

the high performance USB stack

User Guide for USBX Device Stack

Express Logic, Inc.

858.613.6640

Toll Free 888.THREADX

FAX 858.521.4259

<http://www.expresslogic.com>

**©1999-2019 by Express Logic, Inc.**

All rights reserved. This document and the associated USBX software are the sole property of Express Logic, Inc. Each contains proprietary information of Express Logic, Inc. Reproduction or duplication by any means of any portion of this document without the prior written consent of Express Logic, Inc. is expressly forbidden.

Express Logic, Inc. reserves the right to make changes to the specifications described herein at any time and without notice in order to improve design or reliability of USBX. The information in this document has been carefully checked for accuracy; however, Express Logic, Inc. makes no warranty pertaining to the correctness of this document.

**Trademarks**

FileX, and ThreadX are registered trademarks of Express Logic, Inc., and USBX, NetX, *picokernel, preemption-threshold,* and *event-chaining* are trademarks of Express Logic, Inc. All other product and company names are trademarks or registered trademarks of their respective holders.

**Warranty Limitations**

Express Logic, Inc. makes no warranty of any kind that the USBX products will meet the USER’s requirements, or will operate in the manner specified by the USER, or that the operation of the USBX products will operate uninterrupted or error-free, or that any defects that may exist in the USBX products will be corrected after the warranty period. Express Logic, Inc. makes no warranties of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose, with respect to the USBX products. No oral or written information or advice given by Express Logic, Inc., its dealers, distributors, agents, or employees shall create any other warranty or in any way increase the scope of this warranty and licensee may not rely on any such information or advice.

Part Number: 000-1010

Revision 5.9

# Contents

[Contents 3](#_Toc24382168)

[About This Guide 7](#_Toc24382169)

[Chapter 1: Introduction to USBX 8](#_Toc24382170)

[USBX features 8](#_Toc24382171)

[Product Highlights 9](#_Toc24382172)

[Powerful Services of USBX 9](#_Toc24382173)

[Complete USB Device Framework Support 10](#_Toc24382174)

[Easy-To-Use APIs 10](#_Toc24382175)

[Chapter 2: USBX Installation 11](#_Toc24382176)

[Host Considerations 11](#_Toc24382177)

[Computer Type 11](#_Toc24382178)

[Download Interfaces 11](#_Toc24382179)

[Debugging Tools 11](#_Toc24382180)

[Required Hard Disk Space 11](#_Toc24382181)

[Target Considerations 11](#_Toc24382182)

[Configuration Options 14](#_Toc24382183)

[Source Code Tree 16](#_Toc24382184)

[Initialization of USBX resources 17](#_Toc24382185)

[Uninitialization of USBX resources 18](#_Toc24382186)

[Definition of USB Device Controller 18](#_Toc24382187)

[Troubleshooting 20](#_Toc24382188)

[USBX Version ID 20](#_Toc24382189)

[Chapter 3: Functional Components of USBX Device Stack 21](#_Toc24382190)

[Execution Overview: 21](#_Toc24382191)

[Initialization 21](#_Toc24382192)

[Application Interface Calls 22](#_Toc24382193)

[USB Device Stack APIs 22](#_Toc24382194)

[USB Device Class APIs 22](#_Toc24382195)

[Device Framework 22](#_Toc24382196)

[Definition of the Components of the Device Framework 22](#_Toc24382197)

[Definition of the Strings of the Device Framework 23](#_Toc24382198)

[Definition of the Languages Supported by the Device for each String 24](#_Toc24382199)

[VBUS Manager 25](#_Toc24382200)

[Chapter 4: Description of USBX Device Services 26](#_Toc24382201)

[ux\_device\_stack\_alternate\_setting\_get 27](#_Toc24382202)

[ux\_device\_stack\_alternate\_setting\_set 28](#_Toc24382203)

[ux\_device\_stack\_class\_register 29](#_Toc24382204)

[ux\_device\_stack\_class\_unregister 31](#_Toc24382205)

[ux\_device\_stack\_configuration\_get 33](#_Toc24382206)

[ux\_device\_stack\_configuration\_set 34](#_Toc24382207)

[ux\_device\_stack\_descriptor\_send 35](#_Toc24382208)

[ux\_device\_stack\_disconnect 36](#_Toc24382209)

[ux\_device\_stack\_endpoint\_stall 37](#_Toc24382210)

[ux\_device\_stack\_host\_wakeup 38](#_Toc24382211)

[ux\_device\_stack\_initialize 39](#_Toc24382212)

[ux\_device\_stack\_interface\_delete 43](#_Toc24382213)

[ux\_device\_stack\_interface\_get 44](#_Toc24382214)

[ux\_device\_stack\_interface\_set 45](#_Toc24382215)

[ux\_device\_stack\_interface\_start 46](#_Toc24382216)

[ux\_device\_stack\_transfer\_request 47](#_Toc24382217)

[ux\_device\_stack\_transfer\_abort 49](#_Toc24382218)

[ux\_device\_stack\_uninitialize 50](#_Toc24382219)

[Chapter 5: USBX Device Class Considerations 51](#_Toc24382220)

[Device Class registration 51](#_Toc24382221)

[USB Device Storage Class 53](#_Toc24382222)

[Multiple SCSI LUN 56](#_Toc24382223)

[USB Device CDC-ACM Class 58](#_Toc24382224)

[ux\_device\_class\_cdc\_acm\_ioctl 62](#_Toc24382225)

[ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_SET\_LINE\_CODING 64](#_Toc24382226)

[ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_GET\_LINE\_CODING 66](#_Toc24382227)

[ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_GET\_LINE\_STATE 68](#_Toc24382228)

[ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_SET\_LINE\_STATE 70](#_Toc24382229)

[ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_ABORT\_PIPE 71](#_Toc24382230)

[ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_TRANSMISSION\_START 72](#_Toc24382231)

[ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_TRANSMISSION\_STOP 74](#_Toc24382232)

[ux\_device\_class\_cdc\_acm\_read 75](#_Toc24382233)

[ux\_device\_class\_cdc\_acm\_write 77](#_Toc24382234)

[ux\_device\_class\_cdc\_acm\_write\_with\_callback 78](#_Toc24382235)

[USB Device CDC-ECM Class 79](#_Toc24382236)

[USB Device RNDIS Class 83](#_Toc24382237)

[USB Device DFU Class 87](#_Toc24382238)

[USB Device HID Class 93](#_Toc24382239)

[ux\_device\_class\_hid\_event\_set 95](#_Toc24382240)

[hid\_callback 96](#_Toc24382241)

[USB Device PIMA Class (PTP Responder) 97](#_Toc24382242)

[Initialization of the PIMA device class 100](#_Toc24382243)

[ux\_device\_class\_pima\_object\_add 104](#_Toc24382244)

[ux\_device\_class\_pima\_object\_number\_get 105](#_Toc24382245)

[ux\_device\_class\_pima\_object\_handles\_get 106](#_Toc24382246)

[ux\_device\_class\_pima\_object\_info\_get 108](#_Toc24382247)

[ux\_device\_class\_pima\_object\_data\_get 110](#_Toc24382248)

[ux\_device\_class\_pima\_object\_info\_send 113](#_Toc24382249)

[ux\_device\_class\_pima\_object\_data\_send 115](#_Toc24382250)

[ux\_device\_class\_pima\_object\_delete 117](#_Toc24382251)

[USB Device Audio Class 118](#_Toc24382252)

[ux\_device\_class\_audio\_read\_thread\_entry 123](#_Toc24382253)

[ux\_device\_class\_audio\_write\_thread\_entry 124](#_Toc24382254)

[ux\_device\_class\_audio\_stream\_get 125](#_Toc24382255)

[ux\_device\_class\_audio\_reception\_start 126](#_Toc24382256)

[ux\_device\_class\_audio\_sample\_read8 127](#_Toc24382257)

[ux\_device\_class\_audio\_sample\_read16 128](#_Toc24382258)

[ux\_device\_class\_audio\_sample\_read24 129](#_Toc24382259)

[ux\_device\_class\_audio\_sample\_read32 130](#_Toc24382260)

[ux\_device\_class\_audio\_read\_frame\_get 131](#_Toc24382261)

[ux\_device\_class\_audio\_read\_frame\_free 132](#_Toc24382262)

[ux\_device\_class\_audio\_transmission\_start 133](#_Toc24382263)

[ux\_device\_class\_audio\_frame\_write 134](#_Toc24382264)

[ux\_device\_class\_audio\_write\_frame\_get 135](#_Toc24382265)

[ux\_device\_class\_audio\_write\_frame\_commit 136](#_Toc24382266)

[ux\_device\_class\_audio10\_control\_process 137](#_Toc24382267)

[ux\_device\_class\_audio20\_control\_process 139](#_Toc24382268)

[Chapter 6: USBX DPUMP Class Considerations 141](#_Toc24382269)

[USBX DPUMP Device Class 142](#_Toc24382270)

[Chapter 7: USBX Pictbridge implementation 143](#_Toc24382271)

[Pictbridge client implementation 144](#_Toc24382272)

[ux\_pictbridge\_jobinfo\_object\_data\_read 148](#_Toc24382273)

[Pictbridge host implementation 149](#_Toc24382274)

[ux\_pictbridge\_application\_object\_data\_write 151](#_Toc24382275)

[Chapter 8: USBX OTG 152](#_Toc24382276)

[Index 155](#_Toc24382277)

# About This Guide

This guide provides comprehensive information about USBX, the high performance USB foundation software from Express Logic, Inc.

It is intended for the embedded real-time software developer. The developer should be familiar with standard real-time operating system functions, the USB specification, and the C programming language.

For technical information related to USB, see the USB specification and USB Class specifications that can be downloaded at http://www.USB.org/developers

**Organization**

**Chapter 1** contains an introduction to USBX

**Chapter 2** gives the basic steps to install and use USBX with your ThreadX application

**Chapter 3** is titled Functional Components of USBX Device Stack

**Chapter 4** is titled Description of USBX Device Services

**Chapter 5** is titled USBX Device Class Considerations

**Chapter 6** is titled USBX DPUMP Class Considerations

**Chapter 7** is titled USBX Pictbridge Implementation

**Chapter 8** is titled USBX OTG

# Chapter 1: Introduction to USBX

USBX is a full-featured USB stack for deeply embedded applications. This chapter introduces USBX, describing its applications and benefits.

## USBX features

USBX support the three existing USB specifications: 1.1, 2.0 and OTG. It is designed to be scalable and will accommodate simple USB topologies with only one connected device as well as complex topologies with multiple devices and cascading hubs. USBX supports all the data transfer types of the USB protocols: control, bulk, interrupt, and isochronous.

USBX supports both the host side and the device side. Each side is comprised of three layers:

* Controller layer
* Stack layer
* Class layer

The relationship between the USB layers is as follows:



## Product Highlights

Complete ThreadX processor support

No royalties

Complete ANSI C source code

Real-time performance

Responsive technical support

Multiple class support

Multiple class instances

Integration of classes with ThreadX, FileX and NetX

Support for USB devices with multiple configuration

Support for USB composite devices

Support for USB power management

Support for USB OTG

Export trace events for TraceX

## Powerful Services of USBX

.

### Complete USB Device Framework Support

USBX can support the most demanding USB devices, including multiple configurations, multiple interfaces, and multiple alternate settings.

### Easy-To-Use APIs

USBX provides the very best deeply embedded USB stack in a manner that is easy to understand and use. The USBX API makes the services intuitive and consistent. By using the provided USBX class APIs, the user application does not need to understand the complexity of the USB protocols.

# Chapter 2: USBX Installation

## Host Considerations

### Computer Type

Embedded development is usually performed on Windows PC or Unix host computers. After the application is compiled, linked, and located on the host, it is downloaded to the target hardware for execution.

### Download Interfaces

Usually the target download is done over an RS-232 serial interface, although parallel interfaces, USB, and Ethernet are becoming more popular. See the development tool documentation for available options.

### Debugging Tools

Debugging is done typically over the same link as the program image download. A variety of debuggers exist, ranging from small monitor programs running on the target through Background Debug Monitor (BDM) and In-Circuit Emulator (ICE) tools. Of course, the ICE tool provides the most robust debugging of actual target hardware.

### Required Hard Disk Space

The source code for USBX is delivered in ASCII format and requires approximately 500 KBytes of space on the host computer’s hard disk. Please review the supplied ***readme***\_***usbx.txt*** file for additional host system considerations and options.

### Target Considerations

USBX requires between 24 KBytes and 64 KBytes of Read Only Memory (ROM) on the target in host mode. The amount of memory required is dependent on the type of controller used and the USB classes linked to USBX. Another 32 KBytes of the target’s Random Access Memory (RAM) are required for USBX global data structures and memory pool. This memory pool can also be adjusted depending on the expected number of devices on the USB and the type of USB controller. The USBX device side requires roughly 10-12K of ROM depending on the type of device controller. The RAM memory usage depends on the type of class emulated by the device.

USBX also relies on ThreadX semaphores, mutexes, and threads for multiple thread protection, and I/O suspension and periodic processing for monitoring the USB bus topology.

#### Product Distribution

The exact content of the distribution CD depends on the target processor, development tools, and the USBX package. Following is a list of the important files common to most product distributions:

***readme\_usbx.txt*** This file contains specific information about the USBX port, including information about the target processor and the development tools.

***ux\_api.h*** This C header file contains all system equates, data structures, and service prototypes.

***ux\_port.h*** This C header file contains all development-tool-specific data definitions and structures.

***ux.lib*** This is the binary version of the USBX C library. It is distributed with the standard package.

***demo\_usbx.c*** The C file containing a simple USBX demo

All filenames are in lower-case. This naming convention makes it easier to convert the commands to Unix development platforms.

Installation of USBX is straightforward. The following general instructions apply to virtually any installation. However, the ***readme\_usbx\_generic.txt*** file should be examined for changes specific to the actual development tool environment.

Step 1: Backup the USBX distribution disk and store it in a safe location.

Step 2: Use the same directory in which you previously installed ThreadX on the host hard drive. All USBX names are unique and will not interfere with the previous USBX installation.

Step 3: Add a call to **ux\_system\_initialize** at or near the beginning of **tx\_application\_define.** This is where the USBX resources are initialized.

Step 4: Add a call to **ux\_device\_stack\_initialize.**

Step 5: Add one or more calls to initialize the required USBX classes (either host and/or devices classes)

Step 6: Add one or more calls to initialize the device controller available in the system.

Step 7 It may be required to modify the tx\_low\_level\_initialize.c file to add low level hardware initialization and interrupt vector routing. This is specific to the hardware platform and will not be discussed here.

Step 8: Compile application source code and link with the USBX and ThreadX run time libraries (FileX and/or Netx may also be required if the USB storage class and/or USB network classes are to be compiled in), ux.a (or ux.lib) and tx.a (or tx.lib). The resulting can be downloaded to the target and executed!

## Configuration Options

There are several configuration options for building the USBX library. All options are located in the ***ux\_port.h***.

The list below details each configuration option. Additional development tool options are described in the ***readme\_usbx.txt*** file supplied on the distribution disk:

UX\_PERIODIC\_RATE

This value represents how many ticks per seconds for a specific hardware platform. The default is 1000 indicating 1 tick per millisecond.

UX\_THREAD\_STACK\_SIZE

This value is the size of the stack in bytes for the USBX threads. It can be typically 1024 or 2048 bytes depending on the processor used and the host controller.

UX\_THREAD\_PRIORITY\_ENUM

This is the ThreadX priority value for the USBX enumeration threads that monitors the bus topology.

UX\_THREAD\_PRIORITY\_CLASS

This is the ThreadX priority value for the standard USBX threads.

UX\_THREAD\_PRIORITY\_KEYBOARD

This is the ThreadX priority value for the USBX HID keyboard class.

UX\_THREAD\_PRIORITY\_DCD

This is the ThreadX priority value for the device controller thread.

UX\_NO\_TIME\_SLICE

If defined to 1, the ThreadX target port does not use time slice.

UX\_MAX\_SLAVE\_LUN

This value represents the current number of SCSI logical units represented in the device storage class driver.

UX\_SLAVE\_REQUEST\_CONTROL\_MAX\_LENGTH

This value represents the maximum number of bytes received on a control endpoint in the device stack. The default is 256 bytes but can be reduced in memory constraint environments

UX\_SLAVE\_REQUEST\_DATA\_MAX\_LENGTH

This value represents the maximum number of bytes received on a bulk endpoint in the device stack. The default is 4096 bytes but can be reduced in memory constraint environments.

## Source Code Tree

The USBX files are provided in several directories.



In order to make the files recognizable by their names, the following convention has been adopted:

|  |  |
| --- | --- |
| File Suffix Name | File description |
| ux\_host\_stack | usbx host stack core files |
| ux\_host\_class | usbx host stack classes files |
| ux\_hcd | usbx host stack controller driver files |
| ux\_device\_stack | usbx device stack core files |
| ux\_device\_class | usbx device stack classes files |
| ux\_dcd | usbx device stack controller driver files |
| ux\_otg | usbx otg controller driver related files |
| ux\_pictbridge | usbx pictbridge files |
| ux\_utility | usbx utility functions |
| demo\_usbx | demonstration files for USBX |

## Initialization of USBX resources

USBX has its own memory manager. The memory needs to be allocated to USBX before the host or device side of USBX is initialized. USBX memory manager can accommodate systems where memory can be cached.

The following function initializes USBX memory resources with 128K of regular memory and no separate pool for cache safe memory:

/\* Initialize USBX Memory \*/

**ux\_system\_initialize**(memory\_pointer,(128\*1024),UX\_NULL,0);

The prototype for the ux\_system\_initialize is as follows:

UINT **ux\_system\_initialize**(VOID \*regular\_memory\_pool\_start,

ULONG regular\_memory\_size,

VOID \*cache\_safe\_memory\_pool\_start,

ULONG cache\_safe\_memory\_size);

Input parameters:

VOID \*regular\_memory\_pool\_start Beginning of the regular memory pool

ULONG regular\_memory\_size Size of the regular memory pool

VOID \*cache\_safe\_memory\_pool\_start Beginning of the cache safe memory  
 pool

ULONG cache\_safe\_memory\_size Size of the cache safe memory pool

Not all systems require the definition of cache safe memory. In such a system, the values passed during the initialization for the memory pointer will be set to UX\_NULL and the size of the pool to 0. USBX will then use the regular memory pool in lieu of the cache safe pool.

In a system where the regular memory is not cache safe and a controller requires to perform DMA memory it is necessary to define a memory pool in a cache safe zone.

## Uninitialization of USBX resources

USBX can be terminated by releasing its resources. Prior to terminating usbx, all classes and controller resources need to be terminated properly. The following function uninitializes USBX memory ressources :

/\* Unitialize USBX Resources \*/

**ux\_system\_uninitialize**();

The prototype for the ux\_system\_initialize is as follows:

UINT **ux\_system\_uninitialize**(VOID);

## Definition of USB Device Controller

Only one USB device controller can be defined at any time to operate in device mode. The application initialization file should contain this definition.

The following line performs the definition of a generic usbcontroller:

**ux\_dcd\_controller\_initialize**(0x7BB00000, 0, 0xB7A00000);

The USB device initialization has the following prototype:

**UINT ux\_dcd\_controller\_initialize**(ULONG dcd\_io, ULONG dcd\_irq,

ULONG dcd\_vbus\_address);

with the following parameters:

ULONG dcd\_io Address of the controller IO

ULONG dcd\_irq Interrupt used by the controller

ULONG dcd\_vbus\_address Address of the VBUS GPIO

The following example is the initialization of USBX in device mode with the storage device class and a generic controller controller:

/\* Initialize USBX Memory \*/

**ux\_system\_initialize**(memory\_pointer,(128\*1024), 0, 0);

/\* The code below is required for installing the device portion of USBX \*/

status = **ux\_device\_stack\_initialize**(&device\_framework\_high\_speed,

DEVICE\_FRAMEWORK\_LENGTH\_HIGH\_SPEED,

&device\_framework\_full\_speed, DEVICE\_FRAMEWORK\_LENGTH\_FULL\_SPEED,

&string\_framework, STRING\_FRAMEWORK\_LENGTH,

&language\_id\_framework, LANGUAGE\_ID\_FRAMEWORK\_LENGTH,

UX\_NULL);

/\* If status equals UX\_SUCCESS, installation was successful. \*/

/\* Store the number of LUN in this device storage instance: single LUN. \*/

storage\_parameter.ux\_slave\_class\_storage\_parameter\_number\_lun = 1;

/\* Initialize the storage class parameters for reading/writing to the Flash Disk. \*/

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_last\_lba = 0x1e6bfe;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_block\_length = 512;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_type = 0;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_removable\_flag = 0x80;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_read =

tx\_demo\_thread\_flash\_media\_read;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_write =

tx\_demo\_thread\_flash\_media\_write;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_status =

tx\_demo\_thread\_flash\_media\_status;

/\* Initialize the device storage class. The class is connected with interface 0 \*/

status = **ux\_device\_stack\_class\_register**(ux\_system\_slave\_class\_storage\_name,

ux\_device\_class\_storage\_entry,

ux\_device\_class\_storage\_thread,0,

(VOID \*)&storage\_parameter);

/\* Register the device controllers available in this system \*/

status = **ux\_dcd\_controller\_initialize**(0x7BB00000, 0, 0xB7A00000);

/\* If status equals UX\_SUCCESS, registration was successful. \*/

## Troubleshooting

USBX is delivered with a demonstration file and a simulation environment. It is always a good idea to get the demonstration platform running first—either on the target hardware or a specific demonstration platform.

## USBX Version ID

The current version of USBX is available both to the user and the application software during run-time.

The programmer can obtain the USBX version from examination of the ***readme\_usbx.txt*** file. In addition, this file also contains a version history of the corresponding port. Application software can obtain the USBX version by examining the global string ***\_ux\_version\_id***, which is defined in ***ux\_port.h***.

# Chapter 3: Functional Components of USBX Device Stack

This chapter contains a description of the high performance USBX embedded USB device stack from a functional perspective.

## Execution Overview:

USBX for the device is composed of several components:

Initialization

Application interface calls

Device Classes

USB Device Stack

Device controller

VBUS manager

The following diagram illustrates the USBX Device stack:



### Initialization

In order to activate USBX, the function *ux\_system\_initialize* must be called. This function initializes the memory resources of USBX.

In order to activate USBX device facilities, the function *ux\_device\_stack\_initialize* must be called. This function will in turn initialize all the resources used by the USBX device stack such as ThreadX threads, mutexes, and semaphores.

It is up to the application initialization to activate the USB device controller and one or more USB classes. Contrary to the USB host side, the device side can have only one USB controller driver running at any time. When the classes have been registered to the stack and the device controller(s) initialization function has been called, the bus is active and the stack will reply to bus reset and host enumeration commands.

### Application Interface Calls

There are two levels of APIs in USBX:

USB Device Stack APIs

USB Device Class APIs

Normally, a USBX application should not have to call any of the USB device stack APIs. Most applications will only access the USB Class APIs.

### USB Device Stack APIs

The device stack APIs are responsible for the registration of USBX device components such as classes and the device framework.

### USB Device Class APIs

The Class APIs are very specific to each USB class. Most of the common APIs for USB classes provided services such as opening/closing a device and reading from and writing to a device. The APIs are similar in nature to the host side.

## Device Framework

The USB device side is responsible for the definition of the device framework. The device framework is divided into three categories, as described in the following sections.

### Definition of the Components of the Device Framework

The definition of each component of the device framework is related to the nature of the device and the resources utilized by the device. Following are the main categories.

* Device Descriptor
* Configuration Descriptor
* Interface Descriptor
* Endpoint Descriptor

USBX supports device component definition for both high and full speed (low speed being treated the same way as full speed). This allows the device to operate differently when connected to a high speed or full speed host. The typical differences are the size of each endpoint and the power consumed by the device.

The definition of the device component takes the form of a byte string that follows the USB specification. The definition is contiguous and the order in which the framework is represented in memory will be the same as the one returned to the host during enumeration.

Following is an example of a device framework for a high speed USB Flash Disk.

#define DEVICE\_FRAMEWORK\_LENGTH\_HIGH\_SPEED 60

UCHAR device\_framework\_high\_speed[] = {

/\* Device descriptor \*/

0x12, 0x01, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40,

0x0a, 0x07, 0x25, 0x40, 0x01, 0x00, 0x01, 0x02,

0x03, 0x01,

/\* Device qualifier descriptor \*/

0x0a, 0x06, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40,

0x01, 0x00,

/\* Configuration descriptor \*/

0x09, 0x02, 0x20, 0x00, 0x01, 0x01, 0x00, 0xc0,

0x32,

/\* Interface descriptor \*/

0x09, 0x04, 0x00, 0x00, 0x02, 0x08, 0x06, 0x50,

0x00,

/\* Endpoint descriptor (Bulk Out) \*/

0x07, 0x05, 0x01, 0x02, 0x00, 0x02, 0x00,

/\* Endpoint descriptor (Bulk In) \*/

0x07, 0x05, 0x82, 0x02, 0x00, 0x02, 0x00

};

### Definition of the Strings of the Device Framework

Strings are optional in a device. Their purpose is to let the USB host know about the manufacturer of the device, the product name, and the revision number through Unicode strings.

The main strings are indexes embedded in the device descriptors. Additional strings indexes can be embedded into individual interfaces.

Assuming the device framework above has three string indexes embedded into the device descriptor, the string framework definition could look like this:

/\* String Device Framework:

Byte 0 and 1: Word containing the language ID: 0x0904 for US

Byte 2 : Byte containing the index of the descriptor

Byte 3 : Byte containing the length of the descriptor string

\*/

#define STRING\_FRAMEWORK\_LENGTH 38

UCHAR string\_framework[] = {

/\* Manufacturer string descriptor: Index 1 \*/

0x09, 0x04, 0x01, 0x0c,

0x45, 0x78, 0x70, 0x72, 0x65, 0x73, 0x20, 0x4c,

0x6f, 0x67, 0x69, 0x63,

/\* Product string descriptor: Index 2 \*/

0x09, 0x04, 0x02, 0x0c,

0x4D, 0x4C, 0x36, 0x39, 0x36, 0x35, 0x30, 0x30,

0x20, 0x53, 0x44, 0x4B,

/\* Serial Number string descriptor: Index 3 \*/

0x09, 0x04, 0x03, 0x04,

0x30, 0x30, 0x30, 0x31

};

If different strings have to be used for each speed, different indexes must be used as the indexes are speed agnostic.

The encoding of the string is UNICODE-based. For more information on the UNICODE encoding standard refer to the following publication:

The Unicode Standard, Worldwide Character Encoding, Version 1., Volumes 1 and 2, The Unicode Consortium, Addison-Wesley Publishing Company, Reading MA.

### Definition of the Languages Supported by the Device for each String

USBX has the ability to support multiple languages although English is the default. The definition of each language for the string descriptors is in the form of an array of languages definition defined as follows:

#define LANGUAGE\_ID\_FRAMEWORK\_LENGTH 2

UCHAR language\_id\_framework[] = {

/\* English. \*/

0x09, 0x04

};

To support additional languages, simply add the language code double-byte definition after the default English code. The language code has been defined by Microsoft in the document:

Developing International Software for Windows 95 and Windows NT, Nadine Kano, Microsoft Press, Redmond WA

## VBUS Manager

In most USB device designs, VBUS is not part of the USB Device core but rather connected to an external GPIO, which monitors the line signal.

As a result, VBUS has to be managed separately from the device controller driver.

It is up to the application to provide the device controller with the address of the VBUS IO. VBUS must be initialized prior to the device controller initialization.

Depending on the platform specification for monitoring VBUS, it is possible to let the controller driver handle VBUS signals after the VBUS IO is initialized or if this is not possible, the application has to provide the code for handling VBUS.

If the application wishes to handle VBUS by itself, its only requirement is to call the function  
  
 ux\_device\_stack\_disconnect()  
  
when it detects that a device has been extracted. It is not necessary to inform the controller when a device is inserted because the controller will wake up when the BUS RESET assert/deassert signal is detected.

# Chapter 4: Description of USBX Device Services

### ux\_device\_stack\_alternate\_setting\_get

Get current alternate setting for an interface value

**Prototype**

UINT **ux\_device\_stack\_alternate\_setting\_get**(ULONG interface\_value)

**Description**

This function is used by the USB host to obtain the current alternate setting for a specific interface value. It is called by the controller driver when a GET\_INTERFACE request is received.

**Input Parameter**

**interface\_value** Interface value for which the current alternate setting is queried.

**Return Values**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | The data transfer was completed. |
| **UX\_ERROR** | (0xFF) | Wrong interface value. |

**Example**

ULONG interface\_value;

UINT status;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_alternate\_setting\_get**(interface\_value);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_alternate\_setting\_set

Set current alternate setting for an interface value

**Prototype**

UINT **ux\_device\_stack\_alternate\_setting\_set**(ULONG interface\_value,

ULONG alternate\_setting\_value)

**Description**

This function is used by the USB host to set the current alternate setting for a specific interface value. It is called by the controller driver when a SET\_INTERFACE request is received. When the SET\_INTERFACE is completed, the values of the alternate settings are applied to the class.

The device stack will issue a UX\_SLAVE\_CLASS\_COMMAND\_CHANGE to the class that owns this interface to reflect the change of alternate setting.

**Parameters**

**interface\_value** Interface value for which the current alternate setting is set.

**alternate\_setting\_value** The new alternate setting value.

**Return Values**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | The data transfer was completed. |
| **UX\_INTERFACE\_HANDLE\_UNKNOWN** | (0x52) | No interface attached. |
| **UX\_ERROR** | (0xFF) | Wrong interface value. |

**Example**

ULONG interface\_value;

ULONG alternate\_setting\_value;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_alternate\_setting\_set**(interface\_value,

alternate\_setting\_value);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_class\_register

Register a new USB device class

**Prototype**

UINT **ux\_device\_stack\_class\_register**(UCHAR \*class\_name,

UINT (\*class\_entry\_function)(struct UX\_SLAVE\_CLASS\_COMMAND\_STRUCT \*),

ULONG configuration\_number,

ULONG interface\_number,

VOID \*parameter)

**Description**

This function is used by the application to register a new USB device class. This registration starts a class container and not an instance of the class. A class should have an active thread and be attached to a specific interface.

Some classes expect a parameter or parameter list. For instance, the device storage class would expect the geometry of the storage device it is trying to emulate. The parameter field is therefore dependent on the class requirement and can be a value or a pointer to a structure filled with the class values.

Note: The C string of class\_name must be NULL-terminated and the length of it (without the NULL-terminator itself) must be no larger than UX\_MAX\_CLASS\_NAME\_LENGTH.

**Parameters**

**class\_name** Class Name

**class\_entry\_function** The entry function of the class.

**configuration\_number** The configuration number this class is attached to.

**interface\_number** The interface number this class is attached to.

**parameter** A pointer to a class specific parameter list.

**Return Values**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | The class was unregistered |
| **UX\_NO\_CLASS\_MATCH** | (0x57) | Class unknown |

**Example**

UINT status;

/\* The following example illustrates this service. \*/

/\* Initialize the device storage class. The class is connected with

interface 1 \*/

status = **ux\_device\_stack\_class\_register**(\_ux\_system\_slave\_class\_storage\_name,

ux\_device\_class\_storage\_entry,

1, 1, (VOID \*)&parameter);

### ux\_device\_stack\_class\_unregister

Unregister a USB device class

**Prototype**

UINT **ux\_device\_stack\_class\_unregister**(UCHAR \*class\_name,

UINT (\*class\_entry\_function)(struct UX\_SLAVE\_CLASS\_COMMAND\_STRUCT \*))

**Description**

This function is used by the application to unregister a USB device class.

Note: The C string of class\_name must be NULL-terminated and the length of it (without the NULL-terminator itself) must be no larger than UX\_MAX\_CLASS\_NAME\_LENGTH.

**Parameters**

**class\_name** Class Name

**class\_entry\_function** The entry function of the class.

**Return Values**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | The class was registered. |
| **UX\_MEMORY\_INSUFFICIENT** | (0x52) | Not enough memory. |
| **UX\_THREAD\_ERROR** | (0xFF) | Cannot create a class thread. |

**Example**

/\* The following example illustrates this service. \*/

/\* Unitialize the device storage class. \*/

status = **ux\_device\_stack\_class\_unregister**(\_ux\_system\_slave\_class\_storage\_name,

ux\_device\_class\_storage\_entry);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_configuration\_get

Get the current configuration

**Prototype**

UINT **ux\_device\_stack\_configuration\_get**(VOID)

**Description**

This function is used by the host to obtain the current configuration running in the device.

**Input Parameter**

**None**

**Return Value**

**UX\_SUCCESS** (0x00) The data transfer was completed.

**Example**

UINT status;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_configuration\_get**();

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_configuration\_set

Set the current configuration

**Prototype**

UINT **ux\_device\_stack\_configuration\_set**(ULONG configuration\_value)

**Description**

This function is used by the host to set the current configuration running in the device. Upon reception of this command, the USB device stack will activate the alternate setting 0 of each interface connected to this configuration.

**Input Parameter**

**configuration\_value** The configuration value selected by the host.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | The data transfer was completed. |

**Example**

ULONG configuration\_value;

UINT status;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_configuration\_set**(configuration\_value);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_descriptor\_send

Send a descriptor to the host

**Prototype**

UINT **ux\_device\_stack\_descriptor\_send**(ULONG descriptor\_type,

ULONG request\_index, ULONG host\_length)

**Description**

This function is used by the device side to return a descriptor to the host. This descriptor can be a device descriptor, a configuration descriptor or a string descriptor.

**Parameters**

**descriptor\_type** The nature of the descriptor:

UX\_DEVICE\_DESCRIPTOR\_ITEM

UX\_CONFIGURATION\_DESCRIPTOR\_ITEM

UX\_STRING\_DESCRIPTOR\_ITEM

UX\_DEVICE\_QUALIFIER\_DESCRIPTOR\_ITEM

UX\_OTHER\_SPEED\_DESCRIPTOR\_ITEM

**request\_index** The index of the descriptor.

**host\_length** The length required by the host.

**Return Values**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | The data transfer was completed. |
| **UX\_ERROR** | (0xFF) | The transfer was not completed. |

**Example**

ULONG descriptor\_type;

ULONG request\_index;

ULONG host\_length;

UINT status;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_descriptor\_send**(descriptor\_type,

request\_index, host\_length);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_disconnect

Disconnect device stack

**Prototype**

UINT **ux\_device\_stack\_disconnect**(VOID)

**Description**

The VBUS manager calls this function when there is a device disconnection. The device stack will inform all classes registered to this device and will thereafter release all the device resources.

**Input Parameter**

**None**

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | The device was disconnected. |

**Example**

UINT status;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_disconnect**();

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_endpoint\_stall

Request endpoint Stall condition

**Prototype**

UINT **ux\_device\_stack\_endpoint\_stall**(UX\_SLAVE\_ENDPOINT \*endpoint)

**Description**

This function is called by the USB device class when an endpoint should return a Stall condition to the host.

**Input Parameter**

**endpoint** The endpoint on which the Stall condition is requested.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |

**Example**

UINT status;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_endpoint\_stall**(endpoint);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_host\_wakeup

Wake up the host

**Prototype**

UINT **ux\_device\_stack\_host\_wakeup**(VOID)

**Description**

This function is called when the device wants to wake up the host. This command is only valid when the device is in suspend mode. It is up to the device application to decide when it wants to wake up the USB host. For instance, a USB modem can wake up a host when it detects a RING signal on the telephone line.

**Input Parameter**

**None**

**Return values**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | The call was successful. |
| **UX\_ERROR** | (0xFF) | The call failed (the device was probably not in the suspended mode). |

**Example**

UINT status;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_host\_wakeup**();

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_initialize

Initialize USB device stack

**Prototype**

UINT **ux\_device\_stack\_initialize**(UCHAR\_PTR device\_framework\_high\_speed,

ULONG device\_framework\_length\_high\_speed,

UCHAR\_PTR device\_framework\_full\_speed,

ULONG device\_framework\_length\_full\_speed,

UCHAR\_PTR string\_framework,

ULONG string\_framework\_length,

UCHAR\_PTR language\_id\_framework,

ULONG language\_id\_framework\_length),

UINT (\*ux\_system\_slave\_change\_function)(ULONG)))

**Description**

This function is called by the application to initialize the USB device stack. It does not initialize any classes or any controllers. This should be done with separate function calls. This call mainly provides the stack with the device framework for the USB function. It supports both high and full speeds with the possibility to have completely separate device framework for each speed. String framework and multiple languages are supported.

**Parameters**

**device\_framework\_high\_speed** Pointer to the high speed framework.

**device\_framework\_length\_high\_speed** Length of the high speed framework.

**device\_framework\_full\_speed** Pointer to the full speed framework.

**device\_framework\_length\_full\_speed** Length of the full speed framework.

**string\_framework** Pointer to string framework.

**string\_framework\_length** Length of string framework.

**language\_id\_framework** Pointer to string language framework.

**language\_id\_framework\_length** Length of the string language framework.

**ux\_system\_slave\_change\_function** Function to be called when the device state changes.

**Return Values**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_MEMORY\_INSUFFICIENT** | (0x12) | Not enough memory to initialize the stack. |

**Example**

/\* Example of a device framework \*/

#define DEVICE\_FRAMEWORK\_LENGTH\_FULL\_SPEED 50

UCHAR device\_framework\_full\_speed[] = {

/\* Device descriptor \*/

0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x08,

0xec, 0x08, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x01,

/\* Configuration descriptor \*/

0x09, 0x02, 0x20, 0x00, 0x01, 0x01, 0x00, 0xc0,

0x32,

/\* Interface descriptor \*/

0x09, 0x04, 0x00, 0x00, 0x02, 0x08, 0x06, 0x50,

0x00,

/\* Endpoint descriptor (Bulk Out) \*/

0x07, 0x05, 0x01, 0x02, 0x40, 0x00, 0x00,

/\* Endpoint descriptor (Bulk In) \*/

0x07, 0x05, 0x82, 0x02, 0x40, 0x00, 0x00

};

#define DEVICE\_FRAMEWORK\_LENGTH\_HIGH\_SPEED 60

UCHAR device\_framework\_high\_speed[] = {

/\* Device descriptor \*/

0x12, 0x01, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40,

0x0a, 0x07, 0x25, 0x40, 0x01, 0x00, 0x01, 0x02,

0x03, 0x01,

/\* Device qualifier descriptor \*/

0x0a, 0x06, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40,

0x01, 0x00,

/\* Configuration descriptor \*/

0x09, 0x02, 0x20, 0x00, 0x01, 0x01, 0x00, 0xc0,

0x32,

/\* Interface descriptor \*/

0x09, 0x04, 0x00, 0x00, 0x02, 0x08, 0x06, 0x50,

0x00,

/\* Endpoint descriptor (Bulk Out) \*/

0x07, 0x05, 0x01, 0x02, 0x00, 0x02, 0x00,

/\* Endpoint descriptor (Bulk In) \*/

0x07, 0x05, 0x82, 0x02, 0x00, 0x02, 0x00

};

/\* String Device Framework:

Byte 0 and 1: Word containing the language ID: 0x0904 for US

Byte 2 : Byte containing the index of the descriptor

Byte 3 : Byte containing the length of the descriptor string

\*/

#define STRING\_FRAMEWORK\_LENGTH 38

UCHAR string\_framework[] = {

/\* Manufacturer string descriptor: Index 1 \*/

0x09, 0x04, 0x01, 0x0c,

0x45, 0x78, 0x70, 0x72,0x65, 0x73, 0x20, 0x4c,

0x6f, 0x67, 0x69, 0x63,

/\* Product string descriptor: Index 2 \*/

0x09, 0x04, 0x02, 0x0c,

0x4D, 0x4C, 0x36, 0x39, 0x36, 0x35, 0x30, 0x30,

0x20, 0x53, 0x44, 0x4B,

/\* Serial Number string descriptor: Index 3 \*/

0x09, 0x04, 0x03, 0x04,

0x30, 0x30, 0x30, 0x31

};

/\* Multiple languages are supported on the device, to add

a language besides English, the Unicode language code must

be appended to the language\_id\_framework array and the length

adjusted accordingly. \*/

#define LANGUAGE\_ID\_FRAMEWORK\_LENGTH 2

UCHAR language\_id\_framework[] = {

/\* English. \*/

0x09, 0x04

};

The application can request a call back when the controller changes its state. The two main states for the controller are:

UX\_DEVICE\_SUSPENDED

UX\_DEVICE\_RESUMED

If the application does not need Suspend/Resume signals, it would supply a UX\_NULL function.

UINT status;

/\* The code below is required for installing the device portion of USBX. There is no call back for device status change in this example. \*/

status = **ux\_device\_stack\_initialize**(&device\_framework\_high\_speed,

DEVICE\_FRAMEWORK\_LENGTH\_HIGH\_SPEED,

&device\_framework\_full\_speed,

DEVICE\_FRAMEWORK\_LENGTH\_FULL\_SPEED,

&string\_framework,

STRING\_FRAMEWORK\_LENGTH,

&language\_id\_framework,

LANGUAGE\_ID\_FRAMEWORK\_LENGTH,

UX\_NULL);

/\* If status equals UX\_SUCCESS, initialization was successful. \*/

### ux\_device\_stack\_interface\_delete

Delete a stack interface

**Prototype**

UINT **ux\_device\_stack\_interface\_delete**(UX\_SLAVE\_INTERFACE \*interface)

**Description**

This function is called when an interface should be removed. An interface is either removed when a device is extracted, or following a bus reset, or when there is a new alternate setting.

**Input Parameter**

**interface** Pointer to the interface to remove.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |

**Example**

UINT status;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_interface\_delete**(interface);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_interface\_get

Get the current interface value

**Prototype**

UINT **ux\_device\_stack\_interface\_get**(UINT interface\_value)

**Description**

This function is called when the host queries the current interface. The device returns the current interface value.

Note: this function is deprecated. ux\_device\_stack\_alternate\_setting\_get should be used instead.

**Input Parameter**

**interface\_value** Interface value to return.

**Return Values**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | No interface exists. |

**Example**

ULONG interface\_value;

UINT status;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_interface\_get**(interface\_value);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_interface\_set

Change the alternate setting of the interface

**Prototype**

UINT ux\_device\_stack\_interface\_set(UCHAR\_PTR device\_framework,

ULONG device\_framework\_length,

ULONG alternate\_setting\_value)

**Description**

This function is called when the host requests a change of the alternate setting for the interface.

**Parameters**

**device\_framework** Address of the device framework for this interface.

**device\_framework\_length** Length of the device framework.

**alternate\_setting\_value** Alternate setting value to be used by this interface.

**Return Values**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | No interface exists. |

**Example**

UCHAR\_PTR device\_framework

ULONG device\_framework\_length;

ULONG alternate\_setting\_value;

UINT status;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_interface\_set**(device\_framework,

device\_framework\_length,

alternate\_setting\_value);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_interface\_start

Start search for a class to own an interface instance

**Prototype**

UINT **ux\_device\_stack\_interface\_start**(UX\_SLAVE\_INTERFACE \*interface)

**Description**

This function is called when an interface has been selected by the host and the device stack needs to search for a device class to own this interface instance.

**Input Parameter**

**interface** Pointer to the interface created.

**Return Values**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_NO\_CLASS\_MATCH** | (0x57) | No class exists for this interface. |

**Example**

UINT status;

/\* The following example illustrates this service. \*/

status = **ux\_device\_stack\_interface\_start**(interface);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_transfer\_request

Request to transfer data to the host

**Prototype**

UINT **ux\_device\_stack\_transfer\_request**(UX\_SLAVE\_TRANSFER \*transfer\_request,

ULONG slave\_length,

ULONG host\_length)

**Description**

This function is called when a class or the stack wants to transfer data to the host. The host always polls the device but the device can prepare data in advance.

**Parameters**

**transfer\_request** Pointer to the transfer request.

**slave\_length** Length the device wants to return.

**host\_length** Length the host has requested.

**Return Values**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Transport error. |

**Example**

UINT status;

/\* The following example illustrates how to transfer more data

than an application requests. \*/

while(total\_length)

{

/\* How much can we send in this transfer? \*/

if (total\_length > UX\_SLAVE\_CLASS\_STORAGE\_BUFFER\_SIZE)

transfer\_length = UX\_SLAVE\_CLASS\_STORAGE\_BUFFER\_SIZE;

else

transfer\_length = total\_length;

/\* Copy the Storage Buffer into the transfer request memory. \*/

**ux\_utility\_memory\_copy**(transfer\_request ->

ux\_slave\_transfer\_request\_data\_pointer,

media\_memory, transfer\_length);

/\* Send the data payload back to the caller. \*/

status = **ux\_device\_transfer\_request**(transfer\_request,

transfer\_length, transfer\_length);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

/\* Update the buffer address. \*/

media\_memory += transfer\_length;

/\* Update the length to remain. \*/

total\_length -= transfer\_length;

}

### ux\_device\_stack\_transfer\_abort

Cancel a transfer request

**Prototype**

UINT **ux\_device\_stack\_transfer\_abort**(UX\_SLAVE\_TRANSFER \*transfer\_request,

ULONG completion\_code)

**Description**

This function is called when an application needs to cancel a transfer request or when the stack needs to abort a transfer request associated with an endpoint.

**Parameters**

**transfer\_request** Pointer to the transfer request.

**completion\_code** Error code to be returned to the class waiting for this transfer request to complete.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |

**Example**

UINT status;

/\* The following example illustrates how to abort a transfer when

a bus reset has been detected on the bus. \*/

status = **ux\_device\_stack\_transfer\_abort**(transfer\_request,

UX\_TRANSFER\_BUS\_RESET);

/\* If status equals UX\_SUCCESS, the operation was successful. \*/

### ux\_device\_stack\_uninitialize

Unitialize stack

**Prototype**

UINT **ux\_device\_stack\_uninitialize**()

**Description**

This function is called when an application needs to unitialize usbx device stack

**Parameters**

None

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |

# Chapter 5: USBX Device Class Considerations

## Device Class registration

Each device class follows the same principle for registration. A structure containing specific class parameters is passed to the class initialize function :

/\* Set the parameters for callback when insertion/extraction of a HID device. \*/

hid\_parameter.ux\_slave\_class\_hid\_instance\_activate = tx\_demo\_hid\_instance\_activate;

hid\_parameter.ux\_slave\_class\_hid\_instance\_deactivate = tx\_demo\_hid\_instance\_deactivate;

/\* Initialize the hid class parameters for the device. \*/

hid\_parameter.ux\_device\_class\_hid\_parameter\_report\_address = hid\_device\_report;

hid\_parameter.ux\_device\_class\_hid\_parameter\_report\_length = HID\_DEVICE\_REPORT\_LENGTH;

hid\_parameter.ux\_device\_class\_hid\_parameter\_report\_id = UX\_TRUE;

hid\_parameter.ux\_device\_class\_hid\_parameter\_callback = demo\_thread\_hid\_callback;

/\* Initilize the device hid class. The class is connected with interface 0 \*/

status = ux\_device\_stack\_class\_register(\_ux\_system\_slave\_class\_hid\_name, ux\_device\_class\_hid\_entry,1,0, (VOID \*)&hid\_parameter);

Each class can register, optionally, a callback function when an instance of the class gets activated. The callback is then called by the device stack to inform the application that an instance was created.

The application would have in its body the 2 functions for activation and deactivation :

VOID tx\_demo\_hid\_instance\_activate(VOID \*hid\_instance)

{

/\* Save the HID instance. \*/

hid\_slave = (UX\_SLAVE\_CLASS\_HID \*) hid\_instance;

}

VOID tx\_demo\_hid\_instance\_deactivate(VOID \*hid\_instance)

{

/\* Reset the HID instance. \*/

hid\_slave = UX\_NULL;

}

It is not recommended to do anything within these functions but to memorise the instance of the class and synchronize with the rest of the application.

## USB Device Storage Class

The USB device storage class allows for a storage device embedded in the system to be made visible to a USB host.

The USB device storage class does not by itself provide a storage solution. It merely accepts and interprets SCSI requests coming from the host. When one of these requests is a read or a write command, it will invoke a pre-defined call back to a real storage device handler, such as an ATA device driver or a Flash device driver.

When initializing the device storage class, a pointer structure is given to the class that contains all the information necessary. An example is given below.

/\* Initialize the storage class parameters to customize vendor strings. \*/

storage\_parameter.ux\_slave\_class\_storage\_parameter\_vendor\_id =

demo\_ux\_system\_slave\_class\_storage\_vendor\_id;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_product\_id =

demo\_ux\_system\_slave\_class\_storage\_product\_id;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_product\_rev =

demo\_ux\_system\_slave\_class\_storage\_product\_rev;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_product\_serial =

demo\_ux\_system\_slave\_class\_storage\_product\_serial;

/\* Store the number of LUN in this device storage instance: single LUN. \*/

storage\_parameter.ux\_slave\_class\_storage\_parameter\_number\_lun = 1;

/\* Initialize the storage class parameters for reading/writing to the

Flash Disk. \*/

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_last\_lba = 0x1e6bfe;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_block\_length = 512;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_type = 0;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].  
 ux\_slave\_class\_storage\_media\_removable\_flag = 0x80;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].  
 ux\_slave\_class\_storage\_media\_read\_only\_flag = UX\_FALSE;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_read = tx\_demo\_thread\_flash\_media\_read;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_write =

tx\_demo\_thread\_flash\_media\_write;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_status =

tx\_demo\_thread\_flash\_media\_status;

/\* A simple write caching support.

If this callback is assigned, host gets notification from caching page

that write caching is supported, and can send SYNCHRONIZE\_CACHE to

flush cache. But caching options are not allowed to be changed. \*/

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_flush =

tx\_demo\_thread\_flash\_media\_flush;

/\* Initialize the device storage class. The class is connected with

interface 0 \*/

status =

**ux\_device\_stack\_class\_register**(\_ux\_system\_slave\_class\_storage\_name,

ux\_device\_class\_storage\_entry, ux\_device\_class\_storage\_thread,

0, (VOID \*)&storage\_parameter);

In this example, the driver storage strings are customized by assigning string pointers to corresponding parameter. If any one of the string pointer is left to UX\_NULL, the default ExpressLogic string is used.

In this example, the drive’s last block address or LBA is given as well as the logical sector size. The LBA is the number of sectors available in the media –1. The block length is set to 512 in regular storage media. It can be set to 2048 for optical drives.

The application needs to pass three callback function pointers to allow the storage class to read, write and obtain status for the media.

The prototypes for the read and write functions are:

UINT media\_read(VOID \*storage, ULONG lun, UCHAR\* data\_pointer, ULONG number\_blocks, ULONG lba, ULONG \*media\_status);

UINT media\_write(VOID \*storage, ULONG lun, UCHAR\* data\_pointer, ULONG number\_blocks, ULONG lba, ULONG \*media\_status);

Where:

data\_pointer is the address of the buffer to be used for reading or writing

number\_blocks is the number of sectors to read/write

lba is the sector address to read.

The return value can have either the value UX\_SUCCESS or UX\_ERROR indicating a successful or unsuccessful operation. When there is error, the storage class will report the media\_status filled by callback.

The status callback function has the following prototype:

UINT media\_status(VOID \*storage, ULONG lun, ULONG media\_id, ULONG \*media\_status);

The calling parameter media\_id is not currently used and should always be 0.

The return value is a SCSI error code that can have the following format:

Bits 0-7 Sense\_key

Bits 8-15 Additional Sense Code

Bits 16-23 Additional Sense Code Qualifier

The following table provides the possible Sense/ASC/ASCQ combinations.

|  |  |  |  |
| --- | --- | --- | --- |
| Sense Key | ASC | ASCQ | Description |
| 00 | 00 | 00 | NO SENSE |
| 01 | 17 | 01 | RECOVERED DATA WITH RETRIES |
| 01 | 18 | 00 | RECOVERED DATA WITH ECC |
| 02 | 04 | 01 | LOGICAL DRIVE NOT READY - BECOMING READY |
| 02 | 04 | 02 | LOGICAL DRIVE NOT READY - INITIALIZATION REQUIRED |
| 02 | 04 | 04 | LOGICAL UNIT NOT READY - FORMAT IN PROGRESS |
| 02 | 04 | FF | LOGICAL DRIVE NOT READY - DEVICE IS BUSY |
| 02 | 06 | 00 | NO REFERENCE POSITION FOUND |
| 02 | 08 | 00 | LOGICAL UNIT COMMUNICATION FAILURE |
| 02 | 08 | 01 | LOGICAL UNIT COMMUNICATION TIME-OUT |
| 02 | 08 | 80 | LOGICAL UNIT COMMUNICATION OVERRUN |
| 02 | 3A | 00 | MEDIUM NOT PRESENT |
| 02 | 54 | 00 | USB TO HOST SYSTEM INTERFACE FAILURE |
| 02 | 80 | 00 | INSUFFICIENT RESOURCES |
| 02 | FF | FF | UNKNOWN ERROR |
| 03 | 02 | 00 | NO SEEK COMPLETE |
| 03 | 03 | 00 | WRITE FAULT |
| 03 | 10 | 00 | ID CRC ERROR |
| 03 | 11 | 00 | UNRECOVERED READ ERROR |
| 03 | 12 | 00 | ADDRESS MARK NOT FOUND FOR ID FIELD |
| 03 | 13 | 00 | ADDRESS MARK NOT FOUND FOR DATA FIELD |
| 03 | 14 | 00 | RECORDED ENTITY NOT FOUND |
| 03 | 30 | 01 | CANNOT READ MEDIUM - UNKNOWN FORMAT |
| 03 | 31 | 01 | FORMAT COMMAND FAILED |
| 04 | 40 | NN | DIAGNOSTIC FAILURE ON COMPONENT NN (80H-FFH) |
| 05 | 1A | 00 | PARAMETER LIST LENGTH ERROR |
| 05 | 20 | 00 | INVALID COMMAND OPERATION CODE |
| 05 | 21 | 00 | LOGICAL BLOCK ADDRESS OUT OF RANGE |
| 05 | 24 | 00 | INVALID FIELD IN COMMAND PACKET |
| 05 | 25 | 00 | LOGICAL UNIT NOT SUPPORTED |
| 05 | 26 | 00 | INVALID FIELD IN PARAMETER LIST |
| 05 | 26 | 01 | PARAMETER NOT SUPPORTED |
| 05 | 26 | 02 | PARAMETER VALUE INVALID |
| 05 | 39 | 00 | SAVING PARAMETERS NOT SUPPORT |
| 06 | 28 | 00 | NOT READY TO READY TRANSITION – MEDIA CHANGED |
| 06 | 29 | 00 | POWER ON RESET OR BUS DEVICE RESET OCCURRED |
| 06 | 2F | 00 | COMMANDS CLEARED BY ANOTHER INITIATOR |
| 07 | 27 | 00 | WRITE PROTECTED MEDIA |
| 0B | 4E | 00 | OVERLAPPED COMMAND ATTEMPTED |

### Multiple SCSI LUN

The USBX device storage class supports multiple LUNs. It is therefore possible to create a storage device that acts as a CD-ROM and a Flash disk at the same time. In such a case, the initialization would be slightly different. Here is an example for a Flash Disk and CD-ROM:

/\* Store the number of LUN in this device storage instance. \*/

storage\_parameter.ux\_slave\_class\_storage\_parameter\_number\_lun = 2;

/\* Initialize the storage class parameters for reading/writing to the Flash Disk. \*/

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_last\_lba = 0x7bbff;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_block\_length = 512;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_type = 0;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_removable\_flag = 0x80;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_read = tx\_demo\_thread\_flash\_media\_read;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_write =

tx\_demo\_thread\_flash\_media\_write;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[0].

ux\_slave\_class\_storage\_media\_status =

tx\_demo\_thread\_flash\_media\_status;

/\* Initialize the storage class LUN parameters for reading/writing to

the CD-ROM. \*/

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[1].

ux\_slave\_class\_storage\_media\_last\_lba = 0x04caaf;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[1].

ux\_slave\_class\_storage\_media\_block\_length = 2048;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[1].

ux\_slave\_class\_storage\_media\_type = 5;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[1].

ux\_slave\_class\_storage\_media\_removable\_flag = 0x80;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[1].

ux\_slave\_class\_storage\_media\_read = tx\_demo\_thread\_cdrom\_media\_read;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[1].

ux\_slave\_class\_storage\_media\_write =

tx\_demo\_thread\_cdrom\_media\_write;

storage\_parameter.ux\_slave\_class\_storage\_parameter\_lun[1].

ux\_slave\_class\_storage\_media\_status =

tx\_demo\_thread\_cdrom\_media\_status;

/\* Initialize the device storage class for a Flash disk and CD-ROM. The class is connected with interface 0 \*/

status =

**ux\_device\_stack\_class\_register**(\_ux\_system\_slave\_class\_storage\_name,

ux\_device\_class\_storage\_entry, ux\_device\_class\_storage\_thread,0,

(VOID \*) &storage\_parameter);

### Write Caching SCSI LUN

The USBX device storage class supports write caching on LUNs.

The application needs to pass a callback function pointer to allow the storage class to report write caching enable to host and do flushing on host request.

The flush callback function has the following prototype:

UINT media\_flush(VOID \*storage, ULONG lun, ULONG number\_blocks, ULONG lba, ULONG \*media\_status);

The calling parameter number\_blocks and lba specifies the area on LUN that needs flush.

Note that when the callback is not assigned, host is not notified for write caching support, so there is no option for it. When the callback is assigned, host is notified for write caching enabled, but not allowed to change this setting.

## USB Device CDC-ACM Class

The USB device CDC-ACM class allows for a USB host system to communicate with the device as a serial device. This class is based on the USB standard and is a subset of the CDC standard.

A CDC-ACM compliant device framework needs to be declared by the device stack. An example is found here below:

unsigned char device\_framework\_full\_speed[] = {

/\* Device descriptor 18 bytes

0x02 bDeviceClass: CDC class code

0x00 bDeviceSubclass: CDC class sub code

0x00 bDeviceProtocol: CDC Device protocol

idVendor & idProduct - http://www.linux-usb.org/usb.ids

\*/

0x12, 0x01, 0x10, 0x01,

0xEF, 0x02, 0x01, 0x08,

0x84, 0x84, 0x00, 0x00,

0x00, 0x01, 0x01, 0x02,

0x03, 0x01,

/\* Configuration 1 descriptor 9 bytes \*/

0x09, 0x02, 0x4b, 0x00, 0x02, 0x01, 0x00,0x40, 0x00,

/\* Interface association descriptor. 8 bytes. \*/

0x08, 0x0b, 0x00, 0x02, 0x02, 0x02, 0x00, 0x00,

/\* Communication Class Interface Descriptor Requirement. 9 bytes. \*/

0x09, 0x04, 0x00, 0x00,0x01,0x02, 0x02, 0x01, 0x00,

/\* Header Functional Descriptor 5 bytes \*/

0x05, 0x24, 0x00,0x10, 0x01,

/\* ACM Functional Descriptor 4 bytes \*/

0x04, 0x24, 0x02,0x0f,

/\* Union Functional Descriptor 5 bytes \*/

0x05, 0x24, 0x06, 0x00, /\* Master interface \*/

0x01, /\* Slave interface \*/

/\* Call Management Functional Descriptor 5 bytes \*/

0x05, 0x24, 0x01,0x03, 0x01, /\* Data interface \*/

/\* Endpoint 1 descriptor 7 bytes \*/

0x07, 0x05, 0x83, 0x03,0x08, 0x00, 0xFF,

/\* Data Class Interface Descriptor Requirement 9 bytes \*/

0x09, 0x04, 0x01, 0x00, 0x02,0x0A, 0x00, 0x00, 0x00,

/\* First alternate setting Endpoint 1 descriptor 7 bytes\*/

0x07, 0x05, 0x02,0x02,0x40, 0x00,0x00,

/\* Endpoint 2 descriptor 7 bytes \*/

0x07, 0x05, 0x81,0x02,0x40, 0x00, 0x00,

The CDC-ACM class uses a composite device framework to group interfaces (control and data). As a result care should be taken when defining the device descriptor. USBX relies on the IAD descriptor to know internally how to bind interfaces. The IAD descriptor should be declared before the interfaces and contain the first interface of the CDC-ACM class and how many interfaces are attached.

The CDC-ACM class also uses a union functional descriptor which performs the same function as the newer IAD descriptor. Although a Union Functional descriptor must be declared for historical reasons and compatibility with the host side, it is not used by USBX.

The initialization of the CDC-ACM class expects the following parameters:

/\* Set the parameters for callback when insertion/extraction of a

CDC device. \*/

parameter.ux\_slave\_class\_cdc\_acm\_instance\_activate =

tx\_demo\_cdc\_instance\_activate;

parameter.ux\_slave\_class\_cdc\_acm\_instance\_deactivate =

tx\_demo\_cdc\_instance\_deactivate;

parameter.ux\_slave\_class\_cdc\_acm\_parameter\_change =

tx\_demo\_cdc\_instance\_parameter\_change;

/\* Initialize the device cdc class. This class owns both interfaces

starting with 0. \*/

status =

ux\_device\_stack\_class\_register(\_ux\_system\_slave\_class\_cdc\_acm\_name,

ux\_device\_class\_cdc\_acm\_entry, 1,0, &parameter);

The 2 parameters defined are callback pointers into the user applications that will be called when the stack activates or deactivate this class.

The third parameter defined is a callback pointer to the user application that will be called when there is line coding or line states parameter change. E.g., when there is request from host to change DTR state to TRUE, the callback is invoked, in it user application can check line states through IOCTL function to kow host is ready for communication.

The CDC-ACM is based on a USB-IF standard and is automatically recognized by MAC Os and Linux operating systems. On Windows platforms, this class requires a .inf file for Windows version prior to 10. Windows 10 does not require any .inf files. ExpressLogic supplies a template for the CDC-ACM class and it can be found in the usbx\_windows\_host\_files directory. For more recent version of Windows the file CDC\_ACM\_Template\_Win7\_64bit.inf should be used (except Win10). This file needs to be modified to reflect the PID/VID used by the device. The PID/VID will be specific to the final customer when the company and the product are registered with the USB-IF.

In the inf file, the fields to modify are located here:

[DeviceList]

%DESCRIPTION%=DriverInstall, USB\VID\_8484&PID\_0000

[DeviceList.NTamd64]

%DESCRIPTION%=DriverInstall, USB\VID\_8484&PID\_0000

In the device framework of the CDC-ACM device, the PID/VID are stored in the device descriptor (see the device descriptor declared above)

When a USB host systems discovers the USB CDC-ACM device, it will mount a serial class and the device can be used with any serial terminal program. See the host Operating System for reference.

The CDC-ACM class APIs are defined below:

### ux\_device\_class\_cdc\_acm\_ioctl

Perform IOCTL on the CDC-ACM interface

**Prototype**

UINT **ux\_device\_class\_cdc\_acm\_ioctl** (UX\_SLAVE\_CLASS\_CDC\_ACM \*cdc\_acm, ULONG ioctl\_function, VOID \*parameter)

**Description**

This function is called when an application needs to perform various ioctl calls to the cdc acm interface

**Parameters**

**cdc\_acm** Pointer to the cdc class instance.

**ioctl\_function** Ioctl function to be performed.

**parameter** Pointer to a parameter specific to the ioctl call

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Start cdc acm callback transmission. \*/

status = \_ux\_device\_class\_cdc\_acm\_ioctl(cdc\_acm\_slave, UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_TRANSMISSION\_START, &callback);

if(status != UX\_SUCCESS)

return;

**Ioctl functions :**

UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_SET\_LINE\_CODING 1

UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_GET\_LINE\_CODING 2

UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_GET\_LINE\_STATE 3

UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_ABORT\_PIPE 4

UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_SET\_LINE\_STATE 5

UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_TRANSMISSION\_START 6

UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_TRANSMISSION\_STOP 7

### ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_SET\_LINE\_CODING

Perform IOCTL Set Line Coding on the CDC-ACM interface

**Prototype**

UINT **ux\_device\_class\_cdc\_acm\_ioctl** (UX\_SLAVE\_CLASS\_CDC\_ACM \*cdc\_acm, ULONG ioctl\_function, VOID \*parameter)

**Description**

This function is called when an application needs to Set the Line Coding parameters

**Parameters**

**cdc\_acm** Pointer to the cdc class instance.

**ioctl\_function** ux\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_SET\_LINE\_CODING

**parameter** Pointer to a line parameter structure :

typedef struct UX\_SLAVE\_CLASS\_CDC\_ACM\_LINE\_CODING\_PARAMETER\_STRUCT

{

ULONG ux\_slave\_class\_cdc\_acm\_parameter\_baudrate;

UCHAR ux\_slave\_class\_cdc\_acm\_parameter\_stop\_bit;

UCHAR ux\_slave\_class\_cdc\_acm\_parameter\_parity;

UCHAR ux\_slave\_class\_cdc\_acm\_parameter\_data\_bit;

} UX\_SLAVE\_CLASS\_CDC\_ACM\_LINE\_CODING\_PARAMETER;

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Change the line coding values. \*/

line\_coding.ux\_slave\_class\_cdc\_acm\_line\_coding\_dter = 9600;

line\_coding.ux\_slave\_class\_cdc\_acm\_line\_coding\_stop\_bit = UX\_HOST\_CLASS\_CDC\_ACM\_LINE\_CODING\_STOP\_BIT\_15;

line\_coding.ux\_slave\_class\_cdc\_acm\_line\_coding\_parity = UX\_HOST\_CLASS\_CDC\_ACM\_LINE\_CODING\_PARITY\_EVEN;

line\_coding.ux\_slave\_class\_cdc\_acm\_line\_coding\_data\_bits = 5;

status = \_ux\_slave\_class\_cdc\_acm\_ioctl(cdc\_acm, UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_SET\_LINE\_CODING, &line\_coding);

if (status != UX\_SUCCESS)

break;

### ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_GET\_LINE\_CODING

Perform IOCTL Get Line Coding on the CDC-ACM interface

**Prototype**

UINT **ux\_device\_class\_cdc\_acm\_ioctl** (UX\_SLAVE\_CLASS\_CDC\_ACM \*cdc\_acm, ULONG ioctl\_function, VOID \*parameter)

**Description**

This function is called when an application needs to Get the Line Coding parameters

**Parameters**

**cdc\_acm** Pointer to the cdc class instance.

**ioctl\_function** ux\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_GET\_LINE\_CODING

**parameter** Pointer to a line parameter structure :

typedef struct UX\_SLAVE\_CLASS\_CDC\_ACM\_LINE\_CODING\_PARAMETER\_STRUCT

{

ULONG ux\_slave\_class\_cdc\_acm\_parameter\_baudrate;

UCHAR ux\_slave\_class\_cdc\_acm\_parameter\_stop\_bit;

UCHAR ux\_slave\_class\_cdc\_acm\_parameter\_parity;

UCHAR ux\_slave\_class\_cdc\_acm\_parameter\_data\_bit;

} UX\_SLAVE\_CLASS\_CDC\_ACM\_LINE\_CODING\_PARAMETER;

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* This is to retrieve BAUD rate. \*/

status = \_ux\_device\_class\_cdc\_acm\_ioctl(cdc\_acm\_slave, UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_GET\_LINE\_CODING, &line\_coding);

/\* Any error ? \*/

if (status == UX\_SUCCESS)

{

/\* Decode BAUD rate. \*/

switch (line\_coding.ux\_slave\_class\_cdc\_acm\_parameter\_baudrate)

{

case 1200 :

status = tx\_demo\_thread\_slave\_cdc\_acm\_response("1200", 4);

break;

case 2400 :

status = tx\_demo\_thread\_slave\_cdc\_acm\_response("2400", 4);

break;

case 4800 :

status = tx\_demo\_thread\_slave\_cdc\_acm\_response("4800", 4);

break;

case 9600 :

status = tx\_demo\_thread\_slave\_cdc\_acm\_response("9600", 4);

break;

case 115200 :

status = tx\_demo\_thread\_slave\_cdc\_acm\_response("115200", 6);

break;

}

}

### ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_GET\_LINE\_STATE

Perform IOCTL Get Line State on the CDC-ACM interface

**Prototype**

UINT **ux\_device\_class\_cdc\_acm\_ioctl** (UX\_SLAVE\_CLASS\_CDC\_ACM \*cdc\_acm, ULONG ioctl\_function, VOID \*parameter)

**Description**

This function is called when an application needs to Get the Line State parameters

**Parameters**

**cdc\_acm** Pointer to the cdc class instance.

**ioctl\_function** ux\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_GET\_LINE\_STATE

**parameter** Pointer to a line parameter structure :

typedef struct UX\_SLAVE\_CLASS\_CDC\_ACM\_LINE\_STATE\_PARAMETER\_STRUCT

{

UCHAR ux\_slave\_class\_cdc\_acm\_parameter\_rts;

UCHAR ux\_slave\_class\_cdc\_acm\_parameter\_dtr;

} UX\_SLAVE\_CLASS\_CDC\_ACM\_LINE\_STATE\_PARAMETER;

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* This is to retrieve RTS state. \*/

status = \_ux\_device\_class\_cdc\_acm\_ioctl(cdc\_acm\_slave, UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_GET\_LINE\_STATE, &line\_state);

/\* Any error ? \*/

if (status == UX\_SUCCESS)

{

/\* Check state. \*/

if (line\_state.ux\_slave\_class\_cdc\_acm\_parameter\_rts == UX\_TRUE)

/\* State is ON. \*/

status = tx\_demo\_thread\_slave\_cdc\_acm\_response("RTS ON", 6);

else

/\* State is OFF. \*/

status = tx\_demo\_thread\_slave\_cdc\_acm\_response("RTS OFF", 7);

}

### ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_SET\_LINE\_STATE

Perform IOCTL Set Line State on the CDC-ACM interface

**Prototype**

UINT **ux\_device\_class\_cdc\_acm\_ioctl** (UX\_SLAVE\_CLASS\_CDC\_ACM \*cdc\_acm, ULONG ioctl\_function, VOID \*parameter)

**Description**

This function is called when an application needs to Get the Line State parameters

**Parameters**

**cdc\_acm** Pointer to the cdc class instance.

**ioctl\_function** ux\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_SET\_LINE\_STATE

**parameter** Pointer to a line parameter structure :

typedef struct UX\_SLAVE\_CLASS\_CDC\_ACM\_LINE\_STATE\_PARAMETER\_STRUCT

{

UCHAR ux\_slave\_class\_cdc\_acm\_parameter\_rts;

UCHAR ux\_slave\_class\_cdc\_acm\_parameter\_dtr;

} UX\_SLAVE\_CLASS\_CDC\_ACM\_LINE\_STATE\_PARAMETER;

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* This is to set RTS state. \*/

line\_state.ux\_slave\_class\_cdc\_acm\_parameter\_rts = UX\_TRUE;

status = \_ux\_device\_class\_cdc\_acm\_ioctl(cdc\_acm\_slave, UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_SET\_LINE\_STATE, &line\_state);

/\* Any error ? \*/

if (status == UX\_SUCCESS)

{

### ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_ABORT\_PIPE

Perform IOCTL ABORT PIPE on the CDC-ACM interface

**Prototype**

UINT **ux\_device\_class\_cdc\_acm\_ioctl** (UX\_SLAVE\_CLASS\_CDC\_ACM \*cdc\_acm, ULONG ioctl\_function, VOID \*parameter)

**Description**

This function is called when an application needs to abort a pipe

**Parameters**

**cdc\_acm** Pointer to the cdc class instance.

**ioctl\_function** ux\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_ABORT\_PIPE

**parameter** The pipe direction :

UX\_SLAVE\_CLASS\_CDC\_ACM\_ENDPOINT\_XMIT 1

UX\_SLAVE\_CLASS\_CDC\_ACM\_ENDPOINT\_RCV 2

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* This is abort the Xmit pipe. \*/

status = \_ux\_device\_class\_cdc\_acm\_ioctl(cdc\_acm\_slave, UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_ABORT\_PIPE, UX\_SLAVE\_CLASS\_CDC\_ACM\_ENDPOINT\_XMIT);

/\* Any error ? \*/

if (status == UX\_SUCCESS)

{

### ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_TRANSMISSION\_START

Perform IOCTL Transmission Start on the CDC-ACM interface

**Prototype**

UINT **ux\_device\_class\_cdc\_acm\_ioctl** (UX\_SLAVE\_CLASS\_CDC\_ACM \*cdc\_acm, ULONG ioctl\_function, VOID \*parameter)

**Description**

This function is called when an application wants to use transmission with callback

**Parameters**

**cdc\_acm** Pointer to the cdc class instance.

**ioctl\_function** ux\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_TRANSMISSION\_START

**parameter** Pointer to the Start Transmission parameter structure :

typedef struct UX\_SLAVE\_CLASS\_CDC\_ACM\_CALLBACK\_PARAMETER\_STRUCT

{

UINT (\*ux\_device\_class\_cdc\_acm\_parameter\_write\_callback)(struct UX\_SLAVE\_CLASS\_CDC\_ACM\_STRUCT \*cdc\_acm, UINT status, ULONG length);

UINT (\*ux\_device\_class\_cdc\_acm\_parameter\_read\_callback)(struct UX\_SLAVE\_CLASS\_CDC\_ACM\_STRUCT \*cdc\_acm, UINT status, UCHAR \*data\_pointer, ULONG length);

} UX\_SLAVE\_CLASS\_CDC\_ACM\_CALLBACK\_PARAMETER;

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Set the callback parameter. \*/

callback.ux\_device\_class\_cdc\_acm\_parameter\_write\_callback = tx\_demo\_thread\_slave\_write\_callback;

callback.ux\_device\_class\_cdc\_acm\_parameter\_read\_callback = tx\_demo\_thread\_slave\_read\_callback;

/\* Program the start of transmission. \*/

status = \_ux\_device\_class\_cdc\_acm\_ioctl(cdc\_acm\_slave, UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_TRANSMISSION\_START, &callback);

/\* Check if all good. \*/

if (status != UX\_SUCCESS)

return;

### ux\_device\_class\_cdc\_acm\_ioctl : UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_TRANSMISSION\_STOP

Perform IOCTL Transmission Stop on the CDC-ACM interface

**Prototype**

UINT **ux\_device\_class\_cdc\_acm\_ioctl** (UX\_SLAVE\_CLASS\_CDC\_ACM \*cdc\_acm, ULONG ioctl\_function, VOID \*parameter)

**Description**

This function is called when an application wants to stop using transmission with callback

**Parameters**

**cdc\_acm** Pointer to the cdc class instance.

**ioctl\_function** ux\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_TRANSMISSION\_STOP

**parameter** Not used

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Program the stop of transmission. \*/

status = \_ux\_device\_class\_cdc\_acm\_ioctl(cdc\_acm\_slave, UX\_SLAVE\_CLASS\_CDC\_ACM\_IOCTL\_TRANSMISSION\_STOP, UX\_NULL);

/\* Check if all good. \*/

if (status != UX\_SUCCESS)

return;

### ux\_device\_class\_cdc\_acm\_read

Read from CDC-ACM pipe

**Prototype**

UINT **ux\_device\_class\_cdc\_acm\_read**(UX\_SLAVE\_CLASS\_CDC\_ACM \*cdc\_acm,

UCHAR \*buffer, ULONG requested\_length, ULONG \*actual\_length)

**Description**

This function is called when an application needs to read from the OUT data pipe (OUT from the host, IN from the device).

Note the function reads bytes from the host packet by packet. If the prepared buffer size is smaller than a packet and the host sends more data than expected (in other words, the prepared buffer size is not a multiple of the USB endpoint's max packet size), then buffer overflow will occur. To avoid this issue, the recommended way to read is to allocate a buffer exactly one packet size (USB endpoint max packet size). This way if there is more data, the next read can get it and no buffer overflow will occur. If there is less data, the current read can get a short packet instead of generating an error.

**Parameters**

**cdc\_acm** Pointer to the cdc class instance.

**buffer** Buffer address where data will be stored.

**requested\_length** The maximum length we expect

**actual\_length** The length returned into the buffer

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | Device is no longer in the configured state |
| **UX\_TRANSFER\_NO\_ANSWER** | (0x22) | No answer from device. The device was probably disconnected while the transfer was pending. |
| **UX\_TRANSFER\_BUFFER\_OVERFLOW** | (0x27) | Transfer buffer overflow, inside a USB packet, host sending more bytes than available buffer. |

**Example**

/\* Read from the CDC class. \*/

status = **ux\_device\_class\_cdc\_acm\_read**(cdc, buffer, UX\_DEMO\_BUFFER\_SIZE,

&actual\_length);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_cdc\_acm\_write

Writing to a CDC-ACM pipe

**Prototype**

UINT **ux\_device\_class\_cdc\_acm\_write**(UX\_SLAVE\_CLASS\_CDC\_ACM \*cdc\_acm,

UCHAR \*buffer, ULONG requested\_length, ULONG \*actual\_length)

**Description**

This function is called when an application needs to write to the IN data pipe (IN from the host, OUT from the device)

**Parameters**

**cdc\_acm** Pointer to the cdc class instance.

**buffer** Buffer address where data is stored.

**requested\_length** The length of the buffer to write

**actual\_length** The length returned into the buffer after write is performed

**Return Value**

|  |  |  |
| --- | --- | --- |
| UX\_SUCCESS | (0x00) | This operation was successful. |
| UX\_CONFIGURATION\_HANDLE\_UNKNOWN | (0x51) | Device is no longer in the configured state |
| UX\_TRANSFER\_NO\_ANSWER | (0x22) | No answer from device. The device was probably disconnected while the transfer was pending. |

**Example**

/\* Write to the CDC class bulk in pipe. \*/

status = **ux\_device\_class\_cdc\_acm\_write**(cdc, buffer, UX\_DEMO\_BUFFER\_SIZE,

&actual\_length);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_cdc\_acm\_write\_with\_callback

Writing to a CDC-ACM pipe with callback

**Prototype**

UINT **ux\_device\_class\_cdc\_acm\_write\_with\_callback**(UX\_SLAVE\_CLASS\_CDC\_ACM \*cdc\_acm,UCHAR \*buffer, ULONG requested\_length)

**Description**

This function is called when an application needs to write to the IN data pipe (IN from the host, OUT from the device). This function is not blocking and the completion will be done through a callback.

**Parameters**

**cdc\_acm** Pointer to the cdc class instance.

**buffer** Buffer address where data is stored.

**requested\_length** The length of the buffer to write

**actual\_length** The length returned into the buffer after write is performed

**Return Value**

|  |  |  |
| --- | --- | --- |
| UX\_SUCCESS | (0x00) | This operation was successful. |
| UX\_CONFIGURATION\_HANDLE\_UNKNOWN | (0x51) | Device is no longer in the configured state |
| UX\_TRANSFER\_NO\_ANSWER | (0x22) | No answer from device. The device was probably disconnected while the transfer was pending. |

**Example**

/\* Write to the CDC class bulk in pipe non blocking mode. \*/

status = **ux\_device\_class\_cdc\_acm\_write\_with\_callback**(cdc, buffer, UX\_DEMO\_BUFFER\_SIZE);

if(status != UX\_SUCCESS)

return;

## USB Device CDC-ECM Class

The USB device CDC-ECM class allows for a USB host system to communicate with the device as a ethernet device. This class is based on the USB standard and is a subset of the CDC standard.

A CDC-ACM compliant device framework needs to be declared by the device stack. An example is found here below:

unsigned char device\_framework\_full\_speed[] = {

/\* Device descriptor 18 bytes

0x02 bDeviceClass: CDC\_ECM class code

0x06 bDeviceSubclass: CDC\_ECM class sub code

0x00 bDeviceProtocol: CDC\_ECM Device protocol

idVendor & idProduct - http://www.linux-usb.org/usb.ids

0x3939 idVendor: ExpressLogic test.

\*/

0x12, 0x01, 0x10, 0x01,

0x02, 0x00, 0x00, 0x08,

0x39, 0x39, 0x08, 0x08,

0x00, 0x01, 0x01, 0x02, 03,0x01,

/\* Configuration 1 descriptor 9 bytes. \*/

0x09, 0x02, 0x58, 0x00,0x02, 0x01, 0x00,0x40, 0x00,

/\* Interface association descriptor. 8 bytes. \*/

0x08, 0x0b, 0x00, 0x02, 0x02, 0x06, 0x00, 0x00,

/\* Communication Class Interface Descriptor Requirement 9 bytes \*/

0x09, 0x04, 0x00, 0x00,0x01,0x02, 0x06, 0x00, 0x00,

/\* Header Functional Descriptor 5 bytes \*/

0x05, 0x24, 0x00, 0x10, 0x01,

/\* ECM Functional Descriptor 13 bytes \*/

0x0D, 0x24, 0x0F, 0x04,0x00, 0x00, 0x00, 0x00, 0xEA, 0x05, 0x00,

0x00,0x00,

/\* Union Functional Descriptor 5 bytes \*/

0x05, 0x24, 0x06, 0x00,0x01,

/\* Endpoint descriptor (Interrupt) \*/

0x07, 0x05, 0x83, 0x03, 0x08, 0x00, 0x08,

/\* Data Class Interface Descriptor Alternate Setting 0, 0 endpoints. 9 bytes \*/

0x09, 0x04, 0x01, 0x00, 0x00, 0x0A, 0x00, 0x00, 0x00,

/\* Data Class Interface Descriptor Alternate Setting 1, 2 endpoints. 9 bytes \*/

0x09, 0x04, 0x01, 0x01, 0x02, 0x0A, 0x00, 0x00,0x00,

/\* First alternate setting Endpoint 1 descriptor 7 bytes \*/

0x07, 0x05, 0x02, 0x02, 0x40, 0x00, 0x00,

/\* Endpoint 2 descriptor 7 bytes \*/

0x07, 0x05, 0x81, 0x02, 0x40, 0x00,0x00

};The CDC-ECM class uses a very similar device descriptor approach to the CDC-ACM and also requires an IAD descriptor. See the CDC-ACM class for definition.

In addition to the regular device framework, the CDC-ECM requires special string descriptors. An example is given below:

unsigned char string\_framework[] = {

/\* Manufacturer string descriptor: Index 1 - "Express Logic" \*/

0x09, 0x04, 0x01, 0x0c,

0x45, 0x78, 0x70, 0x72, 0x65, 0x73, 0x20, 0x4c,

0x6f, 0x67, 0x69, 0x63,

/\* Product string descriptor: Index 2 - "EL CDCECM Device" \*/

0x09, 0x04, 0x02, 0x10,

0x45, 0x4c, 0x20, 0x43, 0x44, 0x43, 0x45, 0x43,

0x4d, 0x20, 0x44, 0x65, 0x76, 0x69, 0x63, 0x64,

/\* Serial Number string descriptor: Index 3 - "0001" \*/

0x09, 0x04, 0x03, 0x04,

0x30, 0x30, 0x30, 0x31,

/\* MAC Address string descriptor: Index 4 - "001E5841B879" \*/

0x09, 0x04, 0x04, 0x0C,

0x30, 0x30, 0x31, 0x45, 0x35, 0x38,

0x34, 0x31, 0x42, 0x38, 0x37, 0x39

};

The MAC address string descriptor is used by the CDC-ECM class to reply to the host queries as to what MAC address the device is answering to at the TCP/IP protocol. It can be set to the device choice but must be defined here.

The initialization of the CDC-ECM class is as follows:

/\* Set the parameters for callback when insertion/extraction of a CDC device. Set to NULL.\*/

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_instance\_activate = UX\_NULL;

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_instance\_deactivate = UX\_NULL;

/\* Define a NODE ID. \*/

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_local\_node\_id[0] =

0x00;

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_local\_node\_id[1] =

0x1e;

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_local\_node\_id[2] =

0x58;

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_local\_node\_id[3] =

0x41;

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_local\_node\_id[4] =

0xb8;

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_local\_node\_id[5] =

0x78;

/\* Define a remote NODE ID. \*/

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_remote\_node\_id[0] =

0x00;

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_remote\_node\_id[1] =

0x1e;

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_remote\_node\_id[2] =

0x58;

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_remote\_node\_id[3] =

0x41;

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_remote\_node\_id[4] =

0xb8;

cdc\_ecm\_parameter.ux\_slave\_class\_cdc\_ecm\_parameter\_remote\_node\_id[5] =

0x79;

/\* Initialize the device cdc\_ecm class. \*/

status =

**ux\_device\_stack\_class\_register**(\_ux\_system\_slave\_class\_cdc\_ecm\_name,

ux\_device\_class\_cdc\_ecm\_entry, 1,0,

&cdc\_ecm\_parameter);

The initialization of this class expects the same function callback for activation and deactivation, although here as an exercise they are set to NULL so that no callback is performed.

The next parameters are for the definition of the node IDs. 2 Nodes are necessary for the CDC-ECM, a local node and a remote node. The remote node must be the same one as the one declared in the device framework string descriptor.

The CDC-ECM class has built-in APIs for transferring data both ways but they are hidden to the application as the user application will communicate with the USB Ethernet device through NetX.

The USBX CDC-ECM class is closely tied to ExpressLogic NetX Network stack.

An application using both NetX and USBX CDC-ECM class will activate the NetX network stack in its usual way but in addition needs to activate the USB network stack as follows:

/\* Initialize the NetX system. \*/

**nx\_system\_initialize**();

/\* Perform the initialization of the network driver. This will initialize the USBX network layer.\*/

**ux\_network\_driver\_init**();

The USB network stack needs to be activated only once and is not specific to CDC-ECM but is required by any USB class that requires NetX services.

The CDC-ECM class will be recognized by MAC OS and Linux hosts. But there is no driver supplied by Microsoft Windows to recognize CDC-ECM natively. Some commercial products do exist for Windows platforms and they supply their own .inf file. This file will need to be modified the same way as the CDC-ACM inf template to match the PID/VID of the USB network device.

## USB Device RNDIS Class

The USB device RNDIS class allows for a USB host system to communicate with the device as a ethernet device. This class is based on the Microsoft proprietary implementation and is specific to Windows platforms..

A RNDIS compliant device framework needs to be declared by the device stack. An example is found here below:

unsigned char device\_framework\_full\_speed[] = {

/\* VID: 0x04b4

PID: 0x1127

\*/

/\* Device Descriptor \*/

0x12, /\* bLength \*/

0x01, /\* bDescriptorType \*/

0x10, 0x01, /\* bcdUSB \*/

0x02, /\* bDeviceClass - CDC \*/

0x00, /\* bDeviceSubClass \*/

0x00, /\* bDeviceProtocol \*/

0x40, /\* bMaxPacketSize0 \*/

0xb4, 0x04, /\* idVendor \*/

0x27, 0x11, /\* idProduct \*/

0x00, 0x01, /\* bcdDevice \*/

0x01, /\* iManufacturer \*/

0x02, /\* iProduct \*/

0x03, /\* iSerialNumber \*/

0x01, /\* bNumConfigurations \*/

/\* Configuration Descriptor \*/

0x09, /\* bLength \*/

0x02, /\* bDescriptorType \*/

0x38, 0x00, /\* wTotalLength \*/

0x02, /\* bNumInterfaces \*/

0x01, /\* bConfigurationValue \*/

0x00, /\* iConfiguration \*/

0x40, /\* bmAttributes - Self-powered \*/

0x00, /\* bMaxPower \*/

/\* Interface Association Descriptor \*/

0x08, /\* bLength \*/

0x0b, /\* bDescriptorType \*/

0x00, /\* bFirstInterface \*/

0x02, /\* bInterfaceCount \*/

0x02, /\* bFunctionClass - CDC - Communication \*/

0xff, /\* bFunctionSubClass - Vendor Defined – In this case, RNDIS \*/

0x00, /\* bFunctionProtocol - No class specific protocol required \*/

0x00, /\* iFunction \*/

/\* Interface Descriptor \*/

0x09, /\* bLength \*/

0x04, /\* bDescriptorType \*/

0x00, /\* bInterfaceNumber \*/

0x00, /\* bAlternateSetting \*/

0x01, /\* bNumEndpoints \*/

0x02, /\* bInterfaceClass - CDC - Communication \*/

0xff, /\* bInterfaceSubClass - Vendor Defined – In this case, RNDIS \*/

0x00, /\* bInterfaceProtocol - No class specific protocol required \*/

0x00, /\* iInterface \*/

/\* Endpoint Descriptor \*/

0x07, /\* bLength \*/

0x05, /\* bDescriptorType \*/

0x83, /\* bEndpointAddress \*/

0x03, /\* bmAttributes - Interrupt \*/

0x08, 0x00, /\* wMaxPacketSize \*/

0xff, /\* bInterval \*/

/\* Interface Descriptor \*/

0x09, /\* bLength \*/

0x04, /\* bDescriptorType \*/

0x01, /\* bInterfaceNumber \*/

0x00, /\* bAlternateSetting \*/

0x02, /\* bNumEndpoints \*/

0x0a, /\* bInterfaceClass - CDC - Data \*/

0x00, /\* bInterfaceSubClass - Should be 0x00 \*/

0x00, /\* bInterfaceProtocol - No class specific protocol required \*/

0x00, /\* iInterface \*/

/\* Endpoint Descriptor \*/

0x07, /\* bLength \*/

0x05, /\* bDescriptorType \*/

0x02, /\* bEndpointAddress \*/

0x02, /\* bmAttributes - Bulk \*/

0x40, 0x00, /\* wMaxPacketSize \*/

0x00, /\* bInterval \*/

/\* Endpoint Descriptor \*/

0x07, /\* bLength \*/

0x05, /\* bDescriptorType \*/

0x81, /\* bEndpointAddress \*/

0x02, /\* bmAttributes - Bulk \*/

0x40, 0x00, /\* wMaxPacketSize \*/

0x00, /\* bInterval \*/

};

The RNDIS class uses a very similar device descriptor approach to the CDC-ACM and CDC-ECM and also requires a IAD descriptor. See the CDC-ACM class for definition and requirements for the device descriptor.

The activation of the RNDIS class is as follows:

/\* Set the parameters for callback when insertion/extraction of a CDC

device. Set to NULL.\*/

parameter.ux\_slave\_class\_rndis\_instance\_activate = UX\_NULL;

parameter.ux\_slave\_class\_rndis\_instance\_deactivate = UX\_NULL;

/\* Define a local NODE ID. \*/

parameter.ux\_slave\_class\_rndis\_parameter\_local\_node\_id[0] = 0x00;

parameter.ux\_slave\_class\_rndis\_parameter\_local\_node\_id[1] = 0x1e;

parameter.ux\_slave\_class\_rndis\_parameter\_local\_node\_id[2] = 0x58;

parameter.ux\_slave\_class\_rndis\_parameter\_local\_node\_id[3] = 0x41;

parameter.ux\_slave\_class\_rndis\_parameter\_local\_node\_id[4] = 0xb8;

parameter.ux\_slave\_class\_rndis\_parameter\_local\_node\_id[5] = 0x78;

/\* Define a remote NODE ID. \*/

parameter.ux\_slave\_class\_rndis\_parameter\_remote\_node\_id[0] = 0x00;

parameter.ux\_slave\_class\_rndis\_parameter\_remote\_node\_id[1] = 0x1e;

parameter.ux\_slave\_class\_rndis\_parameter\_remote\_node\_id[2] = 0x58;

parameter.ux\_slave\_class\_rndis\_parameter\_remote\_node\_id[3] = 0x41;

parameter.ux\_slave\_class\_rndis\_parameter\_remote\_node\_id[4] = 0xb8;

parameter.ux\_slave\_class\_rndis\_parameter\_remote\_node\_id[5] = 0x79;

/\* Set extra parameters used by the RNDIS query command with certain

OIDs. \*/

parameter.ux\_slave\_class\_rndis\_parameter\_vendor\_id = 0x04b4 ;

parameter.ux\_slave\_class\_rndis\_parameter\_driver\_version = 0x1127;

**ux\_utility\_memory\_copy**(parameter.

ux\_slave\_class\_rndis\_parameter\_vendor\_description,

"ELOGIC RNDIS", 12);

/\* Initialize the device rndis class. This class owns both interfaces. \*/

status =

**ux\_device\_stack\_class\_register**(\_ux\_system\_slave\_class\_rndis\_name,

ux\_device\_class\_rndis\_entry, 1,0,

&parameter);

As for the CDC-ECM, the RNDIS class requires 2 nodes, one local and one remote but there is no requirement to have a string descriptor describing the remote node.

However due to Microsoft proprietary messaging mechanism, some extra parameters are required. First the vendor ID has to be passed. Likewise, the driver version of the RNDIS. A vendor string must also be given.

The RNDIS class has built-in APIs for transferring data both ways but they are hidden to the application as the user application will communicate with the USB Ethernet device through NetX.

The USBX RNDIS class is closely tied to ExpressLogic NetX Network stack.

An application using both NetX and USBX RNDIS class will activate the NetX network stack in its usual way but in addition needs to activate the USB network stack as follows:

/\* Initialize the NetX system. \*/

**nx\_system\_initialize**();

/\* Perform the initialization of the network driver. This will

initialize the USBX network layer.\*/

**ux\_network\_driver\_init**();

The USB network stack needs to be activated only once and is not specific to RNDIS but is required by any USB class that requires NetX services.

The RNDIS class will not be recognized by MAC OS and Linux hosts as it is specific to Microsoft operating systems. On windows platforms a .inf file needs to be present on the host that matches the device descriptor. ExpressLogic supplies a template for the RNDIS class and it can be found in the usbx\_windows\_host\_files directory. For more recent version of Windows the file RNDIS\_Template.inf should be used. This file needs to be modified to reflect the PID/VID used by the device. The PID/VID will be specific to the final customer when the company and the product are registered with the USB-IF.

In the inf file, the fields to modify are located here:

[DeviceList]

%DeviceName%=DriverInstall, USB\VID\_xxxx&PID\_yyyy&MI\_00

[DeviceList.NTamd64]

%DeviceName%=DriverInstall, USB\VID\_xxxx&PID\_yyyy&MI\_00

In the device framework of the RNDIS device, the PID/VID are stored in the device descriptor (see the device descriptor declared above)

When a USB host systems discovers the USB RNDIS device, it will mount a network interface and the device can be used with network protocol stack. See the host Operating System for reference.

## USB Device DFU Class

The USB device DFU class allows for a USB host system to update the device firmware based on a host application. The DFU class is a USB-IF standard class.

USBX DFU class is relatively simple. It device descriptor does not require anything but a control endpoint. Most of the time, this class will be embedded into a USB composite device. The device can be anything such as a storage device or a comm device and the added DFU interface can inform the host that the device can have its firmware updated on the fly.

The DFU class works in 3 steps. First the device mounts as normal using the class exported. An application on the host (Windows or Linux) will exercise the DFU class and send a request to reset the device into DFU mode. The device will disappear from the bus for a short time (enough for the host and the device to detect a RESET sequence) and upon restarting, the device will be exclusively in DFU mode, waiting for the host application to send a firmware upgrade. When the firmware upgrade has been completed, the host application resets the device and upon re-enumeration the device will revert to its normal operation with the new firmware.

A DFU device framework will look like this:

UCHAR device\_framework\_full\_speed[] = {

/\* Device descriptor \*/

0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x40,

0x99, 0x99, 0x00, 0x00, 0x00, 0x00, 0x01, 0x02,

0x03, 0x01,

/\* Configuration descriptor \*/

0x09, 0x02, 0x1b, 0x00, 0x01, 0x01, 0x00, 0xc0,

0x32,

/\* Interface descriptor for DFU. \*/

0x09, 0x04, 0x00, 0x00, 0x00, 0xFE, 0x01, 0x01,

0x00,

/\* Functional descriptor for DFU. \*/

0x09, 0x21, 0x0f, 0xE8, 0x03, 0x40, 0x00, 0x00,

0x01,

};

In this example, the DFU descriptor is not associated with any other classes. It has a simple interface descriptor and no other endpoints attached to it. There is a Functional descriptor that describes the specifics of the DFU capabilities of the device.

The description of the DFU capabilities are as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | Offset | | Size | type | Description |
| bmAttributes | 2 | 1 | | Bit field | Bit 3: device will perform a bus detach-attach sequence when it receives a DFU\_DETACH request.  The host must not issue a USB Reset. (*bitWillDetach*)  0 = no  1 = yes  Bit 2: device is able to communicate via USB after Manifestation phase.  (*bitManifestationTolerant*)  0 = no, must see bus reset  1 = yes  Bit 1: upload capable (*bitCanUpload*)  0 = no  1 = yes  Bit 0: download capable  (*bitCanDnload*)  0 = no  1 = yes |
| wDetachTimeOut | 3 | 2 | | number | Time, in milliseconds, that the device will wait after receipt of the DFU\_DETACH request. If this time elapses without a USB reset, then the device will terminate the Reconfiguration phase and revert  back to normal operation. This represents the maximum time that the device can wait (depending on its timers, etc.). USBX sets this value to 1000 ms. |
| wTransferSize | 5 | 2 | | number | Maximum number of bytes that the device can accept per control-write operation. USBX sets this value to 64 bytes. |

The declaration of the DFU class is as follows:

/\* Store the DFU parameters. \*/

dfu\_parameter.ux\_slave\_class\_dfu\_parameter\_instance\_activate =

tx\_demo\_thread\_dfu\_activate;

dfu\_parameter.ux\_slave\_class\_dfu\_parameter\_instance\_deactivate =

tx\_demo\_thread\_dfu\_deactivate;

dfu\_parameter.ux\_slave\_class\_dfu\_parameter\_read =

tx\_demo\_thread\_dfu\_read;

dfu\_parameter.ux\_slave\_class\_dfu\_parameter\_write =

tx\_demo\_thread\_dfu\_write;

dfu\_parameter.ux\_slave\_class\_dfu\_parameter\_get\_status =

tx\_demo\_thread\_dfu\_get\_status;

dfu\_parameter.ux\_slave\_class\_dfu\_parameter\_notify =

tx\_demo\_thread\_dfu\_notify;

dfu\_parameter.ux\_slave\_class\_dfu\_parameter\_framework =

device\_framework\_dfu;

dfu\_parameter.ux\_slave\_class\_dfu\_parameter\_framework\_length =

DEVICE\_FRAMEWORK\_LENGTH\_DFU;

/\* Initialize the device dfu class. The class is connected with interface

1 on configuration 1. \*/

status =

**ux\_device\_stack\_class\_register**(\_ux\_system\_slave\_class\_dfu\_name,

ux\_device\_class\_dfu\_entry, 1, 0,

(VOID \*)&dfu\_parameter);

if (status!=UX\_SUCCESS)

return;

The DFU class needs to work with a device firmware application specific to the target. Therefore it defines several call back to read and write blocks of firmware and to get status from the firmware update application. The DFU class also has a notify callback function to inform the application when a begin and end of transfer of the firmware occur.

Following is the description of a typical DFU application flow.



The major challenge of the DFU class is getting the right application on the host to perform the download the firmware. There is no application supplied by Microsoft or the USB-IF. Some shareware exist and they work reasonably well on Linux and to a lesser extent on Windows.

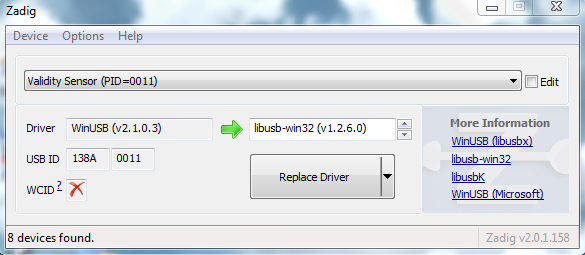
On Linux, one can use dfu-utils to be found here: <http://wiki.openmoko.org/wiki/Dfu-util>

A lot of information on dfu utils can also be found on this link: <http://www.libusb.org/wiki/windows_backend>

The Linux implementation of DFU performs correctly the reset sequence between the host and the device and therefore the device does not need to do it. Linux can accept for the bmAttributes *bitWillDetach* to be0. Windows on the other side requires the device to perform the reset.

On Windows, the USB registry must be able to associate the USB device with its PID/VID and the USB library which will in turn be used by the DFU application. This can be easily done with the free utility Zadig which can be found here: <http://sourceforge.net/projects/libwdi/files/zadig/>.

Running Zadig for the first time will show this screen:



From the device list, find your device and associate it with the libusb windows driver. This will bind the PID/VID of the device with the Windows USB library used by the DFU utilities.

To operate the DFU command, simply unpack the zipped dfu utilities into a directory, making sure the libusb dll is also present in the same directory. The DFU utilities must be run from a DOS box at the command line.

First, type the command **dfu-util –l** to determine whether the device is listed. If not, run Zadig to make sure the device is listed and associated with the USB library. You should see a screen as follows:

C:\usb specs\DFU\dfu-util-0.6>dfu-util -l

dfu-util 0.6

Copyright 2005-2008 Weston Schmidt, Harald Welte and OpenMoko Inc.

Copyright 2010-2012 Tormod Volden and Stefan Schmidt

This program is Free Software and has ABSOLUTELY NO WARRANTY

Found Runtime: [0a5c:21bc] devnum=0, cfg=1, intf=3, alt=0, name="UNDEFINED"

The next step will be to prepare the file to be downloaded. The USBX DFU class does not perform any verification on this file and is agnostic of its internal format. This firmware file is very specific to the target but not to DFU nor to USBX.

Then the dfu-util can be instructed to send the file by typing the following command:

dfu-util –R –t 64 -D file\_to\_download.hex

The dfu-util should display the file download process until the firmware has been completely downloaded.

## USB Device HID Class

The USB device HID class allows for a USB host system to connect to a HID device with specific HID client capabilities.

USBX HID device class is relatively simple compared to the host side. It is closely tied to the behavior of the device and its HID descriptor.

Any HID client requires first to define a HID device framework as the example below:

UCHAR device\_framework\_full\_speed[] = {

/\* Device descriptor \*/

0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x08,

0x81, 0x0A, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00,

0x00, 0x01,

/\* Configuration descriptor \*/

0x09, 0x02, 0x22, 0x00, 0x01, 0x01, 0x00, 0xc0, 0x32,

/\* Interface descriptor \*/

0x09, 0x04, 0x00, 0x00, 0x01, 0x03, 0x00, 0x00, 0x00,

/\* HID descriptor \*/

0x09, 0x21, 0x10, 0x01, 0x21, 0x01, 0x22, 0x3f, 0x00,

/\* Endpoint descriptor (Interrupt) \*/

0x07, 0x05, 0x81, 0x03, 0x08, 0x00, 0x08

};

The HID framework contains an interface descriptor that describes the HID class and the HID device subclass. The HID interface can be a standalone class or part of a composite device. If the HID device supports multiple report, its report\_id parameter should be set to UX\_TRUE, if not it should be set to UX\_FALSE.

The initialization of the HID class is as follow, using a USB keyboard as an example:

/\* Initialize the hid class parameters for a keyboard. \*/

hid\_parameter.ux\_device\_class\_hid\_parameter\_report\_address =

hid\_keyboard\_report;

hid\_parameter.ux\_device\_class\_hid\_parameter\_report\_length =

HID\_KEYBOARD\_REPORT\_LENGTH;

hid\_parameter.ux\_device\_class\_hid\_parameter\_callback =

tx\_demo\_thread\_hid\_callback;

hid\_parameter.ux\_device\_class\_hid\_parameter\_report\_id = UX\_TRUE;

/\* Initialize the device hid class. The class is connected with interface

0 \*/

status =

**ux\_device\_stack\_class\_register**(\_ux\_system\_slave\_class\_hid\_name,

ux\_device\_class\_hid\_entry, 1,0,

(VOID \*)&hid\_parameter);

if (status!=UX\_SUCCESS)

return;

The application needs to pass to the HID class a HID report descriptor and its length. The report descriptor is a collection of items that describe the device. For more information on the HID grammar refer to the HID USB class specification.

In addition to the report descriptor, the application passes a call back when a HID event happens.

The USBX HID class supports the following standard HID commands from the host:

|  |  |  |
| --- | --- | --- |
| Command name | Value | Description |
| UX\_DEVICE\_CLASS\_HID\_COMMAND\_GET\_REPORT | 0x01 | Get a report from the device |
| UX\_DEVICE\_CLASS\_HID\_COMMAND\_GET\_IDLE | 0x02 | Get the idle frequency of the interrupt endpoint |
| UX\_DEVICE\_CLASS\_HID\_COMMAND\_GET\_PROTOCOL | 0x03 | Get the protocol running on the device |
| UX\_DEVICE\_CLASS\_HID\_COMMAND\_SET\_REPORT | 0x09 | Set a report to the device |
| UX\_DEVICE\_CLASS\_HID\_COMMAND\_SET\_IDLE | 0x0a | Set the idle frequency of the interrupt endpoint |
| UX\_DEVICE\_CLASS\_HID\_COMMAND\_SET\_PROTOCOL | 0x0b | Get the protocol running on the device |

The Get and Set report are the most commonly used commands by HID to transfer data back and forth between the host and the device. Most commonly the host sends data on the control endpoint but can receive data either on the interrupt endpoint or by issuing a GET\_REPORT command to fetch the data on the control endpoint.

The HID class can send data back to the host on the interrupt endpoint by using the ux\_device\_class\_hid\_event\_set function.

### ux\_device\_class\_hid\_event\_set

Setting an event to the HID class

**Prototype**

UINT ux\_**device\_class\_hid\_event\_set**(UX\_SLAVE\_CLASS\_HID \*hid,

UX\_SLAVE\_CLASS\_HID\_EVENT \*hid\_event)

**Description**

This function is called when an application needs to send a HID event back to the host. The function is not blocking, it merely puts the report into a circular queue and returns to the application

**Parameters**

**hid** Pointer to the hid class instance.

**hid\_event** Pointer to structure of the hid event.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0x01) | Error on round robin queue |

**Example**

/\* Insert a key into the keyboard event. Length is fixed to 8. \*/

hid\_event.ux\_device\_class\_hid\_event\_length = 8;

/\* First byte is a modifier byte. \*/

hid\_event.ux\_device\_class\_hid\_event\_buffer[0] = 0;

/\* Second byte is reserved. \*/

hid\_event.ux\_device\_class\_hid\_event\_buffer[1] = 0;

/\* The 6 next bytes are keys. We only have one key here. \*/

hid\_event.ux\_device\_class\_hid\_event\_buffer[2] = key;

/\* Set the keyboard event. \*/

**ux\_device\_class\_hid\_event\_set**(hid, &hid\_event);

The callback defined at the initialization of the HID class performs the opposite of sending an event. It gets as input the event sent by the host. The prototype of the callback is as follows:

### hid\_callback

Getting an event from the HID class

**Prototype**

UINT **hid\_callback**(UX\_SLAVE\_CLASS\_HID \*hid,

UX\_SLAVE\_CLASS\_HID\_EVENT \*hid\_event)

**Description**

This function is called when the host sends a HID report to the application.

**Parameters**

**hid** Pointer to the hid class instance.

**hid\_event** Pointer to structure of the hid event.

**Example**

The following example shows how to interpret an event for a HID keyboard:

UINT tx\_demo\_thread\_hid\_callback(UX\_SLAVE\_CLASS\_HID \*hid,

UX\_SLAVE\_CLASS\_HID\_EVENT \*hid\_event)

{

/\* There was an event. Analyze it. Is it NUM LOCK ? \*/

if (hid\_event -> ux\_device\_class\_hid\_event\_buffer[0] &

HID\_NUM\_LOCK\_MASK)

/\* Set the Num lock flag. \*/

num\_lock\_flag = UX\_TRUE;

else

/\* Reset the Num lock flag. \*/

num\_lock\_flag = UX\_FALSE;

/\* There was an event. Analyze it. Is it CAPS LOCK ? \*/

if (hid\_event -> ux\_device\_class\_hid\_event\_buffer[0] &

HID\_CAPS\_LOCK\_MASK)

/\* Set the Caps lock flag. \*/

caps\_lock\_flag = UX\_TRUE;

else

/\* Reset the Caps lock flag. \*/

caps\_lock\_flag = UX\_FALSE;

}

## USB Device PIMA Class (PTP Responder)

The USB device PIMA class allows for a USB host system (Initiator) to connect to a PIMA device (Resonder) to transfer media files. USBX Pima Class is conforming to the USB-IF PIMA 15740 class also known as PTP class (for Picture Transfer Protocol).

USBX device side PIMA class supports the following operations:

|  |  |  |
| --- | --- | --- |
| Operation code | Value | Description |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_GET\_DEVICE\_INFO | 0x1001 | Obtain the device supported operations and events |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_OPEN\_SESSION | 0x1002 | Open a session between the host and the device |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_CLOSE\_SESSION | 0x1003 | Close a session between the host and the device |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_GET\_STORAGE\_IDS | 0x1004 | Returns the storage ID for the device. USBX PIMA uses one storage ID only |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_GET\_STORAGE\_INFO | 0x1005 | Return information about the storage object such as max capacity and free space |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_GET\_NUM\_OBJECTS | 0x1006 | Return the number of objects contained in the storage device |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_GET\_OBJECT\_HANDLES | 0x1007 | Return an array of handles of the objects on the storage device |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_GET\_OBJECT\_INFO | 0x1008 | Return information about an object such as the name of the object, its creation date, modification date … |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_GET\_OBJECT | 0x1009 | Return the data pertaining to a specific object. |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_GET\_THUMB | 0x100A | Send the thumbnail if available about an object |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_DELETE\_OBJECT | 0x100B | Delete an object on the media |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_SEND\_OBJECT\_INFO | 0x100C | Send to the device information about an object for its creation on the media |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_SEND\_OBJECT | 0x100D | Send data for an object to the device |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_FORMAT\_STORE | 0x100F | Clean the device media |
| UX\_DEVICE\_CLASS\_PIMA\_OC\_RESET\_DEVICE | 0x0110 | Reset the target device |

|  |  |  |
| --- | --- | --- |
| Operation Code | Value | Description |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_CANCEL\_TRANSACTION | 0x4001 | Cancels the current transaction |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_OBJECT\_ADDED | 0x4002 | An object has been added to the device media and can be retrieved by the host. |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_OBJECT\_REMOVED | 0x4003 | An object has been deleted from the device media |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_STORE\_ADDED | 0x4004 | A media has been added to the device |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_STORE\_REMOVED | 0x4005 | A media has been deleted from the device |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_DEVICE\_PROP\_CHANGED | 0x4006 | Device properties have changed |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_OBJECT\_INFO\_CHANGED | 0x4007 | An object information has changed |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_DEVICE\_INFO\_CHANGE | 0x4008 | A device has changed |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_REQUEST\_OBJECT\_TRANSFER | 0x4009 | The device requests the transfer of an object from the host |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_STORE\_FULL | 0x400A | Device reports the media is full |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_DEVICE\_RESET | 0x400B | Device reports it was reset |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_STORAGE\_INFO\_CHANGED | 0x400C | Storage information has changed on the device |
| UX\_DEVICE\_CLASS\_PIMA\_EC\_CAPTURE\_COMPLETE | 0x400D | Capture is completed |

The USBX PIMA device class uses a TX Thread to listen to PIMA commands from the host.

A PIMA command is composed of a command block, a data block and a status phase.

The function ux\_device\_class\_pima\_thread posts a request to the stack to receive a PIMA command from the host side. The PIMA command is decoded and verified for content. If the command block is valid, it branches to the appropriate command handler.

Most PIMA commands can only be executed when a session has been opened by the host. The only exception is the command UX\_DEVICE\_CLASS\_PIMA\_OC\_GET\_DEVICE\_INFO. With USBX PIMA implementation, only one session can be opened between an Initiator and Responder at any time. All transactions within the single session are blocking and no new transaction can begin before the previous one completed.

PIMA transactions are composed of 3 phases, a command phase, an optional data phase and a response phase. If a data phase is present, it can only be in one direction.

The Initiator always determines the flow of the PIMA operations but the Responder can initiate events back to the Initiator to inform status changes that happened during a session.

The following diagram shows the transfer of a data object between the host and the PIMA device class:



## Initialization of the PIMA device class

The PIMA device class needs some parameters supplied by the application during the initialization.

The following parameters describe the device and storage information:

* ux\_device\_class\_pima\_manufacturer
* ux\_device\_class\_pima\_model
* ux\_device\_class\_pima\_device\_version
* ux\_device\_class\_pima\_serial\_number
* ux\_device\_class\_pima\_storage\_id
* ux\_device\_class\_pima\_storage\_type
* ux\_device\_class\_pima\_storage\_file\_system\_type
* ux\_device\_class\_pima\_storage\_access\_capability
* ux\_device\_class\_pima\_storage\_max\_capacity\_low
* ux\_device\_class\_pima\_storage\_max\_capacity\_high
* ux\_device\_class\_pima\_storage\_free\_space\_low
* ux\_device\_class\_pima\_storage\_free\_space\_high
* ux\_device\_class\_pima\_storage\_free\_space\_image
* ux\_device\_class\_pima\_storage\_description
* ux\_device\_class\_pima\_storage\_volume\_label

The PIMA class also requires the registration of callback into the application to inform the application of certain events or retrieve/store data from/to the local media. The callbacks are:

* ux\_device\_class\_pima\_object\_number\_get
* ux\_device\_class\_pima\_object\_handles\_get
* ux\_device\_class\_pima\_object\_info\_get
* ux\_device\_class\_pima\_object\_data\_get
* ux\_device\_class\_pima\_object\_info\_send
* ux\_device\_class\_pima\_object\_data\_send
* ux\_device\_class\_pima\_object\_delete

The following example shows how to initialize the client side of PIMA. This example uses Pictbridge as a client for PIMA:

/\* Initialize the first XML object valid in the pictbridge instance.

Initialize the handle, type and file name.

The storage handle and the object handle have a fixed value of 1 in our

implementation. \*/

object\_info = pictbridge -> ux\_pictbridge\_object\_client;

object\_info -> ux\_device\_class\_pima\_object\_format =

UX\_DEVICE\_CLASS\_PIMA\_OFC\_SCRIPT;

object\_info -> ux\_device\_class\_pima\_object\_storage\_id = 1;

object\_info -> ux\_device\_class\_pima\_object\_handle\_id = 2;

ux\_utility\_string\_to\_unicode(\_ux\_pictbridge\_ddiscovery\_name,

object\_info ->

ux\_device\_class\_pima\_object\_filename);

/\* Initialize the head and tail of the notification round robin buffers.

At first, the head and tail are pointing to the beginning of the array. \*/

pictbridge -> ux\_pictbridge\_event\_array\_head = pictbridge ->

ux\_pictbridge\_event\_array;

pictbridge -> ux\_pictbridge\_event\_array\_tail = pictbridge ->

ux\_pictbridge\_event\_array;

pictbridge -> ux\_pictbridge\_event\_array\_end = pictbridge ->

ux\_pictbridge\_event\_array +

UX\_PICTBRIDGE\_MAX\_EVENT\_NUMBER;

/\* Initialialize the pima device parameter. \*/

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_manufacturer = pictbridge ->

ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_vendor\_name;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_model = pictbridge ->

ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_product\_name;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_serial\_number = pictbridge ->

ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_serial\_no;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_storage\_id = 1;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_storage\_type =

UX\_DEVICE\_CLASS\_PIMA\_STC\_FIXED\_RAM;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_storage\_file\_system\_type =

UX\_DEVICE\_CLASS\_PIMA\_FSTC\_GENERIC\_FLAT;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_storage\_access\_capability =

UX\_DEVICE\_CLASS\_PIMA\_AC\_READ\_WRITE;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_storage\_max\_capacity\_low =

pictbridge -> ux\_pictbridge\_dpslocal.

ux\_pictbridge\_devinfo\_storage\_size;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_storage\_max\_capacity\_high = 0;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_storage\_free\_space\_low = pictbridge ->

ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_storage\_size;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_storage\_free\_space\_high = 0;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_storage\_free\_space\_image = 0;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_storage\_description =

\_ux\_pictbridge\_volume\_description;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_storage\_volume\_label =

\_ux\_pictbridge\_volume\_label;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_object\_number\_get =

ux\_pictbridge\_dpsclient\_object\_number\_get;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_object\_handles\_get =

ux\_pictbridge\_dpsclient\_object\_handles\_get;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_object\_info\_get =

ux\_pictbridge\_dpsclient\_object\_info\_get;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_object\_data\_get =

ux\_pictbridge\_dpsclient\_object\_data\_get;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_object\_info\_send =

ux\_pictbridge\_dpsclient\_object\_info\_send;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_object\_data\_send =

ux\_pictbridge\_dpsclient\_object\_data\_send;

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_object\_delete =

ux\_pictbridge\_dpsclient\_object\_delete;

/\* Store the instance owner. \*/

pictbridge -> ux\_pictbridge\_pima\_parameter.

ux\_device\_class\_pima\_parameter\_application = (VOID \*) pictbridge;

/\* Initialize the device pima class. The class is connected with interface

0 \*/

status = **ux\_device\_stack\_class\_register**(\_ux\_system\_slave\_class\_pima\_name,

ux\_device\_class\_pima\_entry, 1, 0,

(VOID \*)&pictbridge ->

ux\_pictbridge\_pima\_parameter);

/\* Check status. \*/

if (status != UX\_SUCCESS)

### ux\_device\_class\_pima\_object\_add

Adding an object and sending the event to the host

**Prototype**

UINT **ux\_device\_class\_pima\_object\_add**(UX\_SLAVE\_CLASS\_PIMA \*pima,

ULONG object\_handle)

**Description**

This function is called when the PIMA class needs to add an object and inform the host.

**Parameters**

**pima** Pointer to the pima class instance.

**object\_handle** Handle of the object.

**Example**

/\* Send the notification to the host that an object has been

added. \*/

status = **ux\_device\_class\_pima\_object\_add**(pima, UX\_PICTBRIDGE\_OBJECT\_HANDLE\_CLIENT\_REQUEST);

### ux\_device\_class\_pima\_object\_number\_get

Getting the object number from the application

**Prototype**

UINT **ux\_device\_class\_pima\_object\_number\_get**(UX\_SLAVE\_CLASS\_PIMA \*pima,

ULONG \*object\_number)

**Description**

This function is called when the PIMA class needs to retrieve the number of objects in the local system and send it back to the host.

**Parameters**

**pima** Pointer to the pima class instance.

**object\_number** Address of the number of objects to be returned.

**Example**

UINT **ux\_pictbridge\_dpsclient\_object\_number\_get**(UX\_SLAVE\_CLASS\_PIMA \*pima,

ULONG \*number\_objects)

{

/\* We force the number of objects to be 1 only here. This will be the XML

scripts. \*/

\*number\_objects = 1;

return(UX\_SUCCESS);

}

### ux\_device\_class\_pima\_object\_handles\_get

Return the object handle array

**Prototype**

UINT **ux\_device\_class\_pima\_object\_handles\_get**(UX\_SLAVE\_CLASS\_PIMA\_STRUCT

\*pima, ULONG object\_handles\_format\_code,

ULONG object\_handles\_association,

ULONG \*object\_handles\_array,

ULONG object\_handles\_max\_number);

**Description**

This function is called when the PIMA class needs to retrieve the object handles array in the local system and send it back to the host.

**Parameters**

**pima** Pointer to the pima class instance.

**object\_handles\_format\_code** Format code for the handles

**object\_handles\_association** Object association code

**object\_handle\_array** Address where to store the handles

**object\_handles\_max\_number** Maximum number of handles in the array

**Example**

UINT ux\_pictbridge\_dpsclient\_object\_handles\_get(UX\_SLAVE\_CLASS\_PIMA \*pima,

ULONG object\_handles\_format\_code, ULONG object\_handles\_association,

ULONG \*object\_handles\_array, ULONG object\_handles\_max\_number)

{

UX\_PICTBRIDGE \*pictbridge;

UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*object\_info;

/\* Get the pointer to the Pictbridge instance. \*/

pictbridge = (UX\_PICTBRIDGE \*) pima -> ux\_device\_class\_pima\_application;

/\* Set the pima pointer to the pictbridge instance. \*/

pictbridge -> ux\_pictbridge\_pima = (VOID \*) pima;

/\* We say we have one object but the caller might specify differnt format

code and associations. \*/

object\_info = pictbridge -> ux\_pictbridge\_object\_client;

/\* Insert in the array the number of found handles so far: 0. \*/

ux\_utility\_long\_put((UCHAR \*)object\_handles\_array, 0);

/\* Check the type demanded. \*/

if (object\_handles\_format\_code == 0 || object\_handles\_format\_code ==

0xFFFFFFFF || object\_info ->

ux\_device\_class\_pima\_object\_format ==

object\_handles\_format\_code)

{

/\* Insert in the array the number of found handles. This handle is

for the client XML script. \*/

ux\_utility\_long\_put((UCHAR \*)object\_handles\_array, 1);

/\* Adjust the array to point after the number of elements. \*/

object\_handles\_array++;

/\* We have a candicate. Store the handle. \*/

ux\_utility\_long\_put((UCHAR \*)object\_handles\_array, object\_info ->

ux\_device\_class\_pima\_object\_handle\_id);

}

return(UX\_SUCCESS);

}

### ux\_device\_class\_pima\_object\_info\_get

Return the object information

**Prototype**

UINT **ux\_device\_class\_pima\_object\_info\_get**(struct

UX\_SLAVE\_CLASS\_PIMA\_STRUCT \*pima, ULONG object\_handle,

UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*\*object);

**Description**

This function is called when the PIMA class needs to retrieve the object handles array in the local system and send it back to the host.

**Parameters**

**pima** Pointer to the pima class instance.

**object\_handles** Handle of the object

**object** Object pointer address

**Example**

UINT ux\_pictbridge\_dpsclient\_object\_info\_get(UX\_SLAVE\_CLASS\_PIMA \*pima,

ULONG object\_handle, UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*\*object)

{

UX\_PICTBRIDGE \*pictbridge;

UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*object\_info;

/\* Get the pointer to the Pictbridge instance. \*/

pictbridge = (UX\_PICTBRIDGE \*)pima -> ux\_device\_class\_pima\_application;

/\* Check the object handle. If this is handle 1 or 2 , we need to return

the XML script object.

If the handle is not 1 or 2, this is a JPEG picture or other object to

be printed. \*/

if ((object\_handle == UX\_PICTBRIDGE\_OBJECT\_HANDLE\_HOST\_RESPONSE) ||

(object\_handle == UX\_PICTBRIDGE\_OBJECT\_HANDLE\_CLIENT\_REQUEST))

{

/\* Check what XML object is requested. It is either a request script

or a response. \*/

if (object\_handle == UX\_PICTBRIDGE\_OBJECT\_HANDLE\_HOST\_RESPONSE)

object\_info = (UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*) pictbridge ->

ux\_pictbridge\_object\_host;

else

object\_info = (UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*) pictbridge ->

ux\_pictbridge\_object\_client;

}

else

/\* Get the object info from the job info structure. \*/

object\_info = (UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*) pictbridge ->

ux\_pictbridge\_jobinfo.ux\_pictbridge\_jobinfo\_object;

/\* Return the pointer to this object. \*/

\*object = object\_info;

/\* We are done. \*/

return(UX\_SUCCESS);

}

### ux\_device\_class\_pima\_object\_data\_get

Return the object data

**Prototype**

UINT **ux\_device\_class\_pima\_object\_info\_get**(UX\_SLAVE\_CLASS\_PIMA \*pima,

ULONG object\_handle, UCHAR \*object\_buffer, ULONG object\_offset,

ULONG object\_length\_requested, ULONG \*object\_actual\_length)

**Description**

This function is called when the PIMA class needs to retrieve the object data in the local system and send it back to the host.

**Parameters**

**pima** Pointer to the pima class instance.

**object\_handle** Handle of the object

**object\_buffer** Object buffer address

**object\_length\_requested** Object data length requested by the client to the application

**object\_actual\_length** Object data length returned by the application

**Example**

UINT ux\_pictbridge\_dpsclient\_object\_data\_get(UX\_SLAVE\_CLASS\_PIMA \*pima,

ULONG object\_handle, UCHAR \*object\_buffer, ULONG object\_offset,

ULONG object\_length\_requested, ULONG \*object\_actual\_length)

{

UX\_PICTBRIDGE \*pictbridge;

UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*object\_info;

UCHAR \*pima\_object\_buffer;

ULONG actual\_length;

UINT status;

/\* Get the pointer to the Pictbridge instance. \*/

pictbridge = (UX\_PICTBRIDGE \*)pima -> ux\_device\_class\_pima\_application;

/\* Check the object handle. If this is handle 1 or 2 , we need to return

the XML script object.

If the handle is not 1 or 2, this is a JPEG picture or other object to

be printed. \*/

if ((object\_handle == UX\_PICTBRIDGE\_OBJECT\_HANDLE\_HOST\_RESPONSE) ||

(object\_handle == UX\_PICTBRIDGE\_OBJECT\_HANDLE\_CLIENT\_REQUEST))

{

/\* Check what XML object is requested. It is either a request script

or a response. \*/

if (object\_handle == UX\_PICTBRIDGE\_OBJECT\_HANDLE\_HOST\_RESPONSE)

object\_info = (UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*) pictbridge ->

ux\_pictbridge\_object\_host;

else

object\_info = (UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*) pictbridge ->

ux\_pictbridge\_object\_client;

/\* Is this the corrent handle ? \*/

if (object\_info -> ux\_device\_class\_pima\_object\_handle\_id ==

object\_handle)

{

/\* Get the pointer to the object buffer. \*/

pima\_object\_buffer = object\_info ->

ux\_device\_class\_pima\_object\_buffer;

/\* Copy the demanded object data portion. \*/

ux\_utility\_memory\_copy(object\_buffer, pima\_object\_buffer +

object\_offset, object\_length\_requested);

/\* Update the length requested. for a demo, we do not do any

checking. \*/

\*object\_actual\_length = object\_length\_requested;

/\* What cycle are we in ? \*/

if (pictbridge -> ux\_pictbridge\_host\_client\_state\_machine &

UX\_PICTBRIDGE\_STATE\_MACHINE\_HOST\_REQUEST)

{

/\* Check if we are blocking for a client request. \*/

if (pictbridge -> ux\_pictbridge\_host\_client\_state\_machine &

UX\_PICTBRIDGE\_STATE\_MACHINE\_CLIENT\_REQUEST\_PENDING)

/\* Yes we are pending, send an event to release the

pending request. \*/

ux\_utility\_event\_flags\_set(&pictbridge ->

ux\_pictbridge\_event\_flags\_group,

UX\_PICTBRIDGE\_EVENT\_FLAG\_STATE\_MACHINE\_READY, TX\_OR);

/\* Since we are in host request, this indicates we are done

with the cycle. \*/

pictbridge -> ux\_pictbridge\_host\_client\_state\_machine =

UX\_PICTBRIDGE\_STATE\_MACHINE\_IDLE;

}

/\* We have copied the requested data. Return OK. \*/

return(UX\_SUCCESS);

}

}

else

{

/\* Get the object info from the job info structure. \*/

object\_info = (UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*) pictbridge ->

ux\_pictbridge\_jobinfo.ux\_pictbridge\_jobinfo\_object;

/\* Obtain the data from the application jobinfo callback. \*/

status = pictbridge ->

ux\_pictbridge\_jobinfo.

ux\_pictbridge\_jobinfo\_object\_data\_read(pictbridge,

object\_buffer, object\_offset,

object\_length\_requested, &actual\_length);

/\* Save the length returned. \*/

\*object\_actual\_length = actual\_length;

/\* Return the application status. \*/

return(status);

}

/\* Could not find the handle. \*/

return(UX\_DEVICE\_CLASS\_PIMA\_RC\_INVALID\_OBJECT\_HANDLE);

}

### ux\_device\_class\_pima\_object\_info\_send

Host sends the object information

**Prototype**

UINT **ux\_device\_class\_pima\_object\_info\_send**(UX\_SLAVE\_CLASS\_PIMA \*pima,

UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*object, ULONG \*object\_handle)

**Description**

This function is called when the PIMA class needs to receive the object information in the local system for future storage.

**Parameters**

**pima** Pointer to the pima class instance.

**object** Pointer to the object

**object\_handle** Handle of the object

**Example**

UINT **ux\_pictbridge\_dpsclient\_object\_info\_send**(UX\_SLAVE\_CLASS\_PIMA \*pima, UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*object, ULONG \*object\_handle)

{

UX\_PICTBRIDGE \*pictbridge;

UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*object\_info;

UCHAR string\_discovery\_name[UX\_PICTBRIDGE\_MAX\_FILE\_NAME\_SIZE];

/\* Get the pointer to the Pictbridge instance. \*/

pictbridge = (UX\_PICTBRIDGE \*)pima -> ux\_device\_class\_pima\_application;

/\* We only have one object. \*/

object\_info = (UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*) pictbridge ->

ux\_pictbridge\_object\_host;

/\* Copy the demanded object info set. \*/

ux\_utility\_memory\_copy(object\_info, object,

UX\_SLAVE\_CLASS\_PIMA\_OBJECT\_DATA\_LENGTH);

/\* Store the object handle. In Pictbridge we only receive XML scripts so

the handle is hardwired to 1. \*/

object\_info -> ux\_device\_class\_pima\_object\_handle\_id = 1;

\*object\_handle = 1;

/\* Check state machine. If we are in discovery pending mode, check file

name of this object. \*/

if (pictbridge -> ux\_pictbridge\_discovery\_state ==

UX\_PICTBRIDGE\_DPSCLIENT\_DISCOVERY\_PENDING)

{

/\* We are in the discovery mode. Check for file name. It must match

HDISCVRY.DPS in Unicode mode. \*/

/\* Check if this is a script. \*/

if (object\_info -> ux\_device\_class\_pima\_object\_format ==

UX\_DEVICE\_CLASS\_PIMA\_OFC\_SCRIPT)

{

/\* Yes this is a script. We need to search for the HDISCVRY.DPS

file name. Get the file name in a ascii format. \*/

ux\_utility\_unicode\_to\_string(object\_info ->

ux\_device\_class\_pima\_object\_filename,

string\_discovery\_name);

/\* Now, compare it to the HDISCVRY.DPS file name. Check length

first. \*/

if (ux\_utility\_string\_length\_get(\_ux\_pictbridge\_hdiscovery\_name)

== ux\_utility\_string\_length\_get(string\_discovery\_name))

{

/\* So far, the length of name of the files are the same.

Compare names now. \*/

if(ux\_utility\_memory\_compare(  
 \_ux\_pictbridge\_hdiscovery\_name,

string\_discovery\_name,

ux\_utility\_string\_length\_get(string\_discovery\_name))

== UX\_SUCCESS)

{

/\* We are done with discovery of the printer. We can now

send notifications when the camera wants to print an

object. \*/

pictbridge -> ux\_pictbridge\_discovery\_state =

UX\_PICTBRIDGE\_DPSCLIENT\_DISCOVERY\_COMPLETE;

/\* Set an event flag if the application is listening. \*/

ux\_utility\_event\_flags\_set(&pictbridge ->

ux\_pictbridge\_event\_flags\_group,

UX\_PICTBRIDGE\_EVENT\_FLAG\_DISCOVERY, TX\_OR);

/\* There is no object during th discovery cycle. \*/

return(UX\_SUCCESS);

}

}

}

}

/\* What cycle are we in ? \*/

if (pictbridge -> ux\_pictbridge\_host\_client\_state\_machine ==

UX\_PICTBRIDGE\_STATE\_MACHINE\_IDLE)

/\* Since we are in idle state, we must have received a request from

the host. \*/

pictbridge -> ux\_pictbridge\_host\_client\_state\_machine =

UX\_PICTBRIDGE\_STATE\_MACHINE\_HOST\_REQUEST;

/\* We have copied the requested data. Return OK. \*/

return(UX\_SUCCESS);

}

### ux\_device\_class\_pima\_object\_data\_send

Host sends the object data

**Prototype**

UINT **ux\_device\_class\_pima\_object\_data\_send**(UX\_SLAVE\_CLASS\_PIMA \*pima,

ULONG object\_handle, ULONG phase, UCHAR \*object\_buffer,

ULONG object\_offset, ULONG object\_length)

**Description**

This function is called when the PIMA class needs to receive the object data in the local system for storage.

**Parameters**

**pima** Pointer to the pima class instance.

**object\_handle** Handle of the object

**phase** phase of the transfer (active or complete)

**object\_buffer** Object buffer address

**object\_offset** Address of data

**object\_length** Object data length sent by application

**Example**

UINT **ux\_pictbridge\_dpsclient\_object\_data\_send**(UX\_SLAVE\_CLASS\_PIMA \*pima,

ULONG object\_handle,

ULONG phase,

UCHAR \*object\_buffer,

ULONG object\_offset,

ULONG object\_length)

{

UINT status;

UX\_PICTBRIDGE \*pictbridge;

UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*object\_info;

ULONG event\_flag;

UCHAR \*pima\_object\_buffer;

/\* Get the pointer to the Pictbridge instance. \*/

pictbridge = (UX\_PICTBRIDGE \*)pima -> ux\_device\_class\_pima\_application;

/\* Get the pointer to the pima object. \*/

object\_info = (UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*) pictbridge ->

ux\_pictbridge\_object\_host;

/\* Is this the corrent handle ? \*/

if (object\_info -> ux\_device\_class\_pima\_object\_handle\_id ==

object\_handle)

{

/\* Get the pointer to the object buffer. \*/

pima\_object\_buffer = object\_info ->

ux\_device\_class\_pima\_object\_buffer;

/\* Check the phase. We should wait for the object to be completed and

the response sent back before parsing the object. \*/

if (phase == UX\_DEVICE\_CLASS\_PIMA\_OBJECT\_TRANSFER\_PHASE\_ACTIVE)

{

/\* Copy the demanded object data portion. \*/

ux\_utility\_memory\_copy(pima\_object\_buffer + object\_offset,

object\_buffer, object\_length);

/\* Save the length of this object. \*/

object\_info -> ux\_device\_class\_pima\_object\_length =

object\_length;

/\* We are not done yet. \*/

return(UX\_SUCCESS);

}

else

{

/\* Completion of transfer. We are done. \*/

return(UX\_SUCCESS);

}

}

}

### ux\_device\_class\_pima\_object\_delete

Delete a local object

**Prototype**

UINT **ux\_device\_class\_pima\_object\_delete**(UX\_SLAVE\_CLASS\_PIMA \*pima,

ULONG object\_handle)

**Description**

This function is called when the PIMA class needs to delete an object on the local storage.

**Parameters**

**pima** Pointer to the pima class instance.

**object\_handle** Handle of the object

**Example**

UINT **ux\_pictbridge\_dpsclient\_object\_delete**(UX\_SLAVE\_CLASS\_PIMA \*pima,

ULONG object\_handle)

{

/\* Delete the object pointer by the handle. \*/

}

## USB Device Audio Class

The USB device Audio class allows for a USB host system to communicate with the device as an audio device. This class is based on the USB standard and USB Audio Class 1.0 or 2.0 standard.

A USB audio compliant device framework needs to be declared by the device stack. An example of an Audio 2.0 speaker follows:

unsigned char device\_framework\_high\_speed[] = {

/\* --- Device Descriptor 18 bytes

0x00 bDeviceClass: Refer to interface

0x00 bDeviceSubclass: Refer to interface

0x00 bDeviceProtocol: Refer to interface

idVendor & idProduct - http://www.linux-usb.org/usb.ids

\*/

/\* 0 bLength, bDescriptorType \*/ 18, 0x01,

/\* 2 bcdUSB : 0x200 (2.00) \*/ 0x00, 0x02,

/\* 4 bDeviceClass : 0x00 (see interface) \*/ 0x00,

/\* 5 bDeviceSubClass : 0x00 (see interface) \*/ 0x00,

/\* 6 bDeviceProtocol : 0x00 (see interface) \*/ 0x00,

/\* 7 bMaxPacketSize0 \*/ 0x08,

/\* 8 idVendor, idProduct \*/ 0x84, 0x84, 0x03, 0x00,

/\* 12 bcdDevice \*/ 0x00, 0x02,

/\* 14 iManufacturer, iProduct, iSerialNumber \*/ 0, 0, 0,

/\* 17 bNumConfigurations \*/ 1,

/\* ---------------- Device Qualifier Descriptor \*/

/\* 0 bLength, bDescriptorType \*/ 10, 0x06,

/\* 2 bcdUSB : 0x200 (2.00) \*/ 0x00,0x02,

/\* 4 bDeviceClass : 0x00 (see interface) \*/ 0x00,

/\* 5 bDeviceSubClass : 0x00 (see interface) \*/ 0x00,

/\* 6 bDeviceProtocol : 0x00 (see interface) \*/ 0x00,

/\* 7 bMaxPacketSize0 \*/ 8,

/\* 8 bNumConfigurations \*/ 1,

/\* 9 bReserved \*/ 0,

/\* --- Configuration Descriptor (9+8+73+55=145, 0x91) \*/

/\* 0 bLength, bDescriptorType \*/ 9, 0x02,

/\* 2 wTotalLength \*/ 145, 0,

/\* 4 bNumInterfaces, bConfigurationValue \*/ 2, 1,

/\* 6 iConfiguration \*/ 0,

/\* 7 bmAttributes, bMaxPower \*/ 0x80, 50,

/\* ----------- Interface Association Descriptor \*/

/\* 0 bLength, bDescriptorType \*/ 8, 0x0B,

/\* 2 bFirstInterface, bInterfaceCount \*/ 0, 2,

/\* 4 bFunctionClass : 0x01 (Audio) \*/ 0x01,

/\* 5 bFunctionSubClass : 0x00 (UNDEFINED) \*/ 0x00,

/\* 6 bFunctionProtocol : 0x20 (VERSION\_02\_00) \*/ 0x20,

/\* 7 iFunction \*/ 0,

/\* --- Interface Descriptor #0: Control (9+64=73) \*/

/\* 0 bLength, bDescriptorType \*/ 9, 0x04,

/\* 2 bInterfaceNumber, bAlternateSetting \*/ 0, 0,

/\* 4 bNumEndpoints \*/ 0,

/\* 5 bInterfaceClass : 0x01 (Audio) \*/ 0x01,

/\* 6 bInterfaceSubClass : 0x01 (AudioControl) \*/ 0x01,

/\* 7 bInterfaceProtocol : 0x20 (VERSION\_02\_00) \*/ 0x20,

/\* 8 iInterface \*/ 0,

/\* --- Audio 2.0 AC Interface Header Descriptor (9+8+17+18+12=64, 0x40) \*/

/\* 0 bLength \*/ 9,

/\* 1 bDescriptorType, bDescriptorSubtype \*/ 0x24, 0x01,

/\* 3 bcdADC \*/ 0x00, 0x02,

/\* 5 bCategory : 0x08 (IO Box) \*/ 0x08,

/\* 6 wTotalLength \*/ 64, 0,

/\* 8 bmControls \*/ 0x00,

/\* -------- Audio 2.0 AC Clock Source Descriptor \*/

/\* 0 bLength \*/ 8,

/\* 1 bDescriptorType, bDescriptorSubtype \*/ 0x24, 0x0A,

/\* 3 bClockID \*/ 0x10,

/\* 4 bmAttributes : 0x05 (Sync|InternalFixedClk) \*/ 0x05,

/\* 5 bmControls : 0x01 (FreqReadOnly) \*/ 0x01,

/\* 6 bAssocTerminal, iClockSource \*/ 0x00, 0,

/\* ------ Audio 2.0 AC Input Terminal Descriptor \*/

/\* 0 bLength \*/ 17,

/\* 1 bDescriptorType, bDescriptorSubtype \*/ 0x24, 0x02,

/\* 3 bTerminalID \*/ 0x04,

/\* 4 wTerminalType : 0x0101 (USB Streaming) \*/ 0x01, 0x01,

/\* 6 bAssocTerminal, bCSourceID \*/ 0x00, 0x10,

/\* 8 bNrChannels \*/ 2,

/\* 9 bmChannelConfig \*/ 0x00, 0x00, 0x00, 0x00,

/\* 13 iChannelNames, bmControls, iTerminal \*/ 0, 0x00, 0x00, 0,

/\* -------- Audio 2.0 AC Feature Unit Descriptor \*/

/\* 0 bLength \*/ 18,

/\* 1 bDescriptorType, bDescriptorSubtype \*/ 0x24, 0x06,

/\* 3 bUnitID, bSourceID \*/ 0x05, 0x04,

/\* 5 bmaControls(0) : 0x0F (VolumeRW|MuteRW) \*/ 0x0F, 0x00, 0x00, 0x00,

/\* 9 bmaControls(1) : 0x00000000 \*/ 0x00, 0x00, 0x00, 0x00,

/\* 13 bmaControls(1) : 0x00000000 \*/ 0x00, 0x00, 0x00, 0x00,

/\* . iFeature \*/ 0,

/\* ----- Audio 2.0 AC Output Terminal Descriptor \*/

/\* 0 bLength \*/ 12,

/\* 1 bDescriptorType, bDescriptorSubtype \*/ 0x24, 0x03,

/\* 3 bTerminalID \*/ 0x06,

/\* 4 wTerminalType : 0x0301 (Speaker) \*/ 0x01, 0x03,

/\* 6 bAssocTerminal, bSourceID, bCSourceID \*/ 0x00, 0x05, 0x10,

/\* 9 bmControls, iTerminal \*/ 0x00, 0x00, 0,

/\* --- Interface Descriptor #1: Stream OUT (9+9+16+6+7+8=55) \*/

/\* 0 bLength, bDescriptorType \*/ 9, 0x04,

/\* 2 bInterfaceNumber, bAlternateSetting \*/ 1, 0,

/\* 4 bNumEndpoints \*/ 0,

/\* 5 bInterfaceClass : 0x01 (Audio) \*/ 0x01,

/\* 6 bInterfaceSubClass : 0x01 (AudioStream) \*/ 0x02,

/\* 7 bInterfaceProtocol : 0x20 (VERSION\_02\_00) \*/ 0x20,

/\* 8 iInterface \*/ 0,

/\* ----------------------- Interface Descriptor \*/

/\* 0 bLength, bDescriptorType \*/ 9, 0x04,

/\* 2 bInterfaceNumber, bAlternateSetting \*/ 1, 1,

/\* 4 bNumEndpoints \*/ 1,

/\* 5 bInterfaceClass : 0x01 (Audio) \*/ 0x01,

/\* 6 bInterfaceSubClass : 0x01 (AudioStream) \*/ 0x02,

/\* 7 bInterfaceProtocol : 0x20 (VERSION\_02\_00) \*/ 0x20,

/\* 8 iInterface \*/ 0,

/\* ----------- Audio 2.0 AS Interface Descriptor \*/

/\* 0 bLength \*/ 16,

/\* 1 bDescriptorType, bDescriptorSubtype \*/ 0x24, 0x01,

/\* 3 bTerminalLink, bmControls \*/ 0x04, 0x00,

/\* 5 bFormatType : 0x01 (FORMAT\_TYPE\_I) \*/ 0x01,

/\* 6 bmFormats : 0x000000001 (PCM) \*/ 0x01, 0x00, 0x00, 0x00,

/\* 10 bNrChannels \*/ 2,

/\* 11 bmChannelConfig \*/ 0x00, 0x00, 0x00, 0x00,

/\* 15 iChannelNames \*/ 0,

/\* --------- Audio 2.0 AS Format Type Descriptor \*/

/\* 0 bLength \*/ 6,

/\* 1 bDescriptorType, bDescriptorSubtype \*/ 0x24, 0x02,

/\* 3 bFormatType : 0x01 (FORMAT\_TYPE\_I) \*/ 0x01,

/\* 4 bSubslotSize, bBitResolution \*/ 2, 16,

/\* ------------------------- Endpoint Descriptor \*/

/\* 0 bLength, bDescriptorType \*/ 7, 0x05,

/\* 2 bEndpointAddress \*/ 0x02,

/\* 3 bmAttributes : 0x0D (Sync|ISO) \*/ 0x0D,

/\* 4 wMaxPacketSize : 0x0100 (256) \*/ 0x00, 0x01,

/\* 6 bInterval : 0x04 (1ms) \*/ 4,

/\* - Audio 2.0 AS ISO Audio Data Endpoint Descriptor \*/

/\* 0 bLength \*/ 8,

/\* 1 bDescriptorType, bDescriptorSubtype \*/ 0x25, 0x01,

/\* 3 bmAttributes, bmControls \*/ 0x00, 0x00,

/\* 5 bLockDelayUnits, wLockDelay \*/ 0x00, 0x00, 0x00,

};

The Audio class uses a composite device framework to group interfaces (control and streaming). As a result care should be taken when defining the device descriptor. USBX relies on the IAD descriptor to know internally how to bind interfaces. The IAD descriptor should be declared before the interfaces (an AudioControl interface followed by one or more AudioStreaming interfaces) and contain the first interface of the Audio class (the AudioControl interface) and how many interfaces are attached.

The way the audio class works depends on whether the device is sending or receiving audio, but both cases use a FIFO for storing audio frame buffers: if the device is sending audio to the host, then the application adds audio frame buffers to the FIFO which are later sent to the host by USBX; if the device is receiving audio from the host, then USBX adds the audio frame buffers received from the host to the FIFO which are later read by the application. Each audio stream has its own FIFO, and each audio frame buffer consists of multiple samples.

The initialization of the Audio class expects the following parts:

1. Audio class expects the following streaming parameters:

/\* Set the parameters for Audio streams. \*/

/\* Set the application-defined callback that is invoked when the host requests a change to the alternate setting. \*/

audio\_stream\_parameter[0].ux\_device\_class\_audio\_stream\_parameter\_callbacks.ux\_device\_class\_audio\_stream\_change = demo\_audio\_read\_change;

/\* Set the application-defined callback that is invoked whenever a USB packet (audio frame) is sent to or received from the host. \*/

audio\_stream\_parameter[0].ux\_device\_class\_audio\_stream\_parameter\_callbacks.ux\_device\_class\_audio\_stream\_frame\_done = demo\_audio\_read\_done;

/\* Set the number of audio frame buffers in the FIFO. \*/

audio\_stream\_parameter[0].ux\_device\_class\_audio\_stream\_parameter\_max\_frame\_buffer\_nb = UX\_DEMO\_FRAME\_BUFFER\_NB;

/\* Set the maximum size of each audio frame buffer in the FIFO. \*/

audio\_stream\_parameter[0].ux\_device\_class\_audio\_stream\_parameter\_max\_frame\_buffer\_size = UX\_DEMO\_MAX\_FRAME\_SIZE;

/\* Set the internally-defined audio processing thread entry pointer. If the application wishes to receive audio from the host (which is the case in this example), ux\_device\_class\_audio\_read\_thread\_entry should be used; if the application wishes to send data to the host, ux\_device\_class\_audio\_write\_thread\_entry should be used. \*/

audio\_stream\_parameter[0].ux\_device\_class\_audio\_stream\_parameter\_thread\_entry = ux\_device\_class\_audio\_read\_thread\_entry;

1. Audio class expects the following function parameters:

/\* Set the parameters for Audio device. \*/

/\* Set the number of streams. \*/

audio\_parameter.ux\_device\_class\_audio\_parameter\_streams\_nb = 1;

/\* Set the pointer to the first audio stream parameter. Note that we initialized this parameter in the previous section. Also note that for more than one streams, this should be an array. \*/

audio\_parameter.ux\_device\_class\_audio\_parameter\_streams = audio\_stream\_parameter;

/\* Set the application-defined callback that is invoked when the audio class is activated i.e. device is connected to host. \*/

audio\_parameter.ux\_device\_class\_audio\_parameter\_callbacks.ux\_slave\_class\_audio\_instance\_activate = demo\_audio\_instance\_activate;

/\* Set the application-defined callback that is invoked when the audio class is deactivated i.e. device is disconnected from host. \*/

audio\_parameter.ux\_device\_class\_audio\_parameter\_callbacks.ux\_slave\_class\_audio\_instance\_deactivate = demo\_audio\_instance\_deactivate;

/\* Set the application-defined callback that is invoked when the stack receives a control request from the host. See below for more details. \*/

audio\_parameter.ux\_device\_class\_audio\_parameter\_callbacks.ux\_device\_class\_audio\_control\_process = demo\_audio20\_request\_process;

/\* Initialize the device Audio class. This class owns interfaces starting with 0. \*/

status = ux\_device\_stack\_class\_register(\_ux\_system\_slave\_class\_cdc\_acm\_name, ux\_device\_class\_audio\_entry, 1, 0, &audio\_parameter);

if(status!=UX\_SUCCESS)

return;

The application-defined control request callback (ux\_device\_class\_audio\_control\_process; set in the previous example) is invoked when the stack receives a control request from the host. If the request is accepted and handled (acknowledged or stalled) the callback must return success, otherwise error should be returned.

The class-specific control request process is defined as an application-defined callback because the control requests are very different between USB Audio versions and a large part of the request process relates to the device framework. The application should handle requests correctly to make the device functional.

Since for an audio device, volume, mute and sampling frequency are common control requests, simple, internally-defined callbacks for different USB audio versions are introduced in later sections for applications to use. Refer to ux\_device\_class\_audio10\_control\_process and ux\_device\_class\_audio\_control\_request for more details.

In the device framework of the Audio device, the PID/VID are stored in the device descriptor (see the device descriptor declared above).

When a USB host system discovers the USB Audio device and mounts the audio class, the device can be used with any audio player or recorder (depending on the framework). See the host Operating System for reference.

The Audio class APIs are defined below:

### ux\_device\_class\_audio\_read\_thread\_entry

Thread entry for reading data for the Audio function

**Prototype**

VOID **ux\_device\_class\_audio\_read\_thread\_entry**(ULONG audio\_stream)

**Description**

This function is passed to the audio stream initialization parameter if reading audio from the host is desired. Internally, a thread is created with this function as its entry function; the thread itself reads audio data through the isochronous OUT endpoint in the Audio function.

**Parameters**

**audio\_stream** Pointer to the audio stream instance.

**Example**

/\* Set parameter to initialize a stream for reading. \*/

audio\_stream\_parameter[0].ux\_device\_class\_audio\_stream\_parameter\_thread\_entry = ux\_device\_class\_audio\_read\_thread\_entry;

### ux\_device\_class\_audio\_write\_thread\_entry

Thread entry for writing data for the Audio function

**Prototype**

VOID **ux\_device\_class\_audio\_write\_thread\_entry**(ULONG audio\_stream)

**Description**

This function is passed to the audio stream initialization parameter if writing audio to the host is desired. Internally, a thread is created with this function as its entry function; the thread itself writes audio data through the isochronous IN endpoint in the Audio function.

**Parameters**

**audio\_stream** Pointer to the audio stream instance.

**Example**

/\* Set parameter to initialize as stream for writing. \*/

audio\_stream\_parameter[0].ux\_device\_class\_audio\_stream\_parameter\_thread\_entry = ux\_device\_class\_audio\_write\_thread\_entry;

### ux\_device\_class\_audio\_stream\_get

Get specific stream instance for the Audio function

**Prototype**

UINT **ux\_device\_class\_audio\_stream\_get**(UX\_DEVICE\_CLASS\_AUDIO \*audio, ULONG stream\_index, UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*\*stream)

**Description**

This function is used to get a stream instance of audio class.

**Parameters**

**audio** Pointer to the audio instance.

**stream\_index** Stream instance index based on 0.

**stream** Pointer to buffer to store the audio stream instance pointer.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Get audio stream instance. \*/

status = ux\_device\_class\_audio\_stream\_get(audio, 0, &stream);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio\_reception\_start

Start audio data reception for the Audio stream

**Prototype**

UINT **ux\_device\_class\_audio\_reception\_start**(UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*stream)

**Description**

This function is used to start audio data reading in audio streams.

**Parameters**

**stream** Pointer to the audio stream instance.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | The interface is down. |
| **UX\_BUFFER\_OVERFLOW** | (0x5d) | FIFO buffer is full. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Start stream data reception. \*/

status = ux\_device\_class\_audio\_reception\_start(stream);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio\_sample\_read8

Read 8-bit sample from the Audio stream

**Prototype**

UINT **ux\_device\_class\_audio\_sample\_read8**(UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*stream, UCHAR \*buffer)

**Description**

This function reads 8-bit audio sample data from the specified stream. Specifically, it reads the sample data from the current audio frame buffer in the FIFO. Upon reading the last sample in an audio frame, the frame will be automatically freed so that it can be used to accept more data from the host.

**Parameters**

**stream** Pointer to the audio stream instance.

**buffer** Pointer to the buffer to save sample byte.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | The interface is down. |
| **UX\_BUFFER\_OVERFLOW** | (0x5d) | FIFO buffer is null. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Read a byte in audio FIFO. \*/

status = ux\_device\_class\_audio\_sample\_read8(stream, &sample\_byte);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio\_sample\_read16

Read 16-bit sample from the Audio stream

**Prototype**

UINT **ux\_device\_class\_audio\_sample\_read16**(UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*stream, USHORT \*buffer)

**Description**

This function reads 16-bit audio sample data from the specified stream. Specifically, it reads the sample data from the current audio frame buffer in the FIFO. Upon reading the last sample in an audio frame, the frame will be automatically freed so that it can be used to accept more data from the host.

**Parameters**

**stream** Pointer to the audio stream instance.

**buffer** Pointer to the buffer to save the 16-bit sample.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | The interface is down. |
| **UX\_BUFFER\_OVERFLOW** | (0x5d) | FIFO buffer is null. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Read a 16-bit sample in audio FIFO. \*/

status = ux\_device\_class\_audio\_sample\_read16(stream, &sample\_word);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio\_sample\_read24

Read 24-bit sample from the Audio stream

**Prototype**

UINT **ux\_device\_class\_audio\_sample\_read24**(UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*stream, ULONG \*buffer)

**Description**

This function reads 24-bit audio sample data from the specified stream. Specifically, it reads the sample data from the current audio frame buffer in the FIFO. Upon reading the last sample in an audio frame, the frame will be automatically freed so that it can be used to accept more data from the host.

**Parameters**

**stream** Pointer to the audio stream instance.

**buffer** Pointer to the buffer to save the 3-byte sample.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | The interface is down. |
| **UX\_BUFFER\_OVERFLOW** | (0x5d) | FIFO buffer is null. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Read 3 bytes to in audio FIFO. \*/

status = ux\_device\_class\_audio\_sample\_read24(stream, &sample\_bytes);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio\_sample\_read32

Read 32-bit sample from the Audio stream

**Prototype**

UINT **ux\_device\_class\_audio\_sample\_read32**(UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*stream, ULONG \*buffer)

**Description**

This function reads 32-bit audio sample data from the specified stream. Specifically, it reads the sample data from the current audio frame buffer in the FIFO. Upon reading the last sample in an audio frame, the frame will be automatically freed so that it can be used to accept more data from the host.

**Parameters**

**stream** Pointer to the audio stream instance.

**buffer** Pointer to the buffer to save the 4-byte data.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | The interface is down. |
| **UX\_BUFFER\_OVERFLOW** | (0x5d) | FIFO buffer is null. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Read 4 bytes in audio FIFO. \*/

status = ux\_device\_class\_audio\_sample\_read32(stream, &sample\_bytes);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio\_read\_frame\_get

Get access to audio frame in the Audio stream

**Prototype**

UINT **ux\_device\_class\_audio\_read\_frame\_get**(UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*stream, UCHAR \*\*frame\_data, ULONG \*frame\_length)

**Description**

This function returns the first audio frame buffer and its length in the specified stream’s FIFO. When the application is done processing the data, ux\_device\_class\_audio\_read\_frame\_free must be used to free the frame buffer in the FIFO.

**Parameters**

**stream** Pointer to the audio stream instance.

**frame\_data** Pointer to data pointer to return the data pointer in.

**frame\_length** Pointer to buffer to save the frame length in number of bytes.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | The interface is down. |
| **UX\_BUFFER\_OVERFLOW** | (0x5d) | FIFO buffer is null. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Get frame access. \*/

status = ux\_device\_class\_audio\_read\_frame\_get(stream, &frame, &frame\_length);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio\_read\_frame\_free

Free an audio frame buffer in Audio stream

**Prototype**

UINT **ux\_device\_class\_audio\_read\_frame\_free**(UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*stream)

**Description**

This function frees the audio frame buffer at the front of the specified stream’s FIFO so that it can receive data from the host.

**Parameters**

**stream** Pointer to the audio stream instance.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | The interface is down. |
| **UX\_BUFFER\_OVERFLOW** | (0x5d) | FIFO buffer is null. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Refree a frame buffer in FIFO. \*/

status = ux\_device\_class\_audio\_read\_frame\_free(stream);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio\_transmission\_start

Start audio data transmission for the Audio stream

**Prototype**

UINT **ux\_device\_class\_audio\_transmission\_start**(UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*stream)

**Description**

This function is used to start sending audio data written to the FIFO in the audio class.

**Parameters**

**stream** Pointer to the audio stream instance.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | The interface is down. |
| **UX\_BUFFER\_OVERFLOW** | (0x5d) | FIFO buffer is null. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Start stream data transmission. \*/

status = ux\_device\_class\_audio\_transmission\_start(stream);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio\_frame\_write

Write an audio frame into the Audio stream

**Prototype**

UINT **ux\_device\_class\_audio\_frame\_write**(UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*stream, UCHAR \*frame, ULONG frame\_length)

**Description**

This function writes a frame to the audio stream’s FIFO. The frame data is copied to the available buffer in the FIFO so that it can be sent to the host.

**Parameters**

**stream** Pointer to the audio stream instance.

**frame** Pointer to frame data.

**frame\_length** Frame length in number of bytes .

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | The interface is down. |
| **UX\_BUFFER\_OVERFLOW** | (0x5d) | FIFO buffer is full. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Get frame access. \*/

status = ux\_device\_class\_audio\_frame\_write(stream, frame, frame\_length);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio\_write\_frame\_get

Get access to audio frame in the Audio stream

**Prototype**

UINT **ux\_device\_class\_audio\_write\_frame\_get**(UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*stream, UCHAR \*\*frame\_data, ULONG \*frame\_length)

**Description**

This function retrieves the address of the last audio frame buffer of the FIFO; it also retrieves the length of the audio frame buffer. After the application fills the audio frame buffer with its desired data, ux\_device\_class\_audio\_write\_frame\_commit must be used to add/commit the frame buffer to the FIFO.

**Parameters**

**stream** Pointer to the audio stream instance.

**frame\_data** Pointer to frame data pointer to return the frame data pointer in.

**frame\_length** Pointer to the buffer to save frame length in number of bytes .

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | The interface is down. |
| **UX\_BUFFER\_OVERFLOW** | (0x5d) | FIFO buffer is full. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Get frame access. \*/

status = ux\_device\_class\_audio\_write\_frame\_get(stream, &frame, &frame\_length);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio\_write\_frame\_commit

Commit an audio frame buffer in Audio stream

**Prototype**

UINT **ux\_device\_class\_audio\_write\_frame\_commit**(UX\_DEVICE\_CLASS\_AUDIO\_STREAM \*stream, ULONG length)

**Description**

This function adds/commits the last audio frame buffer to the FIFO so the buffer is ready to be transferred to host; note the last audio frame buffer should have been filled out via ux\_device\_class\_write\_frame\_get.

**Parameters**

**stream** Pointer to the audio stream instance.

**length** Number of bytes ready in the buffer.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_CONFIGURATION\_HANDLE\_UNKNOWN** | (0x51) | The interface is down. |
| **UX\_BUFFER\_OVERFLOW** | (0x5d) | FIFO buffer is fssull. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Commit a frame after fill values in buffer. \*/

status = ux\_device\_class\_audio\_write\_frame\_commit(stream, 192);

if(status != UX\_SUCCESS)

return;

### ux\_device\_class\_audio10\_control\_process

Process USB Audio 1.0 control requests

**Prototype**

UINT **ux\_device\_class\_audio10\_control\_process**(UX\_DEVICE\_CLASS\_AUDIO \*audio, UX\_SLAVE\_TRANSFER \*transfer\_request, UX\_DEVICE\_CLASS\_AUDIO10\_CONTROL\_GROUP \*group)

**Description**

This function manages basic requests sent by the host on the control endpoint with a USB Audio 1.0 specific type.

Audio 1.0 features of volume and mute requests are processed in the function. When processing the requests, pre-defined data passed by the last parameter (group) is used to answer requests and store control changes.

**Parameters**

**audio** Pointer to the audio instance.

**transfer** Pointer to the transfer request instance.

**group** Data group for request process.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Initialize audio 1.0 control values. \*/

audio\_control[0].ux\_device\_class\_audio10\_control\_fu\_id = 2;

audio\_control[0].ux\_device\_class\_audio10\_control\_mute[0] = 0;

audio\_control[0].ux\_device\_class\_audio10\_control\_volume[0] = 0;

audio\_control[1].ux\_device\_class\_audio10\_control\_fu\_id = 5;

audio\_control[1].ux\_device\_class\_audio10\_control\_mute[0] = 0;

audio\_control[1].ux\_device\_class\_audio10\_control\_volume[0] = 0;

/\* Handle request and update control values.

Note here only mute and volume for master channel is supported.

\*/

status = ux\_device\_class\_audio10\_control\_process(audio, transfer, &group);

if (status == UX\_SUCCESS)

{

/\* Request handled, check changes \*/

switch(audio\_control[0].ux\_device\_class\_audio10\_control\_changed)

{

case UX\_DEVICE\_CLASS\_AUDIO10\_CONTROL\_MUTE\_CHANGED:

case UX\_DEVICE\_CLASS\_AUDIO10\_CONTROL\_VOLUME\_CHANGED:

default:

break;

}

}

### ux\_device\_class\_audio20\_control\_process

Process USB Audio 1.0 control requests

**Prototype**

UINT **ux\_device\_class\_audio20\_control\_process**(UX\_DEVICE\_CLASS\_AUDIO \*audio, UX\_SLAVE\_TRANSFER \*transfer\_request, UX\_DEVICE\_CLASS\_AUDIO20\_CONTROL\_GROUP \*group)

**Description**

This function manages basic requests sent by the host on the control endpoint with a USB Audio 2.0 specific type.

Audio 2.0 sampling rate (assumeds single fixed frequency), features of volume and mute requests are processed in the function. When processing the requests, pre-defined data passed by the last parameter (group) is used to answer requests and store control changes.

**Parameters**

**audio** Pointer to the audio instance.

**transfer** Pointer to the transfer request instance.

**group** Data group for request process.

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0xFF) | Error from function |

**Example**

/\* Initialize audio 2.0 control values. \*/

audio\_control[0].ux\_device\_class\_audio20\_control\_cs\_id = 0x10;

audio\_control[0].ux\_device\_class\_audio20\_control\_sampling\_frequency = 48000;

audio\_control[0].ux\_device\_class\_audio20\_control\_fu\_id = 2;

audio\_control[0].ux\_device\_class\_audio20\_control\_mute[0] = 0;

audio\_control[0].ux\_device\_class\_audio20\_control\_volume\_min[0] = 0;

audio\_control[0].ux\_device\_class\_audio20\_control\_volume\_max[0] = 100;

audio\_control[0].ux\_device\_class\_audio20\_control\_volume[0] = 50;

audio\_control[1].ux\_device\_class\_audio20\_control\_cs\_id = 0x10;

audio\_control[1].ux\_device\_class\_audio20\_control\_sampling\_frequency = 48000;

audio\_control[1].ux\_device\_class\_audio20\_control\_fu\_id = 5;

audio\_control[1].ux\_device\_class\_audio20\_control\_mute[0] = 0;

audio\_control[1].ux\_device\_class\_audio20\_control\_volume\_min[0] = 0;

audio\_control[1].ux\_device\_class\_audio20\_control\_volume\_max[0] = 100;

audio\_control[1].ux\_device\_class\_audio20\_control\_volume[0] = 50;

/\* Handle request and update control values.

Note here only mute and volume for master channel is supported.

\*/

status = ux\_device\_class\_audio20\_control\_process(audio, transfer, &group);

if (status == UX\_SUCCESS)

{

/\* Request handled, check changes \*/

switch(audio\_control[0].ux\_device\_class\_audio20\_control\_changed)

{

case UX\_DEVICE\_CLASS\_AUDIO20\_CONTROL\_MUTE\_CHANGED:

case UX\_DEVICE\_CLASS\_AUDIO20\_CONTROL\_VOLUME\_CHANGED:

default:

break;

}

}

# Chapter 6: USBX DPUMP Class Considerations

USBX contains a DPUMP class for the host and device side. This class is not a standard class per se, but rather an example that illustrates how to create a simple device by using 2 bulk pipes and sending data back and forth on these 2 pipes. The DPUMP class could be used to start a custom class or for legacy RS232 devices.

USB DPUMP flow chart:



## USBX DPUMP Device Class

The device DPUMP class uses a thread which is started upon connection to the USB host. The thread waits for a packet coming on the Bulk Out endpoint. When a packet is received, it copies the content to the Bulk In endpoint buffer and posts a transaction on this endpoint, waiting for the host to issue a request to read from this endpoint. This provides a loopback mechanism between the Bulk Out and Bulk In endpoints.

# Chapter 7: USBX Pictbridge implementation

UBSX supports the full Pictbridge implementation both on the host and the device. Pictbridge sits on top of USBX PIMA class on both sides.

The PictBridge standards allows the connection of a digital still camera or a smart phone directly to a printer without a PC, enabling direct printing to certain Pictbridge aware printers.

When a camera or phone is connected to a printer, the printer is the USB host and the camera is the USB device. However, with Pictbridge, the camera will appear as being the host and commands are driven from the camera. The camera is the storage server, the printer the storage client. The camera is the print client and the printer is of course the print server.

Pictbridge uses USB as a transport layer but relies on PTP (Picture Transfer Protocol) for the communication protocol.

The following is a diagram of the commands/responses between the DPS client and the DPS server when a print job occurs:



## Pictbridge client implementation

The Pictbridge on the client requires the USBX device stack and the PIMA class to be running first.

A device framework describes the PIMA class in the following way:

UCHAR device\_framework\_full\_speed[] =

{

/\* Device descriptor \*/

0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x20,

0xA9, 0x04, 0xB6, 0x30, 0x00, 0x00, 0x00, 0x00,

0x00, 0x01,

/\* Configuration descriptor \*/

0x09, 0x02, 0x27, 0x00, 0x01, 0x01, 0x00, 0xc0, 0x32,

/\* Interface descriptor \*/

0x09, 0x04, 0x00, 0x00, 0x03, 0x06, 0x01, 0x01, 0x00,

/\* Endpoint descriptor (Bulk Out) \*/

0x07, 0x05, 0x01, 0x02, 0x40, 0x00, 0x00,

/\* Endpoint descriptor (Bulk In) \*/

0x07, 0x05, 0x82, 0x02, 0x40, 0x00, 0x00,

/\* Endpoint descriptor (Interrupt) \*/

0x07, 0x05, 0x83, 0x03, 0x08, 0x00, 0x60

};

The Pima class is using the ID field 0x06 and has its subclass is 0x01 for Still Image and the protocol is 0x01 for PIMA 15740.

3 endpoints are defined in this class, 2 bulks for sending/receiving data and one interrupt for events.

Unlike other USBX device implementations, the Pictbridge application does not need to define a class itself. Rather it invokes the function ux\_pictbridge\_dpsclient\_start. An example is below:

/\* Initialize the Pictbridge string components. \*/

ux\_utility\_memory\_copy

(pictbridge.ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_vendor\_name,

"ExpressLogic",13);

ux\_utility\_memory\_copy

(pictbridge.ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_product\_name,

"EL\_Pictbridge\_Camera",21);

ux\_utility\_memory\_copy

(pictbridge.ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_serial\_no,

"ABC\_123",7);

ux\_utility\_memory\_copy

(pictbridge.ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_dpsversions,

"1.0 1.1",7);

pictbridge.ux\_pictbridge\_dpslocal.

ux\_pictbridge\_devinfo\_vendor\_specific\_version = 0x0100;

/\* Start the Pictbridge client. \*/

status = ux\_pictbridge\_dpsclient\_start(&pictbridge);

if(status != UX\_SUCCESS)

return;

The parameters passed to the pictbridge client are as follows:

pictbridge.ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_vendor\_name

: String of Vendor name pictbridge.ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_product\_name

: String of product name

pictbridge.ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_serial\_no,

: String of serial number

pictbridge.ux\_pictbridge\_dpslocal.ux\_pictbridge\_devinfo\_dpsversions

: String of version

pictbridge.ux\_pictbridge\_dpslocal.

ux\_pictbridge\_devinfo\_vendor\_specific\_version

: Value set to 0x0100;

The next step is for the device and the host to synchronize and be ready to exchange information.

This is done by waiting on an event flag as follows:

/\* We should wait for the host and the client to discover one another. \*/

status = ux\_utility\_event\_flags\_get

(&pictbridge.ux\_pictbridge\_event\_flags\_group,

UX\_PICTBRIDGE\_EVENT\_FLAG\_DISCOVERY,TX\_AND\_CLEAR, &actual\_flags,

UX\_PICTBRIDGE\_EVENT\_TIMEOUT);

If the state machine is in the DISCOVERY\_COMPLETE state, the camera side (the DPS client) will gather information regarding the printer and its capabilities.

If the DPS client is ready to accept a print job, its status will be set to UX\_PICTBRIDGE\_NEW\_JOB\_TRUE. It can be checked below:

/\* Check if the printer is ready for a print job. \*/

if (pictbridge.ux\_pictbridge\_dpsclient.ux\_pictbridge\_devinfo\_newjobok ==

UX\_PICTBRIDGE\_NEW\_JOB\_TRUE)

/\* We can print something … \*/

Next some print joib descriptors need to be filled as follows:

/\* We can start a new job. Fill in the JobConfig and PrintInfo structures. \*/

jobinfo = &pictbridge.ux\_pictbridge\_jobinfo;

/\* Attach a printinfo structure to the job. \*/

jobinfo -> ux\_pictbridge\_jobinfo\_printinfo\_start = &printinfo;

/\* Set the default values for print job. \*/

jobinfo -> ux\_pictbridge\_jobinfo\_quality =

UX\_PICTBRIDGE\_QUALITIES\_DEFAULT;

jobinfo -> ux\_pictbridge\_jobinfo\_papersize =

UX\_PICTBRIDGE\_PAPER\_SIZES\_DEFAULT;

jobinfo -> ux\_pictbridge\_jobinfo\_papertype =

UX\_PICTBRIDGE\_PAPER\_TYPES\_DEFAULT;

jobinfo -> ux\_pictbridge\_jobinfo\_filetype =

UX\_PICTBRIDGE\_FILE\_TYPES\_DEFAULT;

jobinfo -> ux\_pictbridge\_jobinfo\_dateprint =

UX\_PICTBRIDGE\_DATE\_PRINTS\_DEFAULT;

jobinfo -> ux\_pictbridge\_jobinfo\_filenameprint =

UX\_PICTBRIDGE\_FILE\_NAME\_PRINTS\_DEFAULT;

jobinfo -> ux\_pictbridge\_jobinfo\_imageoptimize =

UX\_PICTBRIDGE\_IMAGE\_OPTIMIZES\_OFF;

jobinfo -> ux\_pictbridge\_jobinfo\_layout =

UX\_PICTBRIDGE\_LAYOUTS\_DEFAULT;

jobinfo -> ux\_pictbridge\_jobinfo\_fixedsize =

UX\_PICTBRIDGE\_FIXED\_SIZE\_DEFAULT;

jobinfo -> ux\_pictbridge\_jobinfo\_cropping =

UX\_PICTBRIDGE\_CROPPINGS\_DEFAULT;

/\* Program the callback function for reading the object data. \*/

jobinfo -> ux\_pictbridge\_jobinfo\_object\_data\_read =

ux\_demo\_object\_data\_copy;

/\* This is a demo, the fileID is hardwired (1 and 2 for scripts, 3 for photo

to be printed. \*/

printinfo.ux\_pictbridge\_printinfo\_fileid =

UX\_PICTBRIDGE\_OBJECT\_HANDLE\_PRINT;

ux\_utility\_memory\_copy(printinfo.ux\_pictbridge\_printinfo\_filename,

"Pictbridge demo file", 20);

ux\_utility\_memory\_copy(printinfo.ux\_pictbridge\_printinfo\_date, "01/01/2008",

10);

/\* Fill in the object info to be printed. First get the pointer to the

object container in the job info structure. \*/

object = (UX\_SLAVE\_CLASS\_PIMA\_OBJECT \*) jobinfo ->

ux\_pictbridge\_jobinfo\_object;

/\* Store the object format: JPEG picture. \*/

object -> ux\_device\_class\_pima\_object\_format =

UX\_DEVICE\_CLASS\_PIMA\_OFC\_EXIF\_JPEG;

object -> ux\_device\_class\_pima\_object\_compressed\_size = IMAGE\_LEN;

object -> ux\_device\_class\_pima\_object\_offset = 0;

object -> ux\_device\_class\_pima\_object\_handle\_id =

UX\_PICTBRIDGE\_OBJECT\_HANDLE\_PRINT;

object -> ux\_device\_class\_pima\_object\_length = IMAGE\_LEN;

/\* File name is in Unicode. \*/

ux\_utility\_string\_to\_unicode("JPEG Image", object ->

ux\_device\_class\_pima\_object\_filename);

/\* And start the job. \*/

status =ux\_pictbridge\_dpsclient\_api\_start\_job(&pictbridge);

The Pictbridge client now has a print job to do and will fetch the image blocks at a time from the application through the callback defined in the field

jobinfo -> ux\_pictbridge\_jobinfo\_object\_data\_read

The prototype of that function is defined as:

### ux\_pictbridge\_jobinfo\_object\_data\_read

Copying a block of data from user space for printing

**Prototype**

UINT **ux\_pictbridge\_jobinfo\_object\_data\_read**(UX\_PICTBRIDGE \*pictbridge,

UCHAR \*object\_buffer, ULONG object\_offset, ULONG object\_length,

ULONG \*actual\_length)

**Description**

This function is called when the DPS client needs to retrieve a data block to print to the target Pictbridge printer.

**Parameters**

**pictbridge** Pointer to the pictbridge class instance.

**object\_buffer** Pointer to object buffer

**object\_offset** Where we are starting to read the data block

**object\_length** Length to be returned

**actual\_length** Actual length returned

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0x01) | The application could not retrieve data. |

**Example**

/\* Copy the object data. \*/

UINT ux\_demo\_object\_data\_copy(UX\_PICTBRIDGE \*pictbridge,UCHAR \*object\_buffer,

ULONG object\_offset, ULONG object\_length, ULONG \*actual\_length)

{

/\* Copy the demanded object data portion. \*/

ux\_utility\_memory\_copy(object\_buffer, image + object\_offset,

object\_length);

/\* Update the actual length. \*/

\*actual\_length = object\_length;

/\* We have copied the requested data. Return OK. \*/

return(UX\_SUCCESS);

}

## Pictbridge host implementation

The host implementation of Pictbridge is different from the client.

The first thing to do in a Pictbridge host environment is to register the Pima class as the example below shows:

status = ux\_host\_stack\_class\_register(\_ux\_system\_host\_class\_pima\_name,

ux\_host\_class\_pima\_entry);

if(status != UX\_SUCCESS)

return;

This class is the generic PTP layer sitting between the USB stack and the Pictbridge layer.

The next step is to initialize the Pictbridge default values for print services as follows:

|  |  |
| --- | --- |
| Pictbridge field | Value |
| DpsVersion[0]  DpsVersion[1]  DpsVersion[2]  VendorSpecificVersion | 0x00010000  0x00010001  0x00000000  0x00010000 |
| PrintServiceAvailable | 0x30010000 |
| Qualities[0]  Qualities[1]  Qualities[2]  Qualities[3] | UX\_PICTBRIDGE\_QUALITIES\_DEFAULT  UX\_PICTBRIDGE\_QUALITIES\_NORMAL  UX\_PICTBRIDGE\_QUALITIES\_DRAFT  UX\_PICTBRIDGE\_QUALITIES\_FINE |
| PaperSizes[0]  PaperSizes[1]  PaperSizes[2]  PaperSizes[3]  PaperSizes[4] | UX\_PICTBRIDGE\_PAPER\_SIZES\_DEFAULT  UX\_PICTBRIDGE\_PAPER\_SIZES\_4IX6I  UX\_PICTBRIDGE\_PAPER\_SIZES\_L  UX\_PICTBRIDGE\_PAPER\_SIZES\_2L  UX\_PICTBRIDGE\_PAPER\_SIZES\_LETTER |
| PaperTypes[0]  PaperTypes[1]  PaperTypes[2] | UX\_PICTBRIDGE\_PAPER\_TYPES\_DEFAULT  UX\_PICTBRIDGE\_PAPER\_TYPES\_PLAIN  UX\_PICTBRIDGE\_PAPER\_TYPES\_PHOTO |
| FileTypes[0]  FileTypes[1]  FileTypes[2]  FileTypes[3] | UX\_PICTBRIDGE\_FILE\_TYPES\_DEFAULT  UX\_PICTBRIDGE\_FILE\_TYPES\_EXIF\_JPEG  UX\_PICTBRIDGE\_FILE\_TYPES\_JFIF  UX\_PICTBRIDGE\_FILE\_TYPES\_DPOF |
| DatePrints[0]  DatePrints[1]  DatePrints[2] | UX\_PICTBRIDGE\_DATE\_PRINTS\_DEFAULT  UX\_PICTBRIDGE\_DATE\_PRINTS\_OFF  UX\_PICTBRIDGE\_DATE\_PRINTS\_ON |
| FileNamePrints[0]  FileNamePrints[1]  FileNamePrints[2] | UX\_PICTBRIDGE\_FILE\_NAME\_PRINTS\_DEFAULT  UX\_PICTBRIDGE\_FILE\_NAME\_PRINTS\_OFF  UX\_PICTBRIDGE\_FILE\_NAME\_PRINTS\_ON |
| ImageOptimizes[0]  ImageOptimizes[1]  ImageOptimizes[2] | UX\_PICTBRIDGE\_IMAGE\_OPTIMIZES\_DEFAULT  UX\_PICTBRIDGE\_IMAGE\_OPTIMIZES\_OFF  UX\_PICTBRIDGE\_IMAGE\_OPTIMIZES\_ON |
| Layouts[0]  Layouts[1]  Layouts[2]  Layouts[3] | UX\_PICTBRIDGE\_LAYOUTS\_DEFAULT  UX\_PICTBRIDGE\_LAYOUTS\_1\_UP\_BORDER  UX\_PICTBRIDGE\_LAYOUTS\_INDEX\_PRINT  UX\_PICTBRIDGE\_LAYOUTS\_1\_UP\_BORDERLESS |
| FixedSizes[0]  FixedSizes[1]  FixedSizes[2]  FixedSizes[3]  FixedSizes[4]  FixedSizes[5]  FixedSizes[6] | UX\_PICTBRIDGE\_FIXED\_SIZE\_DEFAULT  UX\_PICTBRIDGE\_FIXED\_SIZE\_35IX5I  UX\_PICTBRIDGE\_FIXED\_SIZE\_4IX6I  UX\_PICTBRIDGE\_FIXED\_SIZE\_5IX7I  UX\_PICTBRIDGE\_FIXED\_SIZE\_7CMX10CM  UX\_PICTBRIDGE\_FIXED\_SIZE\_LETTER  UX\_PICTBRIDGE\_FIXED\_SIZE\_A4 |
| Croppings[0]  Croppings[1]  Croppings[2] | UX\_PICTBRIDGE\_CROPPINGS\_DEFAULT  UX\_PICTBRIDGE\_CROPPINGS\_OFF  UX\_PICTBRIDGE\_CROPPINGS\_ON |

The state machine of the DPS host will be set to Idle and ready to accept a new print job.

The host portion of Pictbridge can now be started as the example below shows:

/\* Activate the pictbridge dpshost. \*/

status = ux\_pictbridge\_dpshost\_start(&pictbridge, pima);

if (status != UX\_SUCCESS)

return;

The Pictbridge host function requires a callback when data is ready to be printed. This is accomplished by passing a function pointer in the pictbridge host structure as follows:

/\* Set a callback when an object is being received. \*/

pictbridge.ux\_pictbridge\_application\_object\_data\_write =

tx\_demo\_object\_data\_write;

This function has the following properties:

### ux\_pictbridge\_application\_object\_data\_write

Writing a block of data for printing

**Prototype**

UINT **ux\_pictbridge\_application\_object\_data\_write**(UX\_PICTBRIDGE

\*pictbridge,UCHAR \*object\_buffer, ULONG offset,

ULONG total\_length, ULONG length);

**Description**

This function is called when the DPS server needs to retrieve a data block from the DPS client to print to the local printer.

**Parameters**

**pictbridge** Pointer to the pictbridge class instance.

**object\_buffer** Pointer to object buffer

**object\_offset** Where we are starting to read the data block

**total\_length** Entire length of object

**length** Length of this buffer

**Return Value**

|  |  |  |
| --- | --- | --- |
| **UX\_SUCCESS** | (0x00) | This operation was successful. |
| **UX\_ERROR** | (0x01) | The application could not print data. |

**Example**

/\* Copy the object data. \*/

UINT tx\_demo\_object\_data\_write(UX\_PICTBRIDGE \*pictbridge,

UCHAR \*object\_buffer, ULONG offset, ULONG total\_length, ULONG length);

{

UINT status;

/\* Send the data to the local printer. \*/

status = local\_printer\_data\_send(object\_buffer, length);

/\* We have printed the requested data. Return status. \*/

return(status);

}

# Chapter 8: USBX OTG

USBX supports the OTG functionalities of USB when an OTG compliant USB controller is available in the hardware design.

USBX supports OTG in the core USB stack. But for OTG to function, it requires a specific USB controller. USBX OTG controller functions can be found in the usbx\_otg directory. The current USBX version only supports the NXP LPC3131 with full OTG capabilities.

The regular controller driver functions (host or device) can still be found in the standard USBX usbx\_device\_controllers and usbx\_host\_controllers but the usbx\_otg directory contains the specific OTG functions associated with the USB controller.

There are 4 categories of functions for an OTG controller in addition to the usual host/device functions:

* VBUS specific functions
* Start and Stop of the controller
* USB role manager
* Interrupt handlers

**VBUS functions**

Each controller needs to have a VBUS manager to change the state of VBUS based on power management requirements. Usually, this function only performs turning on or off VBUS

**Start and Stop the controller**

Unlike a regular USB implementation, OTG requires the host and/or the device stack to be activated and deactivated when the role changes.

**USB role Manager**

The USB role manager receives commands to change the state of the USB. There are several states that need transitions to and from:

|  |  |  |
| --- | --- | --- |
| State | Value | Description |
| UX\_OTG\_IDLE | 0 | The device is Idle. Usually not connected to anything |
| UX\_OTG\_IDLE\_TO\_HOST | 1 | Device is connected with type A connector |
| UX\_OTG\_IDLE\_TO\_SLAVE | 2 | Device is connected with type B connector |
| UX\_OTG\_HOST\_TO\_IDLE | 3 | Host device got disconnected |
| UX\_OTG\_HOST\_TO\_SLAVE | 4 | Role swap from Host to Slave |
| UX\_OTG\_SLAVE\_TO\_IDLE | 5 | Slave device is disconnected |
| UX\_OTG\_SLAVE\_TO\_HOST | 6 | Role swap from Slave to Host |

**Interrupt handlers**

Both host and device controller drivers for OTG needs different interrupt handlers to monitor signals beyond traditional USB interrupts, in particular signals due to SRP and VBUS.

How to initialize a USB OTG controller. We use the NXP LPC3131 as an example here:

/\* Initialize the LPC3131 OTG controller. \*/

status = ux\_otg\_lpc3131\_initialize(0x19000000, lpc3131\_vbus\_function,

tx\_demo\_change\_mode\_callback);

In this example, we initialize the LPC3131 in OTG mode by passing a VBUS function and a callback for mode change (from host to slave or vice versa).

The callback function should simply record the new mode and wake up a pending thread to act up the new state:

void tx\_demo\_change\_mode\_callback(ULONG mode)

{

/\* Simply save the otg mode. \*/

otg\_mode = mode;

/\* Wake up the thread that is waiting. \*/

ux\_utility\_semaphore\_put(&mode\_change\_semaphore);

}

The mode value that is passed can have the following values:

* UX\_OTG\_MODE\_IDLE
* UX\_OTG\_MODE\_SLAVE
* UX\_OTG\_MODE\_HOST

The application can always check what the device is by looking at the variable:

\_ux\_system\_otg -> ux\_system\_otg\_device\_type

Its values can be:

* UX\_OTG\_DEVICE\_A
* UX\_OTG\_DEVICE\_B
* UX\_OTG\_DEVICE\_IDLE

A USB OTG host device can always ask for a role swap by issuing the command:

/\* Ask the stack to perform a HNP swap with the device. We relinquish the

host role to A device. \*/

ux\_host\_stack\_role\_swap(storage -> ux\_host\_class\_storage\_device);

For a slave device, there is no command to issue but the slave device can set a state to change the role which will be picked up by the host when it issues a GET\_STATUS and the swap will then be initiated.

/\* We are a B device, ask for role swap. The next GET\_STATUS from the host

will get the status change and do the HNP. \*/

\_ux\_system\_otg -> ux\_system\_otg\_slave\_role\_swap\_flag =

UX\_OTG\_HOST\_REQUEST\_FLAG;

# Index

API

USB device class, 22

USB device stack, 22

bulk in, 23, 40, 140, 142

bulk out, 23, 40, 140, 142

callback, 54, 59, 79, 80, 82, 87, 91, 93, 94, 99, 109, 144, 145, 148, 151

CDC-ACM class, 58, 59, 60, 79, 82, 116, 118, 120

CDC-ECM class, 77

class container, 29

class instance, 9, 61, 63, 65, 67, 69, 70, 71, 73, 74, 75, 76, 93, 94, 102, 103, 104, 106, 108, 111, 113, 115, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 137, 146, 149

Class layer, 8

configuration, 9, 14, 29, 33, 34, 35, 87

configuration descriptor, 22, 23, 40, 85, 91, 142

Controller layer, 8

device descriptor, 22, 23, 40, 58, 77, 85, 91, 142

device framework, 22, 23, 24, 41

device side, 8, 11, 17, 22, 35, 95, 139

DFU class, 85, 86, 87, 88, 90

DPUMP, 7, 139, 140

endpoint descriptor, 22, 23, 40, 77, 91, 142, 143

FileX, 2, 9, 13

firmware, 85, 87, 88, 90

functional descriptor, 59, 85

handle, 25, 99, 104, 105, 106, 108, 109, 110, 111, 113, 115, 145

HID class, 91, 92, 93, 94

host controller, 14

host side, 8, 22, 59, 91, 97

host stack, 17

initialization, 13, 17, 18, 22, 25, 42, 56, 59, 79, 80, 83, 91, 93, 98, 118

interface descriptor, 22, 23, 40, 58, 77, 85, 91, 142

LUN, 14, 19, 53, 55, 56

memory insufficient, 31, 39

NetX, 2, 9, 80, 83, 84

OTG, 7, 8, 9, 150, 151, 152

Picture Transfer Protocol, 95, 141

PIMA class, 95, 99, 102, 103, 104, 106, 108, 111, 113, 115, 141, 142, 143, 147

pipe, 61, 63, 65, 67, 69, 70, 71, 73, 74, 75, 76, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 137

power management, 9, 151

PTP responder, 95

queue, 93

reset sequence, 85, 88

RNDIS class, 81

SCSI logical unit, 14

semaphore, 152

slave, 19, 30, 32, 39, 47, 48, 53, 54, 56, 57, 59, 79, 80, 82, 83, 86, 87, 91, 101, 151, 152

stack layer, 8

target, 11, 12, 13, 14, 20, 87, 90, 96, 146

ThreadX, 2, 7, 9, 11, 12, 13, 14, 22

timer tick, 14

TraceX, 9

UNICODE, 24

USB device, 10, 18, 21, 22, 25, 53, 58, 77, 81, 85, 91, 95, 116

USB device controller, 18

USB device stack, 21, 22

USB host stack, 147

USB IF, 59, 84, 85, 88, 95

USB protocol, 8, 10

USBX pictbridge, 7, 141

USBX thread, 14

VBUS, 18, 21, 25, 36, 150, 151

version\_id, 20