When the hub (root or external) detects there is an external hub device on the hub port, the callback function will be called with this flag. At that time, the device is already enumerated. Data is (HubAddr<<16|HubPort).

**SU\_CB\_HUB\_REMOVED**

When the hub (root or external) detects that an external hub device on the hub port is removed, the callback function will be called with this flag. Data is (HubAddr<<16|HubPort).

**SU\_CB\_OVERCURRENT**

When the host controller's root hub or external hub reports an overcurrent event, the callback function will be called with this flag. Data is (HubAddr<<16|HubPort).

**SU\_CB\_UNKNOWN\_DEV\_REMOVED**

When a previous unsupported/unrecognized device has been removed, the callback function will be called with this flag. Data is (HubAddr<<16|HubPort).

**SU\_CB\_HUB\_UNSUPPORTED**

When a hub device is detected but SU\_HUB is not set, the callback function will be called with this flag. Data is (HubAddr<<16|HubPort).

**SU\_CB\_DEVICE\_UNRECOGNIZED**

When the hub task cannot enumerate the attached device, the callback function will be called with this flag. Data is (HubAddr<<16|HubPort).

**SU\_CB\_DEVICE\_UNSUPPORTED**

When the hub task cannot find a driver for an enumerated device, or when the driver's deviceConn() interface returns an error so the class driver cannot be loaded, the callback function will be called with this flag. Data is (HubAddr<<16|HubPort).

**SU\_CB\_HC\_FATAL\_ERROR**

When the host controller gets an unrecoverable error, the callback function will be called with this flag. For most cases you may need to shutdown smxUSBH and re-init it to recover from this problem. Data is the Host controller index.

**SU\_CB\_STACK\_FATAL\_ERROR**

When smxUSBH detects any internal error, such as data is corrupted or insufficient memory, the callback function will be called with this flag. For most cases you may need to shutdown smxUSBH and re-init it to recover from this problem. Data is the error number SU\_STACK\_FATAL\_ERROR\_XXX.

## 5. Host Controller Drivers

### 5.1 Host Controller Driver Interface

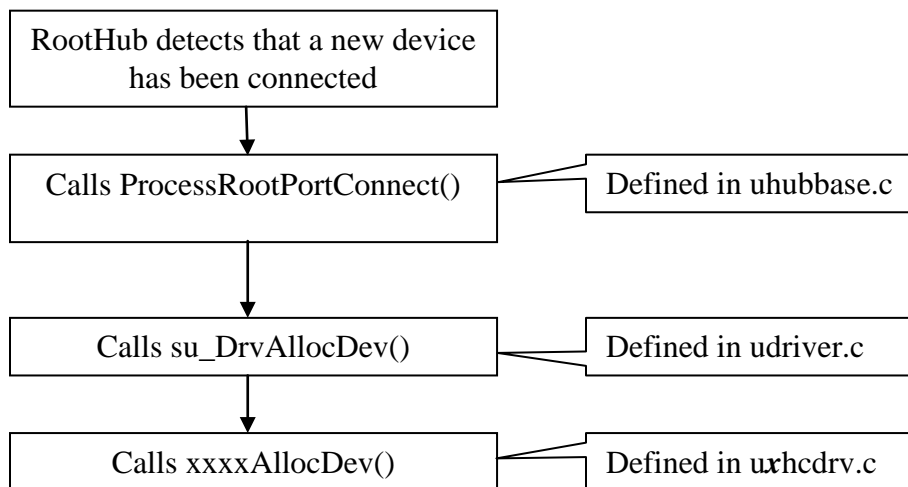
The USB Driver defines a structure `SU_HC_OP` that contains five function pointers. All Host Controller Drivers should implement these functions and call `su_DrvAllocHost()` and `su_DrvApplyUSBHost()` to register the Host Controller with the USB Driver. The functions are described below. They are:

```
int AllocDev(SU_DEV_INFO_PTR pDevInfo)
int FreeDev(SU_DEV_INFO_PTR pDevInfo)
int TransferCmd(SU_REQ_INFO_PTR pReqInfo)
int RemoveCmd(SU_REQ_INFO_PTR pReqInfo)
int RootHubCmd(uint iHCIndex, uint portNum, uint Req, u32 *pData)
```

#### 5.1.1 AllocDev()

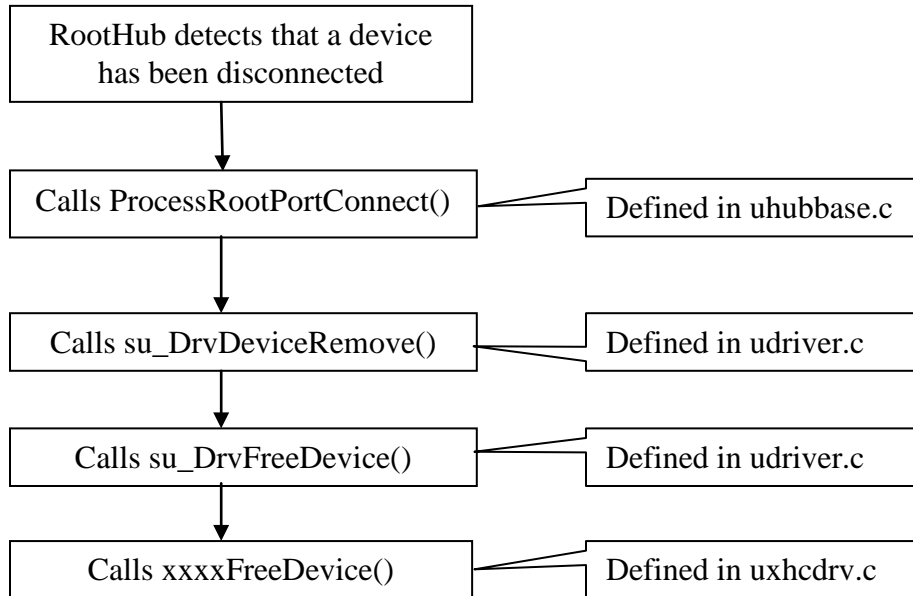
This function is called when the hub task detects that a new device has been connected to a hub port. The Host Controller may allocate the newly added device information structure. The information may only be used internally by the host controller driver. You can save the pointer to that structure in `pDevInfo->pPrivateData` so it can be used later.

The calling procedure is:



### 5.1.2 FreeDev()

This function is called when the hub detects that a device has been removed from a hub port. The Host Controller will free the device's information structure. `pDevInfo->pPrivateData` points to it. The following series of calls is made:



### 5.1.3 TransferCmd()

The USB Driver calls this function to send a request to the device. All device/class drivers' data transfer requests will ultimately call this function. The basic idea is to create a list for each type of transfer request: control, bulk, interrupt, and isochronous. Whenever `HCXXXXTransferCmd()` is called, the request is added to the list. Then general information carried by that request is converted to the specific host controller related data structure, such as some kind of Transfer Descriptor. Then that TD is submitted to the host controller. The request can be sent immediately or sent later by an ISR, usually the SOF interrupt. Within the ISR, if any request is done, it should be removed from the list, and the caller task is notified through the callback function of that request. Then it checks if there are any other pending requests and schedules those requests.

### 5.1.4 RemoveCmd()

The USB Driver calls this function to cancel a pending request sent to the device. Normally the pending interrupt request will be removed through this function. If any request times out or the device is removed, the USBH stack and/or class driver calls this function to remove the pending request from the host controller. If the host controller has any data structure associated with that request, such as a list of Endpoint/Transfer Descriptors, the function should also remove that from the host controller.

### 5.1.5 RoothubCmd()

The Roothub Driver calls this function to get the host controller's roothub information, such as total port number and port status. It will also call this function to set/clear the roothub's features, such as: enable and power to the port, reset the port, and clear connect status changed flag.

## 5.2 Interrupt Handler (ISR)

smxUSBH only processes the host controller interrupt in the ISR in a non-multitasking system (ISB\_MULTITASKING). In a multitasking environment, the interrupt processing is done by a separate high priority task. This approach ensures other higher priority jobs can preempt the USB task. Usually, USB communication is not time-critical.

For the EHCI/OHCI/UHCI controller, if you need to support SMP or multiple controllers and shared interrupts, you need to set SU\_USE\_PIPE to 1 in ucfg.h. The controller driver first reads the interrupt status register to ensure the interrupt is triggered by this controller. If so, it writes to the interrupt status register to clear the interrupt and then passes the value of the interrupt status register to the interrupt handler task through a pipe. Then the ISR returns. The interrupt handler task processes the interrupt according to the received interrupt status register value. The interrupt is never disabled for this case.

For other cases, to improve the performance of the controller driver, smxUSBH will not pass the interrupt status register by pipe. Instead, the controller driver disables the USB interrupt in the ISR first, then wakes up the interrupt handler task, and then the ISR returns. The interrupt handler task reads the interrupt status register and processes it. During this whole time the USB host controller interrupt will remain disabled. After the interrupt processing is done, the USB interrupt is re-enabled by the interrupt handler task.

Interrupt handling is done internally in the porting layer of smxBase not within the host controller driver. For detailed information, please see smxBase User' Guide. Each controller driver has an InterruptHandler() function that must be called from the ISR. The prototype for these is shown in uinit.h. These are defined in uinit.h rather than in another header file for those who purchase the USB stack in binary form, since a minimal set of files is supplied.

If multiple controllers are being used in your system the ISR dispatcher will pass the host controller index to the ISR through the parameter iHCIndex, You can then decide which host controller triggered that interrupt.

## 5.3 Root Hub

The root hub provides the connection between the Host Controller and one or more USB ports. The root hub provides simplified functionality vs. an external hub (see the USB 2.0 specification, "Chapter 11 Hub Specification"). smxUSBH's Host Controller constructs a virtual Root Hub device as the parent device for all attached USB devices. The host controller may start a timer to poll the status of the root hub port if there is no interrupt for it.

Root hub support is included in smxUSBH. External hub support is an add-on option. Some USB devices have a built-in hub, others don't. If external hub driver support is needed, the add-on must be purchased.

## 5.4 EHCI

EHCI only supports high speed devices, so in order to connect to a full speed or low speed device, use a high speed hub (Such as NXP ISP176X) or enable the companion OHCI/UHCI controller driver (such as most PCI cards and Atmel AT91SAM9M10G45). Some processors, such as Freescale's MCF5329/5251/54455/52277 have built-in EHCI compatible controllers, and the on-chip or external transceiver may allow connection to full or low speed devices directly through the USB root hub port.

## 5.5 OHCI

OHCI supports full speed and low speed. Many ARM and other embedded processors have a built-in OHCI-compatible USB host controller.

### 5.5.1 OHCI for x86 Real Mode

OHCI uses memory-mapped I/O for access to the OHCI registers. Unfortunately, the PCI BIOS assigns a high address near the top of the 4GB memory space, so this is inaccessible in real mode, since memory addressing is limited to about 1MB. smxUSBH offers two solutions to this problem. Try them in this order.

1. Change the PCI base address to an address in the Upper Memory Area. Run our OHCIBASE utility from a DOS prompt to find all possible addresses between the address specified and 0xF0000. For example:

```
A:>ohcibase c8000
```

This utility will display a list of possible addresses or it will indicate if none are found. Set SU\_OHCI\_BASE to one of the addresses it returns and try running smxUSBH. Also ensure SB\_CFG\_REAL\_BIG\_MODE is 0. Do not select an address that is used by some other device. For example DiskOnChip uses a memory window in this area, which is typically configured by jumpers. Also, see [www.smxrtos.com/rtos/dos/undwinpb.htm](http://www.smxrtos.com/rtos/dos/undwinpb.htm) for a memory map from 0 to 1MB for more explanation.

This is the easiest technique and adds negligible code (a couple C statements). However, it cannot be used on some systems. In that case, try option 2.

2. Real Big Mode (also called Unreal Mode): This relies on a quirk of the x86 architecture. A search for these terms on the web will yield various articles and code snippets about this. The basic idea is to temporarily enter protected mode to set up a segment register for access to the 4GB memory space and then switch back to real mode, and the register still has 4GB access. It only works on 386 or higher processors. Also, it requires assignment of the FS or GS

register for only the purpose of accessing high addresses. Typically a real mode program uses only ds and es. However, a third-party library or even your own code may use FS or GS.

Do not reload this register in real mode or it loses its “bigness”. In particular, it is not permissible to push and pop FS and GS, because the hidden part of the register is changed. The whole point of switching to protected mode was to set that hidden part, which is done when loading a segment selector into the segment register. If you cannot allocate either FS or GS to this purpose, it should be possible to solve the problem by modifying smxUSBH to call rbm\_init() everywhere prior to accessing the OHCI registers.

Note that the code sets up FS for this purpose but can be easily changed to use GS by searching and replacing in rbm.asm.

This technique should work on any system but has the above disadvantages and adds some code. It also does not work under emm386.exe. If this technique is not acceptable, consider upgrading your application to protected mode.

## **5.6 UHCI**

UHCI USB controllers are usually used only with x86 processors.

### **5.6.1 UHCI for x86 Real Mode**

Since UHCI uses x86 I/O space to access the UHCI registers, there is no problem when running in real mode vs. protected mode. However, it is necessary to have a 386 or higher processor because UHCI requires 32-bit I/O instructions and 32-bit registers.

## **5.7 ISP116x**

As with other NXP USB controllers, NXP ISP116X has one hardware configuration register to set up properties related to the microprocessor, such as the interrupt trigger mode and level, whether to use internal pull-down resistor, etc. Refer to the ISP116x data sheet for the details. Micro Digital provides the function su\_GetISP116XIntSetting() in the hardware porting layer so modifications are needed to return the correct settings for your hardware.

## **5.8 ISP1362**

As with other NXP USB controllers, the NXP ISP1362 has one hardware configuration register to set up properties related to the microprocessor, such as the interrupt trigger mode and level, whether to use internal pull-down resistor, etc. Refer to the ISP1362 data sheet for the details. Micro Digital provides the function su\_GetISP1362IntSetting() in the hardware porting layer so modifications are needed to return the correct settings for your hardware.

## **5.9 ISP176x**

As with other NXP USB controllers, NXP ISP176x has one hardware configuration register to set up properties related to the microprocessor, such as the interrupt trigger mode and level. Refer to the ISP176x data sheet for details. Micro Digital provides the function



su\_GetISP176XIntSetting() in the hardware porting layer so modifications are needed to return the correct settings for your hardware.

If you are using the ISP1763A, remember to set SU\_ISP1763 to 1 in ucfg.h. ISP1763A is basically the same as ISP1760/1 controller, with the following minor differences:

1. Data bus width is 8 or 16 bit for ISP1763
2. Address lines are only 8 bits for ISP1763
3. Number of TDs is only 16 for ISP1763
4. Payload memory is only 20K for ISP1763
5. Register offsets changed and some registers changed to 16 bit.

Remember to check the low level register access functions we implemented in uport.c for the ISP1763A case for how to handle the 32-bit/16-bit register changes.

Set **SU\_ISP1763\_PORT1** to 1 for port 1 of ISP1763A chip to be a dedicated host port.

Set **SU\_ISP1761\_PORT1** to 1 for port 1 of ISP1761 chip to be a dedicated host port.

Set **SU\_ISP1760\_PORT1** to 1 for port 1 of ISP1760 chip to be a dedicated host port.

## 5.10 Blackfin

Analog Devices BF52x/BF54x USB host controller driver currently does not support ISOC transfer mode. This controller IP does not support external hubs and multiple devices. This controller is an IP implementation from Mentor.

## 5.11 CF522xx

Freescale CF522xx USB host controller provides limited USB host functionality. Micro Digital currently does not support external hubs and multiple devices for this host controller. ISOC transfer mode is not tested. This controller cannot schedule multiple USB requests so full-duplex data transfer is not supported.

## 5.12 LM3Sxxxx

The host controller on many LM3S processors (prior to Tempest class, e.g. LM3S9B9x) has a very limited number of endpoints, so for them, connecting a hub only allows supporting two mass storage devices simultaneously. This is not a problem on TI AM17xx/18xx, AM33x, and AM35x processors.

## 5.13 MAX3421

Maxim MAX3421 interfaces with the microprocessor through an SPI bus so it is necessary to implement SPI communications. In addition, the interrupt trigger mode and level must be set up. This should be done in the su\_HdwInit(). Refer to Micro Digital's demo code for the MAX3421 evaluation board and Keil MCB2130 board. ISOC transfer mode is not tested. This controller cannot schedule multiple USB requests, so full-duplex data transfer is not supported.

## 5.14 Renesas

Renesas USB host controller driver currently does not support ISOC transfer mode. This controller does not support high speed interrupt transfer so external hub and multiple device support are not fully tested.

## 5.15 Synopsys

Synopsys DWC2 host controller driver supports only internal DMA/Slave Only mode and external ULPI/FS UTMI. For Dedicated FIFO mode, the size of each FIFO needs to be configured at the top of the controller driver for each processor which is using this USB IP.

Unfortunately, this controller generates a very high number of interrupts, which can greatly reduce system performance. It generates 8,000 SOF interrupts per second. Our driver disables SOF when there are no transfers to do, but while active, these interrupts are handled. Worse is using a high speed hub with a full speed device attached. This causes polling IN requests to be done as SPLIT transactions, sometimes a few per frame. For example, with FTDI232 USB to serial adapter plugged in, we've seen 4 SPLIT IN transfers each microframe, meaning 32,000 channel interrupts per second, just for polling! That's in addition to the 8,000 SOF interrupts, and this incurs so much overhead, there is almost no usable CPU time left, even on a CM7 processor. Therefore, if using a hub, match it to the device speed (HS/HS or FS/FS, not HS/FS). Other controllers, such as EHCI have a very low interrupt rate.

We have had other problems with this controller, such as having channels lock up, and we're unable to restore them. Mass storage works well, fortunately.

Because of the performance overhead and problems, we suggest choosing a processor that uses a different USB controller. The most popular processors using it are the STM32 family.

## 5.16 uPD720150

NEC uPD720150 USB host controller driver has been tested on the LPC1788 Embedded Artists board and uPD720150 Application Board (part #ET-D720150-HP) using a custom interface board. DMA data transfer is not support yet. Port2 as host port has not been tested, but the feature is in the controller driver.

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. If you are only using the host controller, you can build upd720150.c into the XUSBH library.

Set `SU_PD720150_USE_PORT2` to 1 set port 2 as USB host port.

## 5.17 Writing a New Host Controller Driver

**Note: Writing a new USB host controller driver is difficult and may require a lot of assistance from Micro Digital. We do not recommend you do it yourself. Please discuss this with Micro Digital before you decide to do it. The following information only shows the steps to write a new host controller driver. A lot of detailed information is not provided here.**

Micro Digital provides a host controller driver template in the HCD directory to use as a starting point. Check all comments marked “TODO” and implement those sections according to the instructions below.

1. Create an HC initialization and release function (`su_HCXXXXXInit()` and `su_HCXXXXXRelease()`) and call them in `uinit.c`. See 5.17.1 HC Init and Release.
2. Implement the HCD interface functions and register them with USB. See 5.17.2 HCD Interface.
3. Create an ISR to handle the host controller’s interrupt and hook the interrupt vector in `su_HCXXXXXInit()`. See 5.17.3 HC Interrupt.
4. Create a virtual root hub and add it to the USB. See 5.17.4 HC Virtual Root Hub.
5. Use a timer or HC interrupt get the root hub status change event and then report it to the USB. Then the hub thread can run to check the attached or detached device. See 5.17.4 HC Virtual Root Hub.

### 5.17.1 HC Init and Release

In the `su_HCXXXXXInit()` function, do the following typical steps:

1. Check the HC’s ID or Revision number to make sure it is the HC you want to support.
2. Allocate a handle (structure) to record all the important information of this HC.
3. Reset the HC so it resets to the default settings.
4. Allocate any Transfer Descriptor related resources that are needed.
5. Register the HCD with the USB.
6. Initialize the HC itself.
7. Unmask (enable) the HC’s interrupt. Then the HC can begin to work

Our template code provides a framework for the above steps as a starting point for your development.

In the `su_HCXXXXXRelease()` function, you normally need to release all the resources allocated in `su_HCXXXXXInit()` and also disable the HC interrupt.

### 5.17.2 HCD Interface

How you implement `HCXXXXXAllocDev()`, `HCXXXXXFreeDev()`, `HCXXXXXTransferCmd()`, `HCXXXXXRemoveCmd()`, and `HCXXXXXRootHubCmd()` depends on your host controller and your implementation. The EHCI/OHCI/UHCI code is fairly complex; the ISPxxx code is simpler. Refer to the template code and contact Micro Digital for support, if necessary. (see 5.1 Host Controller Driver Interface)

### 5.17.3 HC Interrupt

The interrupt handler depends on your HC and environment. The template cannot do more than provide a stub function.

### 5.17.4 HC Virtual Root Hub

Our template provides most of the code for the root hub implementation. You only need to implement the functions related to your HC, most likely read/write registers.

If you don't want or can't use an interrupt to inform you the roothub change event, you can implement the request `SU_HUB_ROOT_GET_CHG_STATUS` in the `RoothubCmd()` function so the Hub task will keep polling for status changes.

### 5.17.5 Debug the Code

It is recommended to debug the host controller driver following this sequence:

1. Device insert/remove event detection
2. Virtual Root Hub Port Reset procedure when new device is plugged in
3. Control Transfer for device enumeration
4. Bulk/full/high speed transfer for mass storage device
5. Interrupt/low speed transfer for mouse/keyboard device
6. Control/Bulk/Interrupt transfer for external Hub device+one mass storage or mouse device
7. Remove and then plug in device again. Host controller driver error handling and content cleanup
8. Multiple device support through hub. For example, two mass storage devices or one mass storage and one mouse.
9. Split transfer if your host controller is a high speed one.
10. Test ISOC/full/high speed transfer.

## 6. Hardware Porting Notes

General interrupt-related hardware porting layer functions are defined in `smxBase`. Please see the `smxBase` User's Guide for detailed information. The `smxUSBH`-specific hardware porting layer consists of `uport.h` and `uport.c`. These files contain definitions, macros, and functions to port `smxUSBH` to particular target hardware.

### 6.1 `uport.h`

For clarity and simplicity it is recommended to delete the conditionals used around porting definitions. Just keep one of each setting configured as appropriate for your target. There is also no need for the `#error` when just one configuration is supported. Micro Digital code must build for different processors and compilers. Your needs are simpler — you only need to set things for one configuration or a limited number.

- A. Driver BASE and IRQ settings: Set these to the proper addresses for your host controller.
- B. Non-cacheable memory address: If you are using OHCI/UHCI/EHCI compatible host and enable the data cache on your system.

### 6.2 `uport.c`

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

#### `su_HdwInit()`

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

Initializes the hardware platform's USB subsystem. For example, it enables the USB host controller, sets up the clock, and finds the PCI BIOS.

Determines the Host Controller's I/O base, memory base and IRQ number. For systems configured to use an OHCI controller, this function initializes the variables `OHCIbase[]` and `OHCIirq[]` with the controller's memory address and IRQ assignment. For systems using a UHCI controller, this function initializes `UHCIbase[]` and `UHCIirq[]`. For systems using a EHCI controller, this function initializes `EHCIbase[]` and `EHCIirq[]`.

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

#### `su_HdwRelease()`

Disables the USB subsystem of the hardware.

#### `su_FlushData()`, `su_PhysicalToVirtualAddr()`, `su_VirtualToPhysicalAddr()`

These are for MMU support. `su_FlushData()` flushes data in the cacheable memory. The others are address conversion functions. **If you are not using an MMU, use the default implementation.**

### **su\_UHCIRead16(), su\_UHCIWrite16(), su\_UHCIWrite32()**

These functions are used when the smxUSBH UHCI driver accesses I/O registers. Normally these should be mapped onto your compiler's functions (i.e. inpw(), outpw(), outpd()). The default implementation maps these onto functions defined in smx since some versions of the Borland and Microsoft 32-bit compilers have buggy definitions of these.

### **su\_OHCIRead32()/su\_EHCIRead32(), su\_OHCIWrite32()/su\_EHCIWrite32()**

These functions are used when the smxUSBH OHCI/EHCI driver accesses memory mapped registers.

**You may need to check if the default implementations of the above I/O functions meet the hardware timing requirements of your system.**

**su\_GetEHCIBase(), su\_GetEHCIInterrupt()  
su\_GetOHCIBase(), su\_GetOHCIInterrupt()  
su\_GetUHCIBase(), su\_GetUHCIInterrupt()  
su\_GetISP116XBase(), su\_GetISP116XInterrupt()  
su\_GetISP1362Base(), su\_GetISP1362Interrupt()  
su\_GetISP176XBase(), su\_GetISP176XInterrupt()  
su\_GetSynopsysBase(), su\_GetSynopsysInterrupt()**

These functions return the EHCI, OHCI, UHCI, NXP ISP116x/ISP1362/ISP176x or Synopsys DWC host controller I/O base address and IRQ found in su\_HdwInit().

**Typically there is no need to change the default implementation but you need to set the correct base address and IRQ number in the uport.h.**

### **su\_GetISP116XIntSetting(), su\_GetISP1362IntSetting(), su\_GetISP176XIntSetting()**

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

**su\_ISP116XRead32(), su\_ISP116XRead16(), su\_ISP116XWrite32(), su\_ISP116XWrite16()  
su\_ISP1362Read32(), su\_ISP1362Read16(), su\_ISP1362Write32(), su\_ISP1362Write16()  
su\_ISP176XRead32(), su\_ISP176XRead16(), su\_ISP176XWrite32(), su\_ISP176XWrite16()**

These functions are used when the smxUSBH NXP ISP116x/ISP1362/ISP176x host driver accesses I/O registers. **It may be necessary to tune the implementation of these functions to meet the timing requirement of ISP116x/ISP1362/ISP176x according to your hardware implementation.**

**su\_ISP116XReadBuf(), su\_ISP116XWriteBuf()  
su\_ISP1362ReadBuf(), su\_ISP1362WriteBuf()  
su\_ISP176XReadBuf(), su\_ISP176XWriteBuf()**

These functions are used when the smxUSBH NXP ISP116x/ISP1362/ISP176x host driver accesses ATL and or ITL buffer. **It may be necessary to tune the implementation of these functions to meet the timing requirement of ISP116x/ISP1362/ISP176x according to your hardware implementation.**

**su\_MAX3421Read(), su\_MAX3421Write()  
su\_MAX3421ReadBuf(), su\_MAX3421WriteBuf(), su\_MAX3421VBusOn()**

These functions are used when the smxUSBH Maxim MAX3421 host driver accesses registers. MAX3421 is based on the SPI bus so these functions should use the microprocessor's SPI controller to implement the communications. MAX3421VBusOn() is used to turn on/off the VBus of USB.

**su\_SynopsysRead(), su\_SynopsysWrite()**

These functions are used when the smxUSBH Synopsys DWC host driver accesses registers.

**su\_PCIReadConfigWord(), su\_PCIWriteConfigWord()**

These functions are only used to support UHCI's Legacy Device support feature, which is not a required part of the UHCI specification. If there is no need to support it or if there is no PCI bus in your system, make them empty functions.

**su\_PCIReadConfigByte(), su\_PCIReadConfigDWord(), su\_PCIWriteConfigDWord()**

These functions are only used to support EHCI's capability function. If there is no EHCI PCI card on an x86 PC, ignore them.

## **6.3 DMA Transfer**

All smxUSBH external Host Controller Drivers do not use DMA transfer. For ISP116X, ISP1362, and ISP176X external USB host controllers, 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.

## Appendix A. Porting smxUSBH to Another OS

smxUSBH's porting layer maps onto smxBase services. Please see the smxBase User's Guide for the detailed information about how to port it to another OS. The following is the information that is smxUSBH-specific.

### A.1 uheap.h and uheap.c

These files handle smxUSBH memory requests. As shipped, they are ported for the SMX<sup>®</sup> RTOS. Refer to SU\_NEED\_NC\_MEMORY and SU\_USE\_C\_HEAP options described in Miscellaneous Settings.

#### **su\_HeapInit(), su\_HeapRelease()**

Allocate and initialize non-cacheable memory required to support USB functions. This memory may be pre-allocated from the OS's heap. This memory then works as a non-cacheable memory pool for smxUSBH.

#### **su\_AllocNC (uint iSizeInBytes, uint iAlign)**

Allocates *iSizeInBytes* bytes of memory from the pool previously initialized by `su_HeapInit()`. Because some memory used by the smxUSBH stack requires a specific byte alignment (for example the HCCA pointer in the OHCI should be 256-byte aligned), the `su_AllocNC()` API includes an alignment parameter, *iAlign*. Returns a void pointer.

#### **su\_FreeNC (void \* pPreviouslyAllocatedNCMemoryBlock)**

Frees a previously allocated memory block that was allocated using `su_AllocNC()`.

#### **su\_Alloc(uint iSizeInBytes)**

Allocates *iSizeInBytes* bytes of normal memory. Mapped to the C function `malloc()`.

#### **su\_Free(void \* pPreviouslyAllocatedMemoryBlock)**

Frees a previously allocated normal memory block. Mapped to the C function `free()`.

### A.2 Non-Multitasking Support

smxUSBH can work in a non-multitasking environment, such as DOS. smxBase already provides a DOS/NORTOS implementation, so use it. Besides that you also need:

1. In your main function or loop, before calling any smxUSBH API, call `su_Initialize()` first to initialize smxUSBH.
2. In your main function or loop, periodically call `su_CheckRoothubStatus()` to detect the USB device insert/remove event.

Please check the code in `\SMX\APP\NORTOS\usbhdemo.c` for the full demo of the USB host stack.



### A.3 Task Priority

When using a multitasking environment, normally there are three tasks involved for the USB.

smxUSBH has two built-in tasks. One task is the ISR task, which handles processing the host controller's interrupt. The ISR does nothing except wake up that task. Refer to `xxx_HostTaskInit()`. Another task is the Hub task, which enumerates the plugged-in device or cleans up the driver when a device is removed. Refer to the `su_HubStart()` subroutine.

Normally the application task calls the class driver API. The application task is the third task and should have a lower priority than the ISR task and Hub task.

The ISR task should be one of the highest priority tasks in your system.

The Hub task's priority should be higher than any application task that will call the class driver API.

## Appendix B. Memory Usage and Performance Summary

### B.1 Size

#### B.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-2 IAR v6.10</b>	<b>ARM IAR v5.20</b>	<b>ColdFire CodeWar- rior v7.1</b>	<b>X86 Microsoft VC++ v6.0</b>	<b>Blackfin VisualDSP</b>
Core (USBD, Porting Layer, and RootHub driver)	8 KB	10 KB	11 KB	10 KB	13 KB
Audio Device driver	7.5 KB	11 KB	12 KB	8 KB	N/A
CDC ACM (Modem) Device driver	2 KB	3 KB	4 KB	4 KB	2 KB
CDC ECM/NCM (Ethernet) Device driver	4 KB	8 KB	N/A	N/A	N/A
HID Mouse and Keyboard Device driver	2.5 KB	3.5 KB	4 KB	5 KB	2 KB
HID Generic Device driver	4.5 KB	6.5 KB	7 KB	6 KB	3 KB
Hub (External) driver	2 KB	3.0 KB	3.0 KB	3 KB	2 KB
Mass Storage Device driver	4 KB	6.5 KB	7.5 KB	6 KB	6.3 KB
Printer Device driver	1.5 KB	2 KB	3 KB	3 KB	2 KB
Serial Converter driver (FTDI)	1.5 KB	2.5 KB	4 KB	3.5 KB	1.5 KB
Serial Converter driver (Prolific)	2.5 KB	4 KB	4.5 KB	3 KB	1 KB
Serial Device driver	1.5 KB	2.5 KB	3 KB	4 KB	1 KB
Video driver	N/A	13.5 KB	N/A	N/A	N/A
V20K13 driver	N/A	N/A	N/A	5 KB	N/A
<b>Host Controller Drivers</b>					
EHCI	N/A	N/A	14 KB	14 KB	N/A
OHCI	6.5 KB	9 KB	11 KB	12 KB	N/A
UHCI	N/A	N/A	16 KB	14 KB	N/A
Analog Devices Blackfin BF52x	N/A	4 KB	N/A	N/A	5 KB

Freescale MCF522xx	N/A	N/A	5 KB	N/A	N/A
Maxim MAX3421	N/A	4 KB	N/A	N/A	N/A
NEC uPD720150	3 KB	N/A	N/A	N/A	N/A
NXP ISP116x	5.5 KB	6.5 KB	8 KB	7 KB	N/A
NXP ISP1362	6 KB	N/A	8.5 KB	7.5 KB	N/A
NXP ISP176x	9.5 KB	N/A	15 KB	13 KB	N/A
Synopsys	6 KB	N/A	N/A	N/A	N/A
TI LM3Sxxxx	4.5KB	N/A	N/A	N/A	N/A

AM1x, AT91, EP93xx, LPC24xx, LPC3180, LH7A404: See OHCI entry.

AM35x, i.MX31, LPC3131/41/51, MCF525x, MCF532x/7x, MCF5445x: See EHCI entry.

AT91SAM9M10/G45: See OHCI and EHCI entries.

Freescale Kxx: See MCF522xx entry.

STM32F105/7, STM32F205/7: See Synopsys entry.

For example, the total code size on AT91SAM9260 with mouse/keyboard/mass storage support, using IAR EWARM v5.20, is only 29KB (Core + OHCI + Mse/kbd + mass; 10 + 9 + 3.5 + 6.5).

On x86 with the same configuration it is only 33KB (10 + 12 + 5 + 6).

### B.1.2 Data Size (RAM Requirement)

The following is a table of RAM usage

<b>Component</b>	<b>Size</b>
Core (USBD, Porting Layer, and Roothub driver)	2 KB
Audio Device driver	10 KB
CDC ACM (Modem) Device driver	1 KB
CDC ECM/NCM (Ethernet) Device driver	12 KB
HID Mouse and Keyboard Device driver	0.5 KB
HID Generic Device driver	4 KB
Hub (External) driver	2 KB
Mass Storage Device driver	3 KB
Printer Device driver	2 KB
Serial Converter driver (FTDI)	2 KB
Serial Converter driver (Prolific)	2 KB

Serial Device driver	1 KB
Sierra Wireless Device driver	2 KB
Video camera driver	64 KB
V20K13 camera driver	12 KB
<b>Host Controller Drivers</b>	
EHCI	6 KB
OHCI	3 KB
UHCI	70 KB
Analog Devices Blackfin BF52x	1 KB
Freescale MCF522xx	1 KB
Maxim MAX3421	1 KB
NEC uPD720150	1 KB
NXP ISP116x	4 KB
NXP ISP1362	2 KB
NXP ISP176x	2 KB
Synopsys	2 KB
TI LM3Sxxxx	1 KB

Other controllers: See notes below Code Size table.

So the total RAM requirement for OHCI controller + mouse/keyboard/mass storage is only 9KB (Core + OHCI + mouse/keyboard + mass storage = 2 + 3 + .5 + 3).

## B.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.

## B.2.1 Mass Storage Performance

The following is a table of **raw data** read/write performance. The device driver reads/writes 4KB of data at a time from/to the USB flash disk, for an overall transfer of 30MB.

<b><u>Host Controller</u></b>	<b><u>Raw Reading</u></b>	<b><u>Raw Writing</u></b>
EHCI (NEC)	12684 KB/s	8320 KB/s
OHCI (NEC)	891 KB/s	832 KB/s
UHCI (VIA)	639 KB/s	611 KB/s
ISP116x (NXP)	352 KB/s	334 KB/s
ISP1362 (NXP)	621 KB/s	493 KB/s
ISP176x (NXP)	7425 KB/s	3214 KB/s
Blackfin (ADI)	10000 KB/s	8000 KB/s

The following is a table of measured **smxFile** read/write performance. The file operations read/write 4KB data at a time from/to a USB flash disk, for an overall file size of 20MB.

<b><u>Host Controller</u></b>	<b><u>File Reading</u></b>	<b><u>File Writing</u></b>
OHCI (NEC)	645 KB/s	425 KB/s
UHCI (VIA)	425 KB/s	339 KB/s
ISP116x (NXP)	330 KB/s	298 KB/s
ISP1362 (NXP)	402 KB/s	386 KB/s

The following is a table of measured **smxFS** read/write performance. The file operations read/write 4KB data at a time from/to a USB flash disk, for an overall file size of 20MB.

<b><u>Host Controller</u></b>	<b><u>File Reading</u></b>	<b><u>File Writing</u></b>
EHCI (NEC)	10556 KB/s	7787 KB/s
OHCI (NEC)	885 KB/s	817 KB/s
UHCI (VIA)	611 KB/s	590 KB/s
ISP116x (NXP)	336 KB/s	328 KB/s
ISP1362 (NXP)	591 KB/s	478 KB/s
ISP176x (NXP)	7023 KB/s	3072 KB/s

Blackfin (ADI)	9500 KB/s	7500 KB/s
----------------	-----------	-----------

\*The hardware environment for this testing is:

Celeron 300MHz CPU; 32MB 100M SDRAM; PC motherboard; Host Controller connects to System by 33MHz PCI bus.

\*\*Flash Disk is Lexar JumpDrive USB 2.0 512MB

\*\*\*CPU speed, SDRAM speed and size, External Memory Bus speed will affect the performance.

The following is a table of measured **smxFS** read/write performance for some ARM chips. The file operations read/write 4KB data at a time from/to a USB flash disk, for an overall file size of 20MB.

<u>Host Controller</u>	<u>File Reading</u>	<u>File Writing</u>
<u>Atmel SAM9260 OHCI Host Controller</u> (AHB is 105 MHz)	555 KB/s	505 KB/s
<u>Atmel SAM9261 OHCI Host Controller</u> (AHB is 60 MHz)	458 KB/s	414 KB/s
<u>Cirrus Logic EP9315 OHCI Host Controller</u>	575 KB/s	498 KB/s

The following is a table for performance testing of EHCI controller and USB hard disk.

<u>Host Controller</u>	<u>Raw Reading</u>	<u>Raw Writing</u>
<u>VIA EHCI Host Controller</u>	24966 KB/s	19784 KB/s

\*The hardware environment for this testing is:

Celeron 300MHz CPU; 32MB 100M SDRAM; PC motherboard; Host Controller connects to System by 33MHz PCI bus.

\*\* Disk is LACIE USB 2.0 40GB

## B.2.2 Serial Port Performance

The following is a table of serial port read/write performance. The device driver reads/writes 256 bytes of data at a time from/to the USB serial device (not connected to a real RS232 device).

<u>Host Controller</u>	<u>Data Reading</u>	<u>Data Writing</u>
OHCI (NEC)	124 KB/s	124 KB/s

## **Appendix C. Tested Host Controllers and Devices**

### **C.1 Host Controllers**

#### **C.1.1 EHCI Controllers**

- Atmel AT91SAM9M10G45 built-in EHCI controller
- Coldfire 5329 Built-in EHCI Controller
- Freescale i.MX31 built-in EHCI controller
- NEC D720101GJ PCI EHCI Controller
- NXP LPC313x/4x/5x built-in EHCI controller
- TI AM3517 built-in EHCI controller
- VIA VT6212L PCI EHCI Controller

#### **C.1.2 OHCI Controllers**

- ALi M5273 PCI OHCI Controller
- Atmel AT91RM9200 built-in OHCI controller
- Atmel AT91SAM9260/1/3 built-in OHCI controller
- Atmel AT91SAM9M10G45 built-in OHCI controller
- Cirrus Logic EDB9315 built-in OHCI controller
- Cyrix CS5530A (chipset for Geode GX1)
- NEC D720101GJ PCI OHCI Controller
- NXP ISP1561
- NXP LPC2xxx/3xxx built-in OHCI controller
- Samsung S3C2443 built-in OHCI controller
- Sharp LH7A404 built-in OHCI controller
- TI AM17x/18x/35x built-in OHCI controller

#### **C.1.3 UHCI Controllers**

- Intel 82371AB built-in UHCI Controller
- Intel 82801DB built-in UHCI Controller
- VIA VT6212L PCI UHCI Controller
- VIA VT8233 built-in UHCI Controller
- VIA VT82c686B built-in UHCI Controller

### **C.2 Audio Devices**

- Cyber Acoustics AC-850 USB Stereo Headset with Microphone
- Cyber Acoustics CA-2908 USB Speaker
- FD552 USB 3D Sound Adapter
- HP PM107A#ABA 2-Piece Virtual Surround Sound USB Speakers
- Logitech Premium USB Headset 350

- Logitech USB Desktop Microphone
- MIDIPLUS AKM322 MIDI Keyboard

### **C.3 CDC ACM (Modem) Devices**

- MultiTech MT5634ZBA-USB
- Sony Ericsson K800i Mobile Phone
- Sony Ericsson W950i Mobile Phone

### **C.4 HID Devices**

- Apple Pro Keyboard
- Apple Pro Mouse
- Dell USB Keyboard
- Dell USB Mouse
- Logitech Extreme 3D Pro Joystick
- Logitech USB Keyboard
- Logitech M-BD69 USB Mouse
- Microsoft Digital Media Keyboard
- Microsoft Intellimouse Optical USB Mouse

### **C.5 Mass Storage Devices**

Tested by Micro Digital:

- ADISK USB 1.1 32MB flash disk
- Aigo USB 1.1 64M flash disk
- Crucial 1G flash disk
- FPT-D US5B2H01 18-in-1 USB card reader/writer
- HP 1G flash disk
- IBM Portable Diskette Drive (floppy drive)
- IBM USB 2.0 CD-RW / DVD-ROM Combo II Drive (CD-ROM)
- Integral USB 2.0 2GB flash disk
- Kingston DataTraveler 1GB flash disk
- Kingston DataTraveler 100 2GB flash disk
- Kingston DataTraveler 16GB flash disk
- LACIE USB 2.0 40GB mobile hard drive
- Lexar Media JumpDrive Secure USB 2.0 512MB flash disk
- Memorex 2GB flash disk
- NCP XDrivePlus MMC/SD reader
- Newman USB 1.1 64MB flash disk
- PNY Attache USB 1.1 64MB flash disk
- PNY Attache (U3) 1GB
- PNY Attache 2GB
- PNY Attache 8GB



- PQI MMC/SD reader
- RedLeaf USB 2.0 256MB flash disk
- SanDisk Cruzer USB 2.0 256MB flash disk
- SanDisk Cruzer Micro 2GB flash disk
- SanDisk Cruzer Micro (U3) 2GB flash disk
- SanDisk Cruzer Micro (U3) 4GB flash disk
- SONY MICROVAULT USM256U2 USB 2.0 256MB flash disk
- Transcend JF V30 4GB flash disk

Tested by others:

- Edge DiskGO™ 1GB USB Flash Drive Enhanced for ReadyBoost™
- Edge DiskGO™ 2GB USB Flash Drive Enhanced for ReadyBoost™
- Imation 1GB Swivel USB Flash Drive
- Imation 2GB Swivel USB Flash Drive
- Integral 1GB USB Memory Stick
- MARKEM 1GB USB Memory Stick
- Memorex 1GB TravelDrive™ USB Flash Drive
- Memorex 2GB TravelDrive™ USB Flash Drive
- PNY 1GB Attache USB Flash Drive
- SanDisk 2GB Cruzer® Crossfire USB Flash Drive
- SanDisk 512MB Cruzer® Micro USB Flash Drive
- SanDisk 2GB Cruzer® Micro USB Flash Drive
- SanDisk 4GB Cruzer® Micro USB Flash Drive (U3 function not initialized)
- Sony 512MB Micro Vault Tiny USB Flash Drive
- Sony 2GB Micro Vault Tiny USB Flash Drive
- Sony 1GB Micro Vault Classic USB Flash Drive
- Sony 4GB Micro Vault Classic USB Flash Drive
- X Digital Media 1GB Itty Bit USB Flash Drive
- X Digital Media 1GB Poker Chip USB Flash Drive
- X Digital Media 2GB Itty Bit USB Flash Drive

## **C.6 Printer Devices**

- HP Deskjet 2400 series
- HP Deskjet 3500 series
- HP LaserJet 3015
- Panasonic UF 8000 USB facsimile

## **C.7 FTDI Serial Devices**

- QVS USB to RS-232 Cable (FTDI FT232B)
- USB-COM232-PLUS1 (FTDI FT232R)
- VS COM USB to RS-232 Cable (FTDI FT232B)

## **C.8 Windows Mobile 5 Serial Devices**

- Motorola Q Smartphone
- Palm Treo PDA phone

## **C.9 USB to Ethernet Devices**

- D-Link DUB-E100 Adapter (based on ASIX 88772)

## **C.10 Ralink RT250x WiFi Dongles**

- Belkin F5D7050 v3002
- D-Link EWUGRL2700
- Linksys WUSB54GC

## **C.11 Ralink RT2870 WiFi Dongles**

- AmbiCom M600N-USB
- BELKIN N Wireless F5D8053 v3001
- Buffalo WLI-UC-G300N
- D-Link DWA-140

## **C.12 Ralink RT3070 WiFi Dongles**

- AmbiCom WL150N-nUSB
- AmbiCom WL150N-USBx
- Asus USB-N13
- Ralink 3070 WS-WNU682N
- Linksys WUSB54GC ver.3

## **C.13 Ralink RT3572 WiFi Dongles**

- Linksys WUSB600N ver.2
- Linksys AE1000

## **C.14 Ralink RT5370 WiFi Dongles**

- Ralink RT5370 OEM dongle

## **C.15 MediaTek MT7601 WiFi Dongles**

- MediaTek MT7601 OEM dongle

## **C.16 MediaTek MT7612 WiFi Dongles**

- Netgear A6210 AC1200
- SparkLan MT7612 OEM dongle

## **C.17 Sierra Wireless Dongles**

- USB 598

## **C.18 Video Cameras**

- Creative Socialize 1080p
- Logitech C200
- Logitech HD Pro Webcam C920

## **C.19 CDC ECM/NCM (Ethernet) Devices**

- Raspberry Pi Zero W with customized kernel v5.4.80 that enabled `g_ncm` module
- `smxUSBD` Ethernet over USB (RNDIS + ECM/NCM) on LPC55S69 EVK

## ***Appendix D. Testing***

Unlike USB devices, which plug into USB Hosts, there is no protocol compliance testing available for USB Hosts. Instead testing of the host stack and class drivers is done by testing multiple devices of each class as listed in: Appendix C. Tested Host Controllers and Devices.

## **Appendix E. Specification Reference**

smxUSBH is based on the following specifications. USB specifications are available at [www.usb.org](http://www.usb.org) or at [www.usb.org/developers/docs](http://www.usb.org/developers/docs).

### **E.1 USB Specifications**

*Universal Serial Bus Specification, Revision 1.1*

*Universal Serial Bus Specification, Revision 2.0*

### **E.2 Host Controller Specifications**

*Open Host Controller Interface Specification for USB, Release: 1.0a*

*Universal Host Controller Interface (UHCI) Design Guide, Revision 1.1*

*Enhanced Host Controller Interface Specification for Universal Serial Bus, Revision 1.0*

*ISP1161 Full-speed Universal Serial Bus single-chip host and device controller, Rev. 02*

*ISP1161 Embedded Programming Guide, Rev 1.0*

*ISP1362 Single-chip Universal Serial Bus On-The-Go controller, Rev. 04*

*ISP1760 Hi-Speed Universal Serial Bus host controller for embedded applications, Rev. 02*

*ISP1761 Hi-Speed Universal Serial Bus On-The-Go controller, Rev. 02*

*ISP1763A Hi-Speed USB OTG controller, Rev. 01*

### **E.3 PCI Specification**

*PCI Local Bus Specification, Revision 2.1*

### **E.4 Audio Devices Specifications**

*Universal Serial Bus Device Class Definition for Audio Devices, Revision 1.0*

*Universal Serial Bus Device Class Definition for Terminal Types, Revision 1.0*

*Universal Serial Bus Device Class Definition for Audio Data Formats, Revision 1.0*

*Universal Serial Bus Device Class Definition for MIDI Devices, Revision 1.0*

### **E.5 Communication Devices Specifications**

*Universal Serial Bus Device Class Definition for Communication Devices, Revision 1.1*

*Universal Serial Bus Communications Class Subclass Specification for Ethernet Control Model Devices, Revision 1.2*

*Universal Serial Bus Communications Class Subclass Specification for Network Control Model Devices, Revision 1.0*

### **E.6 HID Specifications**

*Universal Serial Bus Device Class Definition for Human Interface Devices, Revision 1.11*

## **E.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*

*Universal Serial Bus Mass Storage Class Bulk/Control/Interrupt Transport, Revision 1.1*

*Universal Serial Bus Mass Storage Class UFI Command Specification, Revision 1.0*

*SCSI Primary Commands - 2 (SPC-2), Revision 20*

*SCSI Block Commands - 2 (SBC-2), Revision 14*

*SFF-8070i Specification for ATAPI Removable Re-writable Media Devices, Rev 1.2*

*SFF-8020i ATA Packet Interface for CD-ROMs, Revision 2.6*

*QIC 157, Revision D*

## **E.8 Printer Specifications**

*Universal Serial Bus Device Class Definition for Printing Devices, Revision 1.1*

## **E.9 Video Class Specifications**

*Universal Serial Bus Device Class Definition for Video Devices, Revision 1.1*

## **E.10 Referenced Documents**

*SMX Quick Start - [www.smxrtos.com/doc](http://www.smxrtos.com/doc)*

*smxWiFi User's Guide - [www.smxrtos.com/doc](http://www.smxrtos.com/doc)*

## ***Appendix F. Glossary***

API	Application Interface
BSP	Board Support Package
CDC	Communications Device Class
EHCI	Enhanced Host Controller Interface
EOI	End Of Interrupt
HCD	Host Controller Driver
HID	Human Interface Device
.inf	Setup Information file, in plain text format
ISOC	Isochronous
ISR	Interrupt Service Routine
LSR	Link Service Routine
MMU	Memory Management Unit
OHCI	Open Host Controller Interface
OS	Operating System
RTOS	Real Time Operating System
UHCI	Universal Host Controller Interface
USB	Universal Serial Bus
USBD	USB Driver
USBDI	USB Driver Interface