Contents

Overview of Azure RTOS USBX

5/18/2020 • 9 minutes to read

Azure RTOS USBX is a high-performance USB host, device, and on-the-go (OTG) embedded stack. Azure RTOS USBX is fully integrated with Azure RTOS ThreadX and available for all ThreadX–supported processors. Like ThreadX, Azure RTOS USBX is designed to have a small footprint and high performance, making it ideal for deeply embedded applications that require an interface with USB devices.

Host, Device, OTG & Extensive Class Support

Azure RTOS USBX Host/Device embedded USB protocol stack is an Industrial Grade embedded USB solution designed specifically for deeply embedded, real-time, and IoT applications. Azure RTOS USBX provides host, device, and OTG support, as well as extensive class support. Azure RTOS USBX is fully integrated with ThreadX Real-Time Operating System, Azure RTOS FileX embedded FAT-compatible file system, Azure RTOS NetX, and Azure RTOS NetX Duo embedded TCP/IP stacks. All of this, combined with an extremely small footprint, fast execution and superior ease-of-use, make Azure RTOS USBX the ideal choice for the most demanding embedded IoT applications requiring USB connectivity.

Small-footprint

Azure RTOS USBX has a remarkably small minimal footprint of 10.5 KB of FLASH and 5.1 KB RAM for Azure RTOS USBX Device CDC/ACM support. Azure RTOS USBX Host requires a minimum of 18 KB of FLASH and 25 KB of RAM for CDC/ACM support.

An additional 10 KB to 13 KB of instruction area memory is needed for TCP functionality. Azure RTOS USBX RAM usage typically ranges from 2.6 KB to 3.6 KB plus the packet pool memory, which is defined by the application.

Like ThreadX, the size of Azure RTOS USBX automatically scales based on the services actually used by the application. This virtually eliminates the need for complicated configuration and build parameters, making things easier for the developer.

Fast execution

Azure RTOS USBX is designed for speed and has minimal internal function call layering and support for cache and DMA utilization. All of this and a general performance-oriented design philosophy helps Azure RTOS USBX achieve the fastest possible performance.

Simple, easy-to-use

Azure RTOS USBX is simple to use. The Azure RTOS USBX API is both intuitive and highly functional. The API names are made of real words and not the "alphabet soup" or highly abbreviated names that are so common in other file system products. All Azure RTOS USBX APIs have a leading "ux_" and follow a noun-verb naming convention. Furthermore, there is a functional consistency throughout the API. For example, all APIs that suspend have an optional timeout that functions in an identical manner for APIs.

USB Interoperability verification

Azure RTOS USBX Device Stack has been rigorously tested with the USB IF standard testing tool USBCV to ensure full compliance with the USB specifications and interoperability with different host systems. In addition, Azure RTOS USBX OTG stack has been verified and certified by the independent test lab Allion in Taiwan.

USB Host controller support

Azure RTOS USBX supports major USB standards like OHCI and EHCI. In addition, Azure RTOS USBX supports proprietary discrete USB host controllers from Atmel, Microchip, Philips, Renesas, ST, TI, and other vendors. Azure RTOS USBX also supports multiple host controllers in the same application. USB Device controller support Azure

RTOS USBX supports popular USB device controllers from Analog Devices, Atmel, Microchip, NXP, Philips, Renesas, ST, TI, and other vendors.

Extensive Host Class support

Azure RTOS USBX Host provides support for most popular classes, including ASIX, AUDIO, CDC/ACM, CDC/ECM, GSER, HID (keyboard, mouse, and remote control), HUB, PIMA (PTP/MTP), PRINTER, PROLIFIC, and STORAGE.

Extensive USB Device Class support

Azure RTOS USBX Device provides support for most popular classes, including CDC/ACM, CDC/ECM, DFU, HID, PIMA (PTP/MTP) (w/MTP), RNDIS, and STORAGE. Support for custom classes is also available.

Pictbridge support

Azure RTOS USBX supports the full Pictbridge implementation both on the host and the device. Pictbridge sits on top of Azure RTOS USBX PIMA (PTP/MTP) class on both sides. The PictBridge standard 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.

Custom class support

Azure RTOS USBX Host and Device support custom classes. An example custom class is provided in the Azure RTOS USBX distribution. This simple data pump class is called DPUMP and can be used as a model for custom application classes. Advanced technology Azure RTOS USBX Host and Device support custom classes. An example custom class is provided in the Azure RTOS USBX distribution. Azure RTOS USBX is advanced technology that includes:

- Host, Device, and OTG support
- USB low, full, and high-speed support
- Automatic scaling
- Fully integrated with ThreadX, Azure RTOS FileX, and Azure RTOS NetX
- Optional performance metrics
- Azure RTOS TraceX system analysis support

Fastest time-to-market

Azure RTOS USBX has a remarkably small footprint of 9 KB to 15 KB for basic IP and UDP support. Azure RTOS USBX is easy to install, learn, use, debug, verify, certify, and maintain. As a result, Azure RTOS USBX is one of the most popular USB solutions for embedded IoT devices. Our consistent time-to-market advantage is built on:

- quality documentation – please review our Azure RTOS USBX Host and Device User Guides and see for yourself!
- complete source code availability
- easy-to-use API
- comprehensive and advanced feature set

One Simple License

There is no cost to use and test the source code and no cost for production licenses when deployed to pre-licensed devices, all other devices need a simple annual license.

Full, highest-quality source code

Throughout the years, Azure RTOS NetX source code has set the bar in quality and ease of understanding. In addition, the convention of having one function per file provides for easy source navigation.

Supports most popular architectures

Azure RTOS USBX runs on most popular 32/64-bit microprocessors, out-of-the-box, fully tested and fully supported, including the following:

- **Analog Devices**: SHARC, Blackfin, CM4xx
- **Andes Core**: RISC-V
- **Ambiqmicro**: Apollo MCUs
- **ARM**: ARM7, ARM9, ARM11, Cortex-M0/M3/M4/M7/A15/A5/A7/A8/A9/A5x 64-bi/A7x 64-bit/R4/R5, TrustZone ARMv8-M
- **Cadence**: Xtensa, Diamond
- **CEVA**: PSoC, PSoC 4, PSoC 5, PSoC 6, FM0+, FM3, MF4, WICED WiFi
- **Cypress**: RISC-V
- **EnSilica**: eSi-RISC
- **Infineon**: XMC1000, XMC4000, TriCore
- **Intel & Intel FPGA**: x36/Pentium, XScale, NIOS II, Cyclone, Arria 10
- **Microchip**: AVR32, ARM7, ARM9, Cortex-M3/M4/M7, SAM3/4/7/9/A/C/D/E/G/L/SV, PIC24/PIC32
- **Microsemi**: RISC-V
- **NXP**: LPC, ARM7, ARM9, PowerPC, 68 K, i.MX, ColdFire, Kinetis Cortex-M3/M4
- **Renesas**: SH, HS, V850, RX, RZ, Synergy Silicon Labs: EFM32
- **Synopsys**: ARC 600, 700, ARC EM, ARC HS
- **ST**: STM32, ARM7, ARM9, Cortex-M3/M4/M7
- **TI**: C5xxx, C6xxx, Stellaris, Sitara, Tiva-C
- **Wave Computing**: MIPS32 4K, 24 K, 34 K, 1004 K, MIPS64 5K, microAptiv, interAptiv, proAptiv, M-Class **Xilinx**: MicroBlaze, PowerPC 405, ZYNQ, ZYNQ UltraSCALE

Azure RTOS USBX APIs

Azure RTOS USBX Host API

The Azure RTOS USBX Host API is an intuitive and consistent API, following a noun-verb naming convention. All APIs have leading ux_host_* to easily identify as USBX. Any blocking APIs have optional thread timeout.

ASIX	Minimal 0.3 KB FLASH, 4 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX host APIs in this form: *ux_host_class_asix_**
AUDIO	Minimal 1.2 KB FLASH, 4 KB RAM Automatic scaling System-level trace via Intuitive Azure RTOS USBX host APIs in this form: *ux_host_class_audio_**
CDC/ACM	Minimal 1.4 KB FLASH, 4 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX host APIs in this form: *ux_host_class_cdc_acm_**

HID	Minimal 0.3 KB FLASH, 4 KB RAM Keyboard, mouse, and remote support Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX host APIs in this form: *ux_host_class_hid_**
HUB	Minimal 1.7 KB FLASH, 2 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX host APIs in this form: *ux_host_class_hub_**
PIMA (PTP/MTP)	Minimal 0.9 KB FLASH, 8 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX host APIs in this form: *ux_host_class_pima_**
PRINTER	Minimal 0.8 KB FLASH, 8 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX host APIs in this form: *ux_host_class_printer_**
PROLIFIC	Minimal 1.5 KB FLASH, 4 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX host APIs in this form: *ux_host_class_prolific_**
STORAGE	Minimal 5.6 KB FLASH, 4 KB RAM Automatic scaling Integrated with Azure RTOS FileX System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX host APIs in this form: *ux_host_class_storage_**
USB Host STACK	Supports many host controllers Minimal 18 KB FLASH, 25 KB RAM Automatic scaling Support for multiple host controllers on same platform USB low, full, and high-speed support System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX host APIs in this form: *ux_host_stack_**
OHCI, EHCI, PROPRIETARY Host CONTROLLERS	

Azure RTOS USBX Device API

The Azure RTOS USBX Device API is an intuitive and consistent API following a noun-verb naming convention. All APIs have leading ux_device_* to easily identify as USBX. Blocking APIs have optional thread timeout. Please see Azure RTOS USBX Host User Guide for more details.

CDC/ACM	Minimal 0.8 KB FLASH, 2 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX device APIs in this form: *ux_device_class_cdc_acm_**
CDC/ECM	Minimal 1.5 KB FLASH, 4 KB to 8 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX device APIs in this form: *ux_device_class_cdc_ecm_**
DFU	Minimal 1.1 KB FLASH, 2 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX device APIs in this form: *ux_device_class_dfu_**
GSER	Minimal 0.6 KB FLASH, 4 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX device APIs in this form: *ux_device_class_gser_**
HID	Minimal 0.9 KB FLASH, 2 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX device APIs in this form: *ux_device_class_hid_**
PIMA (PTP/MTP)	Minimal 5.2 KB FLASH, 8 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX device APIs in this form: *ux_device_class_pima_**
STORAGE	Minimal 2.3 KB FLASH, 4 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX device APIs in this form: *ux_device_class_storage_**
RNDIS	Minimal 2.3 KB FLASH, 4 KB to 8 KB RAM Automatic scaling Integrated with Azure RTOS NetX and Azure RTOS NetX DUO System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX device APIs in this form: *ux_device_class_rndls_**
Azure RTOS USBX Device STACK	Minimal 2.3 KB FLASH, 4 KB RAM Automatic scaling System-level trace via Azure RTOS TraceX Intuitive Azure RTOS USBX device APIs in this form: *ux_device_class_storage_**
PROPRIETARY Host CONTROLLERS	

Next steps

Start working with the Azure RTOS USBX Host and Device Stack by following our Host Stack User Guide or Device Stack User Guide.

Azure RTOS USBX Device Stack User Guide

5/18/2020 • 2 minutes to read

This guide provides comprehensive information about Azure RTOS USBX, the high performance USB foundation software from Microsoft.

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 https://www.USB.org/developers

Organization

- **Chapter 1** - contains an introduction to Azure RTOS USBX
- **Chapter 2** - gives the basic steps to install and use Azure RTOS USBX with your ThreadX application
- **Chapter 3** - describes the functional components of the Azure RTOS USBX device stack
- **Chapter 4** - describes the Azure RTOS USBX device stack services
- **Chapter 5** - describes each Azure RTOS USBX device class including their APIs

Customer Support Center

Please submit a support ticket through the Azure Portal for questions or help using the steps here. Please supply us with the following information in an email message so we can more efficiently resolve your support request:

1. A detailed description of the problem, including frequency of occurrence and whether it can be reliably reproduced.
2. A detailed description of any changes to the application and/or Azure RTOS ThreadX that preceded the problem.
3. The contents of the *_tx_version_id* string found in the *tx_port.h* file of your distribution. This string will provide us valuable information regarding your run-time environment.
4. The contents in RAM of the *_tx_build_options* ULONG variable. This variable will give us information on how your Azure RTOS ThreadX library was built.

Chapter 1 - Introduction to USBX

5/18/2020 • 2 minutes to read

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

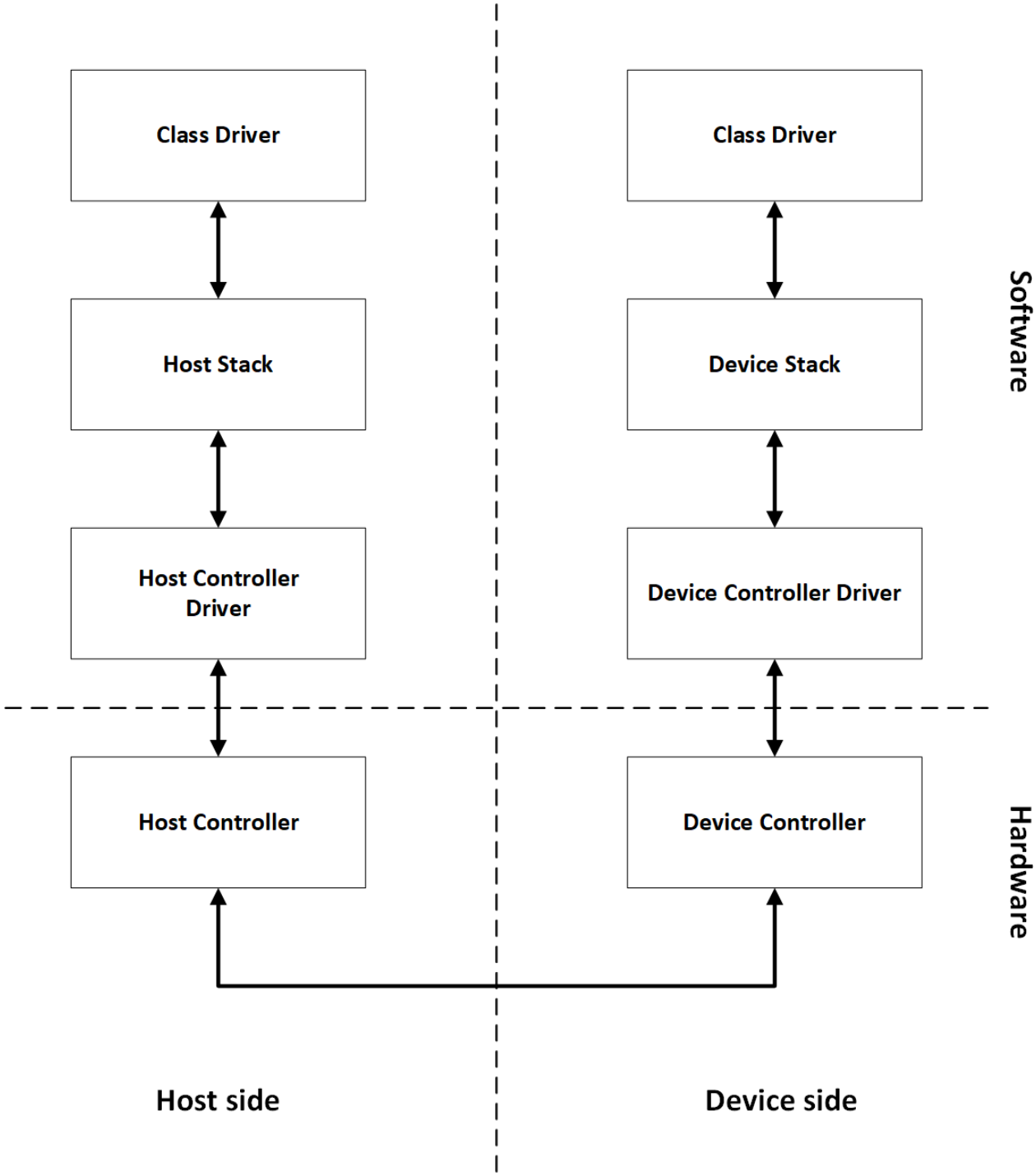
USBX features

USBX supports 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 composed 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 configurations
- 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 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

6/24/2020 • 8 minutes to read

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. 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.

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 10-12 K 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

Azure RTOS USBX can be obtained from our public source code repository at https://github.com/azure-rtos/usbx/.

The following is a list of several important files in the repository:

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.

USBX Installation

USBX is installed by cloning the GitHub repository to your local machine. The following is typical syntax for creating a clone of the USBX repository on your PC:

```
git clone https://github.com/azure-rtos/usbx
```

Alternatively you can download a copy of the repository using the download button on the GitHub main page.

You will also find instructions for building the USBX library on the front page of the online repository.

The following general instructions apply to virtually any installation:

1. 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.
2. Add a call to **ux_system_initialize** at or near the beginning of **tx_application_define.** This is where the USBX resources are initialized.
3. Add a call to **ux_device_stack_initialize.**
4. Add one or more calls to initialize the required USBX classes (either host and/or devices classes)
5. Add one or more calls to initialize the device controller available in the system.
6. 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.|
7. 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_user.h***.

The list below details each configuration option:

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 bytes 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

This value actually defines the time slice that will be used for threads. For example, if defined to 0, the ThreadX target port does not use time slices.

UX_MAX_SLAVE_CLASS_DRIVER

This is the maximum number of USBX classes that can be registered via ux_device_stack_class_register.

UX_MAX_SLAVE_LUN

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

UX_SLAVE_CLASS_STORAGE_INCLUDE_MMC

If defined, the storage class will handle Multi-Media Commands (MMC) that is, DVD-ROM.

UX_DEVICE_CLASS_CDC_ECM_NX_PKPOOL_ENTRIES

This value represents the number of NetX packets in the CDC-ECM class' packet pool. The default is 16.

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_DEVICE_CLASS_HID_EVENT_BUFFER_LENGTH

This value represents the maximum length in bytes of a HID report.

UX_DEVICE_CLASS_HID_MAX_EVENTS_QUEUE

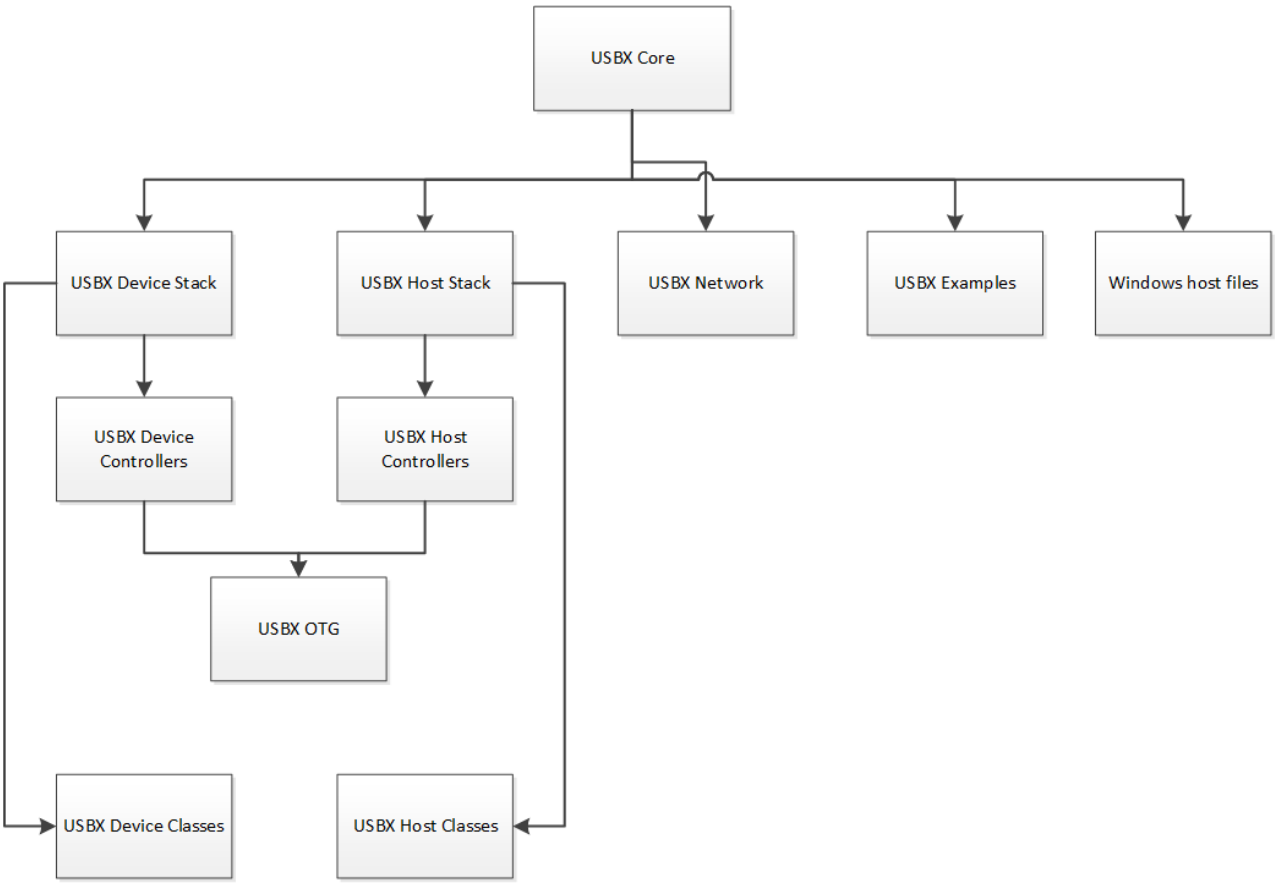
this value represents the maximum number of HID reports that can be queued at once.

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 128 K 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 resources:

```
/* 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 usb controller:

```
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:

```
/* 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 **ux_port.h** 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

6/24/2020 • 5 minutes to read

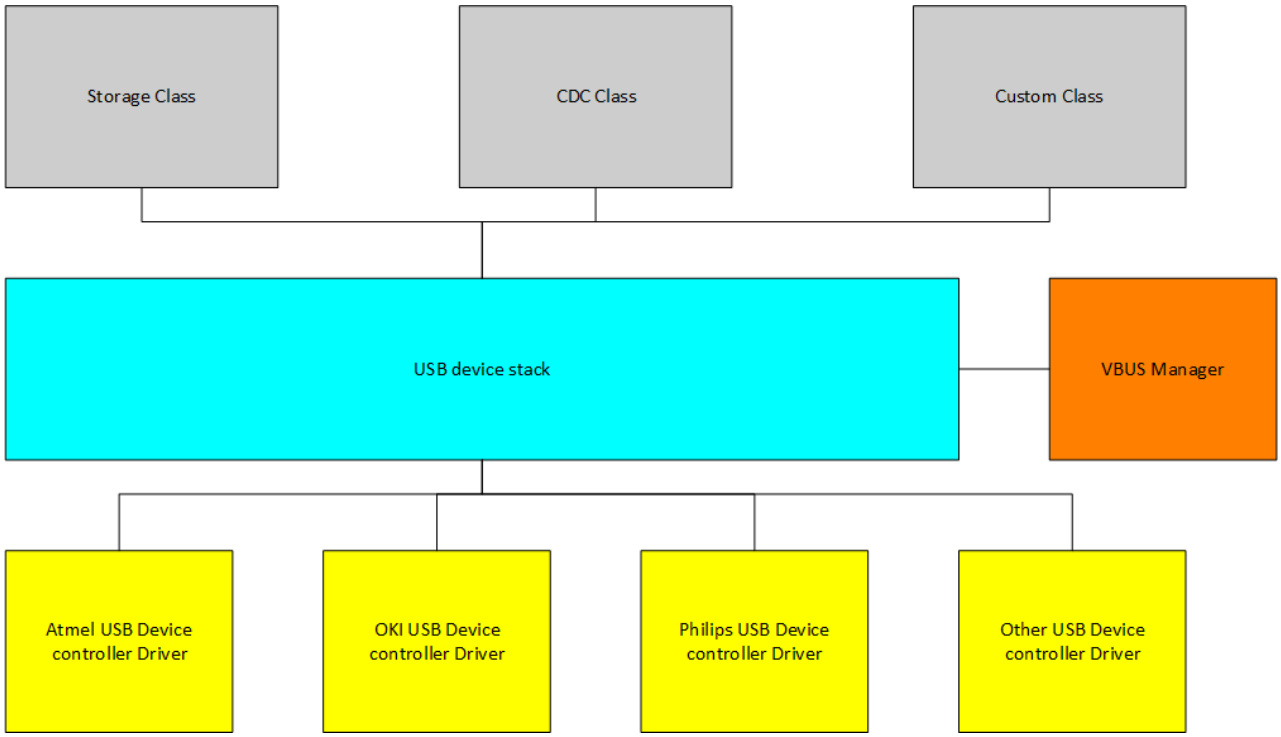
This chapter contains a description of the high performance USBX embedded USB device stack from a functional perspective.

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 devic 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.

Description of USBX Device Services

6/24/2020 • 12 minutes to read

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_FUNCTION_NOT_SUPPORTED** (0x54) Device is not configured.
- **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 registered
- **UX_MEMORY_INSUFFICIENT** (0x12) No entries left in class table.
- **UX_THREAD_ERROR** (0x16) Cannot create a class thread.

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 unregistered.
- **UX_NO_CLASS_MATCH** (0x57) The class isn't registered.

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 configuration was successfully set.

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.
- UX_ERROR (0xFF) The device is in an invalid state.

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_FUNCTION_NOT_SUPPORTED (0x54) 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 *device_framework_high_speed,
    ULONG device_framework_length_high_speed,
    UCHAR *device_framework_full_speed,
    ULONG device_framework_length_full_speed,
    UCHAR *string_framework,
    ULONG string_framework_length,
    UCHAR *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.
- UX_DESCRIPTOR_CORRUPTED (0x42) The descriptor is invalid.

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 *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 * 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_TRANSFER_NOT_READY** (0x25) The device is in an invalid state; it must be ATTACHED, CONFIGURED, or ADDRESSED.
- **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 the USBX device stack – all device stack resources are freed. This should be called after all classes have been unregistered via ux_device_stack_class_unregister.

Parameters

None

Return Value

UX_SUCCESS (0x00) This operation was successful.

Chapter 5 - USBX Device Class Considerations

6/24/2020 • 25 minutes to read

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;

/* 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);
```

Each class can register, optionally, a callback function when a 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;

/* 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 Azure RTOS 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:

- storage is the instance of the storage class.
- lun is the LUN the command is directed to.
- 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.
- media_status should be filled out exactly like the media status callback return value.

The return value can have either the value UX_SUCCESS or UX_ERROR indicating a successful or unsuccessful operation. These operations do not need to return any other error codes. If there is an error in any operation, the storage class will invoke the status call back function.

This function has the following prototype:

```
ULONG 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. In the future it may be used to distinguish multiple storage devices or storage devices with multiple SCSI LUNs. This version of the storage class does not support multiple instances of the storage class or storage devices with multiple SCSI LUNs.

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

SENSE KEY	ASC	ASCQ	DESCRIPTION
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

SENSE KEY	ASC	ASCQ	DESCRIPTION
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

There are two additional, optional callbacks the application may implement; one is for responding to a GET_STATUS_NOTIFICATION command and the other is for responding to the SYNCHRONIZE_CACHE command.

If the application would like to handle the GET_STATUS_NOTIFICATION command from the host, it should implement a callback with the following prototype:

```
UINT ux_slave_class_storage_media_notification(VOID *storage, ULONG lun,
    ULONG media_id, ULONG notification_class,
    UCHAR **media_notification, ULONG *media_notification_length);
```

Where:

- storage is the instance of the storage class.
- media_id is not currently used. notification_class specifies the class of notification.
- media_notification should be set by the application to the buffer containing the response for the notification.
- media_notification_length should be set by the application to contain the length of the response buffer.

The return value indicates whether or not the command succeeded – should be either UX_SUCCESS or UX_ERROR.

If the application does not implement this callback, then upon receiving the GET_STATUS_NOTIFICATION command, USBX will notify the host that the command is not implemented.

The SYNCHRONIZE_CACHE command should be handled if the application is utilizing a cache for writes from the host. A host may send this command if it knows the storage device is about to be disconnected, for example, in Windows, if you right click a flash drive icon in the toolbar and select "Eject [storage device name]", Windows will issue the SYNCHRONIZE_CACHE command to that device.

If the application would like to handle the GET_STATUS_NOTIFICATION command from the host, it should implement a callback with the following prototype:

```
UINT ux_slave_class_storage_media_flush(VOID *storage, ULONG lun,
    ULONG number_blocks, ULONG lba, ULONG *media_status);
```

Where:

- storage is the instance of the storage class.
- lun parameter specifies which LUN the command is directed to.
- number_blocks specifies the number of blocks to synchronize.
- lba is the sector address of the first block to synchronize.
- media_status should be filled out exactly like the media status callback return value.

The return value indicates whether or not the command succeeded – should be either UX_SUCCESS or UX_ERROR.

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);
```

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 - https://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 MACOs 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. We supply 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.

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

```
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.

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.

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.

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);

/* If status is UX_SUCCESS, the operation was successful. */
```

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. For example, to abort an ongoing write, UX_SLAVE_CLASS_CDC_ACM_ENDPOINT_XMIT should be passed as the parameter.

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_ENDPOINT_HANDLE_UNKNOWN** (0x53) Invalid pipe direction.

Example

```
/* This is to 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);

/* If status is UX_SUCCESS, the operation was successful. */
```

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) Transmission already started.
- **UX_MEMORY_INSUFFICIENT** (0x12) A memory allocation failed.

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);

/* If status is UX_SUCCESS, the operation was successful. */
```

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_TRANSMISSI ON_STOP
- **parameter**: Not used

Return Value

- **UX_SUCCESS** (0x00) This operation was successful.
- **UX_ERROR** (0xFF) No ongoing transmission.

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);

/* If status is UX_SUCCESS, the operation was successful. */
```

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). It is blocking.

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.

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

Write 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). It is blocking.

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 non-blocking and the completion will be done through a callback set in UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START.

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 - https://www.linux-usb.org/usb.ids
    0x3939 idVendor: Azure RTOS 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 - "Azure RTOS" */
    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 local node specifies the MAC address of the device, while the remote node specifies the MAC address of the host. 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 Azure RTOS 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 CDCECM 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 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.

Currently, the USBX HID class does not support multiple report IDs, as most applications only require one ID (ID zero). If multiple report IDs is a feature you are interested in, please contact us.

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 = 0;

/* 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** (0xFF) Error no space available in circular 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;
}
```

Chapter 1 - Introduction to the USBX Device Stack User Guide Supplement

5/18/2020 • 2 minutes to read

This document is a supplement to the USBX Device Stack User Guide. It contains documentation for the uncertified USBX Device classes that are not included in the main user guide.

Organization

- **Chapter 1** contains an introduction to USBX
- **Chapter 2** USBX Host Classes API
- **Chapter 3** USBX DPUMP Class Considerations
- **Chapter 4** USBX Pictbridge implementation
- **Chapter 5** USBX OTG

Chapter 2 - USBX Device Class Considerations

6/24/2020 • 40 minutes to read

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 Azure RTOS 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. Microsoft 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:

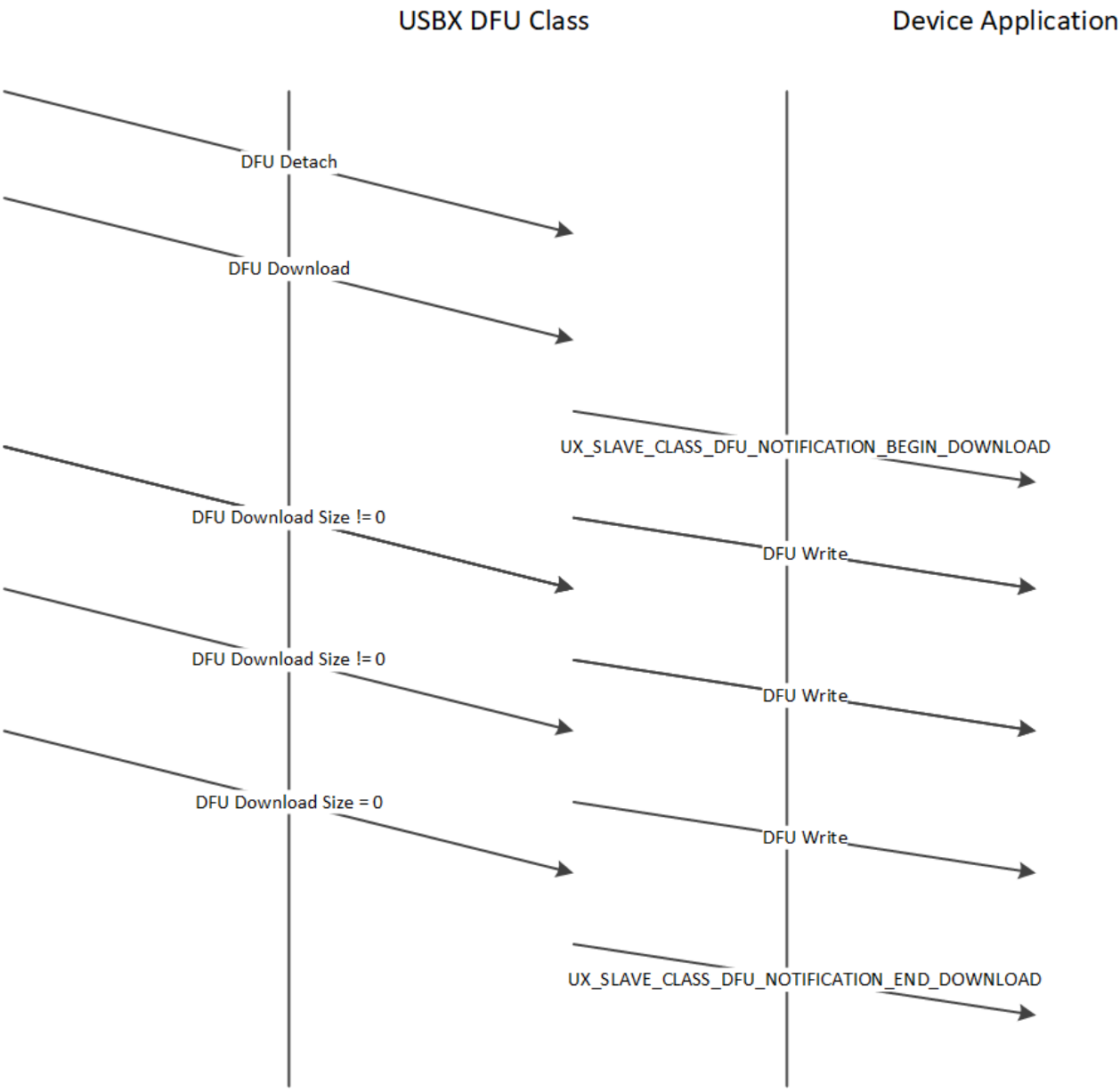
NAME	OFFSET	SIZE	TYPE	DESCRIPTION
bmAttributes	2	1	Bit field	Bit 3: device will perform a bus detach-attach sequence when it receives a DFU_DETACH request. The host must not issue a USB Reset. (bitWillDetach) 0 = no 1 = yes Bit 2: device is able to communicate via USB after Manifestation phase. (bitManifestationTolerant) 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.



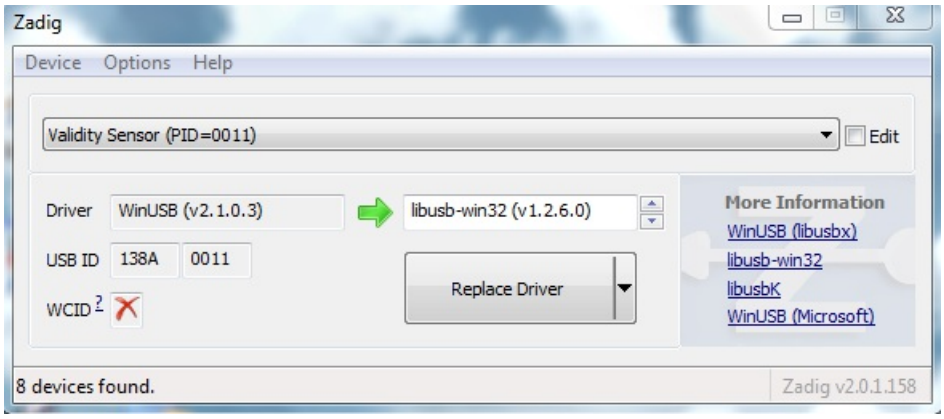
The major challenge of the DFU class is getting the right application on the host to perform the download the firmware. There is no application supplied by Microsoft or the USB-IF. Some shareware exist and they work reasonably well on Linux and to a lesser extent on Windows.

On Linux, one can use dfu-utils to be found here: https://wiki.openmoko.org/wiki/Dfu-util A lot of information on dfu utils can also be found on this link: https://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 be 0. 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: https://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&gt;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 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

OPERATION CODE	VALUE	DESCRIPTION
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

OPERATION CODE	VALUE	DESCRIPTION
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.

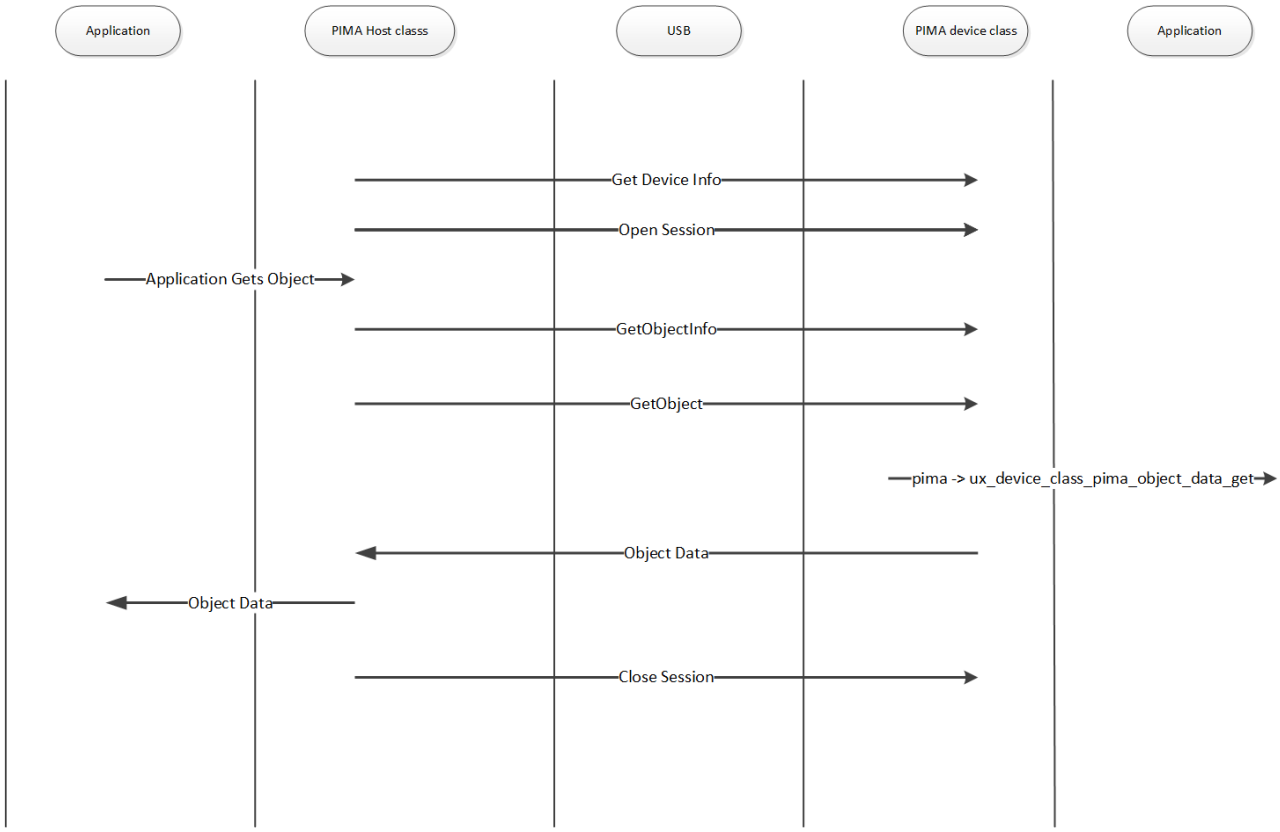
Most PIMA commands can only be executed when a session has been opened by the

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 different 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 - https://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;
```

2. 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_en
    try = 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

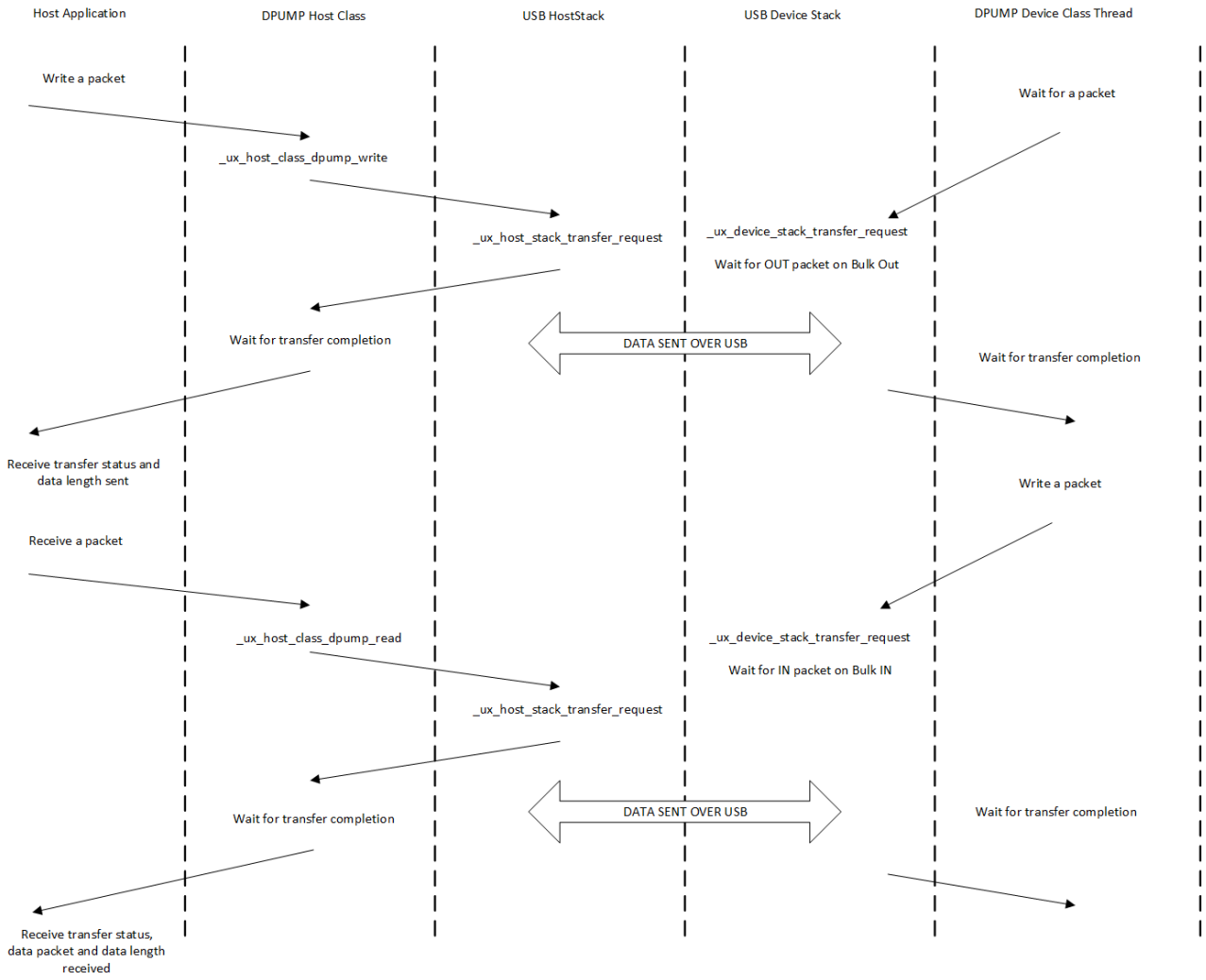
```
/* Initialize audio 2.0 control values. */

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 3 - USBX DPUMP Class Considerations

5/18/2020 • 2 minutes to read

USBX contains a DPUMP class for the host and device side. This class is not a standard class in itself, but rather an example that illustrates how to create a simple device by using two bulk pipes and sending data back and forth on these two 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 4 - USBX Pictbridge implementation

5/18/2020 • 6 minutes to read

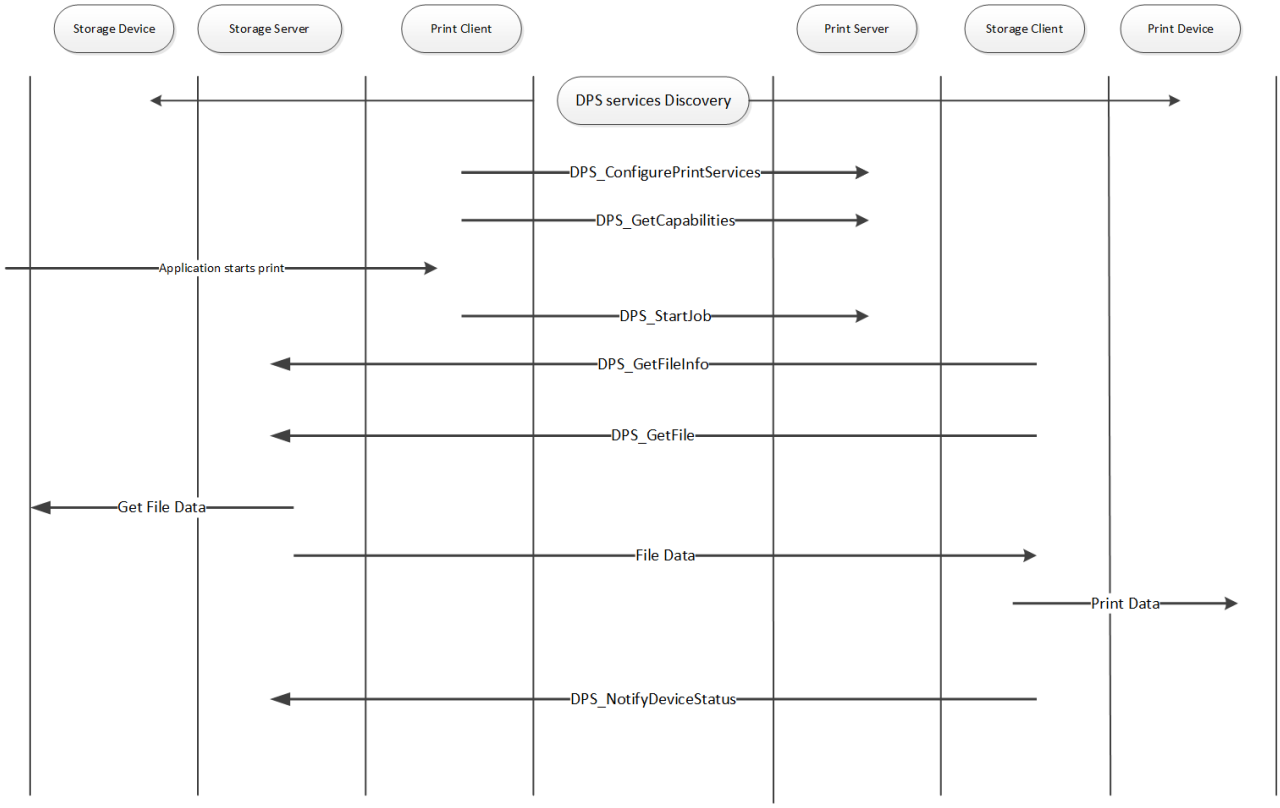
UBSX supports the full Pictbridge implementation both on the host and the device. Pictbridge sits on top of USBX PIMA class on both sides.

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]	0x00010000
DpsVersion[1]	0x00010001
DpsVersion[2]	0x00000000
VendorSpecificVersion	0x00010000
PrintServiceAvailable	0x30010000
Qualities[0]	UX_PICTBRIDGE_QUALITIES_DEFAULT
Qualities[1]	UX_PICTBRIDGE_QUALITIES_NORMAL
Qualities[2]	UX_PICTBRIDGE_QUALITIES_DRAFT
Qualities[3]	UX_PICTBRIDGE_QUALITIES_FINE
PaperSizes[0]	UX_PICTBRIDGE_PAPER_SIZES_DEFAULT
PaperSizes[1]	UX_PICTBRIDGE_PAPER_SIZES_4IX6I
PaperSizes[2]	UX_PICTBRIDGE_PAPER_SIZES_L
PaperSizes[3]	UX_PICTBRIDGE_PAPER_SIZES_2L
PaperSizes[4]	UX_PICTBRIDGE_PAPER_SIZES_LETTER
PaperTypes[0]	UX_PICTBRIDGE_PAPER_TYPES_DEFAULT
PaperTypes[1]	UX_PICTBRIDGE_PAPER_TYPES_PLAIN
PaperTypes[2	UX_PICTBRIDGE_PAPER_TYPES_PHOTO
FileTypes[0]	UX_PICTBRIDGE_FILE_TYPES_DEFAULT
FileTypes[1]	UX_PICTBRIDGE_FILE_TYPES_EXIF_JPEG
FileTypes[2]	UX_PICTBRIDGE_FILE_TYPES_JFIF
FileTypes[3]	UX_PICTBRIDGE_FILE_TYPES_DPOF
DatePrints[0]	UX_PICTBRIDGE_DATE_PRINTS_DEFAULT
DatePrints[1]	UX_PICTBRIDGE_DATE_PRINTS_OFF
DatePrints[2]	UX_PICTBRIDGE_DATE_PRINTS_ON
FileNamePrints[0]	UX_PICTBRIDGE_FILE_NAME_PRINTS_DEFAULT

PICTBRIDGE FIELD	VALUE
FileNamePrints[1]	UX_PICTBRIDGE_FILE_NAME_PRINTS_OFF
FileNamePrints[2]	UX_PICTBRIDGE_FILE_NAME_PRINTS_ON
ImageOptimizes[0]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_DEFAULT
ImageOptimizes[1]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_OFF
ImageOptimizes[2]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_ON
Layouts[0]	UX_PICTBRIDGE_LAYOUTS_DEFAULT
Layouts[1]	UX_PICTBRIDGE_LAYOUTS_1_UP_BORDER
Layouts[2]	UX_PICTBRIDGE_LAYOUTS_INDEX_PRINT
Layouts[3]	UX_PICTBRIDGE_LAYOUTS_1_UP_BORDERLESS
FixedSizes[0]	UX_PICTBRIDGE_FIXED_SIZE_DEFAULT
FixedSizes[1]	UX_PICTBRIDGE_FIXED_SIZE_35IX5I
FixedSizes[2]	UX_PICTBRIDGE_FIXED_SIZE_4IX6I
FixedSizes[3]	UX_PICTBRIDGE_FIXED_SIZE_5IX7I
FixedSizes[4]	UX_PICTBRIDGE_FIXED_SIZE_7CMX10CM
FixedSizes[5]	UX_PICTBRIDGE_FIXED_SIZE_LETTER
FixedSizes[6]	UX_PICTBRIDGE_FIXED_SIZE_A4
Croppings[0]	UX_PICTBRIDGE_CROPPINGS_DEFAULT
Croppings[1]	UX_PICTBRIDGE_CROPPINGS_OFF
Croppings[2]	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 5 - USBX OTG

5/18/2020 • 2 minutes to read

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 four 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. 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

STATE	VALUE	DESCRIPTION
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;
```

Azure RTOS USBX Host Stack User Guide

5/18/2020 • 2 minutes to read

This guide provides comprehensive information about Azure RTOS USBX, the high-performance USB foundation software from Microsoft.

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 https://www.USB.org/developers

Organization

- **Chapter 1** - contains an introduction to Azure RTOS USBX
- **Chapter 2** - gives the basic steps to install and use Azure RTOS USBX with your Azure RTOS ThreadX application
- **Chapter 3** - provides a functional overview of Azure RTOS USBX and basic information about USB
- **Chapter 4** - details the application's interface to Azure RTOS USBX in host mode
- **Chapter 5** - describes the APIs of the Azure RTOS USBX Host classes
- **Chapter 6** - describes the Azure RTOS USBX CDC-ECM class

Customer Support Center

Please submit a support ticket through the Azure Portal for questions or help using the steps here. Please supply us with the following information in an email message so we can more efficiently resolve your support request:

1. A detailed description of the problem, including frequency of occurrence and whether it can be reliably reproduced.
2. A detailed description of any changes to the application and/or Azure RTOS ThreadX that preceded the problem.
3. The contents of the *_tx_version_id* string found in the *tx_port.h* file of your distribution. This string will provide us valuable information regarding your run-time environment.
4. The contents in RAM of the *_tx_build_options* ULONG variable. This variable will give us information on how your Azure RTOS ThreadX library was built.

Chapter 1 - Introduction to USBX

5/18/2020 • 2 minutes to read

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

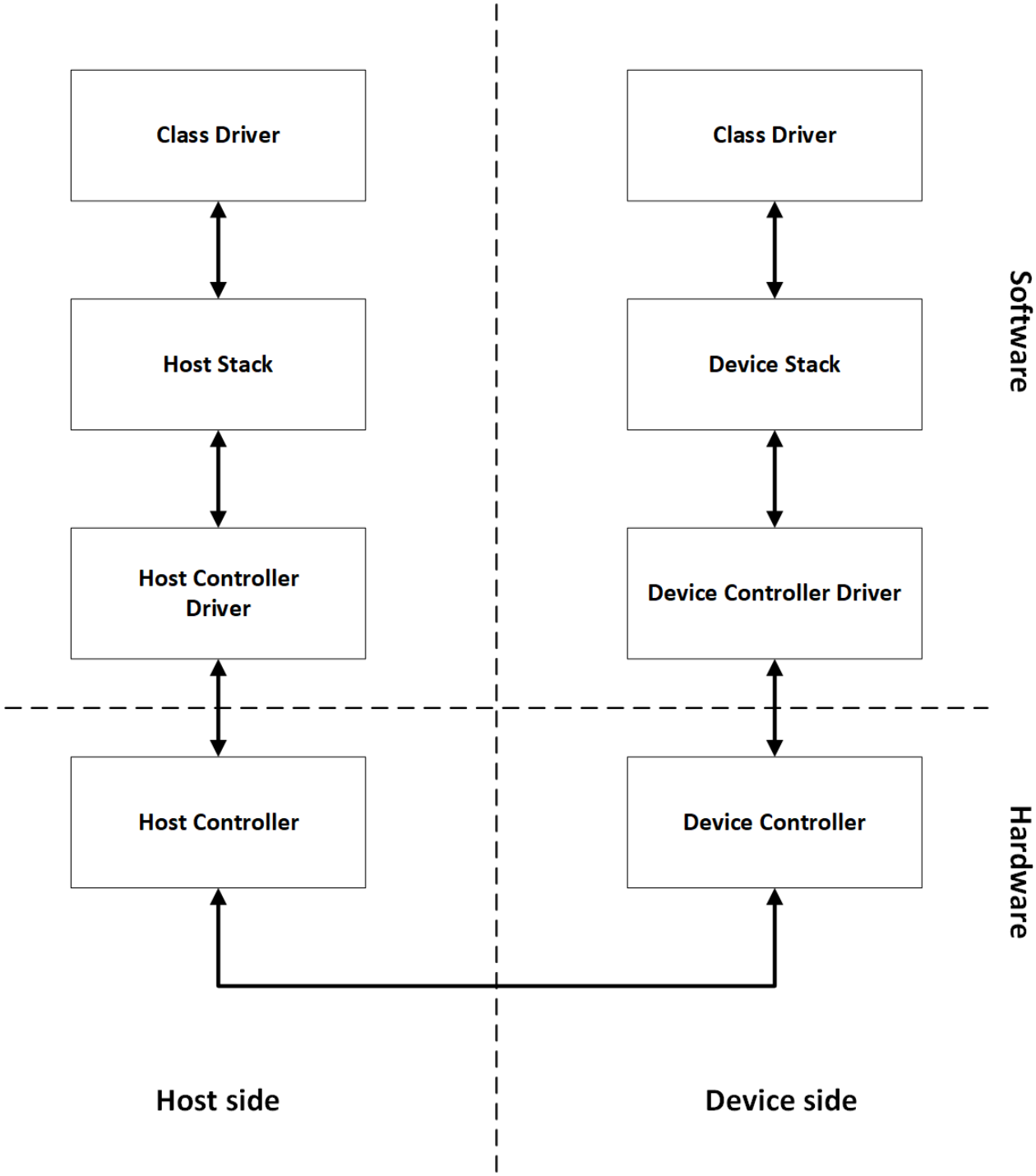
USBX features

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 host controller 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 cascading hubs

- Support for USB power management
- Support for USB OTG
- Export trace events for TraceX

Powerful Services of USBX

Multiple Host Controller Support

USBX can support multiple USB host controllers running concurrently. This feature allows USBX to support the USB 2.0 standard using the backward compatibility scheme associated with most USB 2.0 host controllers on the market today.

USB Software Scheduler

USBX contains a USB software scheduler necessary to support USB controllers that do not have hardware list processing. The USBX software scheduler will organize USB transfers with the correct frequency of service and priority, and will instruct the USB controller to execute each transfer.

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

6/24/2020 • 10 minutes to read

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. 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.

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 10-12 K 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

Azure RTOS USBX can be obtained from our public source code repository at https://github.com/azure-rtos/usbx/.

The following is a list of several important files in the repository:

- *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.

USBX Installation

USBX is installed by cloning the GitHub repository to your local machine. The following is typical syntax for creating a clone of the USBX repository on your PC:

```
git clone https://github.com/azure-rtos/usbx
```

Alternatively you can download a copy of the repository using the download button on the GitHub main page.

You will also find instructions for building the USBX library on the front page of the online repository.

The following general instructions apply to virtually any installation:

1. 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.
2. Add a call to **ux_system_initialize** at or near the beginning of **tx_application_define.** This is where the USBX resources are initialized.
3. Add a call to **ux_host_stack_initialize.**
4. Add one or more calls to initialize the required USBX
5. Add one or more calls to initialize the host controllers available in the system.
6. 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.
7. 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_user.h***.

The list below details each configuration option:

UX_PERIODIC_RATE

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

UX_MAX_CLASS_DRIVER

This value is the maximum number of classes that can be loaded by USBX. This value represents the class container and not the number of instances of a class. For instance, if a particular implementation of USBX needs the hub class, the printer class, and the storage class, then the UX_MAX_CLASS_DRIVER value can be set to 3 regardless of the number of devices that belong to these classes.

UX_MAX_HCD

This value represents the number of different host controllers that are available in the system. For USB 1.1 support, this value will mostly be 1. For USB 2.0 support this value can be more than 1. This value represents the number of concurrent host controllers running at the same time. If for instance, there are two instances of OHCI running or one EHCI and one OHCI controllers running, the UX_MAX_HCD should be set to 2.

UX_MAX_DEVICES

This value represents the maximum number of devices that can be attached to the USB. Normally, the theoretical maximum number on a single USB is 127 devices. This value can be scaled down to conserve memory. It should be noted that this value represents the total number of devices regardless of the number of USB buses in the system.

UX_MAX_ED

This value represents the maximum number of EDs in the controller pool. This number is assigned to one controller only. If multiple instances of controllers are present, this value is used by each individual controller.

UX_MAX_TD and UX_MAX_ISO_TD

This value represents the maximum number of regular and isochronous TDs in the controller pool. This number is assigned to one controller only. If multiple instances of controllers are present, this value is used by each individual controller

UX_THREAD_STACK_SIZE

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

UX_HOST_ENUM_THREAD_STACK_SIZE

This value is the stack size of the USB host enumeration thread. If this symbol I not set, the USBX host enumeration thread stack size is set to UX_THREAD_STACK_SIZE.

UX_HOST_HCD_THREAD_STACK_SIZE

This value is the stack size of the USB host HCD thread. If this symbol I not set, the USBX host HCD thread stack size is set to UX_THREAD_STACK_SIZE.

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_HCD

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

UX_NO_TIME_SLICE

This value actually defines the time slice that will be used for threads. For example, if defined to 0, the ThreadX target port does not use time slices.

UX_MAX_HOST_LUN

This value represents the maximum number of SCSI logical units represented in the host storage class driver

UX_HOST_CLASS_STORAGE_INCLUDE_LEGACY_PROTOCOL_SUPPORT

If defined, this value includes code to handle storage devices that use the CB or CBI protocol (such as floppy disks). It is off by default because these protocols are obsolete, being superseded by the Bulk Only Transport (BOT protocol, which virtually all modern storage devices use.

UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE

If defined, this value causes ux_host_class_hid_keyboard_key_get to only report key changes that is, key presses and key releases. By default, it only reports when a key is down.

UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE_REPORT_KEY_DOWN_ONLY

Only used if UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE is defined. If defined, causes ux_host_class_hid_keyboard_key_get to only report key pressed/down changes; key released/up changes are not reported.

UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE_REPORT_LOCK_KEYS

Only used if UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE is defined. If defined, causes ux_host_class_hid_keyboard_key_get to report lock key (CapsLock/NumLock/ScrollLock) changes.

UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE_REPORT_MODIFIER_KEYS

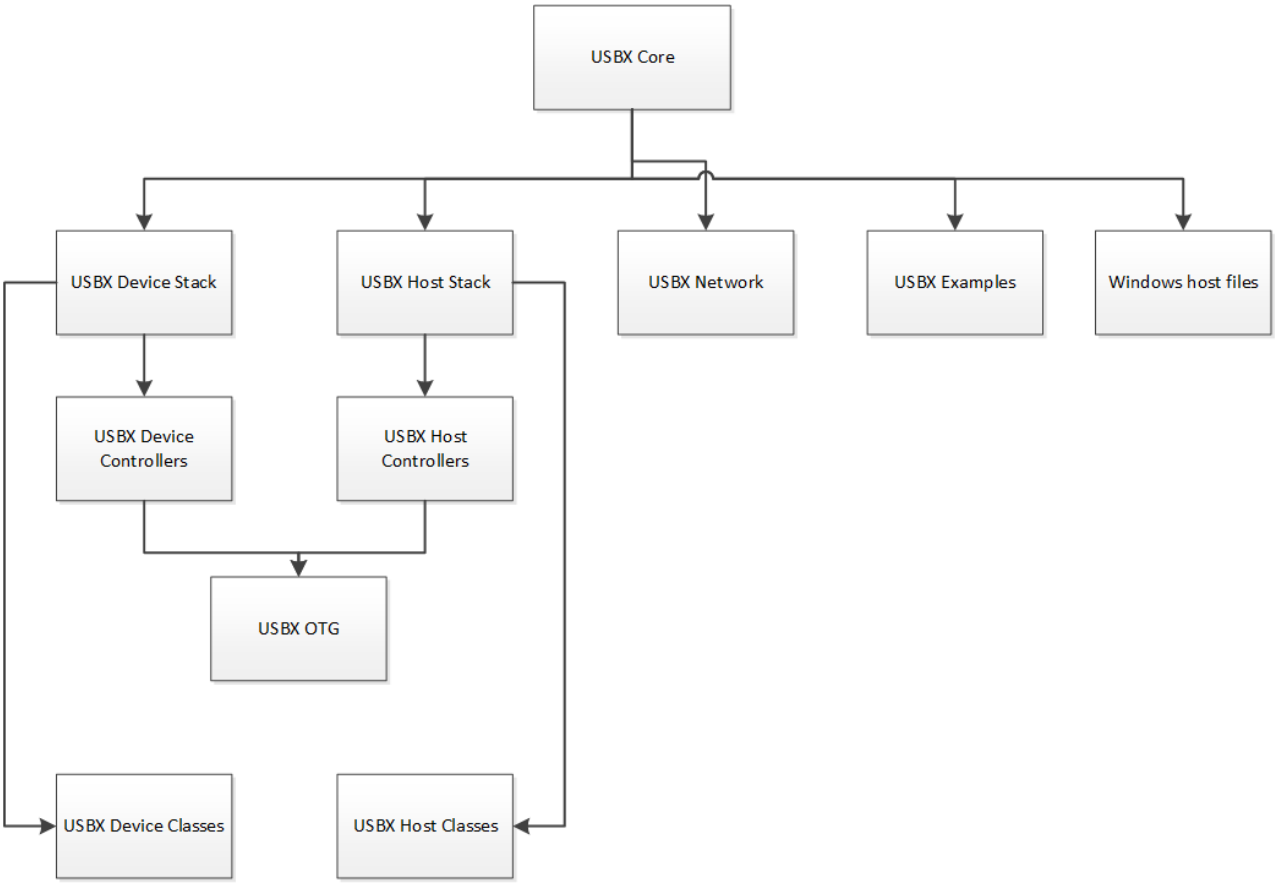
Only used if UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE is defined. If defined, causes ux_host_class_hid_keyboard_key_get to report modifier key (Ctrl/Alt/Shift/GUI) changes.

UX_HOST_CLASS_CDC_ECM_NX_PKPOOL_ENTRIES

If defined, this value represents the number of packets in the CDC-ECM host class. The default is 16.

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

FILE SUFFIX NAME	FILE DESCRIPTION
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 (like OHCI, EHCI controllers amongst others) 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 resources :

```
/* Unitialize USBX Resources */

ux_system_uninitialize();
```

The prototype for the ux_system_initialize is as follows:

```
UINT ux_system_uninitialize(VOID);
```

Definition of USB Host Controllers

It is required to define at least one USB host controller for USBX to operate in host mode. The application initialization file should contain this definition. The following line performs the definition of a generic host controller:

```
ux_host_stack_hcd_register("ux_hcd_controller",
        ux_hcd_controller_initialize, 0xd0000, 0x0a);
```

The ux_host_stack_hcd_register has the following prototype:

```
UINT ux_host_stack_hcd_register(UCHAR *hcd_name,
        UINT (*hcd_initialize_function)(struct UX_HCD_STRUCT *),
        ULONG hcd_param1, ULONG hcd_param2);
```

The ux_host_stack_hcd_register function has the following parameters:

- **hcd_name**: string of the controller name
- **hcd_initialize_function**: initialization function of the controller
- **hcd_param1**: usually the IO value or Memory used by the controller
- **hcd_param2**: usually the IRQ used by the controller

In our previous example:

- "ux_hcd_controller" is the name of the controller
- ux_hcd_controller_initialize is the initialization routine for the host controller
- 0xd0000 is the address at which the host controller registers are visible in memory
- and 0x0a is the IRQ used by the host controller.

Following is an example of the initialization of USBX in host mode with one host controller and several classes.

```
UINT status;

/* Initialize USBX. */
ux_system_initialize(memory_ptr, (128*1024),0,0);

/* The code below is required for installing the USBX host stack. */
status = ux_host_stack_initialize(UX_NULL);

/* If status equals UX_SUCCESS, host stack has been initialized. */

/* Register all the host classes for this USBX implementation. */
status = ux_host_class_register("ux_host_class_hub", ux_host_class_hub_entry);

/* If status equals UX_SUCCESS, host class has been registered. */
status = ux_host_class_register("ux_host_class_storage", ux_host_class_storage_entry);

/* If status equals UX_SUCCESS, host class has been registered. */
status = ux_host_class_register("ux_host_class_printer", ux_host_class_printer_entry);

/* If status equals UX_SUCCESS, host class has been registered. */
status = ux_host_class_register("ux_host_class_audio", ux_host_class_audio_entry);

/* If status equals UX_SUCCESS, host class has been registered. */

/* Register all the USB host controllers available in this system. */
status = ux_host_stack_hcd_register("ux_hcd_controller", ux_hcd_controller_initialize, 0x300000, 0x0a);

/* If status equals UX_SUCCESS, USB host controllers have been registered. */
```

Definition of Host Classes

It is required to define one or more host classes with USBX. A USB class is required to drive a USB device after the USB stack has configured the USB device. A USB class is specific to the device. One or more classes may be required to drive a USB device depending on the number of interfaces contained in the USB device descriptors.

This is an example of the registration of the HUB class:

```
status = ux_host_stack_class_register("ux_host_class_hub", ux_host_class_hub_entry);
```

The function ux_host_class_register has the following prototype:

```
UINT ux_host_stack_class_register(UCHAR *class_name, UINT (*class_entry_address)
            (struct UX_HOST_CLASS_COMMAND_STRUCT *));
```

- **class_name** is the name of the class
- **class_entry_address** is the entry point of the class

In the example of the HUB class initialization:

- "ux_host_class_hub" is the name of the hub class
- ux_host_class_hub_entry is the entry point of the HUB class.

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.

If the demonstration system does not work, try the following things to narrow the problem:

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 **ux_port.h** 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 Host Stack

6/24/2020 • 23 minutes to read

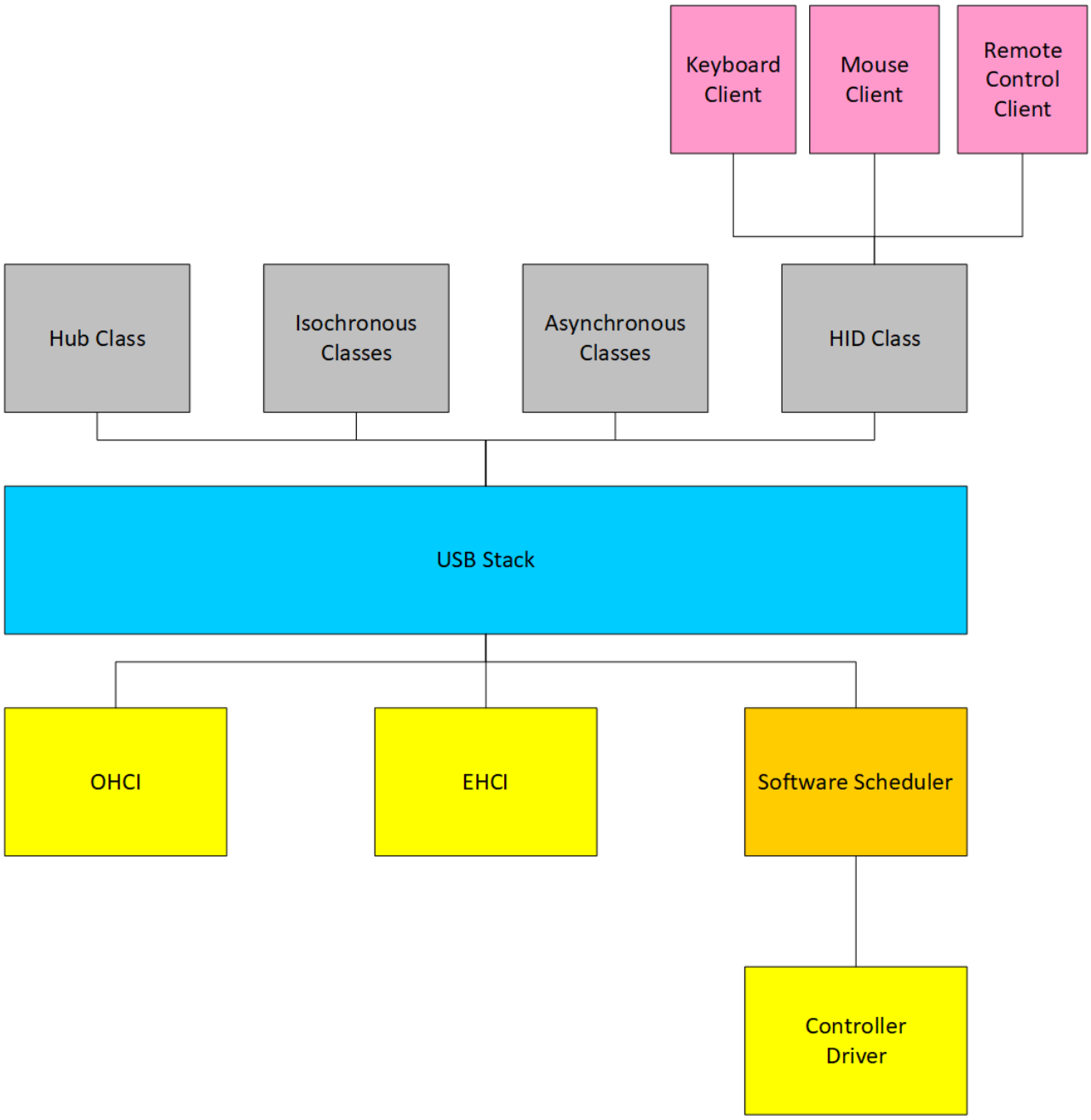
This chapter contains a description of the high performance USBX embedded USB host stack from a functional perspective.

Execution Overview

USBX is composed of several components:

- Initialization
- Application interface calls
- Root Hub
- Hub Class
- Host Classes
- USB Host Stack
- Host controller

The following diagram illustrates the USBX host 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 host facilities, the function ***ux_host_stack_initialize*** must be called. This function will in turn initialize all the resources used by the USBX host stack such as ThreadX threads, mutexes, and semaphores.

It is up to the application initialization to activate at least one USB host controller and one or more USB classes. When the classes have been registered to the stack and the host controller(s) initialization function has been called the bus is active and device discovery can start. If the root hub of the host controller detects an attached device, the USB enumeration thread, in charge of the USB topology, will be wake up and proceed to enumerate the device(s).

It is possible, due to the nature of the root hub and downstream hubs, that all attached USB devices may not have been configured completely when the host controller initialization function returns. It can take several seconds to enumerate all USB devices, especially if there are one or more hubs between the root hub and USB devices.

Application Interface Calls

There are two levels of APIs in USBX:

- USB Host Stack APIs

- USB Host Class APIs

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

USB Host Stack APIs

The host stack APIs are responsible for the registration of USBX components (host classes and host controllers), configuration of devices, and the transfer requests for available device endpoints.

USB Host Class API

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

Root Hub

Each host controller instance has one or more USB root hubs. The number of root hubs is either determined by the nature of the controller or can be retrieved by reading specific registers from the controller.

Hub Class

The hub class is in charge of driving USB hubs. A USB hub can either be a stand-alone hub or as part of a compound device such as a keyboard or a monitor. A hub can be self-powered or bus-powered. Bus-powered hubs have a maximum of four downstream ports and can only allow for the connection of devices that are either self-powered or bus-powered devices that use less than 100mA of power. Hubs can be cascaded. Up to five hubs can be connected to one another.

USB Host Stack

The USB host stack is the centerpiece of USBX. It has three main functions:

- Manage the topology of the USB.
- Bind a USB device to one or more classes.
- Provide an API to classes to perform device descriptor interrogation and USB transfers.

Topology Manager

The USB stack topology thread is awakened when a new device is connected or when a device has been disconnected. Either the root hub or a regular hub can accept device connections. Once a device has been connected to the USB, the topology manager will retrieve the device descriptor. This descriptor will contain the number of possible configurations available for this device. Most devices have one configuration only. Some devices can operate differently according to the available power available on the port where it is connected. If this is the case, the device will have multiple configurations that can be selected depending on the available power. When the device is configured by the topology manager, it is then allowed to draw the amount of power specified in its configuration descriptor.

USB Class Binding

When the device is configured, the topology manager will let the class manager continue the device discovery by looking at the device interface descriptors. A device can have one or more interface descriptors.

An interface represents a function in a device. For instance, a USB speaker has three interfaces, one for audio streaming, one for audio control, and one to manage the various speaker buttons.

The class manager has two mechanisms to join the device interface(s) to one or more classes. It can either use the combination of a PID/VID (product ID and vendor ID) found in the interface descriptor or the combination of Class/Subclass/Protocol.

The PID/VID combination is valid for interfaces that cannot be driven by a generic class. The Class/Subclass/Protocol combination is used by interfaces that belong to a USB-IF certified class such as a printer,

hub, storage, audio, or HID.

The class manager contains a list of registered classes from the initialization of USBX. The class manager will call each class one at a time until one class accepts to manage the interface for that device. A class can only manage one interface. For the example of the USB audio speaker, the class manager will call all the classes for each of the interfaces.

Once a class accepts an interface, a new instance of that class is created. The class manager will then search for the default alternate setting for the interface. A device may have one or more alternate settings for each interface. The alternate setting 0 will be the on used by default until a class decides to change it.

For the default alternate setting, the class manager will mount all the endpoints contained in the alternate setting. If the mounting of each endpoint is successful, the class manager will complete its job by returning to the class that will finish the initialization of the interface.

USBX APIs

The USB stack exports a certain number of APIs for the USB classes to perform interrogation on the device and USB transfers on specific endpoints. These APIs are described in detail in this reference manual.

Host Controller

The host controller driver is responsible for driving a specific type of USB controller. A USB host controller can have multiple controllers inside. For instance, certain Intel PC chipset contain two UHCI controllers. Some USB 2.0 controllers contain multiple instances of an OHCI controller in addition to one instance of the EHCI controller.

The Host controller will manage multiple instance of the same controller only. In order to drive most USB 2.0 host controllers, it will be required to initialize both the OCHI controller and the EHCI controller during the initialization of USBX.

The host controller is responsible for managing the following:

- Root Hub
- Power Management
- Endpoints
- Transfers

Root Hub

The root hub management is responsible for the powering up of each controller port and determining if there is a device inserted or not. This functionality is used by the USBX generic root hub to interrogate the controller downstream ports.

Power Management

The power management processing provides for the handling of suspend/resume signals either in gang mode, therefore affecting all controller downstream ports at the same time, or individually if the controller offers this functionality.

Endpoints

The endpoint management provides for the creation or destruction of physical endpoints to the controller. The physical endpoints are memory entities that are parsed by the controller if the controller supports master DMA or that are written in the controller. The physical endpoints contain transactions information to be performed by the controller.

Transfers

Transfer management provides for a class to perform a transaction on each of the endpoints that have been created. Each logical endpoint contains a component called TRANSFER REQUEST for USB transfer requests. The TRANSFER REQUEST is used by the stack to describe the transaction. This TRANSFER REQUEST is then passed to the stack and to the controller, which may divide it into several sub transactions depending on the capabilities of

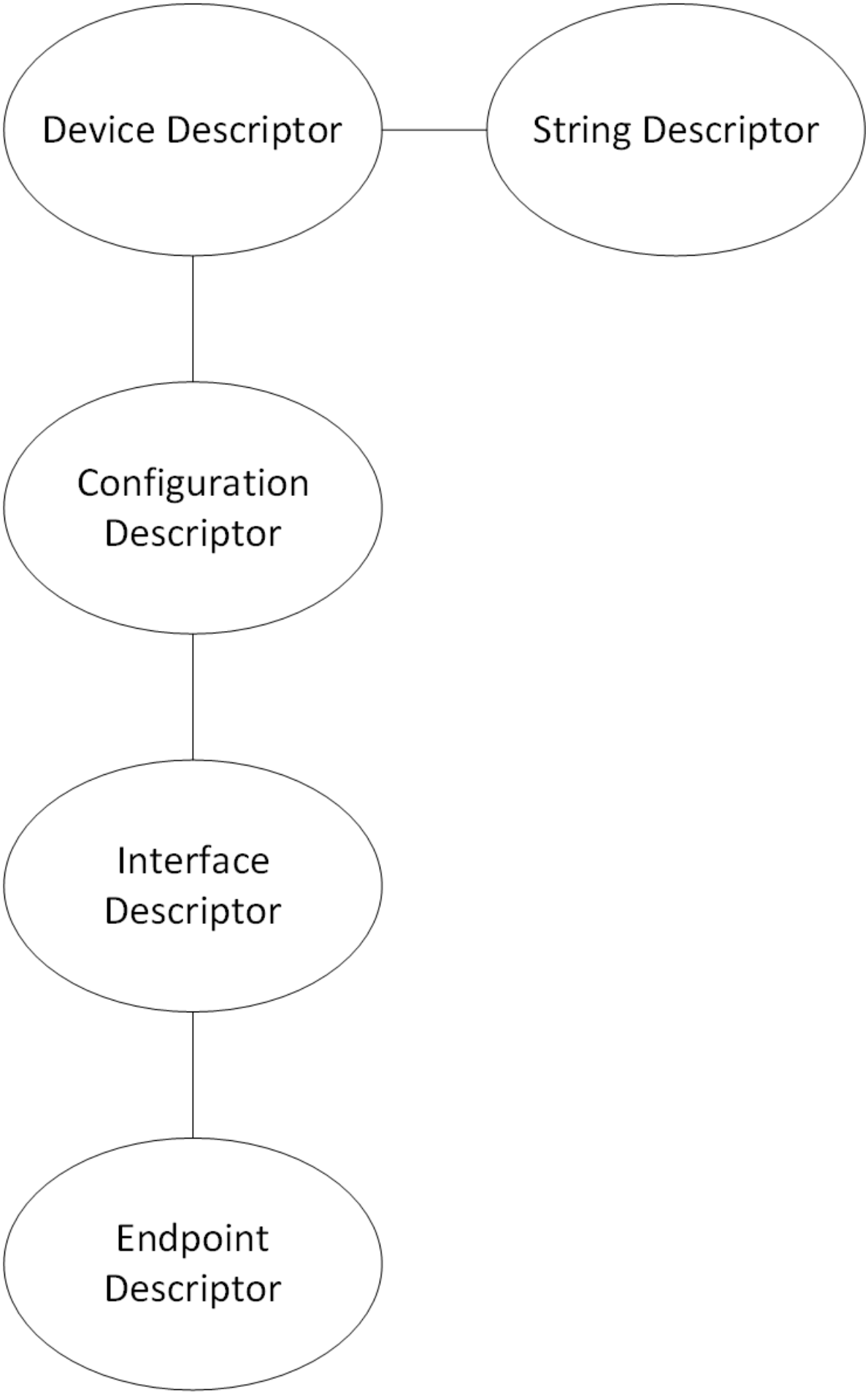
the controller.

USB Device Framework

A USB device is represented by a tree of descriptors. There are six main types of descriptors:

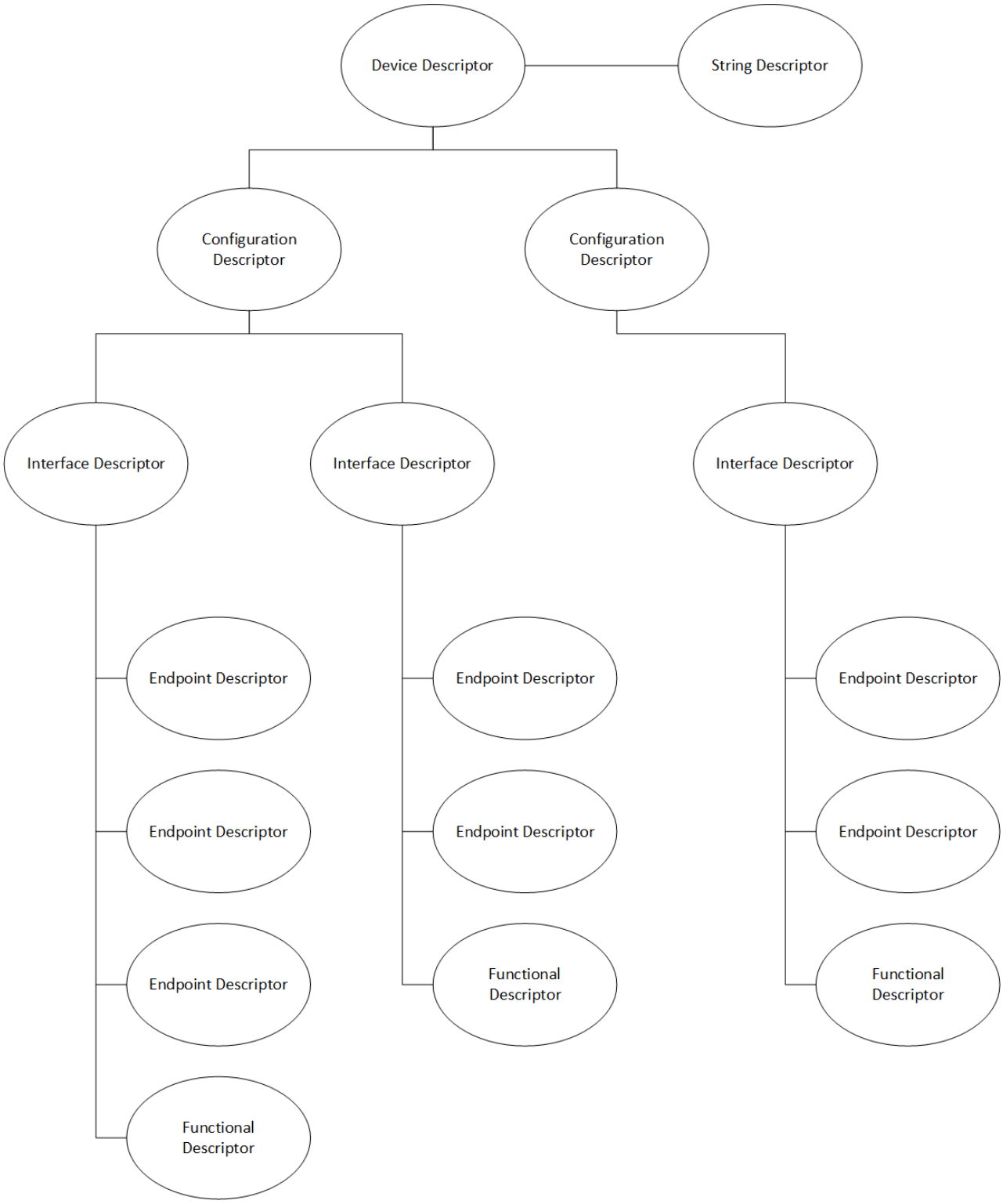
- Device descriptors
- Configuration descriptors
- Interface descriptors
- Endpoint descriptors
- String descriptors
- Functional descriptors

A USB device may have a very simple description and looks like this:



In the above illustration, the device has only one configuration. A single interface is attached to this configuration, indicating that the device has only one function, and it has one endpoint only. Attached to the device descriptor is a string descriptor providing a visible identification of the device.

However, a device may be more complex and may appear as follows:



In the above illustration, the device has two configuration descriptors attached to the device descriptor. This device may indicate that it has two power modes or can be driven by either standard classes or proprietary classes.

Attached to the first configuration are two interfaces indicating that the device has two logical functions. The first function has 3 endpoint descriptors and a functional descriptor. The functional descriptor may be used by the class responsible to drive the interface to obtain further information about this interface normally not found by a generic descriptor.

Device Descriptors

Each USB device has one single device descriptor. This descriptor contains the device identification, the number of configurations supported, and the characteristics of the default control endpoint used for configuring the device.

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
0	BLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	DEVICE Descriptor Type
2	bcdUSB	2	BCD	USB Specification Release Number in BinaryCoded Dec Example: 2.10 is equivalent to 0x210. This field identifies the release of the USB Specification that the device and its descriptors are compliant with.
4	bDeviceClass	1	Class	Class code (assigned by USB-IF). If this field is reset to 0, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and 0xFE, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to 0xFF, the device class is vendor specific.

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
5	bDeviceSubClass	1	SubClass	Subclass code (assigned by USB-IF). These codes are qualified by the value of the bDeviceClass field. If the bDeviceClass field is reset to 0, this field must also be reset to 0. If the bDeviceClass field is not set to 0xFF, all values are reserved for assignment by USB.
6	bDeviceProtocol	1	Protocol	Protocol code (assigned by USB-IF). These codes are qualified by the value of the bDeviceClass and the bDeviceSubClass fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to 0, the device does not use class specific protocols on a device basis. However, it may use class specific protocols on an interface basis. If this field is set to 0xFF, the device uses a vendor specific protocol on a device basis.
7	bMaxPacketSize0	1	Number	Maximum packet size for endpoint zero (only byte sizes of 8, 16, 32, or 64 are valid)
8	idVendor	2	ID	Vendor ID (assigned by USB-IF)

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
10	idProduct	2	ID	Product ID (assigned by the Manufacturer)
12	bcdDevice	2	BCD	Device release number in binary-coded decimal
14	iManufacturer	1	Index	Index of string descriptor describing manufacturer
15	iProduct	1	Index	Index of string descriptor describing product
16	iSerialNumbe	1	Index	Index of string descriptor describing the device's serial number
17	bNumConfigurations	1	Number	Number of possible configurations

USBX defines a USB device descriptor as follows:

```
typedef struct UX_DEVICE_DESCRIPTOR_STRUCT
{
    UINT      bLength;
    UINT      bDescriptorType;
    USHORT    bcdUSB;
    UINT      bDeviceClass;
    UINT      bDeviceSubClass;
    UINT      bDeviceProtocol;
    UINT      bMaxPacketSize0;
    USHORT    idVendor;
    USHORT    idProduct;
    USHORT    bcdDevice;
    UINT      iManufacturer;
    UINT      iProduct;
    UINT      iSerialNumber;
    UINT      bNumConfigurations;
} UX_DEVICE_DESCRIPTOR;
```

The USB device descriptor is part of a device container described as:

```
typedef struct UX_DEVICE_STRUCT
{
    ULONG ux_device_handle;
    ULONG ux_device_type;
    ULONG ux_device_state;
    ULONG ux_device_address;
    ULONG ux_device_speed;
    ULONG ux_device_port_location;
    ULONG ux_device_max_power;
    ULONG ux_device_power_source;
    UINT ux_device_current_configuration;

    TX_SEMAPHORE ux_device_protection_semaphore;
    struct UX_DEVICE_STRUCT *ux_device_parent;
    struct UX_HOST_CLASS_STRUCT *ux_device_class;
    VOID *ux_device_class_instance;
    struct UX_HCD_STRUCT *ux_device_hcd;
    struct UX_CONFIGURATION_STRUCT *ux_device_first_configuration;
    struct UX_DEVICE_STRUCT *ux_device_next_device;
    struct UX_DEVICE_DESCRIPTOR_STRUCT ux_device_descriptor;
    struct UX_ENDPOINT_STRUCT ux_device_control_endpoint;
    struct UX_HUB_TT_STRUCT ux_device_hub_tt[UX_MAX_TT];
} UX_DEVICE;
```

- **ux_device_handle**: Handle of the device. This is typically the address of the instance of this structure for the device.
- **ux_device_type**: Obsolete value. Unused.
- **ux_device_state**: Device State, which can have one of the following values:
  - UX_DEVICE_RESET 0
  - UX_DEVICE_ATTACHED 1
  - UX_DEVICE_ADDRESSED 2
  - UX_DEVICE_CONFIGURED 3
  - UX_DEVICE_SUSPENDED 4
  - UX_DEVICE_RESUMED 5
  - UX_DEVICE_SELF_POWERED_STATE 6
  - UX_DEVICE_SELF_POWERED_STATE 7
  - UX_DEVICE_REMOTE_WAKEUP 8
  - UX_DEVICE_BUS_RESET_COMPLETED 9
  - UX_DEVICE_REMOVED 10
  - UX_DEVICE_FORCE_DISCONNECT 11
- **ux_device_address**: Address of the device after the SET_ADDRESS command has been accepted (from 1 to 127).
- **ux_device_speed**: Speed of the device:
  - UX_LOW_SPEED_DEVICE 0
  - UX_FULL_SPEED_DEVICE 1
  - UX_HIGH_SPEED_DEVICE 2
- **ux_device_port_location**: Index of the port of the parent device (root hub or hub).
- **ux_device_max_power**: Maximum power in mA that the device may take in the selected configuration.
- **ux_device_power_source**: Can be one of the two following values:
  - UX_DEVICE_BUS_POWERED 1
  - UX_DEVICE_SELF_POWERED 2
- **ux_device_current_configuration**: Index of the current configuration being used by this device.
- **ux_device_parent**: Device container pointer of the parent of this device. If the pointer is null, the parent is the

root hub of the controller.

- **ux_device_class**: Pointer to the class type that owns this device.
- **ux_device_class_instance**: Pointer to the instance of the class that owns this device.
- **ux_device_hcd**: USB Host Controller Instance where this device is attached.
- **ux_device_first_configuration**: Pointer to the first configuration container for this device.
- **ux_device_next_device**: Pointer to the next device in the list of device on any of the buses detected by USBX.
- **ux_device_descriptor**: USB device descriptor.
- **ux_device_control_endpoint**: Descriptor of the default control endpoint used by this device.
- **ux_device_hub_tt**: Array of Hub TTs for the device

Configuration Descriptors

The configuration descriptor describes information about a specific device configuration. A USB device may contain one or more configuration descriptors. The *bNumConfigurations* field in the device descriptor indicates the number of configuration descriptors. The descriptor contains a *bConfigurationValue* field with a value that, when used as a parameter to the Set Configuration request, causes the device to assume the described configuration.

The descriptor describes the number of interfaces provided by the configuration. Each interface represents a logical function within the device and may operate independently. For instance a USB audio speaker may have three interfaces, one for audio streaming, one for audio control, and one HID interface to manage the speaker's buttons.

When the host issues a GET_DESCRIPTOR request for the configuration descriptor, all related interface and endpoint descriptors are returned.

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
0	bLength	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	1	Constant	CONFIGURATION
2	wTotalLength	2	Number	Total length of data returned for this configuration. Includes the combined length of all descriptors (configuration, interface, endpoint, and class or vendor specific) returned for this configuration.
4	bNumInterfaces	1	Number	Number of interfaces supported by this configuration.
5	bConfigurationValue	1	Number	Value to use as an argument to Set Configuration to select this configuration.
6	iConfiguration	1	Index	Index of string descriptor describing this configuration.

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
7	bMAttributes	1	Bitmap	Configuration characteristics D7 Bus Powered D6 Self Powered D5 Remote Wakeup D4..0 Reserved (reset to 0) A device configuration that uses power from the bus and a local source sets both D7 and D6. The actual power source at runtime may be determined using the Get Status device request. If a device configuration supports remote wakeup, D5 is set to 1.
8	MaxPower	1	mA	Maximum power consumption of USB device from the bus in this specific configuration when the device is fully operational. Expressed in 2 mA units (e.g., 50 = 100 mA). Note: A device configuration reports whether the configuration is bus-powered or self-powered. Device status reports whether the device is currently self-powered. If a device is disconnected from its external power source, it updates device status to indicate that it is no longer self-powered.

USBX defines a USB configuration descriptor as follows:

```
typedef struct UX_CONFIGURATION_DESCRIPTOR_STRUCT
{
    UINT bLength;
    UINT bDescriptorType;
    USHORT wTotalLength;
    UINT bNumInterfaces;
    UINT bConfigurationValue;
    UINT iConfiguration;
    UINT bmAttributes;
    UINT MaxPower;
} UX_CONFIGURATION_DESCRIPTOR;
```

The USB configuration descriptor is part of a configuration container described as:

```
typedef struct UX_CONFIGURATION_STRUCT
{
    ULONG ux_configuration_handle;
    ULONG ux_configuration_state;
    struct UX_CONFIGURATION_DESCRIPTOR_STRUCT ux_configuration_descriptor;
    struct UX_INTERFACE_STRUCT *ux_configuration_first_interface;
    struct UX_CONFIGURATION_STRUCT *ux_configuration_next_configuration;
    struct UX_DEVICE_STRUCT *ux_configuration_device;
} UX_CONFIGURATION;
```

- **ux_configuration_handle**: Handle of the configuration. This is typically the address of the instance of this structure for the configuration.
- **ux_configuration_state**: State of the configuration.
- **ux_configuration_descriptor**: USB device descriptor.
- **ux_configuration_first_interface**: Pointer to the first interface for this configuration.
- **ux_configuration_next_configuration**: Pointer to the next configuration for the same device.
- **ux_configuration_device**: Pointer to the device owner of this configuration.

Interface Descriptors

The interface descriptor describes a specific interface within a configuration. An interface is a logical function within a USB device. A configuration provides one or more interfaces, each with zero or more endpoint descriptors describing a unique set of endpoints within the configuration. When a configuration supports more than one interface, the endpoint descriptors for a particular interface follow the interface descriptor in the data returned by the GET_DESCRIPTOR request for the specified configuration.

An interface descriptor is always returned as part of a configuration descriptor. An interface descriptor cannot be directly access by a GET_DESCRIPTOR request.

An interface may include alternate settings that allow the endpoints and/or their characteristics to be varied after the device has been configured. The default setting for an interface is always alternate setting zero. A class can select to change the current alternate setting to change the interface behavior and the characteristics of the associated endpoints. The SET_INTERFACE request is used to select an alternate setting or to return to the default setting.

Alternate settings allow a portion of the device configuration to be varied while other interfaces remain in operation. If a configuration has alternate settings for one or more of its interfaces, a separate interface descriptor and its associated endpoints are included for each setting.

If a device configuration contains a single interface with two alternate settings, the GET_DESCRIPTOR request for the configuration would return the configuration descriptor, then the interface descriptor with the *bInterfaceNumber* and *bAlternateSetting* fields set to zero and then the endpoint descriptors for that setting, followed by another interface descriptor and its associated endpoint descriptors. The second interface descriptor's

bInterfaceNumber field would also be set to zero, but the *bAlternateSetting* field of the second interface descriptor would be set to 1 indicating that this alternate setting belongs to the first interface.

An interface may not have any endpoints associated with it, in which case only the default control endpoint is valid for that interface.

Alternate settings are used mainly to change the requested bandwidth for periodic endpoints associated with the interface. For example, a USB speaker streaming interface should have the first alternate setting with a 0 bandwidth demand on its isochronous endpoint. Other alternate settings may select different bandwidth requirements depending on the audio streaming frequency.

The USB descriptor for the interface is as follows:

OFFSET	FIELD	SIZE	VALUE	DESCRIPTOR
0	bLength	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	1	Constant	INTERFACE Descriptor Type
2	bInterfaceNumber	1	Number	Number of interface. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration.
3	bAltenateSetting	1	Number	Value used to select alternate setting for the interface identified in the prior field.
4	bNumEndpoints	1	Number	Number of endpoints used by this interface (excluding endpoint zero). If this value is 0, this interface only uses endpoint zero.
5	bInterfaceClass	1	Class	Class code (assigned by USB) If this field is reset to 0, the interface does not belong to any USB specified device class. If this field is set to 0xFF, the interface class is vendor specific. All other values are reserved for assignment by USB.

OFFSET	FIELD	SIZE	VALUE	DESCRIPTOR
6	bInterfaceSubClass	1	SubClass	Subclass code (assigned by USB). These codes are qualified by the value of the bInterfaceClass field. If the bInterfaceClass field is reset to 0, this field must also be reset to 0. If the bInterfaceClass field is not set to 0xFF, all values are reserved for assignment by USB.
7	bInterfaceProtocol	1	Protocol	Protocol code (assigned by USB). These codes are qualified by the value of the bInterfaceClass and the bInterfaceSubClass fields. If an interface supports class-specific requests, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to 0, the device does not use a class specific protocol on this interface. If this field is set to 0xFF, the device uses a vendor specific protocol for this interface.
8	iInterface	1	Index	Index of string descriptor describing this interface.

USBX defines a USB interface descriptor as follows:

```
typedef struct UX_INTERFACE_DESCRIPTOR_STRUCT
{
    UINT bLength;
    UINT bDescriptorType;
    UINT bInterfaceNumber;
    UINT bAlternateSetting;
    UINT bNumEndpoints;
    UINT bInterfaceClass
    UINT bInterfaceSubClass;
    UINT bInterfaceProtocol;
    UINT iInterface;
} UX_INTERFACE_DESCRIPTOR;
```

The USB interface descriptor is part of an interface container described as:

```
typedef struct UX_INTERFACE_STRUCT
{
    ULONG ux_interface_handle;
    ULONG ux_interface_state;
    ULONG ux_interface_current_alternate_setting;
    struct UX_INTERFACE_DESCRIPTOR_STRUCT ux_interface_descriptor;
    struct UX_HOST_CLASS_STRUCT    *ux_interface_class;
    VOID    *ux_interface_class_instance;
    struct UX_ENDPOINT_STRUCT    *ux_interface_first_endpoint;
    struct UX_INTERFACE_STRUCT    *ux_interface_next_interface;
    struct UX_CONFIGURATION_STRUCT    *ux_interface_configuration;
} UX_INTERFACE;
```

- **ux_interface_handle**: Handle of the interface. This is typically the address of the instance of this structure for the interface.
- **ux_interface_state**: State of the interface.
- **ux_interface_descriptor**: USB interface descriptor.
- **ux_interface_class**: Pointer to the class type that owns this interface.
- **ux_interface_class_instance**: Pointer to the instance of the class that owns this interface.
- **ux_interface_first_endpoint**: Pointer to the first endpoint registered with this interface.
- **ux_interface_next_interface**: Pointer to the next interface associated with the configuration.
- **ux_interface_configuration**: Pointer to the configuration owner of this interface.

Endpoint Descriptors

Each endpoint associated with an interface has its own endpoint descriptor. This descriptor contains the information required by the host stack to determine the bandwidth requirements of each endpoint, the maximum payload associated with the endpoint, its periodicity, and its direction. An endpoint descriptor is always returned by a GET_DESCRIPTOR command for the configuration.

The default control endpoint associated with the device descriptor is not counted as part of the endpoint(s) associated with the interface and therefore not returned in this descriptor.

When the host software requests a change of the alternate setting for an interface, all the associated endpoints and their USB resources are modified according to the new alternate setting.

Except for the default control endpoints, endpoints cannot be share between interfaces.

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
0	bLength	1	Number	Size of this descriptor in bytes.

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
1	bDescriptorType	1	Constant	ENDPOINT Descriptor Type.
2	bEndpointAddress	1	Endpoint	The address of the endpoint on the USB device described by this descriptor. The address is encoded as follows: Bit 3...0: The endpoint number Bit 6...4: Reserved, reset to zero Bit 7: Direction, ignored for control endpoints 0 = OUT endpoint 1 = IN endpoint
3	bmAttributes	1	Bitmap	This field describes the endpoint's attributes when it is configured using the bConfigurationValue. Bits 1..0: Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt If not an isochronous endpoint, bits 5..2 are reserved and must be set to zero. If isochronous, they are defined as follows: Bits 3..2: Synchronization Type 00 = No Synchronization 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4: Usage Type 00 = Data endpoint 01 = Feedback endpoint 10 = Implicit feedback data endpoint 11 = Reserved

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
4	wMaxPacketSize	2	Number	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. For isochronous endpoints, this value is used to reserve the bus time in the schedule, required for the per-(micro)frame data payloads. The pipe may, on an ongoing basis, actually use less bandwidth than that reserved. The device reports, if necessary, the actual bandwidth used via its normal, non-USB defined mechanisms. For all endpoints, bits 10..0 specify the maximum packet size (in bytes). For high-speed isochronous and interrupt endpoints: Bits 12..11 specify the number of additional transaction opportunities per microframe: 00 = None (1 transaction per microframe) 01 = 1 additional (2 per microframe) 10 = 2 additional (3 per microframe) 11 = Reserved Bits 15..13 are reserved and must be set to zero.

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
6	bInterval	1	Number	Number interval for polling endpoint for data transfers. Expressed in frames or microframes depending on the device operating speed (i.e., either 1 millisecond or 125 µs units). For full-/high-speed isochronous endpoints, this value must be in the range from 1 to 16. The **bInterval** is used as the exponent for a 2bInterval-1 value; e.g., a **bInterval** 4 means a period of 8 (24-1). For full-/low-speed interrupt endpoints, the value of this field may be from 1 to 255. For high-speed interrupt endpoints, the **bInterval** is used as the exponent for a 2bInterval-1 value; e.g., a **bInterval** 4 means a period of 8 (24-1). This value must be from 1 to 16. For high-speed bulk/control OUT endpoints, the **bInterval** specify the maximum NAK rate of the endpoint. A value of 0 indicates the endpoint never NAKs. Other values indicate at most one NAK each **bInterval** of microframes. This value must be in the range from 0 to 255.

USBX defines a USB endpoint descriptor as follows:

```
typedef struct UX_ENDPOINT_DESCRIPTOR_STRUCT
{
    UINT bLength;
    UINT bDescriptorType;
    UINT bEndpointAddress;
    UINT bmAttributes;
    USHORT wMaxPacketSize;
    UINT bInterval;
} UX_ENDPOINT_DESCRIPTOR;
```

The USB endpoint descriptor is part of an endpoint container, which is described as follows:

```
typedef struct UX_ENDPOINT_STRUCT {
    ULONG    ux_endpoint_handle;
    ULONG    ux_endpoint_state;
    VOID    *ux_endpoint_ed;
    struct UX_ENDPOINT_DESCRIPTOR_STRUCT    ux_endpoint_descriptor;
    struct UX_ENDPOINT_STRUCT    *ux_endpoint_next_endpoint;
    struct UX_INTERFACE_STRUCT    *ux_endpoint_interface;
    struct UX_DEVICE_STRUCT    *ux_endpoint_device;
    struct UX_TRANSFER REQUEST_STRUCT    ux_endpoint_transfer request;
} UX_ENDPOINT;
```

- **ux_endpoint_handle**: Handle of the endpoint. This is typically the address of the instance of this structure for the endpoint.
- **ux_endpoint_state**: State of the endpoint.
- **ux_endpoint_ed**: Pointer to the physical endpoint at the host controller layer.
- **ux_endpoint_descriptor**: USB endpoint descriptor.
- **ux_endpoint_next_endpoint**: Pointer to the next endpoint that belongs to the same interface.
- **ux_endpoint_interface**: Pointer to the interface that owns this endpoint interface.
- **ux_endpoint_device**: Pointer to the parent device container.
- **ux_endpoint_transfer request**: USB transfer request used to send/receive data from to/from the device.

String descriptors

String descriptors are optional. If a device does not support string descriptors, all references to string descriptors within device, configuration, and interface descriptors must be reset to zero.

String descriptors use UNICODE encoding, thus allowing the support for several character sets. The strings in a USB device may support multiple languages. When requesting a string descriptor, the requester specifies the desired language using a language ID defined by the USB-IF. The list of currently defined USB LANGIDs can be found in the USBX appendix ??. String index zero for all languages returns a string descriptor that contains an array of two-byte LANGID codes supported by the device. It should be noted that the UNICODE string is not 0 terminated. Instead, the size of the string array is computed by subtracting two from the size of the array contained in the first byte of the descriptor.

The USB string descriptor 0 is encoded as follows:

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
0	bLength	1	N+2	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	STRING Descriptor Type

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
2	wLANGID[0]	2	Number	LANGID code 0
...	...]	...	...	...
N	wLANGID[n]	2	Number	LANGID code n

Other USB string descriptors are encoded as follows:

OFFSET	FIELD	SIZE	VALUE	DESCRIPTION
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	STRING Descriptor Type
2	bString	n	Number	UNICODE encoded string

USBX defines a non-zero length USB string descriptor as follows:

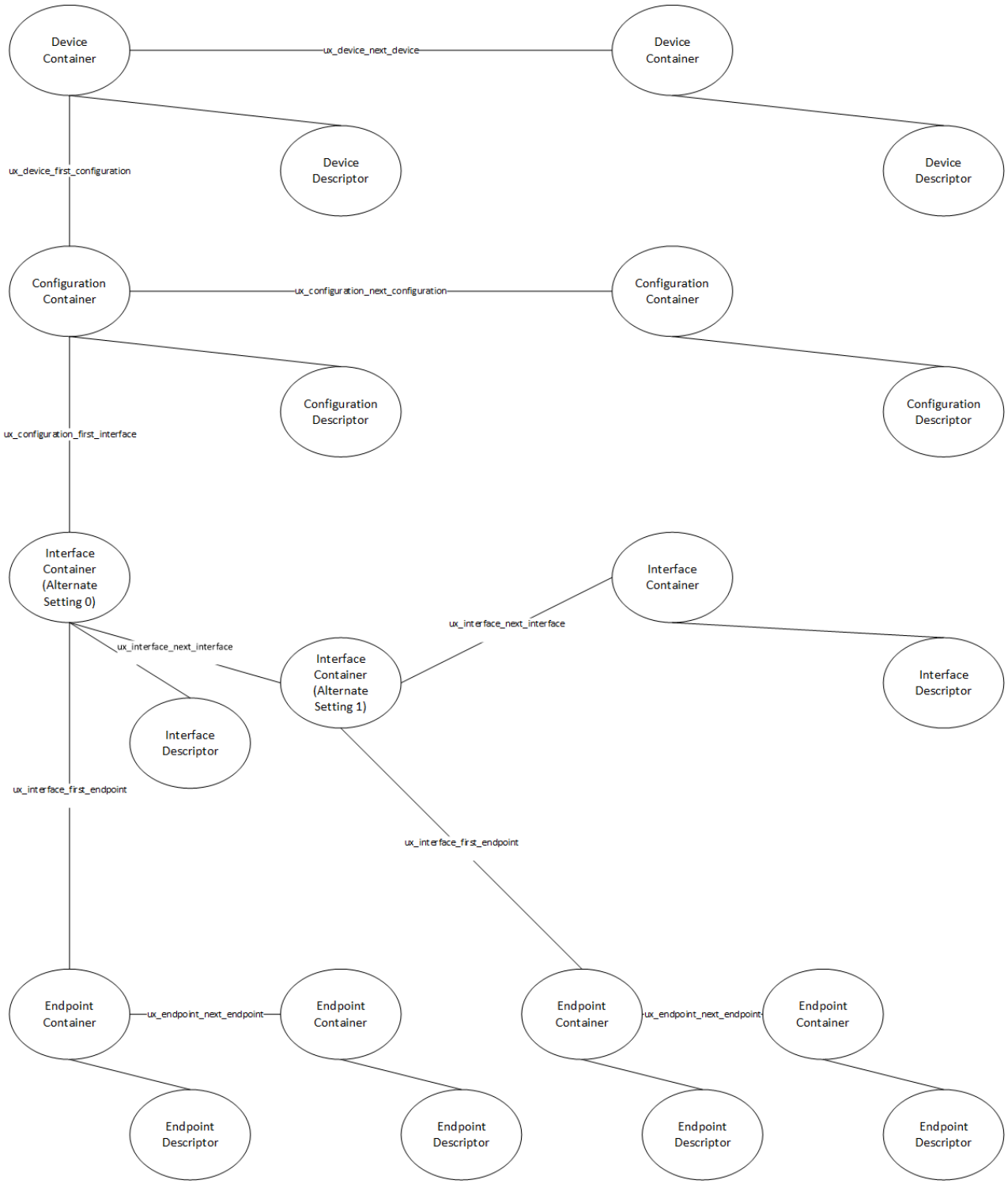
```
typedef struct UX_STRING_DESCRIPTOR_STRUCT
{
    UINT bLength;
    UINT bDescriptorType;
    USHORT bString[1];
} UX_STRING_DESCRIPTOR;
```

Functional Descriptors

Functional descriptors are also known as class-specific descriptors. They normally use the same basic structures as generic descriptors and allow for additional information to be available to the class. For example, in the case of the USB audio speaker, class specific descriptors allow the audio class to retrieve for each alternate setting the type of audio frequency supported.

USBX Device Descriptor Framework in Memory

USBX maintains most device descriptors in memory, that is, all descriptors except the string and functional descriptors. The following diagram shows how these descriptors are stored and related.


Device Container
ux_device_next_device
Device Container
ux_device_first_configuration
Device Descriptor
Device Descriptor
Configuration Container
ux_configuration_next_configuration
Configuration Container
Configuration Descriptor
Configuration Descriptor
ux_configuration_first_interface
Interface Container (Alternate Setting 0)
Interface Container
ux_interface_next_interface
ux_interface_next_interface
Interface Container (Alternate Setting 1)
Interface Descriptor
Interface Descriptor
ux_interface_first_endpoint
ux_interface_first_endpoint
Endpoint Container
ux_endpoint_next_endpoint
Endpoint Container
Endpoint Container
ux_endpoint_next_endpoint
Endpoint Container
Endpoint Descriptor
Endpoint Descriptor
Endpoint Descriptor
Endpoint Descriptor

Chapter 4 - Description of USBX Host Services

6/24/2020 • 12 minutes to read

ux_host_stack_initialize

Initialize USBX for host operation

Prototype

```
UINT ux_host_stack_initialize(UINT (*system_change_function)
    (ULONG, UX_HOST_CLASS *));
```

Description

This function will initialize the USB host stack. The supplied memory area will be setup for USBX internal use. If UX_SUCCESS is returned, USBX is ready for host controller and class registration.

Input Parameter

- **system_change_function** Pointer to optional callback routine for notifying application of device changes.

Return Value

- **UX_SUCCESS** (0x00) Successful initialization.
- **UX_MEMORY_INSUFFICIENT** (0x12) A memory allocation failed.

Example

```
UINT status;

/* Initialize USBX for host operation, without notification. */
status = ux_host_stack_initialize(UX_NULL);

/* If status equals UX_SUCCESS, USBX has been successfully initialized for host operation. */
```

ux_host_stack_endpoint_transfer_abort

Abort all transactions attached to a transfer request for an endpoint

Prototype

```
UINT ux_host_stack_endpoint_transfer_abort(UX_ENDPOINT *endpoint);
```

Description

This function will cancel all transactions active or pending for a specific transfer request attached to an endpoint. It the transfer request has a callback function attached, the callback function will be called with the UX_TRANSACTION_ABORTED status.

Input Parameter

- **endpoint** Pointer to an endpoint.

Return Values

- **UX_SUCCESS** (0x00) No errors.
- **UX_ENDPOINT_HANDLE_UNKNOWN** (0x53) Endpoint handle is not valid.

Example

```
UX_HOST_CLASS_PRINTER *printer;
UINT status;

/* Get the instance for this class. */
printer = (UX_HOST_CLASS_PRINTER *) command ->
    ux_host_class_command_instance;

/* The printer is being shut down. */

printer -> printer_state = UX_HOST_CLASS_INSTANCE_SHUTDOWN;

/* We need to abort transactions on the bulk out pipe. */
status = ux_host_stack_endpoint_transfer_abort
    (printer -> printer_bulk_out_endpoint);

/* If status equals UX_SUCCESS, the operation was successful */
```

ux_host_stack_class_get

Get the pointer to a class container

Prototype

```
UINT ux_host_stack_class_get(UCHAR *class_name, UX_HOST_CLASS **class);
```

Description

This function returns a pointer to the class container. A class needs to obtain its container from the USB stack to search for instances when a class or an application wants to open a device.

> **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** Pointer to the class name.
- **class** A pointer updated by the function call that contains the class container for the name of the class.

Return Values

- **UX_SUCCESS** (0x00) No errors, on return the class field is filed with the pointer to the class container.
- **UX_HOST_CLASS_UNKNOWN** (0x59) Class is unknown by the stack.

Example

```
UX_HOST_CLASS *printer_container;
UINT status;

/* Get the container for this class. */
status = ux_host_stack_class_get("ux_host_class_printer", &printer_container);

/* If status equals UX_SUCCESS, the operation was successful */
```

ux_host_stack_class_register

Register a USB class to the USB stack

Prototype

```
UINT ux_host_stack_class_register(UCHAR *class_name, UINT (*class_entry_address)
    (struct UX_HOST_CLASS_COMMAND_STRUCT *));
```

Description

This function registers a USB class to the USB stack. The class must specify an entry point for the USB stack to send commands such as:

- UX_HOST_CLASS_COMMAND_QUERY
- UX_HOST_CLASS_COMMAND_ACTIVATE
- UX_HOST_CLASS_COMMAND_DESTROY

> **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** Pointer to the name of the class, valid entries are found in the file ux_system_initialize.c under the USB Classes of USBX.
- **class_entry_address** Address of the entry function of the class.

Return Values

- **UX_SUCCESS** (0x00) Class installed successfully.
- **UX_MEMORY_ARRAY_FULL** (0x1a) No more memory to store this class.
- **UX_HOST_CLASS_ALREADY_INSTALLED** (0x58) Host class already installed.

Example:

```
UINT status;

/* Register all the classes for this implementation. */
status = ux_host_stack_class_register("ux_host_class_hub", ux_host_class_hub_entry);

/* If status equals UX_SUCCESS, class was successfully installed. */
```

ux_host_stack_class_instance_create

Create a new class instance for a class container

Prototype

```
UINT ux_host_stack_class_instance_create(UX_HOST_CLASS *class, VOID *class_instance);
```

Description

This function creates a new class instance for a class container. The instance of a class is not contained in the class code to reduce the class complexity. Rather, each class instance is attached to the class container located in the main stack.

Parameters

- **class** Pointer to the class container.
- **class_instance** Pointer to the class instance to be created.

Return Value

- **UX_SUCCESS** (0x00) The class instance was attached to the class container.

Example

```
UINT status;

UX_HOST_CLASS_PRINTER *printer;

/* Obtain memory for this class instance. */

printer = ux_memory_allocate(UX_NO_ALIGN, sizeof(UX_HOST_CLASS_PRINTER));

if (printer == UX_NULL)
    return(UX_MEMORY_INSUFFICIENT);

/* Store the class container into this instance. */
printer -> printer_class = command -> ux_host_class;

/* Create this class instance. */
status = ux_host_stack_class_instance_create(printer -> printer_class, (VOID *)printer);

/* If status equals UX_SUCCESS, the class instance was successfully created and attached to the class
container. */
```

ux_host_stack_class_instance_destroy

Destroy a class instance for a class container

Prototype

```
UINT ux_host_stack_class_instance_destroy(UX_HOST_CLASS *class, VOID *class_instance);
```

Description

This function destroys a class instance for a class container.

Parameters

- **class** Pointer to the class container.
- **class_instance** Pointer to the instance to destroy.

Return Values

- **UX_SUCCESS** (0x00) The class instance was destroyed.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) The class instance is not attached to the class container.

Example

```
UINT status;
UX_HOST_CLASS_PRINTER *printer;

/* Get the instance for this class. */
printer = (UX_HOST_CLASS_PRINTER *) command -> ux_host_class_command_instance;

/* The printer is being shut down. */
printer -> printer_state = UX_HOST_CLASS_INSTANCE_SHUTDOWN;

/* Destroy the instance. */
status = ux_host_stack_class_instance_destroy(printer -> printer_class, (VOID *) printer);

/* If status equals UX_SUCCESS, the class instance was successfully destroyed. */
```

ux_host_stack_class_instance_get

Get a class instance pointer for a specific class

Prototype

```
UINT ux_host_stack_class_instance_get(UX_HOST_CLASS *class,
    UINT class_index, VOID **class_instance);
```

Description

This function returns a class instance pointer for a specific class. The instance of a class is not contained in the class code to reduce the class complexity. Rather, each class instance is attached to the class container. This function is used to search for class instances within a class container.

Parameters

- **class** Pointer to the class container.
- **class_index** An index to be used by the function call within the list of attached classes to the container.
- **class_instance** Pointer to the instance to be returned by the function call.

Return Values

- **UX_SUCCESS** (0x00) The class instance was found.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) There are no more class instances attached to the class container.

Example

```
UINT status;

UX_HOST_CLASS_PRINTER *printer;

/* Obtain memory for this class instance. */
printer = ux_memory_allocate(UX_NO_ALIGN, sizeof(UX_HOST_CLASS_PRINTER));

if (printer == UX_NULL) return(UX_MEMORY_INSUFFICIENT);

/* Search for instance index 2. */
status = ux_host_stack_class_instance_get(class, 2, (VOID *) printer);

/* If status equals UX_SUCCESS, the class instance was found. */
```

ux_host_stack_device_configuration_get

Get a pointer to a configuration container

Prototype

```
UINT ux_host_stack_device_configuration_get(UX_DEVICE *device,
    UINT configuration_index, UX_CONFIGURATION *configuration);
```

Description

This function returns a configuration container based on a device handle and a configuration index.

Parameters

- **device** Pointer to the device container that owns the configuration requested.
- **configuration_index** Index of the configuration to be searched.
- **configuration** Address of the pointer to the configuration container to be returned.

Return Values

- **UX_SUCCESS** (0x00) The configuration was found.
- **UX_DEVICE_HANDLE_UNKNOWN** (0x50) The device container does not exist.
- **UX_CONFIGURATION_HANDLE_UNKNOWN** (0x51) The configuration handle for the index does not exist.

Example

```
UINT status;

UX_HOST_CLASS_PRINTER *printer;

/* If the device has been configured already, we don't need to do it
again. */

if (printer -> printer_device -> ux_device_state == UX_DEVICE_CONFIGURED)
    return(UX_SUCCESS);

/* A printer normally has one configuration, retrieve 1st configuration only. */

status = ux_host_stack_device_configuration_get(printer -> printer_device,
    0, configuration);

/* If status equals UX_SUCCESS, the configuration was found. */
```

ux_host_stack_device_configuration_select

Select a specific configuration for a device

Prototype

```
UINT ux_host_stack_device_configuration_select (UX_CONFIGURATION *configuration);
```

Description

This function selects a specific configuration for a device. When this configuration is set to the device, by default, each device interface and its associated alternate setting 0 is activated on the device. If the device/interface class wishes to change the setting of a particular interface, it needs to issue a **ux_host_stack_interface_setting_select** service call.

Parameters

- **configuration** Pointer to the configuration container that is to be enabled for this device.

Return Values

- **UX_SUCCESS** (0x00) The configuration selection was successful.
- **UX_CONFIGURATION_HANDLE_UNKNOWN** (0x51) The configuration handle does not exist.
- **UX_OVER_CURRENT_CONDITION** (0x43) An over current condition exists on the bus for this configuration.

Example

```
UINT status;

UX_HOST_CLASS_PRINTER *printer;

/* If the device has been configured already, we don't need to do it again. */
if (printer -> printer_device -> ux_device_state == UX_DEVICE_CONFIGURED)
    return(UX_SUCCESS);

/* A printer normally has one configuration - retrieve 1st configuration only. */
status = ux_host_stack_device_configuration_get(printer -> printer_device, 0,configuration);

/* If status equals UX_SUCCESS, the configuration selection was successful. */

/* If valid configuration, ask USBX to set this configuration. */
status = ux_host_stack_device_configuration_select(configuration);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_stack_device_get

Get a pointer to a device container

Prototype

```
UINT ux_host_stack_device_get(ULONG device_index, UX_DEVICE *device);
```

Description

This function returns a device container based on its index. The device index starts with 0. Note that the index is a ULONG because we could have several controllers and a byte index might not be enough. The device index should not be confused with the device address that is bus specific.

Parameters

- **device_index** Index of the device.
- **device** Address of the pointer for the device container to return.

Return Values

- **UX_SUCCESS** (0x00) The device container exists and is returned
- **UX_DEVICE_HANDLE_UNKNOWN** (0x50) Device unknown

Example

```
UINT status;

/* Locate the first device in USBX. */
status = ux_host_stack_device_get(0, device);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_stack_interface_endpoint_get

Get an endpoint container

Prototype

```
UINT ux_host_stack_interface_endpoint_get(UX_INTERFACE *interface,
    UINT endpoint_index, UX_ENDPOINT *endpoint);
```

Description

This function returns an endpoint container based on the interface handle and an endpoint index. It is assumed that the alternate setting for the interface has been selected or the default setting is being used prior to the endpoint(s) being searched.

Parameters

- **interface** Pointer to the interface container that contains the endpoint requested.
- **endpoint_index** Index of the endpoint in this interface.
- **endpoint** Address of the endpoint container to be returned.

Return Values

- **UX_SUCCESS** (0x00) The endpoint container exists and is returned.
- **UX_INTERFACE_HANDLE_UNKNOWN** (0x52) Interface specified does not exist.
- **UX_ENDPOINT_HANDLE_UNKNOWN** (0x53) Endpoint index does not exist.

Example

```
UINT status;
UX_HOST_CLASS_PRINTER *printer;

for(endpoint_index = 0;
    endpoint_index < printer -> printer_interface ->
        ux_interface_descriptor.bNumEndpoints;
    endpoint_index++)
{
    status = ux_host_stack_interface_endpoint_get (printer -> printer_interface,
        endpoint_index, &endpoint);

    if (status == UX_SUCCESS)
    {
        /* Check if endpoint is bulk and OUT. */
        if (((endpoint -> ux_endpoint_descriptor.bEndpointAddress &
            UX_ENDPOINT_DIRECTION) == UX_ENDPOINT_OUT) &&
            ((endpoint -> ux_endpoint_descriptor.bmAttributes &
            UX_MASK_ENDPOINT_TYPE) == UX_BULK_ENDPOINT))
        return(UX_SUCCESS);
    }
}
```

ux_host_stack_hcd_register

Register a USB controller to the USB stack

Prototype

```
UINT ux_host_stack_hcd_register(UCHAR *hcd_name,
    UINT (*hcd_function)(struct UX_HCD_STRUCT *),
    ULONG hcd_param1, ULONG hcd_param2);
```

Description

This function registers a USB controller to the USB stack. It mainly allocates the memory used by this controller and passes the initialization command to the controller.

Parameters

- **hcd_name** Name of the host controller
- **hcd_function** The function in the host controller responsible for the initialization.
- **hcd_param1** The IO or memory resource used by the hcd.
- **hcd_param2** The IRQ used by the host controller.

Return Values

- **UX_SUCCESS** (0x00) The controller was initialized properly.
- **UX_MEMORY_INSUFFICIENT** (0x12) Not enough memory for this controller.
- **UX_PORT_RESET_FAILED** (0x31) The reset of the controller failed.
- **UX_CONTROLLER_INIT_FAILED** (0x32) The controller failed to initialize properly.

Example

```
UINT status;

/* Initialize a host controller mapped at address 0xd0000 and using IRQ 10. */

status = ux_host_stack_hcd_register("ux_hcd_controller",
    ux_hcd_controller_initialize, 0xd0000, 0x0a);

/* If status equals UX_SUCCESS, the controller was initialized properly. */

/* Note that the application must also setup a call to the
    interrupt handler for the controller.
    The function for the controller is called _ux_hch_controller_interrupt_handler. */
```

ux_host_stack_configuration_interface_get

Get an interface container pointer

Prototype

```
UINT ux_host_stack_configuration_interface_get (UX_CONFIGURATION *configuration,
    UINT interface_index, UINT alternate_setting_index, UX_INTERFACE **interface);
```

Description

This function returns an interface container based on a configuration handle, an interface index, and an alternate setting index.

Parameters

- **configuration** Pointer to the configuration container that owns the interface.
- **interface_index** Interface index to be searched.
- **alternate_setting_index** Alternate setting within the interface to search.
- **interface** Address of the interface container pointer to be returned.

Return Values

- **UX_SUCCESS** (0x00) The interface container for the interface index and the alternate setting was found and returned.
- **UX_CONFIGURATION_HANDLE_UNKNOWN** (0x51) The configuration does not exist.
- **UX_INTERFACE_HANDLE_UNKNOWN** (0x52) The interface does not exist.

Example

```
UINT status;

/* Search for the default alternate setting on the first interface for the printer. */
status = ux_host_stack_configuration_interface_get(configuration, 0, 0,
    &printer -> printer_interface);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_stack_interface_setting_select

Select an alternate setting for an interface

Prototype

```
UINT ux_host_stack_interface_setting_select(UX_INTERFACE *interface);
```

Description

This function selects a specific alternate setting for a given interface belonging to the selected configuration. This function is used to change from the default alternate setting to a new setting or to go back to the default alternate setting. When a new alternate setting is selected, the previous endpoint characteristics are invalid and should be reloaded.

Input Parameter

- **interface** Pointer to the interface container whose alternate setting is to be selected.

Return Values

- **UX_SUCCESS** (0x00) The alternate setting for this interface has been successfully selected.
- **UX_INTERFACE_HANDLE_UNKNOWN** (0x52) The interface does not exist.

Example

```
UINT status;

/* Select a new alternate setting for this interface. */
status = ux_host_stack_interface_setting_select(interface);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_stack_transfer_request_abort

Abort a pending transfer request

Prototype

```
UINT ux_host_stack_transfer_request_abort(UX_TRANSFER REQUEST *transfer request);
```

Description

This function aborts a pending transfer request that has been previously submitted. This function only cancels a specific transfer request. The call back to the function will have the UX_TRANSFER REQUEST_STATUS_ABORT status.

Parameters

- **transfer request** Pointer to the transfer request to be aborted.

Return Values

- **UX_SUCCESS** (0x00) The USB transfer for this transfer request was canceled.

Example

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_stack_transfer_request_abort(transfer request);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_stack_transfer_request

Request a USB transfer

Prototype

```
UINT ux_host_stack_transfer_request(UX_TRANSFER REQUEST *transfer request);
```

Description

This function performs a USB transaction. On entry the transfer request gives the endpoint pipe selected for this transaction and the parameters associated with the transfer (data payload, length of transaction). For Control pipe, the transaction is blocking and will only return when the three phases of the control transfer have been completed or if there is a previous error. For other pipes, the USB stack will schedule the transaction on the USB but will not wait for its completion. Each transfer request for non-blocking pipes has to specify a completion routine handler.

When the function call returns, the status of the transfer request should be examined as it contains the result of the transaction.

Input Parameter

- **transfer_request** Pointer to the transfer request. The transfer request contains all the necessary information required for the transfer.

Return Values

- **UX_SUCCESS** (0x00) The USB transfer for this transfer request was scheduled properly. The status code of the transfer request should be examined when the transfer request completes.
- **UX_MEMORY_INSUFFICIENT** (0x12) Not enough memory to allocate the necessary controller resources.
- **UX_TRANSFER_NOT_READY** (0x25) The device was in an invalid state – must be ATTACHED,ADDRESSED, or CONFIGURED.

Example:

```
UINT status;

/* Create a transfer request for the SET_CONFIGURATION request. No data for this request. */
transfer_request -> ux_transfer_request_requested_length = 0;
transfer_request -> ux_transfer_request_function = UX_SET_CONFIGURATION;
transfer_request -> ux_transfer_request_type =
    UX_REQUEST_OUT |
    UX_REQUEST_TYPE_STANDARD |
    UX_REQUEST_TARGET_DEVICE;

transfer_request -> ux_transfer_request_value = (USHORT)
    configuration -> ux_configuration_descriptor.bConfigurationValue;
transfer_request -> ux_transfer_request_index = 0;

/* Send request to HCD layer. */
status = ux_host_stack_transfer_request(transfer_request);

/* If status equals UX_SUCCESS, the operation was successful. */
```

Chapter 5 - USBX Host Classes API

6/24/2020 • 13 minutes to read

This chapter covers all the exposed APIs of the USBX host classes. The following APIs for each class are described in detail:

- HID class
- CDC-ACM class
- CDC-ECM class
- Storage class

ux_host_class_hid_client_register

Register a HID client to the HID class

Prototype

```
UINT ux_host_class_hid_client_register(UCHAR *hid_client_name,
    UINT (*hid_client_handler)
    (struct UX_HOST_CLASS_HID_CLIENT_COMMAND_STRUCT *));
```

Description

This function is used to register a HID client to the HID class. The HID class needs to find a match between a HID device and HID client before requesting data from this device.

> **NOTE**
> The C string of hid_client_name must be NULL-terminated and the length of it (without the NULL-terminator itself) must be no larger than UX_HOST_CLASS_HID_MAX_CLIENT_NAME_LENGTH.

Parameters

- **hid_client_name** Pointer to the HID client name.
- **hid_client_handler** Pointer to the HID client handler.

Return Values

- **UX_SUCCESS** (0x00) The data transfer was completed
- **UX_MEMORY_INSUFFICIENT** (0x12) Memory allocation for client failed.
- **UX_MEMORY_ARRAY_FULL** (0x1a) Max clients already registered.
- **UX_HOST_CLASS_ALREADY_INSTALLED** (0x58) This class already exists

Example

```
UINT status;

/* The following example illustrates how to register a HID client, in
this case a USB mouse, to the HID class. */

status = ux_host_class_hid_client_register("ux_host_class_hid_client_mouse",
    ux_host_class_hid_mouse_entry);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_report_callback_register

Register a callback from the HID class

Prototype

```
UINT ux_host_class_hid_report_callback_register(UX_HOST_CLASS_HID *hid,
    UX_HOST_CLASS_HID_REPORT_CALLBACK *call_back);
```

Description

This function is used to register a callback from the HID class to the HID client when a report is received.

Parameters

- **hid** Pointer to the HID class instance
- **call_back** Pointer to the call_back structure

Return values

- **UX_SUCCESS** (0x00) The data transfer was completed
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) Invalid HID instance.
- **UX_HOST_CLASS_HID_REPORT_ERROR** (0x79) Error in the report callback registration.

Example

```
UINT status;

/* This example illustrates how to register a HID client, in this case a USB mouse, to the HID class. In this
case, the HID client is asking the HID class to call the client for each usage received in the HID report. */

call_back.ux_host_class_hid_report_callback_id = 0;
call_back.ux_host_class_hid_report_callback_function = ux_host_class_hid_mouse_callback;

call_back.ux_host_class_hid_report_callback_buffer = UX_NULL;
call_back.ux_host_class_hid_report_callback_flags = UX_HOST_CLASS_HID_REPORT_INDIVIDUAL_USAGE;

call_back.ux_host_class_hid_report_callback_length = 0;

status = ux_host_class_hid_report_callback_register(hid, &call_back);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_periodic_report_start

Start the periodic endpoint for a HID class instance

Prototype

```
UINT ux_host_class_hid_periodic_report_start(UX_HOST_CLASS_HID *hid);
```

Description

This function is used to start the periodic (interrupt) endpoint for the instance of the HID class that is bound to this HID client. The HID class cannot start the periodic endpoint until the HID client is activated and therefore it is left to the HID client to start this endpoint to receive reports.

Input Parameter

- **hid** Pointer to the HID class instance.

Return Values

- **UX_SUCCESS** (0x00) Periodic reporting successfully started.
- **ux_host_class_hid_PERIODIC_REPORT_ERROR** (0x7A) Error in the periodic report.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID class instance does not exist.

Example

```
UINT status;

/* The following example illustrates how to start the periodic
endpoint. */

status = ux_host_class_hid_periodic_report_start(hid);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_periodic_report_stop

Stop the periodic endpoint for a HID class instance

Prototype

```
UINT ux_host_class_hid_periodic_report_stop(UX_HOST_CLASS_HID *hid);
```

Description

This function is used to stop the periodic (interrupt) endpoint for the instance of the HID class that is bound to this HID client. The HID class cannot stop the periodic endpoint until the HID client is deactivated, all its resources freed and therefore it is left to the HID client to stop this endpoint.

Input Parameter

- **hid** Pointer to the HID class instance.

Return Values

- **UX_SUCCESS** (0x00) Periodic reporting successfully stopped.
- **ux_host_class_hid_PERIODIC_REPORT_ERROR** (0x7A) Error in the periodic report.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID class instance does not exist

Example

```
UINT status;

/* The following example illustrates how to stop the periodic endpoint. */

status = ux_host_class_hid_periodic_report_stop(hid);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_report_get

Get a report from a HID class instance

Prototype

```
UINT ux_host_class_hid_report_get(UX_HOST_CLASS_HID *hid,
    UX_HOST_CLASS_HID_CLIENT_REPORT *client_report);
```

Description

This function is used to receive a report directly from the device without relying on the periodic endpoint. This report is coming from the control endpoint but its treatment is the same as though it were coming on the periodic endpoint.

Parameters

- **hid** Pointer to the HID class instance.
- **client_report** Pointer to the HID client report.

Return Values

- **UX_SUCCESS** (0x00) The report was successfully received.
- **UX_HOST_CLASS_HID_REPORT_ERROR** (0x70) Either client report was invalid or error during transfer.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID class instance does not exist.
- **UX_BUFFER_OVERFLOW** (0x5d) The buffer supplied is not big enough to accommodate the uncompressed report.

Example

```
UX_HOST_CLASS_HID_CLIENT_REPORT input_report;

UINT status;

/* The following example illustrates how to get a report. */

input_report.ux_host_class_hid_client_report = hid_report;
input_report.ux_host_class_hid_client_report_buffer = buffer;
input_report.ux_host_class_hid_client_report_length = length;
input_report.ux_host_class_hid_client_flags = UX_HOST_CLASS_HID_REPORT_INDIVIDUAL_USAGE;

status = ux_host_class_hid_report_get(hid, &input_report);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_report_set

Send a report

Prototype

```
UINT ux_host_class_hid_report_set(UX_HOST_CLASS_HID *hid,
    UX_HOST_CLASS_HID_CLIENT_REPORT *client_report);
```

Description

This function is used to send a report directly to the device.

Parameters

- **hid** Pointer to the HID class instance.
- **client_report** Pointer to the HID client report.

Return Values

- **UX_SUCCESS** (0x00) The report was successfully sent.
- **UX_HOST_CLASS_HID_REPORT_ERROR** (0x70) Either client report was invalid or error during transfer.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID class instance does not exist.
- **UX_HOST_CLASS_HID_REPORT_OVERFLOW** (0x5d) The buffer supplied is not big enough to accommodate the uncompressed report.

Example

```
/* The following example illustrates how to send a report. */

UX_HOST_CLASS_HID_CLIENT_REPORT input_report;

input_report.ux_host_class_hid_client_report = hid_report;
input_report.ux_host_class_hid_client_report_buffer = buffer;
input_report.ux_host_class_hid_client_report_length = length;
input_report.ux_host_class_hid_client_report_flags = UX_HOST_CLASS_HID_REPORT_INDIVIDUAL_USAGE;

status = ux_host_class_hid_report_set(hid, &input_report);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_mouse_buttons_get

Get mouse buttons

Prototype

```
UINT ux_host_class_hid_mouse_buttons_get(UX_HOST_CLASS_HID_MOUSE *mouse_instance,
    ULONG *mouse_buttons);
```

Description

This function is used to get the mouse buttons

Parameters

- **mouse_instance** Pointer to the HID mouse instance.
- **mouse_buttons** Pointer to the return buttons.

Return Values

- **UX_SUCCESS** (0x00) Mouse button successfully retrieved.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID class instance does not exist.

Example

```
/* The following example illustrates how to obtain mouse buttons. */

UX_HOST_CLASS_HID_MOUSE *mouse_instance;

ULONG mouse_buttons;

status = ux_host_class_hid_mouse_button_get(mouse_instance, &mouse_buttons);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_mouse_position_get

Get mouse position

Prototype

```
UINT ux_host_class_hid_mouse_position_get(UX_HOST_CLASS_HID_MOUSE *mouse_instance,
    SLONG *mouse_x_position, SLONG *mouse_y_position);
```

Description

This function is used to get the mouse position in x & y coordinates

Parameters

- **mouse_instance** Pointer to the HID mouse instance.
- **mouse_x_position** Pointer to the x coordinate.
- **mouse_y_position** Pointer to the y coordinate.

Return Values

- **UX_SUCCESS** (0x00) X & Y coordinates successfully retrieved.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID class instance does not exist.

Example

```
/* The following example illustrates how to obtain mouse coordinates. */

UX_HOST_CLASS_HID_MOUSE *mouse_instance;

SLONG mouse_x_position;
SLONG mouse_y_position;

status = ux_host_class_hid_mouse_position_get(mouse_instance,
    &mouse_x_position, &mouse_y_position);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_keyboard_key_get

Get keyboard key and state

Prototype

```
UINT ux_host_class_hid_keyboard_key_get(UX_HOST_CLASS_HID_KEYBOARD *keyboard_instance,
    ULONG *keyboard_key, ULONG *keyboard_state);
```

Description

This function is used to get the keyboard key and state

Parameters

- **keyboard_instance** Pointer to the HID keyboard instance.
- **keyboard_key** Pointer to keyboard key container.
- **keyboard_state** Pointer to the keyboard state container.

Return Values

- **UX_SUCCESS** (0x00) Key and state successfully retrieved.
- **UX_ERROR** (0xff) Nothing to report.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID class instance does not exist.

The keyboard state can have the following values:

- UX_HID_KEYBOARD_STATE_KEY_UP 0x10000
- UX_HID_KEYBOARD_STATE_NUM_LOCK 0x0001
- UX_HID_KEYBOARD_STATE_CAPS_LOCK 0x0002
- UX_HID_KEYBOARD_STATE_SCROLL_LOCK 0x0004
- UX_HID_KEYBOARD_STATE_MASK_LOCK 0x0007
- UX_HID_KEYBOARD_STATE_LEFT_SHIFT 0x0100
- UX_HID_KEYBOARD_STATE_RIGHT_SHIFT 0x0200
- UX_HID_KEYBOARD_STATE_SHIFT 0x0300
- UX_HID_KEYBOARD_STATE_LEFT_ALT 0x0400
- UX_HID_KEYBOARD_STATE_RIGHT_ALT 0x0800
- UX_HID_KEYBOARD_STATE_ALT 0x0a00
- UX_HID_KEYBOARD_STATE_LEFT_CTRL 0x1000
- UX_HID_KEYBOARD_STATE_RIGHT_CTRL 0x2000
- UX_HID_KEYBOARD_STATE_CTRL 0x3000
- UX_HID_KEYBOARD_STATE_LEFT_GUI 0x4000
- UX_HID_KEYBOARD_STATE_RIGHT_GUI 0x8000
- UX_HID_KEYBOARD_STATE_GUI 0xa000

Example

```
while (1)
{

    /* Get a key/state from the keyboard. */
    status = ux_host_class_hid_keyboard_key_get(keyboard,
        &keyboard_char, &keyboard_state);

    /* Check if there is something. */
    if (status == UX_SUCCESS)
    {

        #ifdef UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE
            if (keyboard_state & UX_HID_KEYBOARD_STATE_KEY_UP)
            {
                /* The key was released. */
            } else
            {
                /* The key was pressed. */
            }
        #endif

        /* We have a character in the queue. */
        keyboard_queue[keyboard_queue_index] = (UCHAR) keyboard_char;

        /* Can we accept more ? */
        if(keyboard_queue_index < 1024)
            keyboard_queue_index++;
    }

    tx_thread_sleep(10);

}
```

ux_host_class_hid_keyboard_ioctl

Perform an IOCTL function to the HID keyboard

Prototype

```
UINT ux_host_class_hid_keyboard_ioctl(UX_HOST_CLASS_HID_KEYBOARD *keyboard_instance,
    ULONG ioctl_function, VOID *parameter);
```

Description

This function performs a specific ioctl function to the HID keyboard. The call is blocking and only returns when there is either an error or when the command is completed.

Parameters

- **keyboard_instance** Pointer to the HID keyboard instance.
- **ioctl_function** ioctl function to be performed. See table below for one of the allowed ioctl functions.
- **parameter** Pointer to a parameter specific to the ioctl.

Return Values

- **UX_SUCCESS** (0x00) The ioctl function completed successfully.
- **UX_FUNCTION_NOT_SUPPORTED** (0x54) Unknown IOCTL function

IOCTL functions

- UX_HID_KEYBOARD_IOCTL_SET_LAYOUT
- UX_HID_KEYBOARD_IOCTL_KEY_DECODING_ENABLE
- UX_HID_KEYBOARD_IOCTL_KEY_DECODING_DISABLE

Example – change keyboard layout

```
UINT status;

/* This example shows usage of the SET_LAYOUT IOCTL function.
    USBX receives raw key values from the device (these raw values
    are defined in the HID usage table specification) and optionally
    decodes them for application usage. The decoding is performed
    based on a set of arrays that act as maps – which array is used
    depends on the raw key value (i.e. keypad and non-keypad) and
    the current state of the keyboard (i.e. shift, caps lock, etc.). */

/* When the shift condition is not present and the raw key value
    is not within the keypad value range, this array will be used to decode the raw key value. */

static UCHAR keyboard_layout_raw_to_unshifted_map[] =
{
    0,0,0,0,
    'a','b','c','d','e','f','g',
    'h','i','j','k','l','m','n',
    'o','p','q','r','s','t',
    'u','v','w','x','y','z',
    '1','2','3','4','5','6','7','8','9','0',
    0x0d,0x1b,0x08,0x07,0x20,'-','=','[',']',
    '\\','#',';',0x27,'`',',','.','/',0xf0,
    0xbb,0xbc,0xbd,0xbe,0xbf,0xc0,0xc1,0xc2,0xc3,0xc4,0xc5,0xc6,
    0x00,0xf1,0x00,0xd2,0xc7,0xc9,0xd3,0xcf,0xd1,0xcd,0xcd,0xd0,0xc8,0xf2,
    '/','*','-','+',
    0x0d,'1','2','3','4','5','6','7','8','9','0','.','\\',0x00,0x00,'=',
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
};

/* When the shift condition is present and the raw key value
    is not within the keypad value range, this array will be used to decode the raw key value. */

static UCHAR keyboard_layout_raw_to_shifted_map[] =
{
    0,0,0,0,
    'A','B','C','D','E','F','G',
    'H','I','J','K','L','M','N',
    'O','P','Q','R','S','T',
    'U','V','W','X','Y','Z',
    '!','@','#','$','%','^','&','*','(',')',
    0x0d,0x1b,0x08,0x07,0x20,'_','+','{','}',
    '|','~',':','"','~','<','>','?',0xf0,
    0xbb,0xbc,0xbd,0xbe,0xbf,0xc0,0xc1,0xc2,0xc3,0xc4,0xc5,0xc6,
    0x00,0xf1,0x00,0xd2,0xc7,0xc9,0xd3,0xcf,0xd1,0xcd,0xcd,0xd0,0xc8,0xf2,
    '/','*','-','+',
    0x0d,'1','2','3','4','5','6','7','8','9','0','.','\\',0x00,0x00,'=',
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
};

/* When numlock is on and the raw key value is within the keypad
    value range, this array will be used to decode the raw key value. */

static UCHAR keyboard_layout_raw_to_numlock_on_map[] =
{
    '/','*','-','+',
    0x0d,'1','2','3','4','5','6','7','8','9','0','.','\\',0x00,0x00,'=',
};

/* When numlock is off and the raw key value is within the keypad value
    range, this array will be used to decode the raw key value. */
static UCHAR keyboard_layout_raw_to_numlock_off_map[] =
{
    '/','*','-','+',
    0x0d,0xcf,0xd0,0xd1,0xcb,'5',0xcd,0xc7,0xc8,0xc9,0xd2,0xd3,'\\',0x00,0x
```

```
    00,'=',
};

/* Specify the keyboard layout for USBX usage. */
static UX_HOST_CLASS_HID_KEYBOARD_LAYOUT keyboard_layout =
{
    keyboard_layout_raw_to_shifted_map,
    keyboard_layout_raw_to_unshifted_map,
    keyboard_layout_raw_to_numlock_on_map,
    keyboard_layout_raw_to_numlock_off_map,
    /* The maximum raw key value. Values larger than this are discarded. */
    UX_HID_KEYBOARD_KEYS_UPPER_RANGE,
    /* The raw key value for the letter 'a'. */
    UX_HID_KEYBOARD_KEY_LETTER_A,
    /* The raw key value for the letter 'z'. */
    UX_HID_KEYBOARD_KEY_LETTER_Z,
    /* The lower range raw key value for keypad keys - inclusive. */
    UX_HID_KEYBOARD_KEYS_KEYPAD_LOWER_RANGE,
    /* The upper range raw key value for keypad keys. */
    UX_HID_KEYBOARD_KEYS_KEYPAD_UPPER_RANGE
};

/* Call the IOCTL function to change the keyboard layout. */
status = ux_host_class_hid_keyboard_ioctl(keyboard,
    UX_HID_KEYBOARD_IOCTL_SET_LAYOUT, (VOID *)&keyboard_layout);

/* If status equals UX_SUCCESS, the operation was successful. */
```

Example – disable keyboard key decode

```
UINT status;

/* The following example illustrates IOCTL function of Disable key decode from keyboard layout. */
status = ux_host_class_hid_keyboard_ioctl(keyboard,
    UX_HID_KEYBOARD_IOCTL_DISABLE_KEYS_DECODE, UX_NULL);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_hid_remote_control_usage_get

Get remote control usage

Prototype

```
UINT ux_host_class_hid_remote_control_usage_get (UX_HOST_CLASS_HID_REMOTE_CONTROL *remote_control_instance,
    LONG *usage, ULONG *value);
```

Description

This function is used to get the remote control usages.

Parameters

- **remote_control_instance** Pointer to the HID remote control instance.
- **usage** Pointer to the usage.
- **value** Pointer to the value for the usage.

Return Values

- **UX_SUCCESS** (0x00) The data transfer was completed.
- **UX_ERROR** (0xff) Nothing to report.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) HID class instance does not exist.

The list of all possible usages is too long to fit in this user guide. For a full description, the ux_host_class_hid.h has the entire set of possible values.

Example

```
/* Read usages and values as the user changes the vol/bass/treble buttons on the speaker */

while (remote_control != UX_NULL)
{
    status = ux_host_class_hid_remote_control_usage_get(remote_control, &usage, &value);
    if (status == UX_SUCCESS)
    {
        /* We have something coming from the HID remote control,
        we filter the usage here and only allow the
        volume usage which can be VOLUME, VOLUME_INCREMENT or VOLUME_DECREMENT */
        switch(usage)
        {
            case UX_HOST_CLASS_HID_CONSUMER_VOLUME :
            case UX_HOST_CLASS_HID_CONSUMER_VOLUME_INCREMENT :
            case UX_HOST_CLASS_HID_CONSUMER_VOLUME_DECREMENT :

            if (value<0x80)
            {
                if (current_volume + audio_control.ux_host_class_audio_control_res < 0xffff)
                    current_volume = current_volume + audio_control.ux_host_class_audio_control_res;
                } else {
                if (current_volume > audio_control.ux_host_class_audio_control_res)
                    current_volume = current_volumeaudio_control.ux_host_class_audio_control_res;
            }

            audio_control.ux_host_class_audio_control_channel = 1;
            audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;
            audio_control.ux_host_class_audio_control_cur = current_volume;
            status = ux_host_class_audio_control_value_set(audio, &audio_control);
            audio_control.ux_host_class_audio_control_channel = 2;
            audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;
            audio_control.ux_host_class_audio_control_cur = current_volume;
            status = ux_host_class_audio_control_value_set(audio, &audio_control);
            break;

        }
    }
    tx_thread_sleep(10);

}
```

ux_host_class_cdc_acm_read

Read from the cdc_acm interface

Prototype

```
UINT ux_host_class_cdc_acm_read(UX_HOST_CLASS_CDC_ACM *cdc_acm,
    UCHAR *data_pointer, ULONG requested_length, ULONG *actual_length);
```

Description

This function reads from the cdc_acm interface. The call is blocking and only returns when there is either an error or when the transfer is complete.

Parameters

- **cdc_acm** Pointer to the cdc_acm class instance.
- **data_pointer** Pointer to the buffer address of the data payload.

- **requested_length** Length to be received.
- **actual_length** Length actually received.

Return Values

- **UX_SUCCESS** (0x00) The data transfer was completed.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) The cdc_acm instance is invalid.
- **UX_TRANSFER_TIMEOUT** (0x5c) Transfer timeout, reading incomplete.

Example

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_cdc_acm_read(cdc_acm, data_pointer,
    requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_cdc_acm_write

Write to the cdc_acm interface

Prototype

```
UINT ux_host_class_cdc_acm_write(UX_HOST_CLASS_CDC_ACM *cdc_acm,
    UCHAR *data_pointer, ULONG requested_length, ULONG *actual_length);
```

Description

This function writes to the cdc_acm interface. The call is blocking and only returns when there is either an error or when the transfer is complete.

Parameters

- **cdc_acm** Pointer to the cdc_acm class instance.
- **data_pointer** Pointer to the buffer address of the data payload.
- **requested_length** Length to be sent.
- **actual_length** Length actually sent.

Return Values

- **UX_SUCCESS** (0x00) The data transfer was completed.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) The cdc_acm instance is invalid.
- **UX_TRANSFER_TIMEOUT** (0x5c) Transfer timeout, writing incomplete.

Example

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_cdc_acm_write(cdc_acm, data_pointer,
    requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_cdc_acm_ioctl

Perform an IOCTL function to the cdc_acm interface

Prototype

```
UINT ux_host_class_cdc_acm_ioctl(UX_HOST_CLASS_CDC_ACM *cdc_acm,
    ULONG ioctl_function, VOID *parameter);
```

Description

This function performs a specific ioctl function to the cdc_acm interface. The call is blocking and only returns when there is either an error or when the command is completed.

Parameters

- **cdc_acm** Pointer to the cdc_acm class instance.
- **ioctl_function** ioctl function to be performed. See table below for one of the allowed ioctl functions.
- **parameter** Pointer to a parameter specific to the ioctl

Return Value

- **UX_SUCCESS** (0x00) The data transfer was completed.
- **UX_MEMORY_INSUFFICIENT** (0x12) Not enough memory.
- **UX_HOST_CLASS_INSTANCE_UNKNOWN** (0x5b) CDC-ACM instance is in an invalid state.
- **UX_FUNCTION_NOT_SUPPORTED** (0x54) Unknown IOCTL function.

IOCTL functions:

- UX_HOST_CLASS_CDC_ACM_IOCTL_SET_LINE_CODING
- UX_HOST_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING
- UX_HOST_CLASS_CDC_ACM_IOCTL_SET_LINE_STATE
- UX_HOST_CLASS_CDC_ACM_IOCTL_SEND_BREAK
- UX_HOST_CLASS_CDC_ACM_IOCTL_ABORT_IN_PIPE
- UX_HOST_CLASS_CDC_ACM_IOCTL_ABORT_OUT_PIPE
- UX_HOST_CLASS_CDC_ACM_IOCTL_NOTIFICATION_CALLBACK
- UX_HOST_CLASS_CDC_ACM_IOCTL_GET_DEVICE_STATUS

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_cdc_acm_ioctl(cdc_acm,
    UX_HOST_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING, (VOID *)&line_coding);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_cdc_acm_reception_start

Begins background reception of data from the device.

Prototype

```
UINT ux_host_class_cdc_acm_reception_start(UX_HOST_CLASS_CDC_ACM *cdc_acm,
    UX_HOST_CLASS_CDC_ACM_RECEPTION *cdc_acm_reception);
```

Description

This function causes USBX to continuously read data from the device in the background. Upon completion of each transaction, the callback specified in **cdc_acm_reception** is invoked so the application may perform further processing of the transaction's data.

> **NOTE**
>
> **ux_host_class_cdc_acm_read** must not be used while background reception is in use.

Parameters

- **cdc_acm** Pointer to the cdc_acm class instance.
- **cdc_acm_reception** Pointer to parameter that contains values defining behavior of background reception. The layout of this parameter follows:

```
typedef struct UX_HOST_CLASS_CDC_ACM_RECEPTION_STRUCT
{
    ULONG ux_host_class_cdc_acm_reception_state;
    ULONG ux_host_class_cdc_acm_reception_block_size;
    UCHAR *ux_host_class_cdc_acm_reception_data_buffer;
    ULONG ux_host_class_cdc_acm_reception_data_buffer_size;
    UCHAR *ux_host_class_cdc_acm_reception_data_head;
    UCHAR *ux_host_class_cdc_acm_reception_data_tail;
    VOID (*ux_host_class_cdc_acm_reception_callback)(struct UX_HOST_CLASS_CDC_ACM_STRUCT *cdc_acm,
        UINT status, UCHAR *reception_buffer, ULONG reception_size);
} UX_HOST_CLASS_CDC_ACM_RECEPTION;
```

Return Value

- **UX_SUCCESS** (0x00) Background reception successfully started.
- **UX_HOST_CLASS_INSTANCE _UNKNOWN** (0x5b) Wrong class instance.

```
UINT status;
UX_HOST_CLASS_CDC_ACM_RECEPTION cdc_acm_reception;

/* Setup the background reception parameter. */

/* Set the desired max read size for each transaction.
    For example, if this value is 64, then the maximum amount of
    data received from the device in a single transaction is 64.
    If the amount of data received from the device is less than this value,
    the callback will still be invoked with the actual amount of data received. */
cdc_acm_reception.ux_host_class_cdc_acm_reception_block_size = block_size;

/* Set the buffer where the data from the device is read to. */
cdc_acm_reception.ux_host_class_cdc_acm_reception_data_buffer = cdc_acm_reception_buffer;

/* Set the size of the data reception buffer.
    Note that this should be at least as large as ux_host_class_cdc_acm_reception_block_size. */
cdc_acm_reception.ux_host_class_cdc_acm_reception_data_buffer_size = cdc_acm_reception_buffer_size;

/* Set the callback that is to be invoked upon each reception transfer completion. */
cdc_acm_reception.ux_host_class_cdc_acm_reception_callback = reception_callback;

/* Start background reception using the values we defined in the reception parameter. */
status = ux_host_class_cdc_acm_reception_start(cdc_acm_host_data, &cdc_acm_reception);

/* If status equals UX_SUCCESS, background reception has successfully started. */
```

ux_host_class_cdc_acm_reception_stop

Stops background reception of packets.

Prototype

```
UINT ux_host_class_cdc_acm_reception_stop(UX_HOST_CLASS_CDC_ACM *cdc_acm,
    UX_HOST_CLASS_CDC_ACM_RECEPTION *cdc_acm_reception);
```

Description

This function causes USBX to stop background reception previously started by **ux_host_class_cdc_acm_reception_start**.

Parameters

- **cdc_acm** Pointer to the cdc_acm class instance.
- **cdc_acm_reception** Pointer to the same parameter that was used to start background reception. The layout of this parameter follows:

```
typedef struct UX_HOST_CLASS_CDC_ACM_RECEPTION_STRUCT
{
    ULONG ux_host_class_cdc_acm_reception_state;
    ULONG ux_host_class_cdc_acm_reception_block_size;
    UCHAR *ux_host_class_cdc_acm_reception_data_buffer;
    ULONG ux_host_class_cdc_acm_reception_data_buffer_size;
    UCHAR *ux_host_class_cdc_acm_reception_data_head;
    UCHAR *ux_host_class_cdc_acm_reception_data_tail;
    VOID (*ux_host_class_cdc_acm_reception_callback)(
            struct UX_HOST_CLASS_CDC_ACM_STRUCT *cdc_acm, UINT status,
            UCHAR *reception_buffer, ULONG reception_size);
} UX_HOST_CLASS_CDC_ACM_RECEPTION;
```

Return Value

- **UX_SUCCESS** (0x00) Background reception successfully stopped.
- **UX_HOST_CLASS_INSTANCE _UNKNOWN** (0x5b) Wrong class instance.

```
UINT status;
UX_HOST_CLASS_CDC_ACM_RECEPTION cdc_acm_reception;

/* Stop background reception. The reception parameter should be the same
    that was passed to ux_host_class_cdc_acm_reception_start. */
status = ux_host_class_cdc_acm_reception_stop(cdc_acm, &cdc_acm_reception);

/* If status equals UX_SUCCESS, background reception has successfully stopped. */
```

Chapter 6 - USBX CDC-ECM Class Usage

6/24/2020 • 2 minutes to read

USBX contains a CDC-ECM class for the host and device side. This class is designed to be used with NetX, specifically, the USBX CDC-ECM class acts as the driver for NetX. This is why there are no CDC-ECM APIs listed in Chapter 5.

Once NetX and USBX are initialized and an instance of a CDC-ECM device is found by USBX, the application exclusively uses NetX to communicate with the device. Initialization follows:

```
UINT status;

/* The USB controller should be the last component initialized so that
everything is ready when data starts being received. */

/* Initialize USBX. */

ux_system_initialize(memory_pointer, UX_USBX_MEMORY_SIZE, UX_NULL, 0);

/* The code below is required for installing the host portion of USBX */
status = ux_host_stack_initialize(UX_NULL);

/* Register cdc_ecm class. */

status = ux_host_stack_class_register(_ux_system_host_class_cdc_ecm_name,
    ux_host_class_cdc_ecm_entry);

/* Perform the initialization of the network driver. */

_ux_network_driver_init();

/* Initialize NetX. Refer to NetX user guide for details to add initialization code. */

/* Register the platform-specific USB controller. */

status = ux_host_stack_hcd_register("controller_name", controller_entry, param1, param2);

/* Find the CDC-ECM class. */
class_cdc_ecm_get();

/* Now wait for the link to be up. */

while (cdc_ecm -> ux_host_class_cdc_ecm_link_state != UX_HOST_CLASS_CDC_ECM_LINK_STATE_UP)
    tx_thread_sleep(10);

/* At this point, everything has been initialized, and we've found a CDC-ECM device.
    Now NetX can be used to communicate with the device. */
```

Chapter 1 - Introduction to the USBX Host Stack User Guide Supplement

5/18/2020 • 2 minutes to read

This document is a supplement to the USBX Host Stack User Guide. It contains documentation for the uncertified USBX Host classes that are not included in the main user guide.

Organization

- **Chapter 1** contains an introduction to USBX
- **Chapter 2** USBX Host Classes API
- **Chapter 3** USBX DPUMP Class Considerations
- **Chapter 4** USBX Pictbridge implementation
- **Chapter 5** USBX OTG

Chapter 2: USBX Host Classes API

5/18/2020 • 21 minutes to read

This chapter covers all the exposed APIs of the USBX host classes. The following APIs for each class are described in detail:

- Printer class
- Audio class
- Asix class
- Pima/PTP class
- Prolific class
- Generic Serial class

ux_host_class_printer_read

Read from the printer interface

Prototype

```
UINT ux_host_class_printer_read(UX_HOST_CLASS_PRINTER *printer,
    UCHAR *data_pointer,
    ULONG requested_length,
    ULONG *actual_length)
```

Description

This function reads from the printer interface. The call is blocking and only returns when there is either an error or when the transfer is complete. A read is allowed only on bi-directional printers.

Parameters

- **printer**: Pointer to the printer class instance.
- **data_pointer**: Pointer to the buffer address of the data payload.
- **requested_length**: Length to be received.
- **actual_length**: Length actually received.

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_FUNCTION_NOT_SUPPORTED**: (0x54) Function not supported because the printer is not bi-directional.
- **UX_TRANSFER_TIMEOUT**: (0x5c) Transfer timeout, reading incomplete.

Example

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_printer_read(printer, data_pointer, requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_printer_write

Write to the printer interface

Prototype

```
UINT ux_host_class_printer_write(UX_HOST_CLASS_PRINTER *printer,
    UCHAR *data_pointer, ULONG requested_length,
    ULONG *actual_length)
```

Description

This function writes to the printer interface. The call is blocking and only returns when there is either an error or when the transfer is complete.

Parameters

- **printer**: Pointer to the printer class instance.
- **data_pointer**: Pointer to the buffer address of the data payload.
- **requested_length**: Length to be sent.
- **actual_length**: Length actually sent.

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_TRANSFER_TIMEOUT**: (0x5c) Transfer timeout, writing incomplete.

Example

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_printer_write(printer, data_pointer, requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_printer_soft_reset

Perform a soft reset to the printer

Prototype

```
UINT ux_host_class_printer_soft_reset(UX_HOST_CLASS_PRINTER *printer)
```

Description

This function performs a soft reset to the printer.

Input Parameter

- **printer**: Pointer to the printer class instance.

Return Value

- **UX_SUCCESS**: (0x00)The reset was completed.
- **UX_TRANSFER_TIMEOUT**: (0x5c) Transfer timeout, reset not completed.

Example

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_printer_soft_reset(printer);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_printer_status_get

Get the printer status

Prototype

```
UINT ux_host_class_printer_status_get(UX_HOST_CLASS_PRINTER *printer,
    ULONG *printer_status)
```

Description

This function obtains the printer status. The printer status is similar to the LPT status (1284 standard).

Parameters

- **printer**: Pointer to the printer class instance.
- **printer_status**: Address of the status to be returned.

Return Value

- **UX_SUCCESS** (0x00): The reset was completed.
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory to perform the operation.
- **UX_TRANSFER_TIMEOUT**: (0x5c) Transfer timeout, reset not completed

Example

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_printer_status_get(printer, printer_status);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_read

Read from the audio interface

Prototype

```
UINT ux_host_class_audio_read(UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_TRANSFER_REQUEST
    *audio_transfer_request)
```

Description

This function reads from the audio interface. The call is non-blocking. The application must ensure that the appropriate alternate setting has been selected for the audio streaming interface.

Parameters

- **audio**: Pointer to the audio class instance.
- **audio_transfer_request**: Pointer to the audio transfer structure.

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed
- **UX_FUNCTION_NOT_SUPPORTED**" (0x54) Function not supported

Example

```
/* The following example reads from the audio interface. */

audio_transfer_request.ux_host_class_audio_transfer_request_completion_function =
tx_audio_transfer_completion_function;
audio_transfer_request.ux_host_class_audio_transfer_request_class_instance = audio;
audio_transfer_request.ux_host_class_audio_transfer_request_next_audio_audio_transfer_request = UX_NULL;
audio_transfer_request.ux_host_class_audio_transfer_request_data_pointer = audio_buffer;
audio_transfer_request.ux_host_class_audio_transfer_request_requested_length = requested_length;
audio_transfer_request.ux_host_class_audio_transfer_request_packet_length = AUDIO_FRAME_LENGTH;

status = ux_host_class_audio_read(audio, audio_transfer_request);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_write

Write to the audio interface

Prototype

```
UINT ux_host_class_audio_write(UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_TRANSFER_REQUEST *audio_transfer_request)
```

Description

This function writes to the audio interface. The call is non-blocking. The application must ensure that the appropriate alternate setting has been selected for the audio streaming interface.

Parameters

- **audio**: Pointer to the audio class instance
- **audio_transfer_request**: Pointer to the audio transfer structure

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_FUNCTION_NOT_SUPPORTED**: (0x54) Function not supported.
- **ux_host_CLASS_AUDIO_WRONG_INTERFACE**: (0x81) Interface incorrect.

Example

```
UINT status;

/* The following example writes to the audio interface */

audio_transfer_request.ux_host_class_audio_transfer_request_completion_function =
tx_audio_transfer_completion_function;
audio_transfer_request.ux_host_class_audio_transfer_request_class_instance = audio;
audio_transfer_request.ux_host_class_audio_transfer_request_next_audio_audio_transfer_request = UX_NULL;
audio_transfer_request.ux_host_class_audio_transfer_request_data_pointer = audio_buffer;
audio_transfer_request.ux_host_class_audio_transfer_request_requested_length = requested_length;
audio_transfer_request.ux_host_class_audio_transfer_request_packet_length = AUDIO_FRAME_LENGTH;
status = ux_host_class_audio_write(audio, audio_transfer_request);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_control_get

Get a specific control from the audio control interface

Prototype

```
UINT ux_host_class_audio_control_get(UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_CONTROL
    *audio_control)
```

Description

This function reads a specific control from the audio control interface.

Parameters

- **audio**: Pointer to the audio class instance
- **audio_control**: Pointer to the audio control structure

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed
- **UX_FUNCTION_NOT_SUPPORTED**: (0x54) Function not supported
- **UX_HOST_CLASS_AUDIO_WRONG_INTERFACE**: (0x81) Interface incorrect

Example

```
UINT status;

/* The following example reads the volume control from a stereo USB speaker. */

UX_HOST_CLASS_AUDIO_CONTROL audio_control;

audio_control.ux_host_class_audio_control_channel = 1;
audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;

status = ux_host_class_audio_control_get(audio, &audio_control);

/* If status equals UX_SUCCESS, the operation was successful. */

audio_control.ux_host_class_audio_control_channel = 2;
audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;

status = ux_host_class_audio_control_get(audio, &audio_control);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_control_value_set

Set a specific control to the audio control interface

Prototype

```
UINT ux_host_class_audio_control_value_set(UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_CONTROL *audio_control)
```

**Description **

This function sets a specific control to the audio control interface.

Parameters

- **audio**: Pointer to the audio class instance
- **audio_control**: Pointer to the audio control structure

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed
- **UX_FUNCTION_NOT_SUPPORTED**: (0x54) Function not supported
- **UX_HOST_CLASS_AUDIO_WRONG_INTERFACE**: (0x81) Interface incorrect

Example

```
/* The following example sets the volume control of a stereo USB speaker. */

UX_HOST_CLASS_AUDIO_CONTROL audio_control;

UINT status;

audio_control.ux_host_class_audio_control_channel = 1;
audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;
audio_control.ux_host_class_audio_control_cur = 0xf000;

status = ux_host_class_audio_control_value_set(audio, &audio_control);
/* If status equals UX_SUCCESS, the operation was successful. */

current_volume = audio_control.audio_control_cur;
audio_control.ux_host_class_audio_control_channel = 2;
audio_control.ux_host_class_audio_control = UX_HOST_CLASS_AUDIO_VOLUME_CONTROL;
audio_control.ux_host_class_audio_control_cur = 0xf000;

status = ux_host_class_audio_control_value_set(audio, &audio_control);
/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_streaming_sampling_set

Set an alternate setting interface of the audio streaming interface

Prototype

```
UINT ux_host_class_audio_streaming_sampling_set(UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_SAMPLING *audio_sampling)
```

Description

This function sets the appropriate alternate setting interface of the audio streaming interface according to a specific sampling structure.

Parameters

- **audio**: Pointer to the audio class instance.
- **audio_sampling**: Pointer to the audio sampling structure.

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed
- **UX_FUNCTION_NOT_SUPPORTED**: (0x54) Function not supported
- **UX_HOST_CLASS_AUDIO_WRONG_INTERFACE**: (0x81) Interface incorrect
- **UX_NO_ALTERNATE_SETTING**: (0x5e) No alternate setting for the sampling values

Example

```
/* The following example sets the alternate setting interface of a stereo USB speaker. */

UX_HOST_CLASS_AUDIO_SAMPLING audio_sampling;

UINT status;

sampling.ux_host_class_audio_sampling_channels = 2;
sampling.ux_host_class_audio_sampling_frequency = AUDIO_FREQUENCY;
sampling. ux_host_class_audio_sampling_resolution = 16;

status = ux_host_class_audio_streaming_sampling_set(audio, &sampling);
/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_audio_streaming_sampling_get

Get possible sampling settings of audio streaming interface

Prototype

```
UINT ux_host_class_audio_streaming_sampling_get(UX_HOST_CLASS_AUDIO *audio,
    UX_HOST_CLASS_AUDIO_SAMPLING_CHARACTERISTICS *audio_sampling)
```

Description

This function gets, one by one, all the possible sampling settings available in each of the alternate settings of the audio streaming interface. The first time the function is used, all the fields in the calling structure pointer must be reset. The function will return a specific set of streaming values upon return unless the end of the alternate settings has been reached. When this function is reused, the previous sampling values will be used to find the next sampling values.

Parameters

- **audio**: Pointer to the audio class instance
- **audio_sampling**: Pointer to the audio sampling structure

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed
- **UX_FUNCTION_NOT_SUPPORTED**: (0x54) Function not supported
- **UX_HOST_CLASS_AUDIO_WRONG_INTERFACE**: (0x81) Interface incorrect
- **UX_NO_ALTERNATE_SETTING**: (0x5e) No alternate setting for the sampling values

Example

```
/* The following example gets the sampling values for the first alternate setting interface of a stereo USB
speaker. */

UX_HOST_CLASS_AUDIO_SAMPLING_CHARACTERISTICS audio_sampling;

UINT status;

sampling.ux_host_class_audio_sampling_channels=0;
sampling.ux_host_class_audio_sampling_frequency_low=0;
sampling.ux_host_class_audio_sampling_frequency_high=0;
sampling.ux_host_class_audio_sampling_resolution=0;

status = ux_host_class_audio_streaming_sampling_get(audio, &sampling);

/* If status equals UX_SUCCESS, the operation was successful and information could be displayed as follows:

printf("Number of channels %d, Resolution %d bits, frequency range %d-%d\n",
    sampling.audio_channels, sampling.audio_resolution,
    sampling.audio_frequency_low, sampling.audio_frequency_high);

*/
```

ux_host_class_asix_read

Read from the asix interface

Prototype

```
UINT ux_host_class_asix_read(UX_HOST_CLASS_ASIX *asix, UCHAR *data_pointer,
    ULONG requested_length, ULONG *actual_length)
```

Description

This function reads from the asix interface. The call is blocking and only returns when there is either an error or when the transfer is complete.

Parameters

- **asix**: Pointer to the asix class instance.
- **data_pointer**: Pointer to the buffer address of the data payload.
- **requested_length**: Length to be received.
- **actual_length**: Length actually received.

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_TRANSFER_TIMEOUT**: (0x5c) Transfer timeout, reading incomplete.

Example

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_asix_read(asix, data_pointer, requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_asix_write

Write to the asix interface

Prototype

```
UINT ux_host_class_asix_write(VOID *asix_class, NX_PACKET *packet)
```

Description

This function writes to the asix interface. The call is non blocking.

Parameters

- **asix**: Pointer to the asix class instance.
- **packet**: Netx data packet

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_ERROR**: (0xFF) Transfer could not be requested.

Example

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_asix_write(asix, packet);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_pima_session_open

Open a session between Initiator and Responder

Prototype

```
UINT ux_host_class_pima_session_open(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session)
```

Description

This function opens a session between a PIMA Initiator and a PIMA Responder. Once a session is successfully opened, most PIMA commands can be executed.

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_sessio**: Pointer to PIMA session<

Return Value

- **UX_SUCCESS**: (0x00) Session successfully opened
- **UX_HOST_CLASS_PIMA_RC_SESSION_ALREADY_OPENED**: (0x201E) Session already opened

Example

```
/* Open a pima session. */

status = ux_host_class_pima_session_open(pima, pima_session);

if (status != UX_SUCCESS)
    return(UX_PICTBRIDGE_ERROR_SESSION_NOT_OPEN);
```

ux_host_class_pima_session_close

Close a session between Initiator and Responder

Prototype

```
UINT ux_host_class_pima_session_close(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session)
```

Description

This function closes a session that was previously opened between a PIMA Initiator and a PIMA Responder. Once a session is closed, most PIMA commands can no longer be executed.

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session

Return Value

- **UX_SUCCESS**: (0x00) The session was closed
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened

Example

```
/* Close the pima session. */

status = ux_host_class_pima_session_close(pima, pima_session);
```

ux_host_class_pima_storage_ids_get

Obtain the storage ID array from Responder

Prototype

```
UINT ux_host_class_pima_storage_ids_get(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG *storage_ids_array,
    ULONG storage_id_length)
```

Description

This function obtains the storage ID array from the responder.

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session
- **storage_ids_array**: Array where storage IDs will be returned
- **storage_id_length**: Length of the storage array

Return Value

- **UX_SUCCESS**: (0x00) The storage ID array has been populated
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory to create PIMA command.

Example

```
/* Get the number of storage IDs. */
status = ux_host_class_pima_storage_ids_get(pima, pima_session,
    pictbridge ->ux_pictbridge_storage_ids, 64);

if (status != UX_SUCCESS)
{
    /* Close the pima session. */
    status = ux_host_class_pima_session_close(pima, pima_session);

    return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
}
```

ux_host_class_pima_storage_info_get

Obtain the storage information from Responder

Prototype

```
UINT ux_host_class_pima_storage_info_get(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG storage_id,
    UX_HOST_CLASS_PIMA_STORAGE *storage)
```

Description

This function obtains the storage information for a storage container of value storage_id

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session
- **storage_id**: ID of the storage container
- **storage**: Pointer to storage information container

Return Value

- **UX_SUCCESS**: (0x00) The storage information was retrieved
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened
- **UX_MEMORY_INSUFFICIENT** (0x12) Not enough memory to create PIMA command.

Example

```
/* Get the first storage ID info container. */
status = ux_host_class_pima_storage_info_get(pima, pima_session,
    pictbridge ->ux_pictbridge_storage_ids[0],
    (UX_HOST_CLASS_PIMA_STORAGE *)pictbridge ->ux_pictbridge_storage);

if (status != UX_SUCCESS)
{
    /* Close the pima session. */
    status = ux_host_class_pima_session_close(pictbridge ->
        ux_pictbridge_pima, pima_session);
    return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
}
```

ux_host_class_pima_num_objects_get

Obtain the number of objects on a storage container from Responder

Prototype

```
UINT ux_host_class_pima_num_objects_get(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG storage_id, ULONG object_format_code)
```

Description

This function obtains the number of objects stored on a specific storage container of value storage_id matching a specific format code. The number of objects is returned in the field: ux_host_class_pima_session_nb_objects of the pima_session structure.

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session
- **storage_id**: ID of the storage container
- **object_format_code**: Objects format code filter.

The Object Format Codes can have one of the following values:

OBJECT FORMAT CODE	DESCRIPTION	USBX CODE
0x3000	Undefined Undefined non-image object	UX_HOST_CLASS_PIMA_OFC_UNDEFINED
0x3001	Association Association (e.g. folder)	UX_HOST_CLASS_PIMA_OFC_ASSOCIATION
0x3002	Script Device-model specific script	UX_HOST_CLASS_PIMA_OFC_SCRIPT
0x3003	Executable Device model-specific binary executable	UX_HOST_CLASS_PIMA_OFC_EXECUTABLE
0x3004	Text Text file	UX_HOST_CLASS_PIMA_OFC_TEXT
0x3005	HTML HyperText Markup Language file (text)	UX_HOST_CLASS_PIMA_OFC_HTML

OBJECT FORMAT CODE	DESCRIPTION	USBX CODE
0x3006	DPOF Digital Print Order Format file (text)	UX_HOST_CLASS_PIMA_OFC_DPOF
0x3007	AIFF Audio clip	UX_HOST_CLASS_PIMA_OFC_AIFF
0x3008	WAV Audio clip	UX_HOST_CLASS_PIMA_OFC_WAV
0x3009	MP3 Audio clip	UX_HOST_CLASS_PIMA_OFC_MP3
0x300A	AVI Video clip	UX_HOST_CLASS_PIMA_OFC_AVI
0x300B	MPEG Video clip	UX_HOST_CLASS_PIMA_OFC_MPEG
0x300C	ASF Microsoft Advanced Streaming Format (video)	UX_HOST_CLASS_PIMA_OFC_ASF
0x3800	Undefined Unknown image object	UX_HOST_CLASS_PIMA_OFC_QT
0x3801	EXIF/JPEG Exchangeable File Format, JEIDA standard	UX_HOST_CLASS_PIMA_OFC_EXIF_JPEG
0x3802	TIFF/EP Tag Image File Format for Electronic Photography	UX_HOST_CLASS_PIMA_OFC_TIFF_EP
0x3803	FlashPix Structured Storage Image Format	UX_HOST_CLASS_PIMA_OFC_FLASHPIX
0x3804	BMP Microsoft Windows Bitmap file	UX_HOST_CLASS_PIMA_OFC_BMP
0x3805	CIFF Canon Camera Image File Format	UX_HOST_CLASS_PIMA_OFC_CIFF
0x3806	Undefined Reserved	
0x3807	GIF Graphics Interchange Format	UX_HOST_CLASS_PIMA_OFC_GIF
0x3808	JFIF JPEG File Interchange Format	UX_HOST_CLASS_PIMA_OFC_JFIF
0x3809	PCD PhotoCD Image Pac	UX_HOST_CLASS_PIMA_OFC_PCD
0x380A	PICT Quickdraw Image Format	UX_HOST_CLASS_PIMA_OFC_PICT
0x380B	PNG Portable Network Graphics	UX_HOST_CLASS_PIMA_OFC_PNG
0x380C	Undefined Reserved	
0x380D	TIFF Tag Image File Format	UX_HOST_CLASS_PIMA_OFC_TIFF
0x380E	TIFF/IT Tag Image File Format for Information Technology (graphic arts)	UX_HOST_CLASS_PIMA_OFC_TIFF_IT
0x380F	JP2 JPEG2000 Baseline File Format	UX_HOST_CLASS_PIMA_OFC_JP2

OBJECT FORMAT CODE	DESCRIPTION	USBX CODE
0x3810	JPX JPEG2000 Extended File Format	UX_HOST_CLASS_PIMA_OFC_JPX
All other codes with MSN of 0011	Any Undefined Reserved for future use	
All other codes with MSN of 1011	Any Vendor-Defined Vendor-Defined type: Image	

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory to create PIMA command.

Example

```
/* Get the number of objects on all containers matching a SCRIPT object. */
status = ux_host_class_pima_num_objects_get(pima, pima_session,
    UX_PICTBRIDGE_ALL_CONTAINERS, UX_PICTBRIDGE_OBJECT_SCRIPT);

if (status != UX_SUCCESS)
{
    /* Close the pima session. */
    status = ux_**host_class_pima_session_close(pima, pima_session);

    return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
} else
    /* The number of objects is returned in the field: pima_session -> ux_host_class_pima_session_nb_objects
*/
```

ux_host_class_pima_object_handles_get

Obtain object handles from Responder

Prototype

```
UINT ux_host_class_pima_object_handles_get(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG *object_handles_array,
    ULONG object_handles_length,
    ULONG storage_id,
    ULONG object_format_code,
    ULONG object_handle_association)
```

Description

Returns an array of Object Handles present in the storage container indicated by the storage_id parameter. If an aggregated list across all stores is desired, this value shall be set to 0xFFFFFFFF.

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session
- **object_handes_array**: Array where handles are returned
- **object_handles_length**: Length of the array
- **storage_id**: ID of the storage container
- **object_format_code**: Format code for object (see table for function ux_host_class_pima_num_objects_get)

- **object_handle_association**: Optional object association value

The object handle association can be one of the value from the table below:

ASSOCIATIONCODE	ASSOCIATIONTYPE	INTERPRETATION
0x0000	Undefined	Undefined
0x0001	GenericFolder	Unused
0x0002	Album	Reserved
0x0003	TimeSequence	DefaultPlaybackDelta
0x0004	HorizontalPanoramic	Unused
0x0005	VerticalPanoramic	Unused
0x0006	2DPanoramic	ImagesPerRow
0x0007	AncillaryData	Undefined
All other values with bit 15 set to 0	Reserved	Undefined
All values with bit 15 set to 1	Vendor-Defined	Vendor-Defined

Return Value

- UX_SUCCESS: (0x00) The data transfer was completed.
- UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN: (0x2003) Session not opened
- UX_MEMORY_INSUFFICIENT: (0x12) Not enough memory to create PIMA command.

Example

```
/* Get the array of objects handles on the container. */
status = ux_**host_class_pima_object_handles_get(pima, pima_session,
    pictbridge ->ux_pictbridge_object_handles_array,
    4 * pima_session ->ux_host_class_pima_session_nb_objects,
    UX_PICTBRIDGE_ALL_CONTAINERS,
    UX_PICTBRIDGE_OBJECT_SCRIPT, 0);

if (status != UX_SUCCESS)
{
    /* Close the pima session. */
    status = ux_host_class_pima_session_close(pima, pima_session);
    return(UX_PICTBRIDGE_ERROR_STORE_NOT_AVAILABLE);
}
```

ux_host_class_pima_object_info_get

Obtain the object information from Responder

Prototype

```
UINT ux_host_class_pima_object_info_get(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle,
    UX_HOST_CLASS_PIMA_OBJECT *object)
```

Description

This function obtains the object information for an object handle.

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session
- **object_handle**: Handle of the object
- **object**: Pointer to object information container

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory to create PIMA command.

Example

```
/* We search for an object that is a picture or a script. */
object_index = 0;

while (object_index < pima_session -> ux_host_class_pima_session_nb_objects)
{
    /* Get the object info structure. */
    status = ux_**host_class_pima_object_info_get(pima, pima_session,
        pictbridge -> ux_pictbridge_object_handles_array[object_index],
        pima_object);

    if (status != UX_SUCCESS)
    {
        /* Close the pima session. */
        status = ux_host_class_pima_session_close(pima, pima_session);

        return(UX_PICTBRIDGE_ERROR_INVALID_OBJECT_HANDLE );
    }
}
```

ux_host_class_pima_object_info_send

Send the object information to Responder

Prototype

```
UINT ux_host_class_pima_object_info_send(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG storage_id,
    ULONG parent_object_id,
    UX_HOST_CLASS_PIMA_OBJECT *object)
```

Description

This function sends the storage information for a storage container of value storage_id. The Initiator should use this command before sending an object to the responder.

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session
- **storage_id**: Destination storage ID
- **parent_object_id**: Parent ObjectHandle on Responder where object should be placed
- **object**: Pointer to object information container

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory to create PIMA command.

Example

```
/* Send a script info. */
status = ux_host_class_pima_object_info_send(pima, pima_session,
    0, 0, pima_object);

if (status != UX_SUCCESS)
{
    /* Close the pima session. */
    status = ux_host_class_pima_session_close(pima, pima_session);

    return(UX_ERROR);
}
```

ux_host_class_pima_object_open

Open an object stored in the Responder

Prototype

```
UINT ux_host_class_pima_object_open(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle,
    UX_HOST_CLASS_PIMA_OBJECT *object)
```

Description

This function opens an object on the responder before reading or writing.

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session
- **object_handle**: handle of the object
- **objec**: Pointer to object information container

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened
- **UX_HOST_CLASS_PIMA_RC_OBJECT_ALREADY_OPENED**: (0x2021) Object already opened.
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory to create PIMA command.

Example

```
/* Open the object. */

status = ux_host_class_pima_object_open(pima, pima_session,
    object_handle, pima_object);

/* Check status. */
if (status != UX_SUCCESS)
    return(status);
```

ux_host_class_pima_object_get

Get an object stored in the Responder

Prototype

```
UINT ux_host_class_pima_object_get(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle,
    UX_HOST_CLASS_PIMA_OBJECT *object,
    UCHAR *object_buffer,
    ULONG object_buffer_length,
    ULONG *object_actual_length)
```

Description

This function gets an object on the responder.

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session
- **object_handle**: handle of the object
- **object**: Pointer to object information container
- **object_buffer**: Address of object data
- **object_buffer_length**: Requested length of object
- **object_actual_length**: Length of object returned

Return Value

- **UX_SUCCESS**: (0x00) The object was transfered
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened
- **UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED**: (0x2023) Object not opened.
- **UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED**: (0x200f) Access to object denied
- **UX_HOST_CLASS_PIMA_RC_INCOMPLETE_TRANSFER**: (0x2007) Transfer is incomplete
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory to create PIMA command.
- **UX_TRANSFER_ERROR**: (0x23) Transfer error while reading object

Example

```
/* Open the object. */

status = ux_host_class_pima_object_open(pima, pima_session,
    object_handle, pima_object);

/* Check status. */
if (status != UX_SUCCESS)
    return(status);

/* Set the object buffer pointer. */
object_buffer = pima_object ->ux_host_class_pima_object_buffer;

/* Obtain all the object data. */
while(object_length != 0)
{
    /* Calculate what length to request. */
    if (object_length > UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER)
        /* Request maximum length. */
        requested_length = UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER;
    else
        /* Request remaining length. */
        requested_length = object_length;

    /* Get the object data. */
    status = ux_host_class_pima_object_get(pima, pima_session,
        object_handle, pima_object, object_buffer,
        requested_length, &actual_length);

    if (status != UX_SUCCESS)
    {
        /* We had a problem, abort the transfer. */
        ux_host_class_pima_object_transfer_abort(pima, pima_session,
            object_handle, pima_object);

        /* And close the object. */
        ux_host_class_pima_object_close(pima, pima_session,
            object_handle, pima_object, object);

        return(status);

    }

    /* We have received some data, update the length remaining. */
    object_length -= actual_length;

    /* Update the buffer address. */
    object_buffer += actual_length;

}

/* Close the object. */
status = ux_host_class_pima_object_close(pima, pima_session,
    object_handle, pima_object, object);
```

ux_host_class_pima_object_send

Send an object stored in the Responder

Prototype

```
UINT ux_host_class_pima_object_send(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    UX_HOST_CLASS_PIMA_OBJECT *object,
    UCHAR *object_buffer, ULONG object_buffer_length)
```

Description

This function sends an object to the responder

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_sessio**: Pointer to PIMA session
- **object_handle**: handle of the object
- **object**: Pointer to object information container
- **object_buffer**: Address of object data
- **object_buffer_length**: Requested length of object

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened
- **UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED**: (0x2023) Object not opened.
- **UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED**: (0x200f) Access to object denied
- **UX_HOST_CLASS_PIMA_RC_INCOMPLETE_TRANSFER**: (0x2007) Transfer is incomplete
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory to create PIMA command.
- **UX_TRANSFER_ERROR**: (0x23) Transfer error while writing object

Example

```
/* Open the object. */
status = ux_host_class_pima_object_open(pima, pima_session,
    object_handle, pima_object);

/* Get the object length. */
object_length = pima_object ->ux_host_class_pima_object_compressed_size;

/* Recall the object buffer address. */
pima_object_buffer = pima_object ->ux_host_class_pima_object_buffer;

/* Send all the object data. */
while(object_length != 0)
{
    /* Calculate what length to request. */
    if (object_length > UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER)
        /* Request maximum length. */
        requested_length = UX_PICTBRIDGE_MAX_PIMA_OBJECT_BUFFER;
    else
        /* Request remaining length. */
        requested_length = object_length;

    /* Send the object data. */
    status = ux_host_class_pima_object_send(pima,
        pima_session, pima_object,
        pima_object_buffer, requested_length);

    if (status != UX_SUCCESS)
    {
        /* Abort the transfer. */
        ux_host_class_pima_object_transfer_abort(pima, pima_session,
            object_handle, pima_object);

        /* Return status. */
        return(status);
    }

    /* We have sent some data, update the length remaining. */
    object_length -= requested_length;
}

/* Close the object. */
status = ux_host_class_pima_object_close(pima, pima_session, object_handle,
    pima_object, object);
```

ux_host_class_pima_thumb_get

Get a thumb object stored in the Responder

Prototype

```
UINT ux_host_class_pima_thumb_get(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle, UX_HOST_CLASS_PIMA_OBJECT *object,
    UCHAR *thumb_buffer, ULONG thumb_buffer_length,
    ULONG *thumb_actual_length)
```

Description

This function gets a thumb object on the responder

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session

- **object_handle**: handle of the object
- **object**: Pointer to object information container
- **thumb_buffer**: Address of thumb object data
- **thumb_buffer_length**: Requested length of thumb object
- **thumb_actual_length**: Length of thumb object returned

Return Value

- **UX_SUCCESS **: (0x00) The data transfer was completed
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened
- **UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED**: (0x2023) Object not opened.
- **UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED**: (0x200f) Access to object denied
- **UX_HOST_CLASS_PIMA_RC_INCOMPLETE_TRANSFER**: (0x2007) Transfer is incomplete
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory to create PIMA command.
- **UX_TRANSFER_ERROR**: (0x23) Transfer error while reading object

Example

```
/* Get the thumb object data. */

status = ux_host_class_pima_thumb_get(pima, pima_session,
    object_handle, pima_object, object_buffer,
    requested_length, &actual_length);

if (status != UX_SUCCESS)
{
    /* And close the object. */
    ux_host_class_pima_object_close(pima, pima_session, object_handle, pima_object, object);

    return(status);
}
```

ux_host_class_pima_object_delete

Delete an object stored in the Responder

Prototype

```
UINT ux_host_class_pima_object_delete(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle)
```

Description

This function deletes an object on the responder

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session
- **object_handle**: handle of the object

Return Value

- **UX_SUCCESS**: (0x00) The object was deleted.
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened
- **UX_HOST_CLASS_PIMA_RC_ACCESS_DENIED**: (0x200f) Cannot delete object
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory to create PIMA command.

Example

```
/* Delete the object. */
status = ux_host_class_pima_object_delete(pima, pima_session, object_handle, pima_object);

/* Check status. */
if (status != UX_SUCCESS)
    return(status);
```

ux_host_class_pima_object_close

Close an object stored in the Responder

Prototype

```
UINT ux_host_class_pima_object_close(UX_HOST_CLASS_PIMA *pima,
    UX_HOST_CLASS_PIMA_SESSION *pima_session,
    ULONG object_handle, UX_HOST_CLASS_PIMA_OBJECT *object)
```

Description

This function closes an object on the responder

Parameters

- **pima**: Pointer to the pima class instance.
- **pima_session**: Pointer to PIMA session
- **object_handle**: Pandle of the object
- **object**: Pointer to object

Return Value

- **UX_SUCCESS**: (0x00) The object was closed
- **UX_HOST_CLASS_PIMA_RC_SESSION_NOT_OPEN**: (0x2003) Session not opened
- **UX_HOST_CLASS_PIMA_RC_OBJECT_NOT_OPENED**: (0x2023) Object not opened.
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory to create PIMA command.

Example

```
/* Close the object. */
status = ux_host_class_pima_object_close(pima, pima_session, object_handle, object);
```

ux_host_class_gser_read

Read from the generic serial interface

Prototype

```
UINT ux_host_class_gser_read(UX_HOST_CLASS_GSER *gser,
    ULONG interface_index, UCHAR *data_pointer,
    ULONG requested_length,
    ULONG *actual_length)
```

Description

This function reads from the generic serial interface. The call is blocking and only returns when there is either an error or when the transfer is complete.

Parameters

- **gser**: Pointer to the gser class instance.
- **interface_index**: Interface index to read from
- **data_pointer**: Pointer to the buffer address of the data payload
- **requested_length**: Length to be received.
- **actual_length**: Length actually received.

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_TRANSFER_TIMEOUT**: (0x5c) Transfer timeout, reading incomplete.

Example

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_gser_read(cdc_acm, interface_index,data_pointer, requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_gser_write

Write to the generic serial interface

Prototype

```
UINT ux_host_class_gser_write(UX_HOST_CLASS_GSER *gser,
    ULONG interface_index, UCHAR *data_pointer,
    ULONG requested_length, ULONG *actual_length)
```

Description

This function writes to the generic serial interface. The call is blocking and only returns when there is either an error or when the transfer is complete.

Parameters

- **gser**: Pointer to the gser class instance.
- **interface_index**: Interface to which to write
- **data_pointer**: Pointer to the buffer address of the data payload.
- **requested_length**: Length to be sent.
- **actual_length**: Length actually sent.

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_TRANSFER_TIMEOUT**: (0x5c) Transfer timeout, writing incomplete.

Example

```
UINT status;

/* The following example illustrates this service. */
status = ux_host_class_cdc_acm_write(gser, data_pointer, requested_length, &actual_length);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_gser_ioctl

Perform an IOCTL function to the generic serial interface

Prototype

```
UINT ux_host_class_gser_ioctl(UX_HOST_CLASS_GSER *gser,
    ULONG ioctl_function,
    VOID *parameter)
```

Description

This function performs a specific ioctl function to the gser interface. The call is blocking and only returns when there is either an error or when the command is completed.

Parameters

- **gser**: Pointer to the gser class instance.
- **ioctl_function**: ioctl function to be performed. See table below for one of the allowed ioctl functions.
- **parameter**: Pointerto a parameter specific to the ioctl

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_MEMORY_INSUFFICIENT**: (0x12) Not enough memory.
- **UX_HOST_CLASS_UNKNOWN**: (0x59) Wrong class instance
- **UX_FUNCTION_NOT_SUPPORTED**: (0x54) Unknown IOCTL function

IOCTL functions:

- UX_HOST_CLASS_GSER_IOCTL_SET_LINE_CODING
- UX_HOST_CLASS_GSER_IOCTL_GET_LINE_CODING
- UX_HOST_CLASS_GSER_IOCTL_SET_LINE_STATE
- UX_HOST_CLASS_GSER_IOCTL_SEND_BREAK
- UX_HOST_CLASS_GSER_IOCTL_ABORT_IN_PIPE
- UX_HOST_CLASS_GSER_IOCTL_ABORT_OUT_PIPE
- UX_HOST_CLASS_GSER_IOCTL_NOTIFICATION_CALLBACK
- UX_HOST_CLASS_GSER_IOCTL_GET_DEVICE_STATUS

Example

```
UINT status;

/* The following example illustrates this service. */

status = ux_host_class_gser_ioctl(gser,
    UX_HOST_CLASS_GSER_IOCTL_GET_LINE_CODING,
    (VOID *)&line_coding);

/* If status equals UX_SUCCESS, the operation was successful. */
```

ux_host_class_gser_reception_start

Start reception on the generic serial interface

Prototype

```
UINT ux_host_class_gser_reception_start(UX_HOST_CLASS_GSER *gser,
    UX_HOST_CLASS_GSER_RECEPTION *gser_reception)
```

Description

This function starts the reception on the generic serial class interface. This function allows for non-blocking reception. When a buffer is received, a callback in invoked into the application.

Parameters

- **gser**: Pointer to the gser class instance.
- **gser_reception**: Structure containing the reception parameters

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_HOST_CLASS_UNKNOWN**: (0x59) Wrong class instance
- **UX_ERROR** (0x01) Error

Example

```
/* Start the reception for gser. AT commands are on interface 2. */
gser_reception.ux_host_class_gser_reception_interface_index =
    UX_DEMO_GSER_AT_INTERFACE;
gser_reception.ux_host_class_gser_reception_block_size =
    UX_DEMO_RECEPTION_BLOCK_SIZE;
gser_reception.ux_host_class_gser_reception_data_buffer =
    gser_reception_buffer;
gser_reception.ux_host_class_gser_reception_data_buffer_size =
    UX_DEMO_RECEPTION_BUFFER_SIZE;
gser_reception.ux_host_class_gser_reception_callback =
    tx_demo_thread_callback;

ux_host_class_gser_reception_start(gser, &gser_reception);
```

ux_host_class_gser_reception_stop

Stop reception on the generic serial interface

Prototype

```
UINT ux_host_class_gser_reception_stop(UX_HOST_CLASS_GSER *gser,
    UX_HOST_CLASS_GSER_RECEPTION *gser_reception)
```

Description

This function stops the reception on the generic serial class interface.

Parameters

- **gser**: Pointer to the gser class instance.
- **gser_reception**: Structure containing the reception parameters

Return Value

- **UX_SUCCESS**: (0x00) The data transfer was completed.
- **UX_HOST_CLASS_UNKNOWN**: (0x59) Wrong class instance
- **UX_ERROR**: (0x01) Error

Example

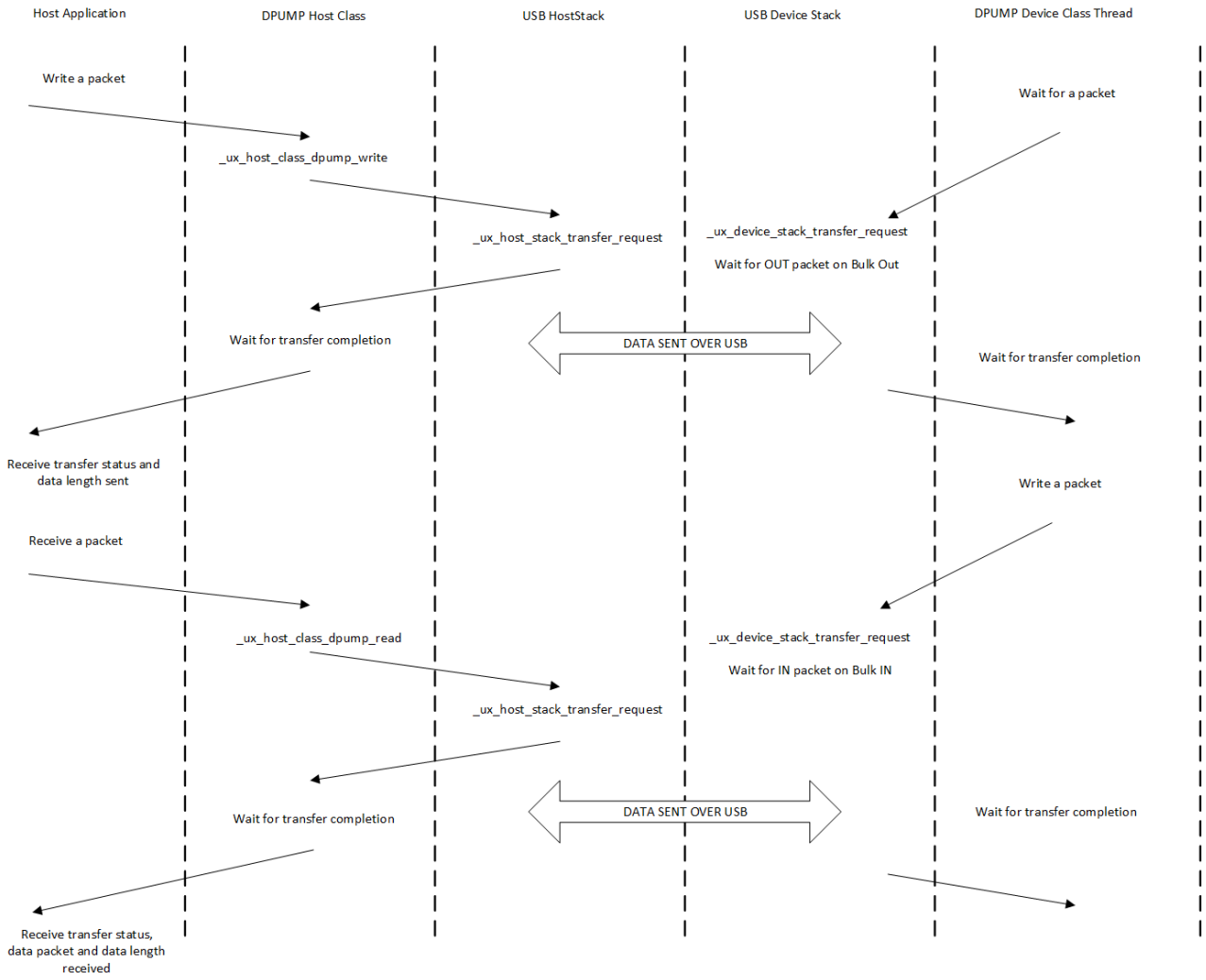
```
/* Stops the reception for gser. */
ux_host_class_gser_reception_stop(gser, &gser_reception);
```

Chapter 3: USBX DPUMP Class Considerations

5/18/2020 • 2 minutes to read

USBX contains a DPUMP class for the host and device side. This class is not a standard class in itself, but rather an example that illustrates how to create a simple device by using two bulk pipes and sending data back and forth on these two pipes. The DPUMP class could be used to start a custom class or for legacy RS232 devices.

USB DPUMP flow chart:



USBX DPUMP Host Class

The host side of the DPUMP Class has two functions, one for sending data and one for receiving data:

- `ux_host_class_dpump_write`
- `ux_host_class_dpump_read`

Both functions are blocking to make the DPUMP application easier. If it is necessary to have both pipes (IN and OUT) running at the same time, the application will be required to create a transmit thread and a receive thread.

The prototype for the writing function is as follows:

```
UINT ux_host_class_dpump_write(UX_HOST_CLASS_DPUMP *dpump,
    UCHAR * data_pointer,
    ULONG requested_length, ULONG *actual_length)
```

Where:

- dpump is the instance of the class
- data_pointer is the pointer to the buffer to be sent
- requested_length is the length to send
- actual_length is the length sent after completion of the transfer, either successfully or partially.

The prototype for the receiving function is the same:

```
UINT ux_host_class_dpump_read(UX_HOST_CLASS_DPUMP *dpump,
    UCHAR *data_pointer,
    ULONG requested_length, ULONG *actual_length)
```

Here is an example of the host DPUMP class where an application writes a packet to the device side and receives the same packet on the reception:

```
/* We start with a 'A' in buffer. */
current_char = 'A';

while(1)
{
    /* Initialize the write buffer. */
    ux_utility_memory_set(out_buffer, current_char,
        UX_HOST_CLASS_DPUMP_PACKET_SIZE);

    /* Increment the character in buffer. */
    current_char++;

    /* Check for upper alphabet limit. */
    if (current_char > 'Z')
        current_char = 'A';

    /* Write to the Data Pump Bulk out endpoint. */
    status = ux_host_class_dpump_write (dpump, out_buffer,
        UX_HOST_CLASS_DPUMP_PACKET_SIZE,
        &actual_length);

    /* Verify that the status and the amount of data is correct. */
    if ((status == UX_SUCCESS) && actual_length == UX_HOST_CLASS_DPUMP_PACKET_SIZE)
    {
        /* Read to the Data Pump Bulk out endpoint. */
        status = ux_host_class_dpump_read (dpump, in_buffer,
            UX_HOST_CLASS_DPUMP_PACKET_SIZE, &actual_length);
    }
}
```

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 4: USBX Pictbridge implementation

5/18/2020 • 6 minutes to read

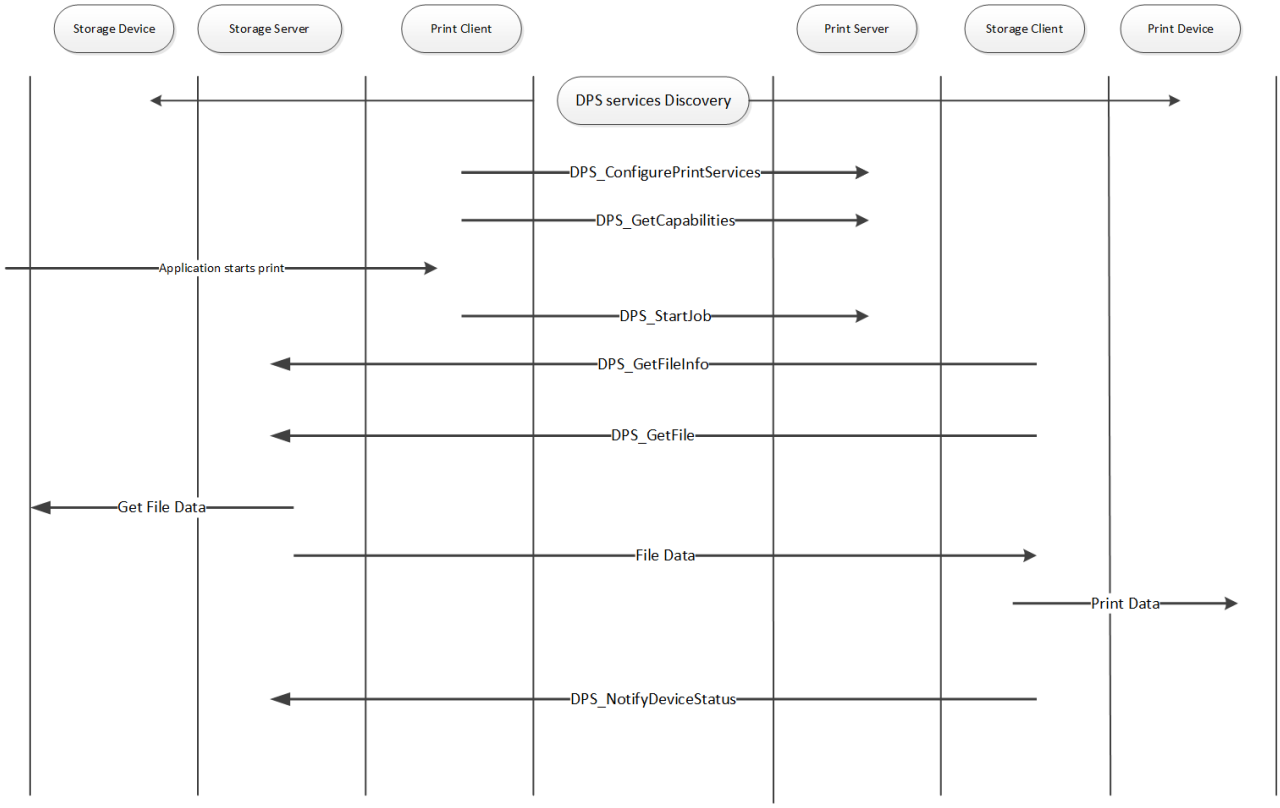
UBSX supports the full Pictbridge implementation both on the host and the device. Pictbridge sits on top of USBX PIMA class on both sides.

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,
    "Azure RTOS",10);
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 host 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]	0x00010000
DpsVersion[1]	0x00010001
DpsVersion[2]	0x00000000
VendorSpecificVersion	0x00010000
PrintServiceAvailable	0x30010000
Qualities[0]	UX_PICTBRIDGE_QUALITIES_DEFAULT
Qualities[1]	UX_PICTBRIDGE_QUALITIES_NORMAL
Qualities[2]	UX_PICTBRIDGE_QUALITIES_DRAFT
Qualities[3]	UX_PICTBRIDGE_QUALITIES_FINE
PaperSizes[0]	UX_PICTBRIDGE_PAPER_SIZES_DEFAULT
PaperSizes[1]	UX_PICTBRIDGE_PAPER_SIZES_4IX6I
PaperSizes[2]	UX_PICTBRIDGE_PAPER_SIZES_L
PaperSizes[3]	UX_PICTBRIDGE_PAPER_SIZES_2L
PaperSizes[4]	UX_PICTBRIDGE_PAPER_SIZES_LETTER
PaperTypes[0]	UX_PICTBRIDGE_PAPER_TYPES_DEFAULT
PaperTypes[1]	UX_PICTBRIDGE_PAPER_TYPES_PLAIN
PaperTypes[2]	UX_PICTBRIDGE_PAPER_TYPES_PHOTO
FileTypes[0]	UX_PICTBRIDGE_FILE_TYPES_DEFAULT
FileTypes[1]	UX_PICTBRIDGE_FILE_TYPES_EXIF_JPEG
FileTypes[2]	UX_PICTBRIDGE_FILE_TYPES_JFIF
FileTypes[3]	UX_PICTBRIDGE_FILE_TYPES_DPOF

PICTBRIDGE FIELD	VALUE
DatePrints[0]	UX_PICTBRIDGE_DATE_PRINTS_DEFAULT
DatePrints[1]	UX_PICTBRIDGE_DATE_PRINTS_OFF
DatePrints[2]	UX_PICTBRIDGE_DATE_PRINTS_ON
FileNamePrints[0]	UX_PICTBRIDGE_FILE_NAME_PRINTS_DEFAULT
FileNamePrints[1]	UX_PICTBRIDGE_FILE_NAME_PRINTS_OFF
FileNamePrints[2]	UX_PICTBRIDGE_FILE_NAME_PRINTS_ON
ImageOptimizes[0]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_DEFAULT
ImageOptimizes[1]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_OFF
ImageOptimizes[2]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_ON
Layouts[0]	UX_PICTBRIDGE_LAYOUTS_DEFAULT
Layouts[1]	UX_PICTBRIDGE_LAYOUTS_1_UP_BORDER
Layouts[2]	UX_PICTBRIDGE_LAYOUTS_INDEX_PRINT
Layouts[3]	UX_PICTBRIDGE_LAYOUTS_1_UP_BORDERLESS
FixedSizes[0]	UX_PICTBRIDGE_FIXED_SIZE_DEFAULT
FixedSizes[1]	UX_PICTBRIDGE_FIXED_SIZE_35IX5I
FixedSizes[2]	UX_PICTBRIDGE_FIXED_SIZE_4IX6I
FixedSizes[3]	UX_PICTBRIDGE_FIXED_SIZE_5IX7I
FixedSizes[4]	UX_PICTBRIDGE_FIXED_SIZE_7CMX10CM
FixedSizes[5]	UX_PICTBRIDGE_FIXED_SIZE_LETTER
FixedSizes[6]	UX_PICTBRIDGE_FIXED_SIZE_A4
Croppings[0]	UX_PICTBRIDGE_CROPPINGS_DEFAULT
Croppings[1]	UX_PICTBRIDGE_CROPPINGS_OFF
Croppings[2]	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 5: USBX OTG

5/18/2020 • 2 minutes to read

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 four 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. 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

STATE	VALUE	DESCRIPTION
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;
```