User's Manual

USBX™ Host Stack

User Guide

Renesas Synergy™ Platform
Synergy Software
Synergy Software (SSP) Component

Renesas Electronics
www.renesas.com

Nov 8, 2018

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

Renesas Synergy Specific Information

If you are using USBX Host Stack for the Renesas Synergy platform, please use the following information.

Customer Support

For Renesas Synergy platform support, please contact Renesas directly:

Support: https://synergygallery.renesas.com/support

America: https://www.renesas.com/en-us/support/contact.html

Europe: https://www.renesas.com/en-eu/support/contact.html

Japan: https://www.renesas.com/ja-jp/support/contact.html

Isochronous Transfer Type Not Supported

Page 7: Isochronous transfer is only supported for USB Host. This transfer type is not supported for USB Device.

Installation and Use of USBX Host

Page 10: If you are using Renesas Synergy SSP and the e² studio ISDE, USBX Host will already be installed. You can ignore the Installation section.

Unsupported Classes/Features

Page 66 and following: The following classes or features are not supported in SSP v1.5.0:

- USBX host class pima
- USBX host class asix
- USBX host class audio
- USBX host class gser
- USBX host class printer
- USBX host class dpump
- USBX Pictbridge implementation
- USBX OTG
- USB composite devices

the high performance USB stack

User Guide for USBX Host Stack

Express Logic, Inc.
858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

http://www.expresslogic.com

Part Number: 000-1010

Revision 5.8SP2

Contents

About This Guide

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

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

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

Organization

Chapter 1 contains an introduction to USBX

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

Chapter 3 provides a functional overview of USBX and basic information about USB

Chapter 4 details the application's interface to USBX in host mode

Chapter 5 is titled USBX DPUMP Class Considerations

Chapter 6 is titled USBX Pictbridge Implementation

Chapter 7 is titled USBX OTG

Chapter 1: Introduction to USBX

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

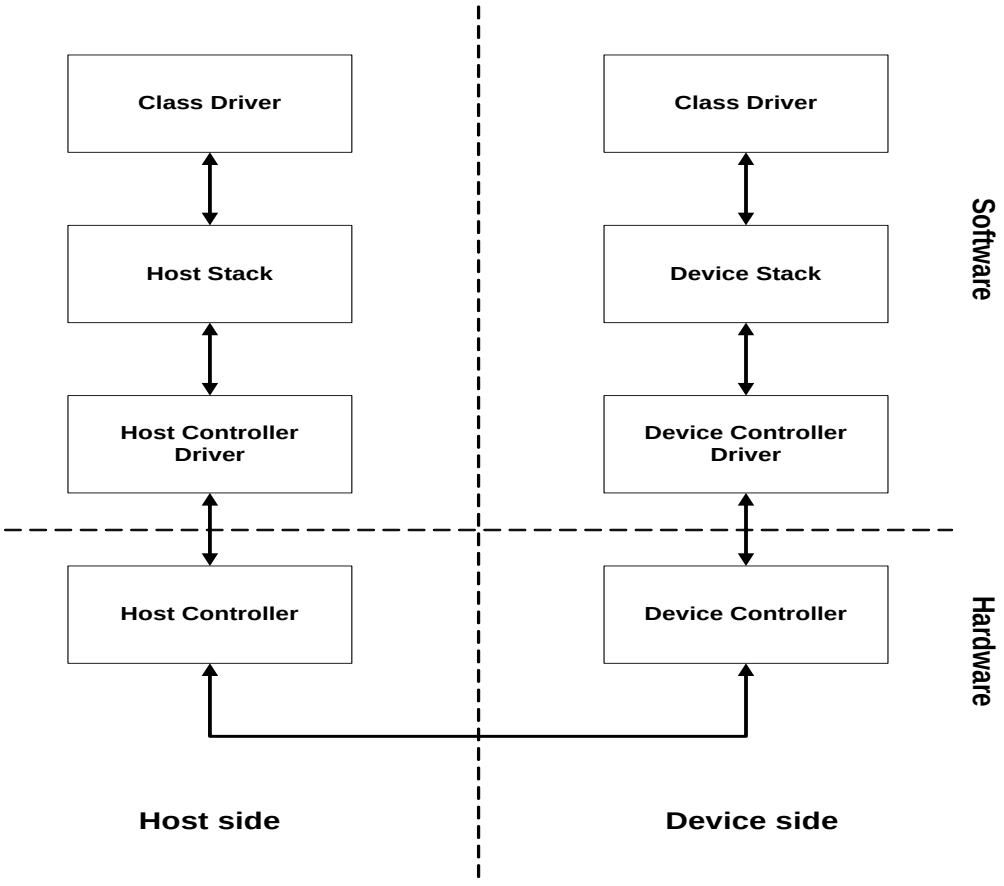
USBX features

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

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

- Controller layer
- Stack layer
- Class layer

The relationship between the USB layers is as follows:



Product Highlights

Complete ThreadX processor support
No royalties
Complete ANSI C source code
Real-time performance
Responsive technical support
Multiple 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

Host Considerations

Computer Type

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

Download Interfaces

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

Debugging Tools

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

Required Hard Disk Space

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

Target Considerations

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

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

Product Distribution

Two USBX packages are available—standard and premium. The standard package includes minimal source code, while the premium package contains the complete USBX source code. Either package is shipped on a single CD.

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

readme_usbx.txt	This file contains specific information about the USBX port, including information about the target processor and the development tools.
ux_api.h	This C header file contains all system equates, data structures, and service prototypes.
ux_port.h	This C header file contains all development-tool-specific data definitions and structures.
ux.lib	This is the binary version of the USBX C library. It is distributed with the standard package.
demo_usbx.c	The C file containing a simple USBX demo

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

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

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

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

Step 3: Add a call to **ux_system_initialize** at or near the beginning of **tx_application_define.** This is where the USBX resources are initialized.

Step 4: Add a call to **ux_host_stack_initialize.**

Step 5: Add one or more calls to initialize the required USBX

Step 6: Add one or more calls to initialize the host controllers available in the system.

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

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

Configuration Options

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

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

UX_PERIODIC_RATE

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

UX_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 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_HOST_HNP_POLLING_THREAD_STACK_SIZE

This value is the stack size of the USB host HNP polling thread. If this symbol I not set, the USBX host HNP polling thread stack size is set to UX_THREAD_STACK_SIZE. Note that the HNP Polling thread is only used in USB OTG.

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

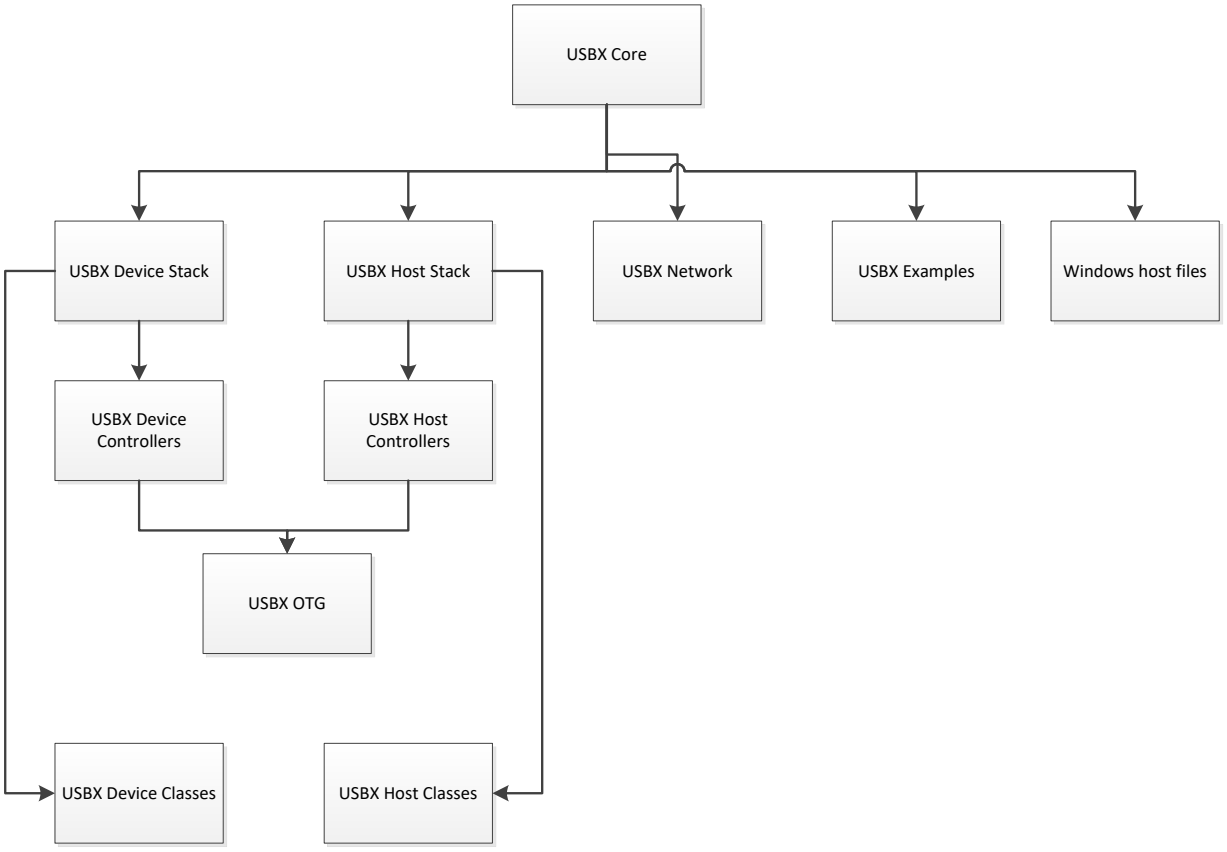
If defined to 1, the ThreadX target port does not use time slice.

UX_MAX_HOST_LUN

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

Source Code Tree

The USBX files are provided in several directories.



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

File Suffix Name	File description
ux_host_stack	usbx host stack core files
ux_host_class	usbx host stack classes files
ux_hcd	usbx host stack controller driver files
ux_device_stack	usbx device stack core files
ux_device_class	usbx device stack classes files
ux_dcd	usbx device stack controller driver files
ux_otg	usbx otg controller driver related files
ux_pictbridge	usbx pictbridge files
ux_utility	usbx utility functions
demo_usbx	demonstration files for USBX

Initialization of USBX resources

USBX has its own memory manager. The memory needs to be allocated to USBX before the host or device side of USBX is initialized. USBX memory manager can accommodate systems where memory can be cached.
The following function initializes USBX memory resources with 128K of regular memory and no separate pool for cache safe memory:

```
/* Initialize USBX Memory */
ux_system_initialize(memory_pointer,(128*1024),UX_NULL,0);
```

The prototype for the ux_system_initialize is as follows:

```
UINT  ux_system_initialize(VOID *regular_memory_pool_start,
                           ULONG regular_memory_size,
                           VOID *cache_safe_memory_pool_start,
                           ULONG cache_safe_memory_size)
```

Input parameters:

VOID *regular_memory_pool_start	Beginning of the regular memory pool
ULONG regular_memory_size	Size of the regular memory pool
VOID *cache_safe_memory_pool_start	Beginning of the cache safe memory pool
ULONG cache_safe_memory_size	Size of the cache safe memory pool

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

In a system where the regular memory is not cache safe and a controller requires to perform DMA memory (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(CHAR_PTR 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 very 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(CHAR_PTR 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 ***usbx_generic.txt*** file. In addition, this file also contains a version history of the corresponding port. Application software can obtain the USBX version by examining the global string ***_ux_version_id***, which is defined in ***ux_port.h***.

Chapter 3: Functional Components of USBX Host Stack

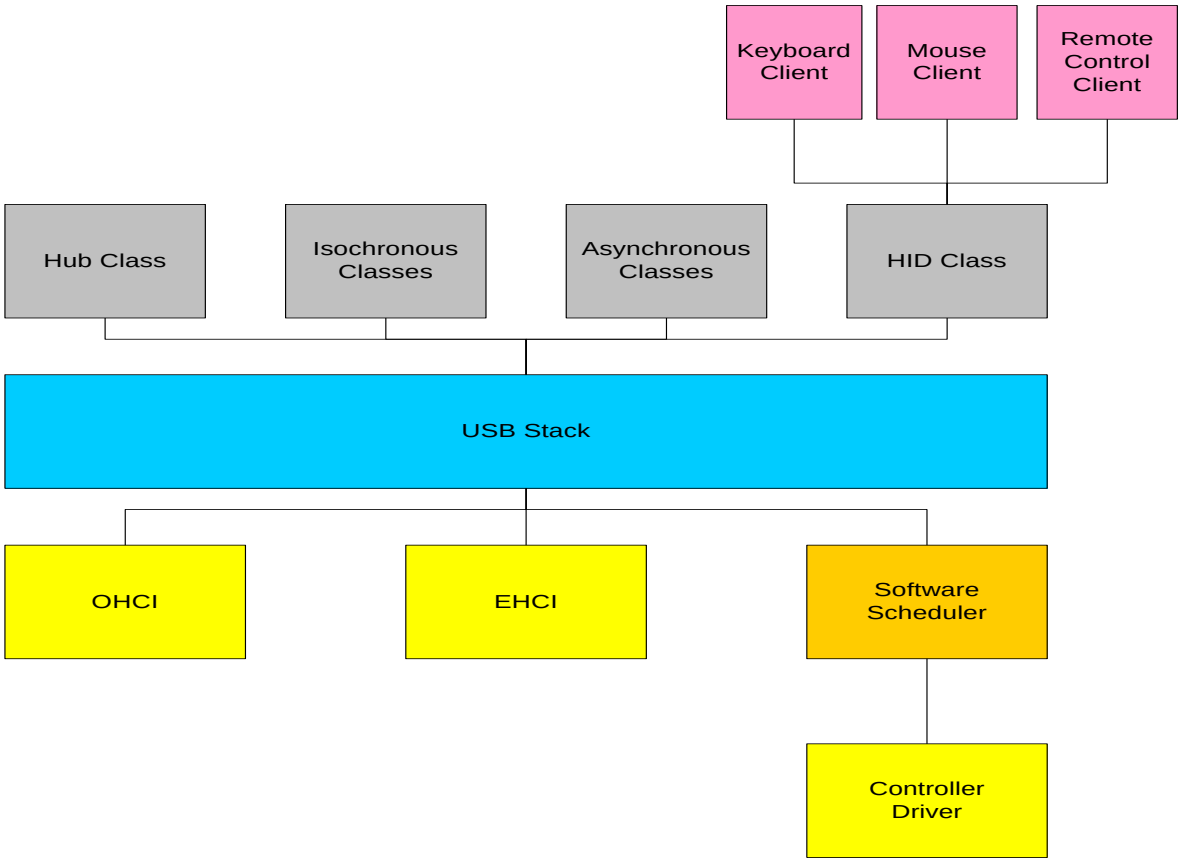
This chapter contains a description of the high performance USBX embedded USB host stack from a functional perspective.

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 APIs

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 one 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

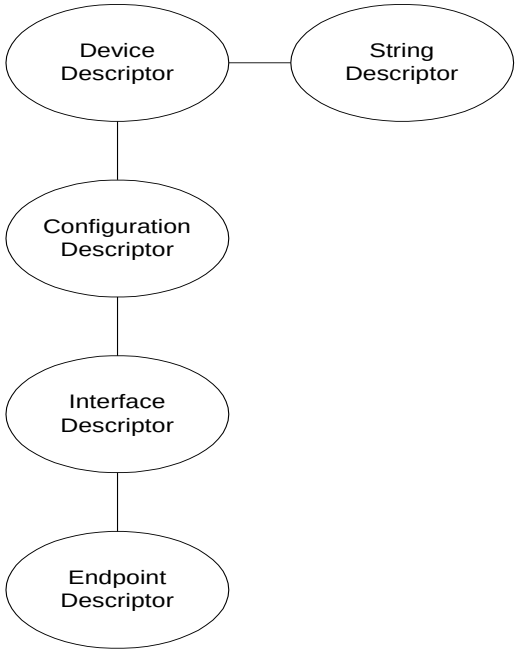
Transfer management provides for a class to perform a transaction on each of the endpoints that have been created. Each logical endpoint contains a component called TRANSFER REQUEST for USB transfer requests. The TRANSFER REQUEST is used by the stack to describe the transaction. This TRANSFER REQUEST 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 Binary-Coded Decimal 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.
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)

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	iSerialNumber	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;
```

Variable Name	Variable Description
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.
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;
```

Variable Name	Variable Description
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.
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;
```

Variable Name	Variable Description
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 shared between interfaces.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes.
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
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.
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* value is used as the exponent for a 2bInterval-1 value; e.g., a *bInterval* of 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* value is used as the exponent for a 2bInterval-1 value; e.g., a *bInterval* of 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* must 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* number 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;
```

Variable Name	Variable Description
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
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:

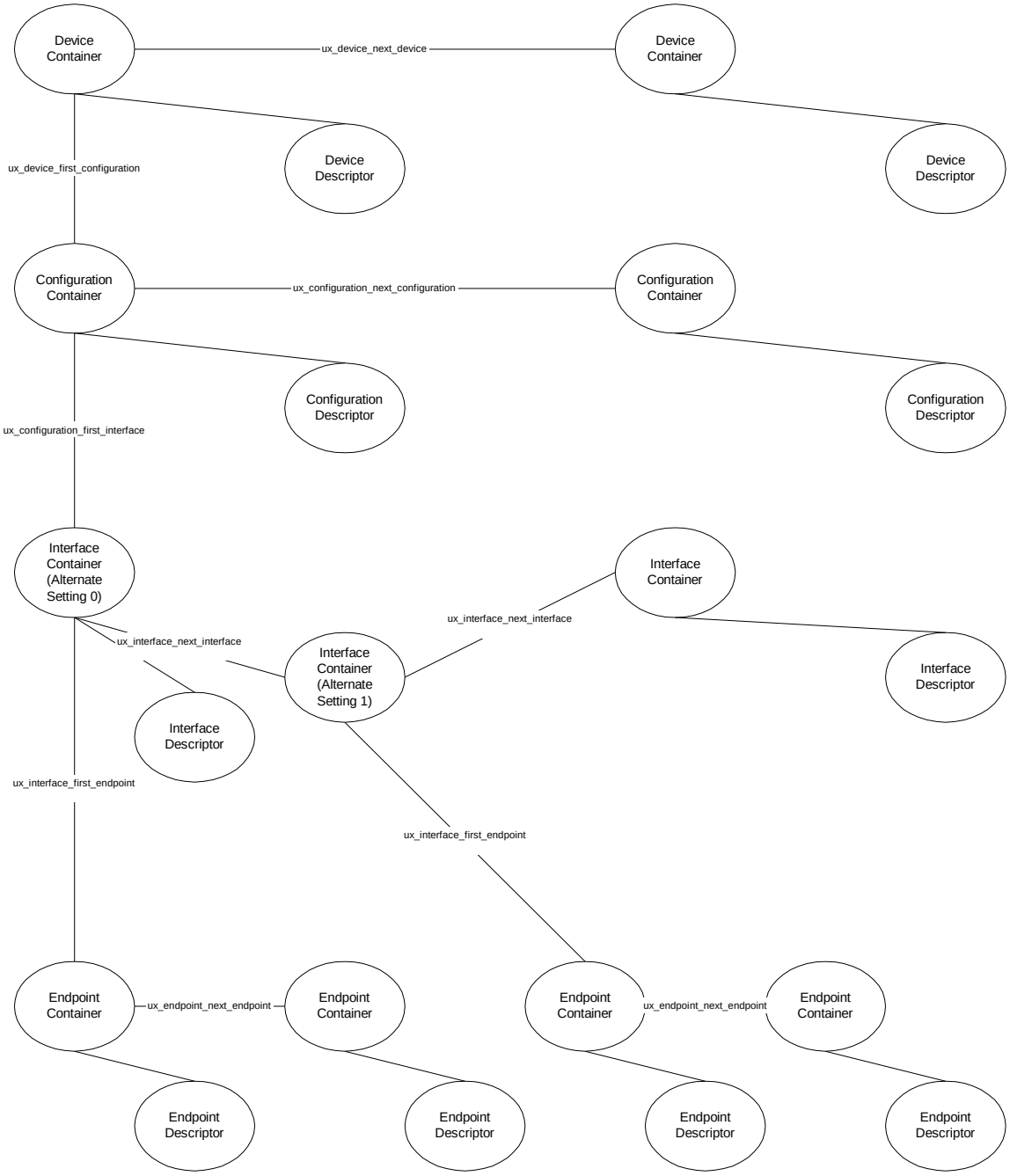
```
typedef struct UX_STRING_DESCRIPTOR_STRUCT
{

      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
Device Descriptor
Device Descriptor
ux_device_first_configuration
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

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.

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.

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(CHAR_PTR 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

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_INSUFFICIENT	(0x12)	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(CHAR_PTR 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.

Example:

```
UINT   status;

/* Create a transfer request for the SET_CONFIGURATION request.
   No data for this request. */
transfer_request -> ux_transfer_endpoint_handle = control_endpoint;
transfer_request -> ux_transfer_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

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

Printer class
HID class
Audio class
Asix class
CDC-ACM class
CDC-ECM class
Pima/PTP class
Prolific class
Storage 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 Values

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 Values

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 Values

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 Values

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 values

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 Values

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 Values

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 Values

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 Values

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 Values

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_hid_client_register

Register a HID client to the HID class

Prototype

```
UINT  ux_host_class_hid_client_register(UCHAR_PTR 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.

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_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported
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_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported
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)	The data transfer was completed.
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported.
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)	The data transfer was completed.
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported.
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 data transfer was completed.
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported.
UX_HOST_CLASS_HID_REPORT_ERROR	(0x70)	Error in the periodic report.
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 data transfer was completed.
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Function not supported.
UX_HOST_CLASS_HID_REPORT_ERROR	(0x70)	Error in the periodic report.
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

```
/* 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_button_get

Get mouse buttons

Prototype

```
UINT  ux_host_class_hid_mouse_button_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)	The data transfer was completed.
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)	The data transfer was completed.
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)	The data transfer was completed.
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_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)
        {
            /* 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_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,
                                         ULONG *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_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_volume-
audio_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_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 Values

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 Values

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_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_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_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	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_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

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_ecm_read

Read from the cdc_ecm interface

Prototype

```
UINT  ux_host_class_cdc_ecm_read(UX_HOST_CLASS_CDC_ECM *cdc_ecm,
                                 UCHAR *data_pointer,
                                 ULONG requested_length,
                                 ULONG *actual_length)
```

Description

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

Parameters

cdc_ecm	Pointer to the cdc_ecm 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_TRANSFER_TIMEOUT	(0x5c)	Transfer timeout, reading incomplete.

Example

```
UINT   status;

/* The following example illustrates this service. */

status = ux_host_class_cdc_ecm_read(cdc_cm, data_pointer,
                                    requested_length, &actual_length);

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

ux_host_class_cdc_ecm _write

Write to the cdc_ecm interface

Prototype

```
UINT  ux_host_class_cdc_ecm_write(VOID *cdc_ecm_class, NX_PACKET *packet)
```

Description

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

Parameters

cdc_ecm	Pointer to the cdc_ecm class instance.
packet	Netx data packet

Return Values

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_cdc_ecm_write(cdc_ecm, 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_session	Pointer to PIMA session

Return Values

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 Values

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 Values

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 Values

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
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
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 Values

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	AssociationDesc 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 Values

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 Values

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 Values

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
object	Pointer to object information container

Return Values

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 Values

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_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

Return Values

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

UX_TRANSFER_ERROR	(0x23)	PIMA command. 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 Values

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

UX_TRANSFER_ERROR	(0x23)	PIMA command. 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 Values

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 Values

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 Values

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_CDC_ACM *cdc_acm,
                               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 Values

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(cdc_acm, 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_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 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 Values

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_write(gser, data_pointer, interface_index,
                                  requested_length, &actual_length);
/* 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 Values

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 Values

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

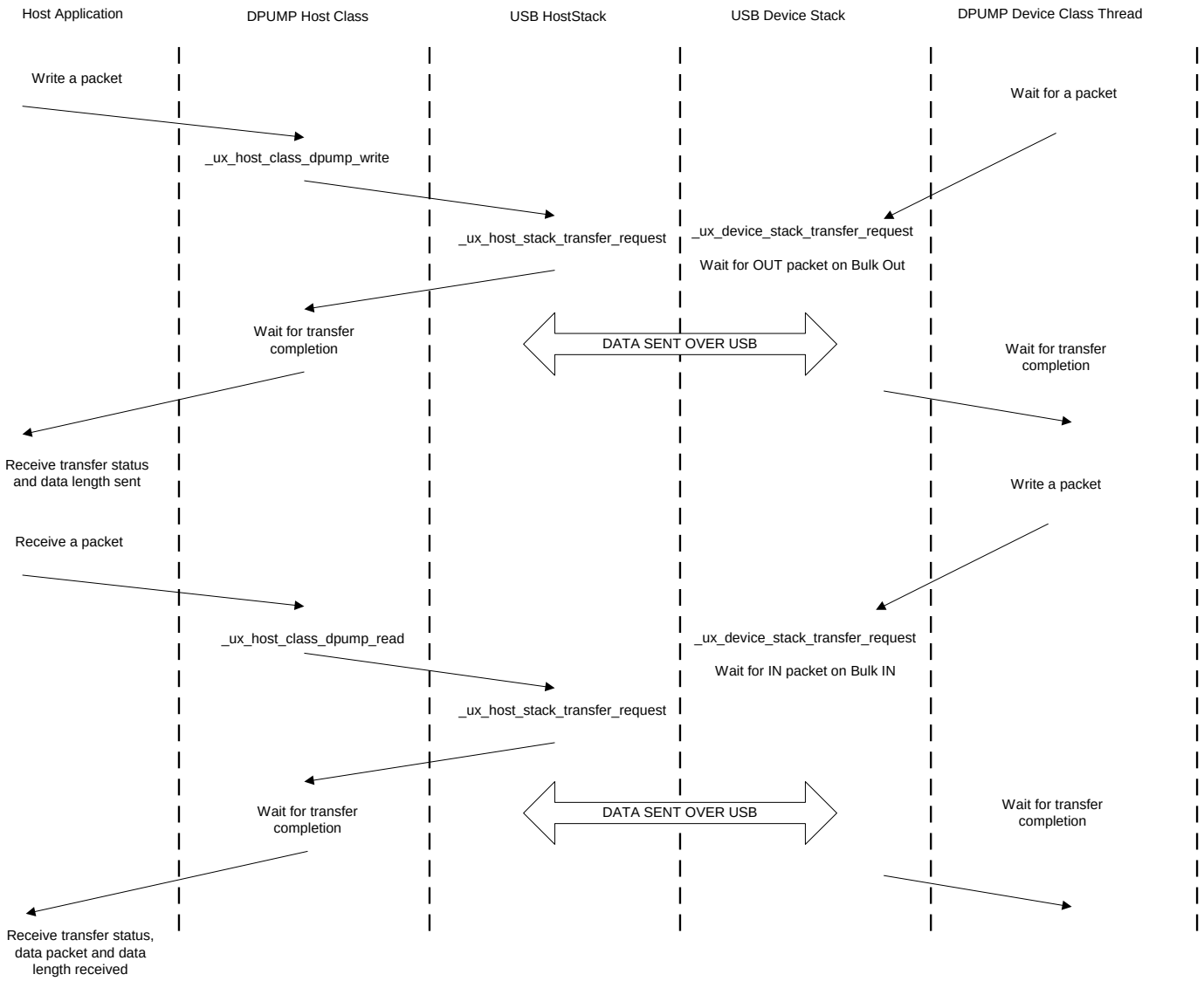
Example

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

Chapter 6: USBX DPUMP Class Considerations

USBX contains a DPUMP class for the host and device side. This class is not a standard class per se, but rather an example that illustrates how to create a simple device by using 2 bulk pipes and sending data back and forth on these 2 pipes. The DPUMP class could be used to start a custom class or for legacy RS232 devices.

USB DPUMP flow chart:



USBX DPUMP Host Class

The host side of the DPUMP Class has 2 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 7: USBX Pictbridge implementation

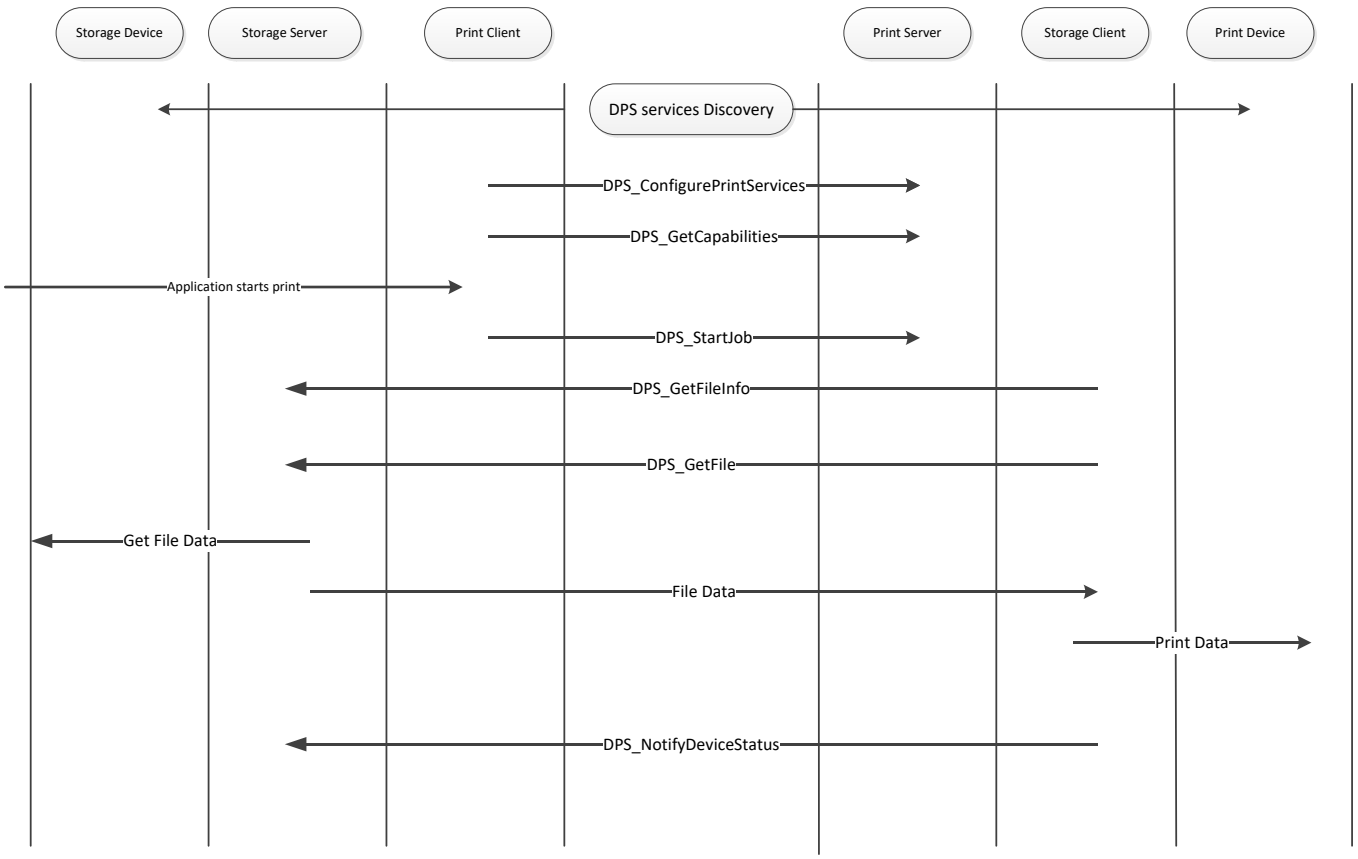
UBSX supports the full Pictbridge implementation both on the host and the device. Pictbridge sits on top of USBX PIMA class on both sides.

The PictBridge standards allows the connection of a digital still camera or a smart phone directly to a printer without a PC, enabling direct printing to certain Pictbridge aware printers.

When a camera or phone is connected to a printer, the printer is the USB host and the camera is the USB device. However, with Pictbridge, the camera will appear as being the host and commands are driven from the camera. The camera is the storage server, the printer the storage client. The camera is the print client and the printer is, of course, the print server.

Pictbridge uses USB as a transport layer but relies on PTP (Picture Transfer Protocol) for the communication protocol.

The following is a diagram of the commands/responses between the DPS client and the DPS server when a print job occurs:



Pictbridge client implementation

The Pictbridge on the client requires the USBX device stack and the PIMA class to be running first.

A device framework describes the PIMA class in the following way:

```
UCHAR device_framework_full_speed[] =
{

    /* Device descriptor */
        0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x20,
        0xA9, 0x04, 0xB6, 0x30, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x01,

    /* Configuration descriptor */
        0x09, 0x02, 0x27, 0x00, 0x01, 0x01, 0x00, 0xc0, 0x32,

    /* Interface descriptor */
        0x09, 0x04, 0x00, 0x00, 0x03, 0x06, 0x01, 0x01, 0x00,

    /* Endpoint descriptor (Bulk Out) */
        0x07, 0x05, 0x01, 0x02, 0x40, 0x00, 0x00,

    /* Endpoint descriptor (Bulk In) */
        0x07, 0x05, 0x82, 0x02, 0x40, 0x00, 0x00,
```

```
    /* Endpoint descriptor (Interrupt) */
        0x07, 0x05, 0x83, 0x03, 0x08, 0x00, 0x60

};
```

The Pima class is using the ID field 0x06 and has its subclass is 0x01 for Still Image and the protocol is 0x01 for PIMA 15740.

3 endpoints are defined in this class, 2 bulks for sending/receiving data and one interrupt for events.

Unlike other USBX device implementations, the Pictbridge application does not need to define a class itself. Rather it invokes the function ux_pictbridge_dpsclient_start. An example is below:

```
    /* Initialize the Pictbridge string components. */
    ux_utility_memory_copy
       (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_name,
       "ExpressLogic",13);
    ux_utility_memory_copy
       (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_product_name,
       "EL_Pictbridge_Camera",21);
    ux_utility_memory_copy
       (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_serial_no,
       "ABC_123",7);
    ux_utility_memory_copy
       (pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_dpsversions,
       "1.0 1.1",7);
    pictbridge.ux_pictbridge_dpslocal.
       ux_pictbridge_devinfo_vendor_specific_version = 0x0100;
    /* Start the Pictbridge client.  */
    status = ux_pictbridge_dpsclient_start(&pictbridge);

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

The parameters passed to the pictbridge client are as follows:

```
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_name
      : String of Vendor name
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_product_name
      : String of product name
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_serial_no,
      : String of serial number
pictbridge.ux_pictbridge_dpslocal.ux_pictbridge_devinfo_dpsversions
      : String of version
pictbridge.ux_pictbridge_dpslocal.
      ux_pictbridge_devinfo_vendor_specific_version
      : Value set to  0x0100;
```

The next step is for the device and the host to synchronize and be ready to exchange information.

This is done by waiting on an event flag as follows:

```
/* We should wait for the host and the client to discover one another.  */
status =  ux_utility_event_flags_get
      (&pictbridge.ux_pictbridge_event_flags_group,
       UX_PICTBRIDGE_EVENT_FLAG_DISCOVERY,TX_AND_CLEAR, &actual_flags,
       UX_PICTBRIDGE_EVENT_TIMEOUT);
```

If the state machine is in the DISCOVERY_COMPLETE state, the camera side (the DPS client) will gather information regarding the printer and its capabilities.

If the DPS client is ready to accept a print job, its status will be set to UX_PICTBRIDGE_NEW_JOB_TRUE. It can be checked below:

```
/* Check if the printer is ready for a print job.  */
if (pictbridge.ux_pictbridge_dpsclient.ux_pictbridge_devinfo_newjobok ==
                  UX_PICTBRIDGE_NEW_JOB_TRUE)
      /* We can print something … */
```

Next some print joib descriptors need to be filled as follows:

```
/* We can start a new job. Fill in the JobConfig and PrintInfo structures. */
jobinfo = &pictbridge.ux_pictbridge_jobinfo;

/* Attach a printinfo structure to the job.  */
jobinfo -> ux_pictbridge_jobinfo_printinfo_start = &printinfo;


/* Set the default values for print job.  */
jobinfo -> ux_pictbridge_jobinfo_quality  =
                                  UX_PICTBRIDGE_QUALITIES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_papersize  =
                                 UX_PICTBRIDGE_PAPER_SIZES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_papertype  =
                                 UX_PICTBRIDGE_PAPER_TYPES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_filetype  =
                                  UX_PICTBRIDGE_FILE_TYPES_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_dateprint  =
                                 UX_PICTBRIDGE_DATE_PRINTS_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_filenameprint  =
                            UX_PICTBRIDGE_FILE_NAME_PRINTS_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_imageoptimize  =
                                 UX_PICTBRIDGE_IMAGE_OPTIMIZES_OFF;
jobinfo -> ux_pictbridge_jobinfo_layout  =
                                     UX_PICTBRIDGE_LAYOUTS_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_fixedsize  =
                                  UX_PICTBRIDGE_FIXED_SIZE_DEFAULT;
jobinfo -> ux_pictbridge_jobinfo_cropping  =
                                  UX_PICTBRIDGE_CROPPINGS_DEFAULT;

/* Program the callback function for reading the object data.  */
jobinfo -> ux_pictbridge_jobinfo_object_data_read  =
                                                 ux_demo_object_data_copy;
```

```
/* This is a demo, the fileID is hardwired (1 and 2 for scripts, 3 for photo
   to be printed.  */
printinfo.ux_pictbridge_printinfo_fileid  =
                                    UX_PICTBRIDGE_OBJECT_HANDLE_PRINT;
ux_utility_memory_copy(printinfo.ux_pictbridge_printinfo_filename,
                       "Pictbridge demo file", 20);
ux_utility_memory_copy(printinfo.ux_pictbridge_printinfo_date, "01/01/2008",
                       10);

/* Fill in the object info to be printed.  First get the pointer to the
   object container in the job info structure. */
object = (UX_SLAVE_CLASS_PIMA_OBJECT *) jobinfo ->
                                        ux_pictbridge_jobinfo_object;

/* Store the object format: JPEG picture.  */
object -> ux_device_class_pima_object_format  =
      UX_DEVICE_CLASS_PIMA_OFC_EXIF_JPEG;
object -> ux_device_class_pima_object_compressed_size   =  IMAGE_LEN;
object -> ux_device_class_pima_object_offset            =  0;
object -> ux_device_class_pima_object_handle_id         =
                                        UX_PICTBRIDGE_OBJECT_HANDLE_PRINT;
object -> ux_device_class_pima_object_length            =  IMAGE_LEN;

/* File name is in Unicode.  */
ux_utility_string_to_unicode("JPEG Image", object ->
                                    ux_device_class_pima_object_filename);

/* And start the job.  */
status =ux_pictbridge_dpsclient_api_start_job(&pictbridge);
```

The Pictbridge client now has a print job to do and will fetch the image blocks at a time from the application through the callback defined in the field

```
jobinfo -> ux_pictbridge_jobinfo_object_data_read
```

The prototype of that function is defined as:

ux_pictbridge_jobinfo_object_data_read

Copying a block of data from user space for printing

Prototype

```
UINT ux_pictbridge_jobinfo_object_data_read(UX_PICTBRIDGE *pictbridge,
      UCHAR *object_buffer, ULONG object_offset, ULONG object_length,
      ULONG *actual_length)
```

Description

This function is called when the DPS client needs to retrieve a data block to print to the target Pictbridge printer.

Parameters

pictbridge	Pointer to the pictbridge class instance.
object_buffer	Pointer to object buffer
object_offset	Where we are starting to read the data block
object_length	Length to be returned
actual_length	Actual length returned

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0x01)	The application could not retrieve data.

Example

```
/* Copy the object data.  */
UINT ux_demo_object_data_copy(UX_PICTBRIDGE *pictbridge,UCHAR *object_buffer,
      ULONG object_offset, ULONG object_length, ULONG *actual_length)
{

    /* Copy the demanded object data portion.  */
    ux_utility_memory_copy(object_buffer, image  + object_offset,
                           object_length);

    /* Update the actual length.  */
    *actual_length = object_length;

    /* We have copied the requested data. Return OK.  */
    return(UX_SUCCESS);

}
```

Pictbridge host implementation

The host implementation of Pictbridge is different from the client.

The first thing to do in a Pictbridge host environment is to register the Pima class as the example below shows:

```
status =  ux_host_stack_class_register(ux_system_host_class_pima_name,
                                       ux_host_class_pima_entry);
if(status != UX_SUCCESS)
        return;
```

This class is the generic PTP layer sitting between the USB 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] DpsVersion[1] DpsVersion[2] VendorSpecificVersion	0x00010000 0x00010001 0x00000000 0x00010000
PrintServiceAvailable	0x30010000
Qualities[0] Qualities[1] Qualities[2] Qualities[3]	UX_PICTBRIDGE_QUALITIES_DEFAULT UX_PICTBRIDGE_QUALITIES_NORMAL UX_PICTBRIDGE_QUALITIES_DRAFT UX_PICTBRIDGE_QUALITIES_FINE
PaperSizes[0] PaperSizes[1] PaperSizes[2] PaperSizes[3] PaperSizes[4]	UX_PICTBRIDGE_PAPER_SIZES_DEFAULT UX_PICTBRIDGE_PAPER_SIZES_4IX6I UX_PICTBRIDGE_PAPER_SIZES_L UX_PICTBRIDGE_PAPER_SIZES_2L UX_PICTBRIDGE_PAPER_SIZES_LETTER
PaperTypes[0] PaperTypes[1] PaperTypes[2]	UX_PICTBRIDGE_PAPER_TYPES_DEFAULT UX_PICTBRIDGE_PAPER_TYPES_PLAIN UX_PICTBRIDGE_PAPER_TYPES_PHOTO
FileTypes[0] FileTypes[1] FileTypes[2] FileTypes[3]	UX_PICTBRIDGE_FILE_TYPES_DEFAULT UX_PICTBRIDGE_FILE_TYPES_EXIF_JPEG UX_PICTBRIDGE_FILE_TYPES_JFIF UX_PICTBRIDGE_FILE_TYPES_DPOF
DatePrints[0] DatePrints[1] DatePrints[2]	UX_PICTBRIDGE_DATE_PRINTS_DEFAULT UX_PICTBRIDGE_DATE_PRINTS_OFF UX_PICTBRIDGE_DATE_PRINTS_ON
FileNamePrints[0] FileNamePrints[1] FileNamePrints[2]	UX_PICTBRIDGE_FILE_NAME_PRINTS_DEFAULT UX_PICTBRIDGE_FILE_NAME_PRINTS_OFF UX_PICTBRIDGE_FILE_NAME_PRINTS_ON
ImageOptimizes[0]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_DEFAULT

ImageOptimizes[1] ImageOptimizes[2]	UX_PICTBRIDGE_IMAGE_OPTIMIZES_OFF UX_PICTBRIDGE_IMAGE_OPTIMIZES_ON
Layouts[0] Layouts[1] Layouts[2] Layouts[3]	UX_PICTBRIDGE_LAYOUTS_DEFAULT UX_PICTBRIDGE_LAYOUTS_1_UP_BORDER UX_PICTBRIDGE_LAYOUTS_INDEX_PRINT UX_PICTBRIDGE_LAYOUTS_1_UP_BORDERLESS
FixedSizes[0] FixedSizes[1] FixedSizes[2] FixedSizes[3] FixedSizes[4] FixedSizes[5] FixedSizes[6]	UX_PICTBRIDGE_FIXED_SIZE_DEFAULT UX_PICTBRIDGE_FIXED_SIZE_35IX5I UX_PICTBRIDGE_FIXED_SIZE_4IX6I UX_PICTBRIDGE_FIXED_SIZE_5IX7I UX_PICTBRIDGE_FIXED_SIZE_7CMX10CM UX_PICTBRIDGE_FIXED_SIZE_LETTER UX_PICTBRIDGE_FIXED_SIZE_A4
Croppings[0] Croppings[1] Croppings[2]	UX_PICTBRIDGE_CROPPINGS_DEFAULT UX_PICTBRIDGE_CROPPINGS_OFF UX_PICTBRIDGE_CROPPINGS_ON

The state machine of the DPS host will be set to Idle and ready to accept a new print job.
The host portion of Pictbridge can now be started as the example below shows:

```
/* Activate the pictbridge dpshost.  */
status = ux_pictbridge_dpshost_start(&pictbridge, pima);

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

The Pictbridge host function requires a callback when data is ready to be printed. This is accomplished by passing a function pointer in the pictbridge host structure as follows:

```
/* Set a callback when an object is being received.  */
pictbridge.ux_pictbridge_application_object_data_write =
                                        tx_demo_object_data_write;
```

This function has the following properties:

ux_pictbridge_application_object_data_write

Writing a block of data for printing

Prototype

```
UINT  ux_pictbridge_application_object_data_write(UX_PICTBRIDGE
            *pictbridge,UCHAR *object_buffer, ULONG offset,
            ULONG total_length, ULONG length);
```

Description

This function is called when the DPS server needs to retrieve a data block from the DPS client to print to the local printer.

Parameters

pictbridge	Pointer to the pictbridge class instance.
object_buffer	Pointer to object buffer
object_offset	Where we are starting to read the data block
total_length	Entire length of object
length	Length of this buffer

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0x01)	The application could not print data.

Example

```
/* Copy the object data.  */
UINT tx_demo_object_data_write(UX_PICTBRIDGE *pictbridge,
      UCHAR *object_buffer, ULONG offset, ULONG total_length, ULONG length);
{
UINT status;

    /* Send the data to the local printer. */
    status = local_printer_data_send(object_buffer, length);

    /* We have printed the requested data. Return status.  */
    return(status);

}
```

Chapter 8: USBX OTG

USBX supports the OTG functionalities of USB when an OTG compliant USB controller is available in the hardware design.

USBX supports OTG in the core USB stack. But for OTG to function, it requires a specific USB controller. USBX OTG controller functions can be found in the usbx_otg directory. The current USBX version only supports the NXP LPC3131 with full OTG capabilities.

The regular controller driver functions (host or device) can still be found in the standard USBX usbx_device_controllers and usbx_host_controllers but the usbx_otg directory contains the specific OTG functions associated with the USB controller.

There are 4 categories of functions for an OTG controller in addition to the usual host/device functions:

- VBUS specific functions
- Start and Stop of the controller
- USB role manager
- Interrupt handlers

VBUS functions
Each controller needs to have a VBUS manager to change the state of VBUS based on power management requirements. Usually this function only performs turning on or off VBUS

Start and Stop the controller
Unlike a regular USB implementation, OTG requires the host and/or the device stack to be activated and deactivated when the role changes.

USB role Manager
The USB role manager receives commands to change the state of the USB. There are several states that need transitions to and from:

State	Value	Description
UX_OTG_IDLE	0	The device is Idle. Usually not connected to anything
UX_OTG_IDLE_TO_HOST	1	Device is connected with type A connector
UX_OTG_IDLE_TO_SLAVE	2	Device is connected with type B connector
UX_OTG_HOST_TO_IDLE	3	Host device got disconnected
UX_OTG_HOST_TO_SLAVE	4	Role swap from Host to Slave
UX_OTG_SLAVE_TO_IDLE	5	Slave device is disconnected
UX_OTG_SLAVE_TO_HOST	6	Role swap from Slave to Host

Interrupt handlers
Both host and device controller drivers for OTG needs different interrupt handlers to monitor signals beyond traditional USB interrupts, in particular signals due to SRP and VBUS.

How to initialize a USB OTG controller. We use the NXP LPC3131 as an example here:

```
/* Initialize the LPC3131 OTG controller.  */
 status = ux_otg_lpc3131_initialize(0x19000000, lpc3131_vbus_function,
                                    tx_demo_change_mode_callback);
```

In this example, we initialize the LPC3131 in OTG mode by passing a VBUS function and a callback for mode change (from host to slave or vice versa).

The callback function should simply record the new mode and wake up a pending thread to act up the new state:

```
void  tx_demo_change_mode_callback(ULONG mode)
{

    /* Simply save the otg mode.  */
    otg_mode = mode;
```

```
    /* Wake up the thread that is waiting.  */
    ux_utility_semaphore_put(&mode_change_semaphore);

}
```

The mode value that is passed can have the following values:

- UX_OTG_MODE_IDLE
- UX_OTG_MODE_SLAVE
- UX_OTG_MODE_HOST

The application can always check what the device is by looking at the variable:

```
ux_system_otg -> ux_system_otg_device_type
```

Its values can be:

- UX_OTG_DEVICE_A
- UX_OTG_DEVICE_B
- UX_OTG_DEVICE_IDLE

A USB OTG host device can always ask for a role swap by issuing the command:

```
/* Ask the stack to perform a HNP swap with the device.  We relinquish the
   host role to A device. */
ux_host_stack_role_swap(storage -> ux_host_class_storage_device);
```

For a slave device, there is no command to issue but the slave device can set a state to change the role which will be picked up by the host when it issues a GET_STATUS and the swap will then be initiated.

```
/* We are a B device, ask for role swap.  The next GET_STATUS from the host
   will get the status change and do the HNP. */
ux_system_otg -> ux_system_otg_slave_role_swap_flag =
                                          UX_OTG_HOST_REQUEST_FLAG;
```

Index

USBX™ Host Stack User Guide

Publication Date: Rev.5.83 Nov 8, 2018

Published by: Renesas Electronics Corporation

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics Corporation
TOYOSU FORESIA, 3-2-24 Toyosu, Koto-ku, Tokyo 135-0061, Japan

Renesas Electronics America Inc.
1001 Murphy Ranch Road, Milpitas, CA 95035, U.S.A.
Tel: +1-408-432-8888, Fax: +1-408-434-5351

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709 Quantum Plaza, No.27 ZhichunLu, Haidian District, Beijing, 100191 P. R. China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, 200333 P. R. China
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL 2nd Stage, Indiranagar, Bangalore 560 038, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
17F, KAMCO Yangjae Tower, 262, Gangnam-daero, Gangnam-gu, Seoul, 06265 Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5338

Colophon 5.2

USBX™ Host Stack User Guide

Renesas Electronics Corporation

R11UM0008EU0583