User's Manual

USBX™ Device Stack

User's Manual: Software

Renesas Synergy™ Platform

synergygallery.renesas.com

Renesas Electronics
www.renesas.com

Rev. 5.7

the high performance USB stack

User Guide for USBX Device Stack

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

http://www.expresslogic.com

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

Trademarks

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

Warranty Limitations

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

Part Number: 000-1010

Revision 5.7

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 is titled Functional Components of USBX Device Stack

Chapter 4 is titled Description of USBX Device Services

Chapter 5 is titled USBX Device Class Considerations

Chapter 6 is titled USBX DPUMP Class Considerations

Chapter 7 is titled USBX Pictbridge Implementation

Chapter 8 is titled USBX OTG

Chapter 1: Introduction to USBX

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

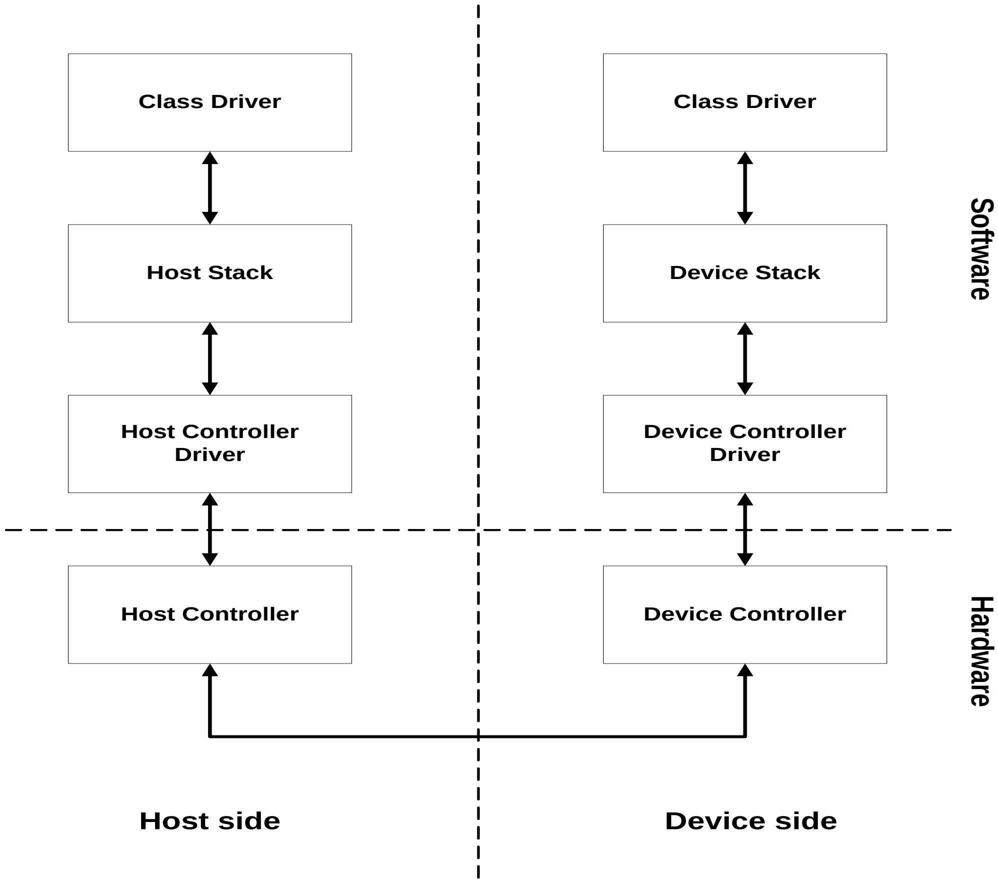
USBX features

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

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

- Controller layer
- Stack layer
- Class layer

The relationship between the USB layers is as follows:



Product Highlights

- Complete ThreadX processor support
- No royalties
- Complete ANSI C source code
- Real-time performance
- Responsive technical support
- Multiple class support
- Multiple class instances
- Integration of classes with ThreadX, FileX and NetX
- Support for USB devices with multiple configuration
- Support for USB composite devices
- Support for USB power management
- Support for USB OTG
- Export trace events for TraceX

Powerful Services of USBX

.

Complete USB Device Framework Support

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

Easy-To-Use APIs

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

Chapter 2: USBX Installation

Host Considerations

Computer Type

Embedded development is usually performed on IBM-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_device_stack_initialize.**

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

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

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

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

Configuration Options

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

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

UX_PERIODIC_RATE

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

UX_THREAD_STACK_SIZE

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

UX_THREAD_PRIORITY_ENUM

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

UX_THREAD_PRIORITY_CLASS

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

UX_THREAD_PRIORITY_KEYBOARD

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

UX_THREAD_PRIORITY_DCD

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

UX_NO_TIME_SLICE

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

UX_MAX_SLAVE_LUN

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

UX_SLAVE_REQUEST_CONTROL_MAX_LENGTH

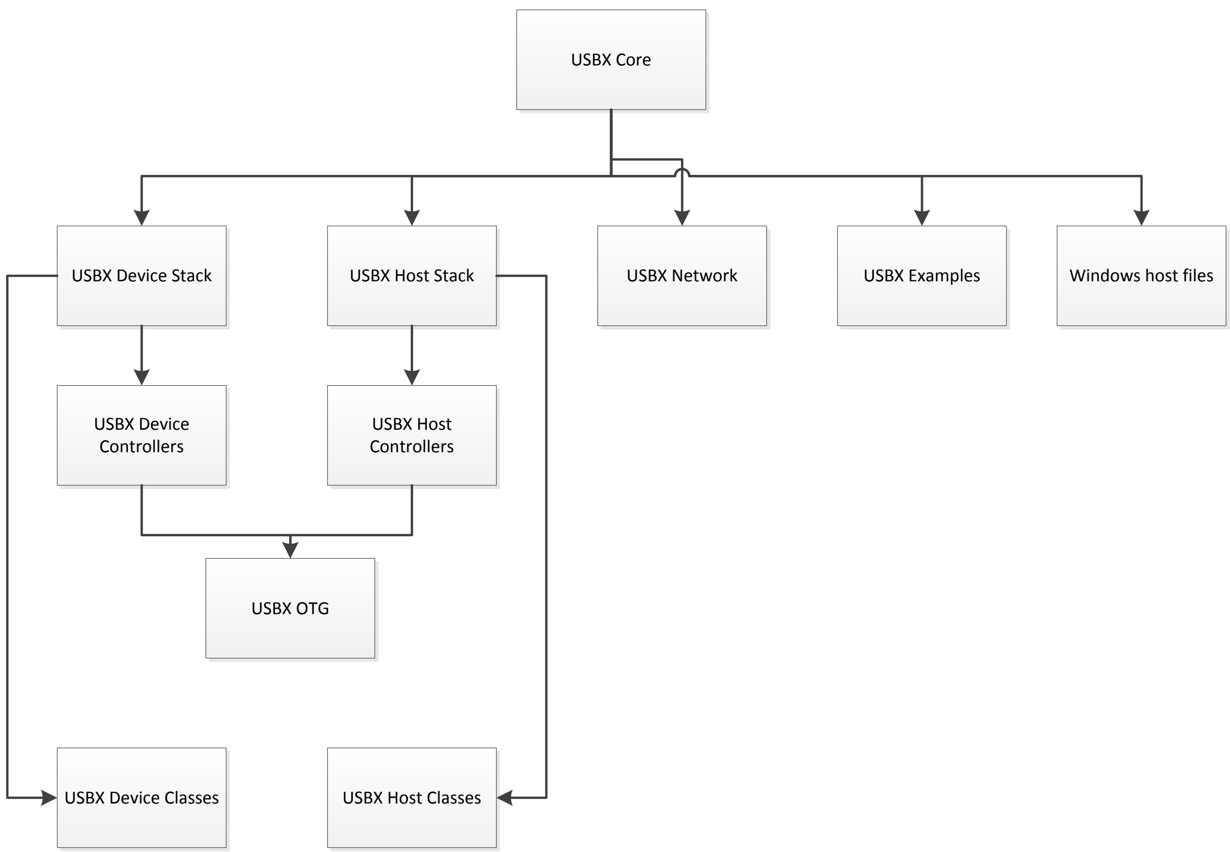
This value represents the maximum number of bytes received on a control endpoint in the device stack. The default is 256 bytes but can be reduced in memory constraint environments

UX_SLAVE_REQUEST_DATA_MAX_LENGTH

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

Source Code Tree

The USBX files are provided in several directories.



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

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

Initialization of USBX resources

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

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

The prototype for the ux_system_initialize is as follows:

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

Input parameters:

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

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

In a system where the regular memory is not cache safe and a controller requires to perform DMA memory (like OHCI, EHCI controllers amongst others) it is necessary to define a memory pool in a cache safe zone.

Definition of USB Device Controller

Only one USB device controller can be defined at any time to operate in device mode. The application initialization file should contain this definition. The example below refers to the OKI USB device controller. For other controllers, the function entry definition has to be changed accordingly.

The following line performs the definition of an OKI controller:

```
ux_dcd_ml6965_initialize(0x7BB00000, 0, 0xB7A00000);
```

The USB device initialization has the following prototype:

```
UINT ux_dcd_ml6965_initialize(ULONG dcd_io, ULONG dcd_irq,
                               ULONG dcd_vbus_address);
```

with the following parameters:

ULONG dcd_io	Address of the controller IO
ULONG dcd_irq	Interrupt used by the controller
ULONG dcd_vbus_address	Address of the VBUS GPIO

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

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

/* The code below is required for installing the device portion of USBX */
status = ux_device_stack_initialize(&device_framework_high_speed,
                           DEVICE_FRAMEWORK_LENGTH_HIGH_SPEED,
                           &device_framework_full_speed,
                           DEVICE_FRAMEWORK_LENGTH_FULL_SPEED,
                           &string_framework, STRING_FRAMEWORK_LENGTH,
                           &language_id_framework, LANGUAGE_ID_FRAMEWORK_LENGTH,
                           UX_NULL);

/* If status equals UX_SUCCESS, installation was successful. */

/* Store the number of LUN in this device storage instance: single LUN.  */
storage_parameter.ux_slave_class_storage_parameter_number_lun = 1;

/* Initialize the storage class parameters for reading/writing to the Flash Disk.  */
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                 ux_slave_class_storage_media_last_lba = 0x1e6bfe;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                 ux_slave_class_storage_media_block_length =  512;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
```

```
                                  ux_slave_class_storage_media_type =  0;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                  ux_slave_class_storage_media_removable_flag =  0x80;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                  ux_slave_class_storage_media_read =
                                                tx_demo_thread_flash_media_read;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                  ux_slave_class_storage_media_write =
                                         tx_demo_thread_flash_media_write;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                  ux_slave_class_storage_media_status =
                                         tx_demo_thread_flash_media_status;

/* Initialize the device storage class. The class is connected with interface 0 */
status =  ux_device_stack_class_register(ux_system_slave_class_storage_name,
                                                ux_device_class_storage_entry,
                                                ux_device_class_storage_thread,0,
                                                (VOID *)&storage_parameter);

/* Register the OKI USB device controllers available in this system */
status = ux_dcd_ml6965_initialize(0x7BB00000, 0, 0xB7A00000);

/* If status equals UX_SUCCESS, registration was successful. */
```

Troubleshooting

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

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.txt*** file. In addition, this file also contains a version history of the corresponding port. Application software can obtain the USBX version by examining the global string ***_ux_version_id***, which is defined in ***ux_port.h***.

Chapter 3: Functional Components of USBX Device Stack

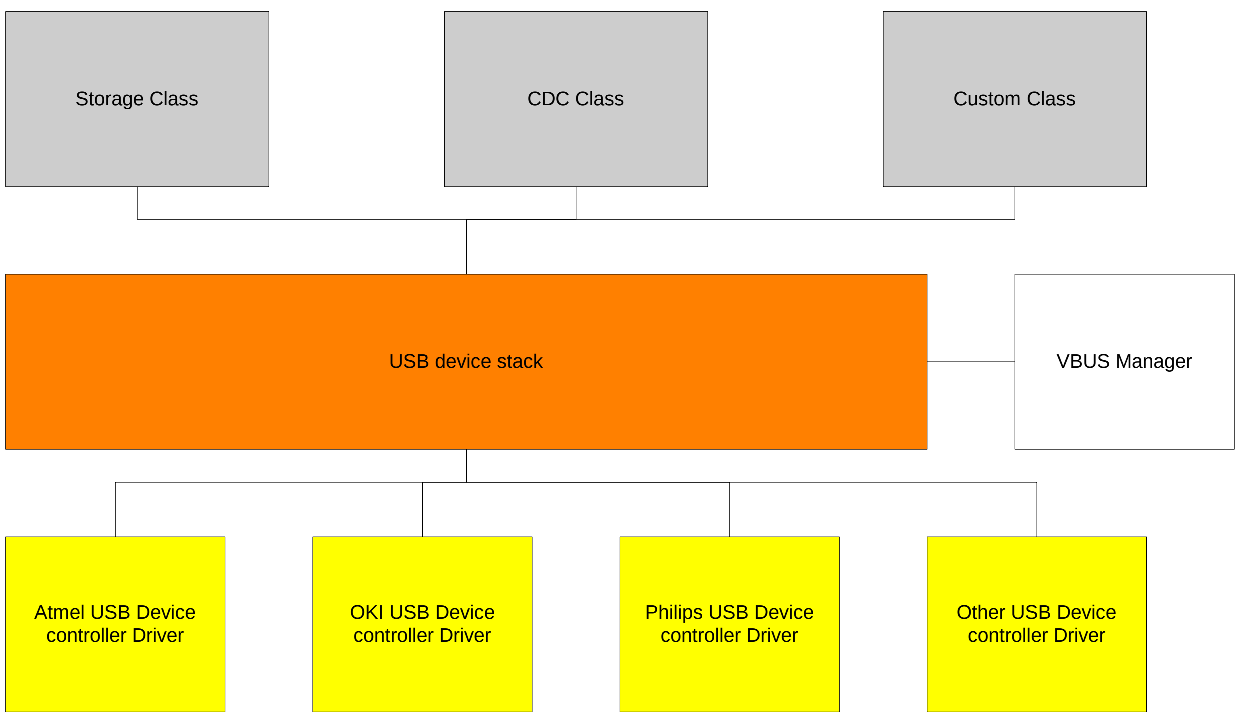
This chapter contains a description of the high performance USBX embedded USB device stack from a functional perspective.

Execution Overview:

USBX for the device is composed of several components:

Initialization
Application interface calls
Device Classes
USB Device Stack
Device controller
VBUS manager

The following diagram illustrates the USBX Device stack:



Initialization

In order to activate USBX, the function *ux_system_initialize* must be called. This function initializes the memory resources of USBX.

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

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

Application Interface Calls

There are two levels of APIs in USBX:

USB Device Stack APIs
USB Device Class APIs

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

USB Device Stack APIs

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

USB Device Class APIs

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

Device Framework

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

Definition of the Components of the Device Framework

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

- Device Descriptor
- Configuration Descriptor
- Interface Descriptor
- Endpoint Descriptor

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

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

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

```
#define DEVICE_FRAMEWORK_LENGTH_HIGH_SPEED 60
UCHAR device_framework_high_speed[] = {

      /* Device descriptor */
            0x12, 0x01, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40,
            0x0a, 0x07, 0x25, 0x40, 0x01, 0x00, 0x01, 0x02,
            0x03, 0x01,

      /* Device qualifier descriptor */
            0x0a, 0x06, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40,
            0x01, 0x00,

      /* Configuration descriptor */
            0x09, 0x02, 0x20, 0x00, 0x01, 0x01, 0x00, 0xc0,
            0x32,

      /* Interface descriptor */
            0x09, 0x04, 0x00, 0x00, 0x02, 0x08, 0x06, 0x50,
            0x00,

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

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

Definition of the Strings of the Device Framework

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

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

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

```
/* String Device Framework:
     Byte 0 and 1: Word containing the language ID: 0x0904 for US
     Byte 2      : Byte containing the index of the descriptor
     Byte 3      : Byte containing the length of the descriptor string
*/

#define STRING_FRAMEWORK_LENGTH 38
UCHAR string_framework[] = {

/* Manufacturer string descriptor: Index 1 */
        0x09, 0x04, 0x01, 0x0c,
        0x45, 0x78, 0x70, 0x72, 0x65, 0x73, 0x20, 0x4c,
        0x6f, 0x67, 0x69, 0x63,

/* Product string descriptor: Index 2 */
        0x09, 0x04, 0x02, 0x0c,
        0x4D, 0x4C, 0x36, 0x39, 0x36, 0x35, 0x30, 0x30,
        0x20, 0x53, 0x44, 0x4B,

/* Serial Number string descriptor: Index 3 */
        0x09, 0x04, 0x03, 0x04,
        0x30, 0x30, 0x30, 0x31
};
```

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

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

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

Definition of the Languages Supported by the Device for each String

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

```
#define LANGUAGE_ID_FRAMEWORK_LENGTH 2
UCHAR language_id_framework[] = {

      /* English. */
           0x09, 0x04
      };
```

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

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

VBUS Manager

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

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

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

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

If the application wishes to handle VBUS by itself, its only requirement is to call the function

```
ux_device_stack_disconnect()
```

when it detects that a device has been extracted. It is not necessary to inform the controller when a device is inserted because the controller will wake up when the BUS RESET assert/deassert signal is detected.

Chapter 4: Description of USBX Device Services

ux_device_stack_alternate_setting_get

Get current alternate setting for an interface value

Prototype

```
UINT  ux_device_stack_alternate_setting_get(ULONG interface_value)
```

Description

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

Input Parameter

interface_value	Interface value for which the current alternate setting is queried.

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_ERROR	(0xFF)	Wrong interface value.

Example

```
ULONG   interface_value;
UINT    status;

/* The following example illustrates this service. */

status = ux_device_stack_alternate_setting_get(interface_value);

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

ux_device_stack_alternate_setting_set

Set current alternate setting for an interface value

Prototype

```
UINT  ux_device_stack_alternate_setting_set(ULONG interface_value,
                                             ULONG alternate_setting_value)
```

Description

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

The device stack will issue a UX_SLAVE_CLASS_COMMAND_CHANGE to the class that owns this interface to reflect the change of alternate setting.

Parameters

interface_value	Interface value for which the current alternate setting is set.
alternate_setting_value	The new alternate setting value.

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_INTERFACE_HANDLE_UNKNOWN	(0x52)	No interface attached.
UX_ERROR	(0xFF)	Wrong interface value.

Example

```
ULONG   interface_value;
ULONG   alternate_setting_value;

/* The following example illustrates this service. */
status = ux_device_stack_alternate_setting_set(interface_value,
                                               alternate_setting_value);

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

ux_device_stack_class_register

Register a new USB device class

Prototype

```
UINT  ux_device_stack_class_register(UCHAR *class_name,
        UINT (*class_entry_function)(struct UX_SLAVE_CLASS_COMMAND_STRUCT *),
        ULONG configuration_number,
        ULONG interface_number,
        VOID *parameter)
```

Description

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

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

Parameters

class_entry_function	The entry function of the class.
Configuration_number	The configuration number this class is attached to.
interface_number	The interface number this class is attached to.
parameter	A pointer to a class specific parameter list.

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_MEMORY_INSUFFICIENT	(0x52)	Not enough memory.
UX_THREAD_ERROR	(0xFF)	Cannot create a class thread.

Example

```
UINT   status;

/* The following example illustrates this service. */

/* Initialize the device storage class. The class is connected with
   interface 1 */
status =
ux_device_stack_class_register(_ux_system_slave_class_storage_name,
                               ux_device_class_storage_entry,
                               1, 1,  (VOID *)&parameter);

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

ux_device_stack_configuration_get

Get the current configuration

Prototype

```
UINT  ux_device_stack_configuration_get(VOID)
```

Description

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

Input Parameter

None

Return Value

UX_SUCCESS	(0x00)	The data transfer was completed.

Example

```
UINT   status;

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

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

ux_device_stack_configuration_set

Set the current configuration

Prototype

```
UINT  ux_device_stack_configuration_set(ULONG configuration_value)
```

Description

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

Input Parameter

configuration_value	The configuration value selected by the host.

Return Value

UX_SUCCESS	(0x00)	The data transfer was completed.

Example

```
ULONG   configuration_value;
UINT    status;

/* The following example illustrates this service. */
status = ux_device_stack_configuration_set(configuration_value);

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

ux_device_stack_descriptor_send

Send a descriptor to the host

Prototype

```
UINT  ux_device_stack_descriptor_send(ULONG descriptor_type,
                                      ULONG request_index, ULONG host_length)
```

Description

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

Parameters

descriptor_type The nature of the descriptor:

UX_DEVICE_DESCRIPTOR_ITEM
UX_CONFIGURATION_DESCRIPTOR_ITEM
UX_STRING_DESCRIPTOR_ITEM
UX_DEVICE_QUALIFIER_DESCRIPTOR_ITEM
UX_OTHER_SPEED_DESCRIPTOR_ITEM

request_index The index of the descriptor.
host_length The length required by the host.

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_ERROR	(0xFF)	The transfer was not completed.

Example

```
ULONG   descriptor_type;
ULONG   request_index;
ULONG   host_length;
UINT    status;

/* The following example illustrates this service. */
status = ux_device_stack_configuration_send(descriptor_type,
                                        request_index, host_length);

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

ux_device_stack_disconnect

Disconnect device stack

Prototype

```
UINT  ux_device_stack_disconnect(VOID)
```

Description

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

Input Parameter

None

Return Value

UX_SUCCESS	(0x00)	The device was disconnected.

Example

```
UINT   status;

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

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

ux_device_stack_endpoint_stall

Request endpoint Stall condition

Prototype

```
UINT  ux_device_stack_endpoint_stall(UX_SLAVE_ENDPOINT *endpoint)
```

Description

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

Input Parameter

endpoint	The endpoint on which the Stall condition is requested.

Return Value

UX_SUCCESS	(0x00)	This operation was successful.

Example

```
UINT   status;

/* The following example illustrates this service. */
status = ux_device_stack_endpoint_stall(endpoint);

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

ux_device_stack_host_wakeup

Wake up the host

Prototype

```
UINT  ux_device_stack_host_wakeup(VOID)
```

Description

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

Input Parameter

None

Return values

UX_SUCCESS	(0x00)	The call was successful.
UX_ERROR	(0xFF)	The call failed (the device was probably not in the suspended mode).

Example

```
UINT   status;

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

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

ux_device_stack_initialize

Initialize USB device stack

Prototype

```
UINT  ux_device_stack_initialize(UCHAR_PTR device_framework_high_speed,
                                 ULONG device_framework_length_high_speed,
                                 UCHAR_PTR device_framework_full_speed,
                                 ULONG device_framework_length_full_speed,
                                 UCHAR_PTR string_framework,
                                 ULONG string_framework_length,
                                 UCHAR_PTR language_id_framework,
                                 ULONG language_id_framework_length),
                                 UINT (*ux_system_slave_change_function)(ULONG)))
```

Description

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

Parameters

device_framework_high_speed	Pointer to the high speed framework.
device_framework_length_high_speed	Length of the high speed framework.
device_framework_full_speed	Pointer to the full speed framework.
device_framework_length_full_speed	Length of the full speed framework.
string_framework	Pointer to string framework.
string_framework_length	Length of string framework.
language_id_framework	Pointer to string language framework.
language_id_framework_length	Length of the string language framework.
ux_system_slave_change_function	Function to be called when the device state changes.

Return Values

UX_SUCCESS	(0x00)	This operation was successful.
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to initialize the stack.

Example

```
/* Example of a device framework */

#define DEVICE_FRAMEWORK_LENGTH_FULL_SPEED 50
UCHAR device_framework_full_speed[] = {

      /* Device descriptor */
            0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x08,
            0xec, 0x08, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00,
            0x00, 0x01,

      /* Configuration descriptor */
            0x09, 0x02, 0x20, 0x00, 0x01, 0x01, 0x00, 0xc0,
            0x32,

      /* Interface descriptor */
            0x09, 0x04, 0x00, 0x00, 0x02, 0x08, 0x06, 0x50,
            0x00,

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

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


#define DEVICE_FRAMEWORK_LENGTH_HIGH_SPEED 60
UCHAR device_framework_high_speed[] = {

      /* Device descriptor */
            0x12, 0x01, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40,
            0x0a, 0x07, 0x25, 0x40, 0x01, 0x00, 0x01, 0x02,
            0x03, 0x01,

      /* Device qualifier descriptor */
            0x0a, 0x06, 0x00, 0x02, 0x00, 0x00, 0x00, 0x40,
            0x01, 0x00,

      /* Configuration descriptor */
            0x09, 0x02, 0x20, 0x00, 0x01, 0x01, 0x00, 0xc0,
            0x32,

      /* Interface descriptor */
            0x09, 0x04, 0x00, 0x00, 0x02, 0x08, 0x06, 0x50,
            0x00,

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

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

```
/* String Device Framework:
Byte 0 and 1: Word containing the language ID: 0x0904 for US
Byte 2      : Byte containing the index of the descriptor
Byte 3      : Byte containing the length of the descriptor string
*/

#define STRING_FRAMEWORK_LENGTH 38
UCHAR string_framework[] = {

      /* Manufacturer string descriptor: Index 1 */
            0x09, 0x04, 0x01, 0x0c,
            0x45, 0x78, 0x70, 0x72,0x65, 0x73, 0x20, 0x4c,
            0x6f, 0x67, 0x69, 0x63,

      /* Product string descriptor: Index 2 */
            0x09, 0x04, 0x02, 0x0c,
            0x4D, 0x4C, 0x36, 0x39, 0x36, 0x35, 0x30, 0x30,
            0x20, 0x53, 0x44, 0x4B,

      /* Serial Number string descriptor: Index 3 */
            0x09, 0x04, 0x03, 0x04,
            0x30, 0x30, 0x30, 0x31
      };

      /* Multiple languages are supported on the device, to add
         a language besides English, the Unicode language code must
         be appended to the language_id_framework array and the length
         adjusted accordingly. */

      #define LANGUAGE_ID_FRAMEWORK_LENGTH 2
      UCHAR language_id_framework[] = {

      /* English. */
            0x09, 0x04
      };
```

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

UX_DEVICE_SUSPENDED
UX_DEVICE_RESUMED

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

```
UINT   status;

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

status = ux_device_stack_initialize(&device_framework_high_speed,
                                 DEVICE_FRAMEWORK_LENGTH_HIGH_SPEED,
```

```
                                    &device_framework_full_speed,
                                    DEVICE_FRAMEWORK_LENGTH_FULL_SPEED,
                                    &string_framework,
                                    STRING_FRAMEWORK_LENGTH,
                                    &language_id_framework,
                                    LANGUAGE_ID_FRAMEWORK_LENGTH,
                                    UX_NULL);

/* If status equals UX_SUCCESS, initialization was successful. */
```

ux_device_stack_interface_delete

Delete a stack interface

Prototype

```
UINT  ux_device_stack_interface_delete(UX_SLAVE_INTERFACE *interface)
```

Description

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

Input Parameter

interface	Pointer to the interface to remove.

Return Value

UX_SUCCESS	(0x00)	This operation was successful.

Example

```
UINT   status;

/* The following example illustrates this service. */
status = ux_device_stack_interface_delete(interface);

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

ux_device_stack_interface_get

Get the current interface value

Prototype

```
UINT  ux_device_stack_interface_get(UINT interface_value)
```

Description

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

Input Parameter

interface_value	Interface value to return.

Return Values

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0xFF)	No interface exists.

Example

```
ULONG   interface_value;
UINT    status;

/* The following example illustrates this service. */
status = ux_device_stack_interface_delete(interface_value);

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

ux_device_stack_interface_set

Change the alternate setting of the interface

Prototype

```
UINT  ux_device_stack_interface_set(UCHAR_PTR device_framework,
                                    ULONG device_framework_length,
                                    ULONG alternate_setting_value)
```

Description

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

Parameters

device_framework	Address of the device framework for this interface.
device_framework_length	Length of the device framework.
alternate_setting_value	Alternate setting value to be used by this interface.

Return Values

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0xFF)	No interface exists.

Example

```
UCHAR_PTR  device_framework
ULONG      device_framework_length;
ULONG      alternate_setting_value;
UINT       status;

/* The following example illustrates this service. */
status = ux_device_stack_interface_set(device_framework,
                                       device_framework_length,
                                       alternate_setting_value);

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

ux_device_stack_interface_start

Start search for a class to own an interface instance

Prototype

```
UINT  ux_device_stack_interface_start(UX_SLAVE_INTERFACE *interface)
```

Description

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

Input Parameter

interface	Pointer to the interface created.

Return Values

UX_SUCCESS	(0x00)	This operation was successful.
UX_NO_CLASS_MATCH	(0x57)	No class exists for this interface.

Example

```
UINT   status;

/* The following example illustrates this service. */
status = ux_device_stack_interface_start(interface);

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

ux_device_stack_transfer_request

Request to transfer data to the host

Prototype

```
UINT ux_device_stack_transfer_request(UX_SLAVE_TRANSFER *transfer_request,
                                      ULONG slave_length,
                                      ULONG host_length)
```

Description

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

Parameters

transfer_request	Pointer to the transfer request.
slave_length	Length the device wants to return.
host_length	Length the host has requested.

Return Values

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0xFF)	Transport error.

Example

```
UINT   status;

/* The following example illustrates how to transfer more data
   than an application requests. */
while(total_length)
{
   /* How much can we send in this transfer? */
   if (total_length > UX_SLAVE_CLASS_STORAGE_BUFFER_SIZE)
      transfer_length = UX_SLAVE_CLASS_STORAGE_BUFFER_SIZE;
   else
      transfer_length = total_length;

   /* Copy the Storage Buffer into the transfer request memory. */
   ux_utility_memory_copy(transfer_request ->
                                ux_slave_transfer_request_data_pointer,
                                media_memory, transfer_length);
   /* Send the data payload back to the caller. */
   status = ux_device_transfer_request(transfer_request,
                                     transfer_length, transfer_length);

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

   /* Update the buffer address.  */
   media_memory += transfer_length;

   /* Update the length to remain. */
   total_length -= transfer_length;
}
```

ux_device_stack_transfer_request_abort

Cancel a transfer request

Prototype

```
UINT  ux_device_stack_transfer_abort(UX_SLAVE_TRANSFER *transfer_request,
                                     ULONG completion_code)
```

Description

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

Parameters

transfer_request	Pointer to the transfer request.
completion_code	Error code to be returned to the class waiting for this transfer request to complete.

Return Value

UX_SUCCESS	(0x00)	This operation was successful.

Example

```
UINT   status;

/* The following example illustrates how to abort a transfer when
   a bus reset has been detected on the bus. */
status = ux_device_stack_transfer_abort(transfer_request,
                                        UX_TRANSFER_BUS_RESET);

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

Chapter 5: USBX Device Class Considerations

USB Device Storage Class

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

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

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

```
/* Store the number of LUN in this device storage instance: single LUN. */
storage_parameter.ux_slave_class_storage_parameter_number_lun = 1;

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

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_last_lba  =  0x1e6bfe;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_block_length  =  512;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_type  =  0;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_removable_flag =  0x80;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_read  =  tx_demo_thread_flash_media_read;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_write  =
     tx_demo_thread_flash_media_write;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_status  =
     tx_demo_thread_flash_media_status;


/* Initialize the device storage class. The class is connected with
   interface 0 */
status =
  ux_device_stack_class_register(_ux_system_slave_class_storage_name,
```

```
ux_device_class_storage_entry, ux_device_class_storage_thread,
0, (VOID *)&storage_parameter);
```

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

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

The prototypes for the read and write functions are:

```
UINT  media_read(UCHAR_PTR data_pointer, ULONG number_blocks, ULONG lba);
UINT  media_write(UCHAR_PTR data_pointer, ULONG number_blocks, ULONG lba);
```

Where:

data_pointer is the address of the buffer to be used for reading or writing
number_blocks is the number of sectors to read/write
lba is the sector address to read.

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

This function has the following prototype:

```
ULONG    tx_demo_thread_media_status(ULONG media_id);
```

The calling parameter media_id is not currently used and should always be 0. In the future it may be used to distinguish multiple storage devices or storage devices with multiple SCSI LUNs. This version of the storage class does not support multiple instances of the storage class or storage devices with multiple SCSI LUNs.

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

Bits 0-7	Sense_key
Bits 8-15	Additional Sense Code
Bits 16-23	Additional Sense Code Qualifier

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

Sense Key	ASC	ASCQ	Description
00	00	00	NO SENSE
01	17	01	RECOVERED DATA WITH RETRIES
01	18	00	RECOVERED DATA WITH ECC
02	04	01	LOGICAL DRIVE NOT READY - BECOMING READY

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

Multiple SCSI LUN

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

```
/* Store the number of LUN in this device storage instance.  */
storage_parameter.ux_slave_class_storage_parameter_number_lun = 2;

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

storage_parameter.ux_slave_class_storage_parameter_lun[0].
    ux_slave_class_storage_media_last_lba  =  0x7bbff;
```

```
storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_block_length  =  512;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_type  =  0;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_removable_flag  =  0x80;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_read  =  tx_demo_thread_flash_media_read;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_write  =
        tx_demo_thread_flash_media_write;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_status  =
        tx_demo_thread_flash_media_status;

/* Initialize the storage class LUN parameters for reading/writing to
   the CD-ROM.  */

storage_parameter.ux_slave_class_storage_parameter_lun[1].
  ux_slave_class_storage_media_last_lba  = 0x04caaf;

storage_parameter.ux_slave_class_storage_parameter_lun[1].
  ux_slave_class_storage_media_block_length  =  2048;

storage_parameter.ux_slave_class_storage_parameter_lun[1].
  ux_slave_class_storage_media_type  =  5;

storage_parameter.ux_slave_class_storage_parameter_lun[1].
  ux_slave_class_storage_media_removable_flag  =  0x80;

storage_parameter.ux_slave_class_storage_parameter_lun[1].
  ux_slave_class_storage_media_read  =  tx_demo_thread_cdrom_media_read;

storage_parameter.ux_slave_class_storage_parameter_lun[1].
  ux_slave_class_storage_media_write  =
        tx_demo_thread_cdrom_media_write;

storage_parameter.ux_slave_class_storage_parameter_lun[1].
  ux_slave_class_storage_media_status  =
        tx_demo_thread_cdrom_media_status;


/* Initialize the device storage class for a Flash disk and CD-ROM. The
class is connected with interface 0 */
status =
  ux_device_stack_class_register(_ux_system_slave_class_storage_name,
        ux_device_class_storage_entry, ux_device_class_storage_thread,0,
        (VOID *) &storage_parameter);
```

USB Device CDC-ACM Class

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

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

```
unsigned char device_framework_full_speed[] = {

    /* Device descriptor     18 bytes
       0x02 bDeviceClass:    CDC class code
       0x00 bDeviceSubclass: CDC class sub code
       0x00 bDeviceProtocol: CDC Device protocol

       idVendor & idProduct - http://www.linux-usb.org/usb.ids
    */
    0x12, 0x01, 0x10, 0x01,
    0xEF, 0x02, 0x01, 0x08,
    0x84, 0x84, 0x00, 0x00,
    0x00, 0x01, 0x01, 0x02,
    0x03, 0x01,

    /* Configuration 1 descriptor 9 bytes */
    0x09, 0x02, 0x4b, 0x00, 0x02, 0x01, 0x00,0x40, 0x00,

    /* Interface association descriptor. 8 bytes.  */
    0x08, 0x0b, 0x00, 0x02, 0x02, 0x02, 0x00, 0x00,

    /* Communication Class Interface Descriptor Requirement. 9 bytes.   */
    0x09, 0x04, 0x00, 0x00,0x01,0x02, 0x02, 0x01, 0x00,

    /* Header Functional Descriptor 5 bytes */
    0x05, 0x24, 0x00,0x10, 0x01,

    /* ACM Functional Descriptor 4 bytes */
    0x04, 0x24, 0x02,0x0f,

    /* Union Functional Descriptor 5 bytes */
    0x05, 0x24, 0x06, 0x00,                          /* Master interface */
    0x01,                                            /* Slave interface  */

    /* Call Management Functional Descriptor 5 bytes */
    0x05, 0x24, 0x01,0x03, 0x01,                     /* Data interface   */

    /* Endpoint 1 descriptor 7 bytes */
    0x07, 0x05, 0x83, 0x03,0x08, 0x00,  0xFF,

    /* Data Class Interface Descriptor Requirement 9 bytes */
    0x09, 0x04, 0x01, 0x00, 0x02,0x0A, 0x00, 0x00, 0x00,

    /* First alternate setting Endpoint 1 descriptor 7 bytes*/
    0x07, 0x05, 0x02,0x02,0x40, 0x00,0x00,
```

```
/* Endpoint 2 descriptor 7 bytes */
0x07, 0x05, 0x81,0x02,0x40, 0x00, 0x00,
```

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

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

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

```
/* Set the parameters for callback when insertion/extraction of a
   CDC device.  */
parameter.ux_slave_class_cdc_acm_instance_activate  =
                                          tx_demo_cdc_instance_activate;
parameter.ux_slave_class_cdc_acm_instance_deactivate =
                                          tx_demo_cdc_instance_deactivate;

/* Initialize the device cdc class. This class owns both interfaces
   starting with 0. */
status =
  ux_device_stack_class_register(_ux_system_slave_class_cdc_acm_name,
  ux_device_class_cdc_acm_entry, 1,0,  &parameter);
```

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

The CDC-ACM is based on a USB-IF standard and is automatically recognized by MAC Os and Linux operating systems. On Windows platforms, this class requires a .inf file. ExpressLogic supplies a template for the CDC-ACM class and it can be found in the usbx_windows_host_files directory. For more recent version of Windows the file CDC_ACM_Template_Win7_64bit.inf should be used. This file needs to be modified to reflect the PID/VID used by the device. The PID/VID will be specific to the final customer when the company and the product are registered with the USB-IF.
In the inf file, the fields to modify are located here:

```
[DeviceList]
%DESCRIPTION%=DriverInstall, USB\VID_8484&PID_0000

[DeviceList.NTamd64]
%DESCRIPTION%=DriverInstall, USB\VID_8484&PID_0000
```

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

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

The CDC-ACM class APIs are defined below:

ux_device_class_cdc_acm_read

Read from CDC-ACM pipe

Prototype

```
UINT ux_device_class_cdc_acm_read(UX_SLAVE_CLASS_CDC_ACM *cdc_acm,
               UCHAR *buffer, ULONG requested_length, ULONG *actual_length)
```

Description

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

Parameters

cdc_acm	Pointer to the cdc class instance.
buffer	Buffer address where data will be stored.
requested_length	The maximum length we expect
actual_length	The length returned into the buffer

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_CONFIGURATION_HANDLE_UNKNOWN	(0x51)	Device is no longer in the configured state
UX_TRANSFER_NO_ANSWER	(0x22)	No answer from device. The device was probably disconnected while the transfer was pending.

Example

```
/* Read from the CDC class.  */
status = ux_device_class_cdc_acm_read(cdc, buffer, UX_DEMO_BUFFER_SIZE,
                                       &actual_length);

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

ux_device_class_cdc_acm_write

Writing to a CDC-ACM pipe

Prototype

```
UINT ux_device_class_cdc_acm_write(UX_SLAVE_CLASS_CDC_ACM *cdc_acm,
                UCHAR *buffer, ULONG requested_length, ULONG *actual_length)
```

Description

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

Parameters

cdc_acm	Pointer to the cdc class instance.
buffer	Buffer address where data is stored.
requested_length	The length of the buffer to write
actual_length	The length returned into the buffer after write is performed

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_CONFIGURATION_HANDLE_UNKNOWN	(0x51)	Device is no longer in the configured state
UX_TRANSFER_NO_ANSWER	(0x22)	No answer from device. The device was probably disconnected while the transfer was pending.

Example

```
/* Write to the CDC class bulk in pipe.  */
status = ux_device_class_cdc_acm_write(cdc, buffer, UX_DEMO_BUFFER_SIZE,
                                        &actual_length);

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

USB Device CDC-ECM Class

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

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

```
unsigned char device_framework_full_speed[] = {

    /* Device descriptor     18 bytes
       0x02 bDeviceClass:    CDC_ECM class code
       0x06 bDeviceSubclass: CDC_ECM class sub code
       0x00 bDeviceProtocol: CDC_ECM Device protocol

       idVendor & idProduct - http://www.linux-usb.org/usb.ids
       0x3939 idVendor:     ExpressLogic test.
    */
    0x12, 0x01, 0x10, 0x01,
    0x02, 0x00, 0x00, 0x08,
    0x39, 0x39, 0x08, 0x08,
    0x00, 0x01, 0x01, 0x02, 03,0x01,

    /* Configuration 1 descriptor 9 bytes. */
    0x09, 0x02, 0x58, 0x00,0x02, 0x01, 0x00,0x40, 0x00,

    /* Interface association descriptor. 8 bytes.  */
    0x08, 0x0b, 0x00, 0x02, 0x02, 0x06, 0x00, 0x00,

    /* Communication Class Interface Descriptor Requirement 9 bytes */
    0x09, 0x04, 0x00, 0x00,0x01,0x02, 0x06, 0x00, 0x00,

    /* Header Functional Descriptor 5 bytes */
    0x05, 0x24, 0x00, 0x10, 0x01,

    /* ECM Functional Descriptor 13 bytes */
    0x0D, 0x24, 0x0F, 0x04,0x00, 0x00, 0x00, 0x00, 0xEA, 0x05, 0x00,
    0x00,0x00,

    /* Union Functional Descriptor 5 bytes */
    0x05, 0x24, 0x06, 0x00,0x01,

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

    /* Data Class Interface Descriptor  Alternate Setting 0, 0 endpoints. 9
          bytes */
    0x09, 0x04, 0x01, 0x00, 0x00, 0x0A, 0x00, 0x00, 0x00,

    /* Data Class Interface Descriptor Alternate Setting 1, 2 endpoints. 9
          bytes */
    0x09, 0x04, 0x01, 0x01, 0x02, 0x0A, 0x00, 0x00,0x00,
```

```
    /* First alternate setting Endpoint 1 descriptor 7 bytes */
    0x07, 0x05, 0x02, 0x02, 0x40, 0x00, 0x00,

    /* Endpoint 2 descriptor 7 bytes */
    0x07, 0x05, 0x81, 0x02, 0x40, 0x00,0x00
};
```

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

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

```
unsigned char string_framework[] = {

    /* Manufacturer string descriptor: Index 1 - "Express Logic" */
        0x09, 0x04, 0x01, 0x0c,
        0x45, 0x78, 0x70, 0x72, 0x65, 0x73, 0x20, 0x4c,
        0x6f, 0x67, 0x69, 0x63,

    /* Product string descriptor: Index 2 - "EL CDCECM Device" */
        0x09, 0x04, 0x02, 0x10,
        0x45, 0x4c, 0x20, 0x43, 0x44, 0x43, 0x45, 0x43,
        0x4d, 0x20, 0x44, 0x65, 0x76, 0x69, 0x63, 0x64,

    /* Serial Number string descriptor: Index 3 - "0001" */
        0x09, 0x04, 0x03, 0x04,
        0x30, 0x30, 0x30, 0x31,

    /* MAC Address string descriptor: Index 4 - "001E5841B879" */
        0x09, 0x04, 0x04, 0x0C,
        0x30, 0x30, 0x31, 0x45, 0x35, 0x38,
        0x34, 0x31, 0x42, 0x38, 0x37, 0x39
    };
```

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

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

```
    /* Set the parameters for callback when insertion/extraction of a CDC
    device.  Set to NULL.*/
    cdc_ecm_parameter.ux_slave_class_cdc_ecm_instance_activate   =  UX_NULL;
    cdc_ecm_parameter.ux_slave_class_cdc_ecm_instance_deactivate =  UX_NULL;


    /* Define a NODE ID.  */
    cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[0] =
                                                                      0x00;
    cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[1] =
                                                                      0x1e;
    cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[2] =
                                                                      0x58;
    cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[3] =
                                                                      0x41;
    cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[4] =
                                                                      0xb8;
    cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[5] =
                                                                      0x78;
```

```
/* Define a remote NODE ID.  */
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[0] =
                                                                      0x00;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[1] =
                                                                      0x1e;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[2] =
                                                                      0x58;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[3] =
                                                                      0x41;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[4] =
                                                                      0xb8;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_remote_node_id[5] =
                                                                      0x79;

/* Initialize the device cdc_ecm class. */
status =
  ux_device_stack_class_register(_ux_system_slave_class_cdc_ecm_name,
                                   ux_device_class_cdc_ecm_entry, 1,0,
                                   &cdc_ecm_parameter);
```

The initialization of this class expects the same function callback for activation and deactivation, although here as an exercise they are set to NULL so that no callback is performed.

The next parameters are for the definition of the node IDs. 2 Nodes are necessary for the CDC-ECM, a local node and a remote node. The remote node must be the same one as the one declared in the device framework string descriptor.
The CDC-ECM class has built-in APIs for transferring data both ways but they are hidden to the application as the user application will communicate with the USB Ethernet device through NetX.

The USBX CDC-ECM class is closely tied to ExpressLogic NetX Network stack.
An application using both NetX and USBX CDC-ECM class will activate the NetX network stack in its usual way but in addition needs to activate the USB network stack as follows:

```
/* Initialize the NetX system.  */
nx_system_initialize();

/* Perform the initialization of the network driver.  This will initialize
the USBX network layer.*/
ux_network_driver_init();
```

The USB network stack needs to be activated only once and is not specific to CDC-ECM but is required by any USB class that requires NetX services.

The CDC-ECM class will be recognized by MAC OS and Linux hosts. But there is no driver supplied by Microsoft Windows to recognize CDC-ECM natively. Some commercial products do exist for Windows platforms and they supply their own .inf file. This file will need to be modified the same way as the CDC-ACM inf template to match the PID/VID of the USB network device.

USB Device RNDIS Class

The USB device RNDIS class allows for a USB host system to communicate with the device as a ethernet device. This class is based on the Microsoft proprietary implementation and is specific to Windows platforms..

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

```
unsigned char device_framework_full_speed[] = {

    /* Device descriptor
       0x02 bDeviceClass:    RNDIS class code
       0x00 bDeviceSubclass: RNDIS class sub code
       0x00 bDeviceProtocol: RNDIS Device protocol

       idVendor & idProduct - http://www.linux-usb.org/usb.ids
       0x3939 idVendor:     ExpressLogic test.
    */
    0x12, 0x01, 0x10, 0x01, 0x02, 0x00, 0x00,
    0x40, 0xb4, 0x04, 0x27, 0x11, 0x00, 0x01,
    0x01, 0x02, 0x03, 0x01,

    /* Configuration 1 descriptor */
    0x09, 0x02, 0x4b, 0x00, 0x02, 0x01, 0x00, 0x40, 0x00,

    /* Interface association descriptor. 8 bytes.  */
    0x08, 0x0b, 0x00, 0x02, 0x02, 0x02, 0x00, 0x00,

    /* Communication Class Interface Descriptor Requirement */
    0x09, 0x04, 0x00, 0x00, 0x01, 0x02, 0x02, 0x00, 0x00,

    /* Header Functional Descriptor */
    0x05, 0x24, 0x00, 0x10, 0x01,

    /* ACM Functional Descriptor */
    0x04, 0x24, 0x02, 0x00,

    /* Union Functional Descriptor */
    0x05, 0x24, 0x06, 0x00, 0x01,

    /* Call Management Functional Descriptor */
    0x05, 0x24, 0x01, 0x00, 0x01,

    /* Endpoint 1 descriptor */
    0x07, 0x05, 0x83, 0x03, 0x08, 0x00, 0xFF,

    /* Data Class Interface Descriptor Requirement */
    0x09, 0x04, 0x01, 0x00, 0x02,  0x0A, 0x00, 0x00, 0x00,

    /* First alternate setting Endpoint 1 descriptor */
    0x07, 0x05, 0x02, 0x02, 0x40, 0x00, 0x00,

    /* Endpoint 2 descriptor */
```

```
    0x07, 0x05, 0x81, 0x02, 0x40, 0x00, 0x00
};
```

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

The activation of the RNDIS class is as follows:

```
  /* Set the parameters for callback when insertion/extraction of a CDC
     device.  Set to NULL.*/
    parameter.ux_slave_class_rndis_instance_activate   =  UX_NULL;
    parameter.ux_slave_class_rndis_instance_deactivate =  UX_NULL;

    /* Define a local NODE ID.  */
    parameter.ux_slave_class_rndis_parameter_local_node_id[0] = 0x00;
    parameter.ux_slave_class_rndis_parameter_local_node_id[1] = 0x1e;
    parameter.ux_slave_class_rndis_parameter_local_node_id[2] = 0x58;
    parameter.ux_slave_class_rndis_parameter_local_node_id[3] = 0x41;
    parameter.ux_slave_class_rndis_parameter_local_node_id[4] = 0xb8;
    parameter.ux_slave_class_rndis_parameter_local_node_id[5] = 0x78;

    /* Define a remote NODE ID.  */
    parameter.ux_slave_class_rndis_parameter_remote_node_id[0] = 0x00;
    parameter.ux_slave_class_rndis_parameter_remote_node_id[1] = 0x1e;
    parameter.ux_slave_class_rndis_parameter_remote_node_id[2] = 0x58;
    parameter.ux_slave_class_rndis_parameter_remote_node_id[3] = 0x41;
    parameter.ux_slave_class_rndis_parameter_remote_node_id[4] = 0xb8;
    parameter.ux_slave_class_rndis_parameter_remote_node_id[5] = 0x79;

    /* Set extra parameters used by the RNDIS query command with certain
       OIDs.  */
    parameter.ux_slave_class_rndis_parameter_vendor_id          =  0x04b4 ;
    parameter.ux_slave_class_rndis_parameter_driver_version     =  0x1127;
    ux_utility_memory_copy(parameter.
      ux_slave_class_rndis_parameter_vendor_description,
      "ELOGIC RNDIS", 12);

    /* Initialize the device rndis class. This class owns both interfaces. */
    status =
      ux_device_stack_class_register(_ux_system_slave_class_rndis_name,
                                     ux_device_class_rndis_entry, 1,0,
                                     &parameter);
```

As for the CDC-ECM, the RNDIS class requires 2 nodes, one local and one remote but there is no requirement to have a string descriptor describing the remote node.

However due to Microsoft proprietary messaging mechanism, some extra parameters are required. First the vendor ID has to be passed. Likewise, the driver version of the RNDIS. A vendor string must also be given.

The RNDIS class has built-in APIs for transferring data both ways but they are hidden to the application as the user application will communicate with the USB Ethernet device through NetX.

The USBX RNDIS class is closely tied to ExpressLogic NetX Network stack.
An application using both NetX and USBX RNDIS class will activate the NetX network stack in its usual way but in addition needs to activate the USB network stack as follows:

```
/* Initialize the NetX system.  */
nx_system_initialize();

/* Perform the initialization of the network driver.  This will
   initialize the USBX network layer.*/
ux_network_driver_init();
```

The USB network stack needs to be activated only once and is not specific to RNDIS but is required by any USB class that requires NetX services.

The RNDIS class will not be recognized by MAC OS and Linux hosts as it is specific to Microsoft operating systems. On windows platforms a .inf file needs to be present on the host that matches the device descriptor. ExpressLogic supplies a template for the RNDIS class and it can be found in the usbx_windows_host_files directory. For more recent version of Windows the file RNDIS_Template.inf should be used. This file needs to be modified to reflect the PID/VID used by the device. The PID/VID will be specific to the final customer when the company and the product are registered with the USB-IF. In the inf file, the fields to modify are located here:

```
[ELogicDevices]
%ELogicDevice%    = RNDIS, USB\VID_xxxx&PID_0000

[ELogicDevices.NT.5.1]
%ELogicDevice%    = RNDIS.NT.5.1, USB\VID_xxxx&PID_0000
```

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

When a USB host systems discovers the USB RNDIS device, it will mount a network interface and the device can be used with network protocol stack. See the host Operating System for reference.

USB Device DFU Class

The USB device DFU class allows for a USB host system to update the device firmware based on a host application. The DFU class is a USB-IF standard class.

USBX DFU class is relatively simple. It device descriptor does not require anything but a control endpoint. Most of the time, this class will be embedded into a USB composite device. The device can be anything such as a storage device or a comm device and the added DFU interface can inform the host that the device can have its firmware updated on the fly.

The DFU class works in 3 steps. First the device mounts as normal using the class exported. An application on the host (Windows or Linux) will exercise the DFU class and send a request to reset the device into DFU mode. The device will disappear from the bus for a short time (enough for the host and the device to detect a RESET sequence) and upon restarting, the device will be exclusively in DFU mode, waiting for the host application to send a firmware upgrade. When the firmware upgrade has been completed, the host application resets the device and upon re-enumeration the device will revert to its normal operation with the new firmware.

A DFU device framework will look like this:

```
UCHAR device_framework_full_speed[] = {

    /* Device descriptor */
        0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x40,
        0x99, 0x99, 0x00, 0x00, 0x00, 0x00, 0x01, 0x02,
        0x03, 0x01,

    /* Configuration descriptor */
        0x09, 0x02, 0x1b, 0x00, 0x01, 0x01, 0x00, 0xc0,
        0x32,

    /* Interface descriptor for DFU. */
        0x09, 0x04, 0x00, 0x00, 0x00, 0xFE, 0x01, 0x01,
        0x00,

    /* Functional descriptor for DFU. */
        0x09, 0x21, 0x0f, 0xE8, 0x03, 0x40, 0x00, 0x00,
        0x01,

    };
```

In this example, the DFU descriptor is not associated with any other classes. It has a simple interface descriptor and no other endpoints attached to it. There is a Functional descriptor that describes the specifics of the DFU capabilities of the device.

The description of the DFU capabilities are as follows:

Name	Offset	Size	type	Description
bmAttributes	2	1	Bit field	Bit 3: device will perform a bus detach-

				attach sequence when it receives a DFU_DETACH request. The host must not issue a USB Reset. (*bitWillDetach*) 0 = no 1 = yes Bit 2: device is able to communicate via USB after Manifestation phase. (*bitManifestationTolerant*) 0 = no, must see bus reset 1 = yes Bit 1: upload capable (*bitCanUpload*) 0 = no 1 = yes Bit 0: download capable (*bitCanDnload*) 0 = no 1 = yes
wDetachTimeOut	3	2	number	Time, in milliseconds, that the device will wait after receipt of the DFU_DETACH request. If this time elapses without a USB reset, then the device will terminate the Reconfiguration phase and revert back to normal operation. This represents the maximum time that the device can wait (depending on its timers, etc.). USBX sets this value to 1000 ms.
wTransferSize	5	2	number	Maximum number of bytes that the device can accept per control-write operation. USBX sets this value to 64 bytes.

The declaration of the DFU class is as follows:

```
  /* Store the DFU parameters.   */
    dfu_parameter.ux_slave_class_dfu_parameter_instance_activate =
                                             tx_demo_thread_dfu_activate;
    dfu_parameter.ux_slave_class_dfu_parameter_instance_deactivate =
                                             tx_demo_thread_dfu_deactivate
;
    dfu_parameter.ux_slave_class_dfu_parameter_read =
                                             tx_demo_thread_dfu_read;
    dfu_parameter.ux_slave_class_dfu_parameter_write =
                                             tx_demo_thread_dfu_write;
    dfu_parameter.ux_slave_class_dfu_parameter_get_status =
                                             tx_demo_thread_dfu_get_status
;
    dfu_parameter.ux_slave_class_dfu_parameter_notify =
```

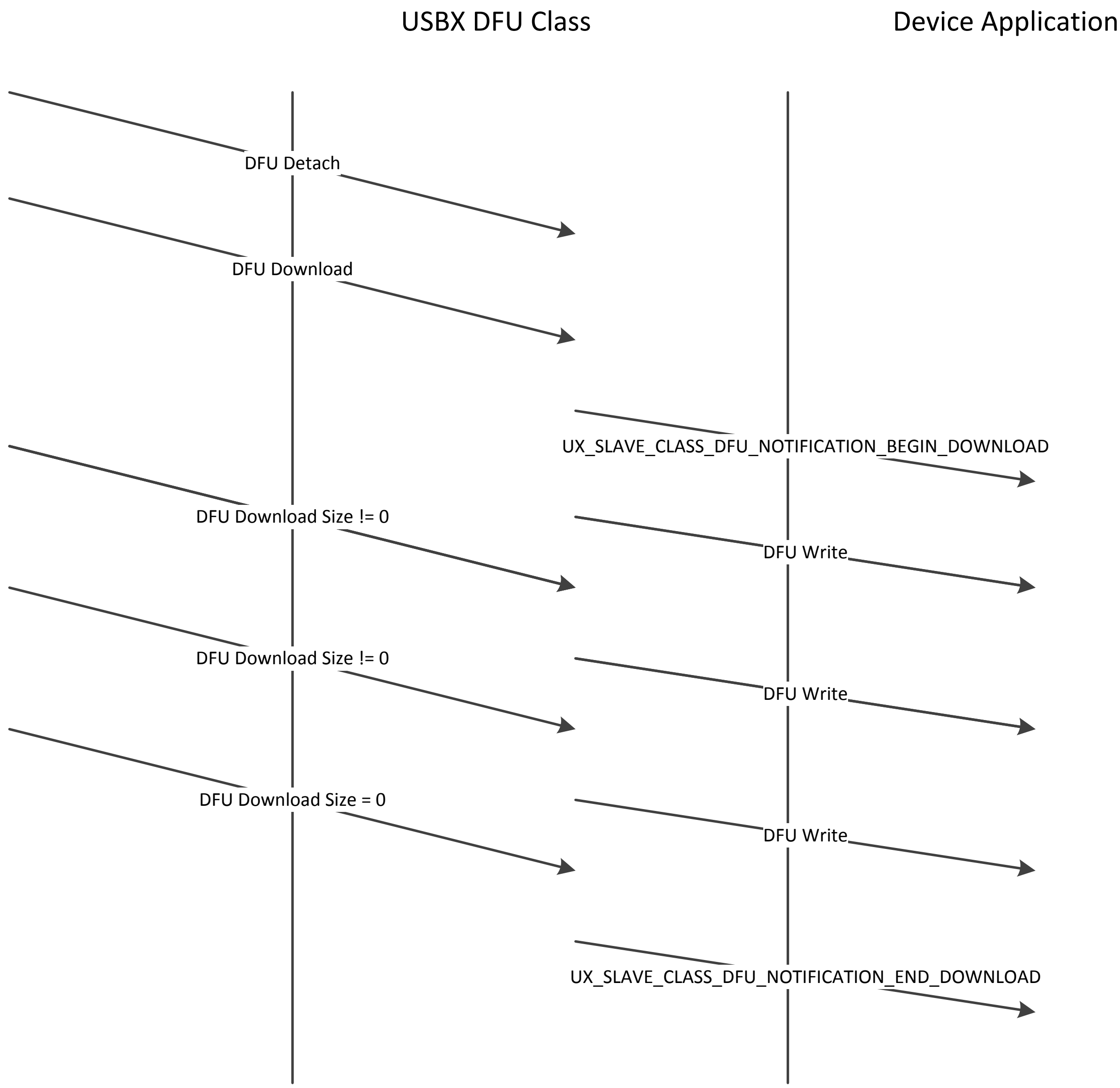
```
                                              tx_demo_thread_dfu_notify;
dfu_parameter.ux_slave_class_dfu_parameter_framework =
                                              device_framework_dfu;
dfu_parameter.ux_slave_class_dfu_parameter_framework_length =
                                              DEVICE_FRAMEWORK_LENGTH_DFU;

/* Initialize the device dfu class. The class is connected with interface
   1 on configuration 1. */
status =
  ux_device_stack_class_register(_ux_system_slave_class_dfu_name,
                                  ux_device_class_dfu_entry, 1, 0,
                                  (VOID *)&dfu_parameter);

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

The DFU class needs to work with a device firmware application specific to the target. Therefore it defines several call back to read and write blocks of firmware and to get status from the firmware update application. The DFU class also has a notify callback function to inform the application when a begin and end of transfer of the firmware occur.

Following is the description of a typical DFU application flow.



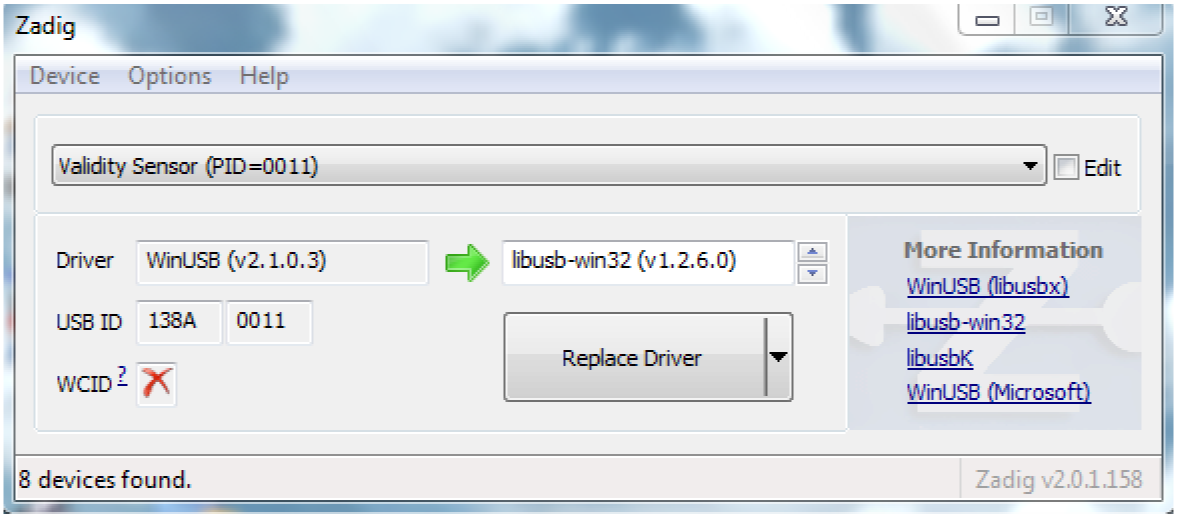
The major challenge of the DFU class is getting the right application on the host to perform the download the firmware. There is no application supplied by Microsoft or the USB-IF. Some shareware exist and they work reasonably well on Linux and to a lesser extent on Windows.

On Linux, one can use dfu-utils to be found here: http://wiki.openmoko.org/wiki/Dfu-util
A lot of information on dfu utils can also be found on this link:
http://www.libusb.org/wiki/windows_backend

The Linux implementation of DFU performs correctly the reset sequence between the host and the device and therefore the device does not need to do it. Linux can accept for the bmAttributes *bitWillDetach* to be 0. Windows on the other side requires the device to perform the reset.

On Windows, the USB registry must be able to associate the USB device with its PID/VID and the USB library which will in turn be used by the DFU application. This can be easily done with the free utility Zadig which can be found here: http://sourceforge.net/projects/libwdi/files/zadig/.

Running Zadig for the first time will show this screen:



From the device list, find your device and associate it with the libusb windows driver. This will bind the PID/VID of the device with the Windows USB library used by the DFU utilities.

To operate the DFU command, simply unpack the zipped dfu utilities into a directory, making sure the libusb dll is also present in the same directory. The DFU utilities must be run from a DOS box at the command line.

First, type the command **dfu-util –l** to determine whether the device is listed. If not, run Zadig to make sure the device is listed and associated with the USB library. You should see a screen as follows:

```
C:\usb specs\DFU\dfu-util-0.6>dfu-util -l
dfu-util 0.6

Copyright 2005-2008 Weston Schmidt, Harald Welte and OpenMoko Inc.
Copyright 2010-2012 Tormod Volden and Stefan Schmidt
This program is Free Software and has ABSOLUTELY NO WARRANTY

Found Runtime: [0a5c:21bc] devnum=0, cfg=1, intf=3, alt=0,
name="UNDEFINED"
```

The next step will be to prepare the file to be downloaded. The USBX DFU class does not perform any verification on this file and is agnostic of its internal format. This firmware file is very specific to the target but not to DFU nor to USBX.

Then the dfu-util can be instructed to send the file by typing the following command:

```
dfu-util –R –t 64  -D file_to_download.hex
```

The dfu-util should display the file download process until the firmware has been completely downloaded.

USB Device HID Class

The USB device HID class allows for a USB host system to connect to a HID device with specific HID client capabilities.

USBX HID device class is relatively simple compared to the host side. It is closely tied to the behavior of the device and its HID descriptor.

Any HID client requires first to define a HID device framework as the example below:

```
UCHAR device_framework_full_speed[] = {

    /* Device descriptor */
        0x12, 0x01, 0x10, 0x01, 0x00, 0x00, 0x00, 0x08,
        0x81, 0x0A, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x01,

    /* Configuration descriptor */
        0x09, 0x02, 0x22, 0x00, 0x01, 0x01, 0x00, 0xc0, 0x32,

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

    /* HID descriptor */
        0x09, 0x21, 0x10, 0x01, 0x21, 0x01, 0x22, 0x3f, 0x00,

    /* Endpoint descriptor (Interrupt) */
        0x07, 0x05, 0x81, 0x03, 0x08, 0x00, 0x08

    };
```

The HID framework contains an interface descriptor that describes the HID class and the HID device subclass. The HID interface can be a standalone class or part of a composite device. Follows is a HID descriptor and the interrupt endpoint.

The initialization of the HID class is as follow, using a USB keyboard as an example:

```
    /* Initialize the hid class parameters for a keyboard.  */
    hid_parameter.ux_device_class_hid_parameter_report_address =
                                                 hid_keyboard_report;
    hid_parameter.ux_device_class_hid_parameter_report_length  =
                                                 HID_KEYBOARD_REPORT_LENGTH;
    hid_parameter.ux_device_class_hid_parameter_callback  =
                                                 tx_demo_thread_hid_callback;

    /* Initialize the device hid class. The class is connected with interface
       0 */
    status =
      ux_device_stack_class_register(_ux_system_slave_class_hid_name,
                                    ux_device_class_hid_entry, 1,0,
                                    (VOID *)&hid_parameter);
    if (status!=UX_SUCCESS)
        return;
```

The application needs to pass to the HID class a HID report descriptor and its length. The report descriptor is a collection of items that describe the device. For more information on the HID grammar refer to the HID USB class specification.

In addition to the report descriptor, the application passes a call back when a HID event happens.

The USBX HID class supports the following standard HID commands from the host:

Command name	Value	Description
UX_DEVICE_CLASS_HID_COMMAND_GET_REPORT	0x01	Get a report from the device
UX_DEVICE_CLASS_HID_COMMAND_GET_IDLE	0x02	Get the idle frequency of the interrupt endpoint
UX_DEVICE_CLASS_HID_COMMAND_GET_PROTOCOL	0x03	Get the protocol running on the device
UX_DEVICE_CLASS_HID_COMMAND_SET_REPORT	0x09	Set a report to the device
UX_DEVICE_CLASS_HID_COMMAND_SET_IDLE	0x0a	Set the idle frequency of the interrupt endpoint
UX_DEVICE_CLASS_HID_COMMAND_SET_PROTOCOL	0x0b	Get the protocol running on the device

The Get and Set report are the most commonly used commands by HID to transfer data back and forth between the host and the device. Most commonly the host sends data on the control endpoint but can receive data either on the interrupt endpoint or by issuing a GET_REPORT command to fetch the data on the control endpoint.

The HID class can send data back to the host on the interrupt endpoint by using the ux_device_class_hid_event_set function. Its definition is below:

ux_device_class_hid_event_set

Setting an event to the HID class

Prototype

```
UINT  ux_device_class_hid_event_set(UX_SLAVE_CLASS_HID *hid,
                                     UX_SLAVE_CLASS_HID_EVENT *hid_event)
```

Description

This function is called when an application needs to send a HID event back to the host. The function is not blocking, it merely puts the report into a circular queue and returns to the application

Parameters

hid	Pointer to the hid class instance.
hid_event	Pointer to structure of the hid event.

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0x01)	Error on round robin queue

Example

```
/* Insert a key into the keyboard event. Length is fixed to 8. */
hid_event.ux_device_class_hid_event_length = 8;

/* First byte is a modifier byte.  */
hid_event.ux_device_class_hid_event_buffer[0] = 0;

/* Second byte is reserved. */
hid_event.ux_device_class_hid_event_buffer[1] = 0;

/* The 6 next bytes are keys. We only have one key here.  */
hid_event.ux_device_class_hid_event_buffer[2] = key;

/* Set the keyboard event.  */
ux_device_class_hid_event_set(hid, &hid_event);
```

The callback defined at the initialization of the HID class performs the opposite of sending an event. It gets as input the event sent by the host. The prototype of the callback is as follows:

hid_callback

Getting an event from the HID class

Prototype

```
UINT  hid_callback(UX_SLAVE_CLASS_HID *hid,
                   UX_SLAVE_CLASS_HID_EVENT *hid_event)
```

Description

This function is called when the host sends a HID report to the application.

Parameters

hid	Pointer to the hid class instance.
hid_event	Pointer to structure of the hid event.

Example

The following example shows how to interpret an event for a HID keyboard:

```
UINT    tx_demo_thread_hid_callback(UX_SLAVE_CLASS_HID *hid,
                              UX_SLAVE_CLASS_HID_EVENT *hid_event)
{

    /* There was an event.  Analyze it.  Is it NUM LOCK ? */
    if (hid_event -> ux_device_class_hid_event_buffer[0] &
                                                HID_NUM_LOCK_MASK)

        /* Set the Num lock flag.  */
        num_lock_flag = UX_TRUE;
    else

        /* Reset the Num lock flag.  */
        num_lock_flag = UX_FALSE;

    /* There was an event.  Analyze it.  Is it CAPS LOCK ? */
    if (hid_event -> ux_device_class_hid_event_buffer[0] &
                                                 HID_CAPS_LOCK_MASK)

        /* Set the Caps lock flag.  */
        caps_lock_flag = UX_TRUE;

    else

        /* Reset the Caps lock flag.  */
        caps_lock_flag = UX_FALSE;
}
```

USB Device PIMA Class (PTP Responder)

The USB device PIMA class allows for a USB host system (Initiator) to connect to a PIMA device (Resonder) to transfer media files. USBX Pima Class is conforming to the USB-IF PIMA 15740 class also known as PTP class (for Picture Transfer Protocol).

USBX device side PIMA class supports the following operations:

Operation code	Value	Description
UX_DEVICE_CLASS_PIMA_OC_GET_DEVICE_INFO	0x1001	Obtain the device supported operations and events
UX_DEVICE_CLASS_PIMA_OC_OPEN_SESSION	0x1002	Open a session between the host and the device
UX_DEVICE_CLASS_PIMA_OC_CLOSE_SESSION	0x1003	Close a session between the host and the device
UX_DEVICE_CLASS_PIMA_OC_GET_STORAGE_IDS	0x1004	Returns the storage ID for the device. USBX PIMA uses one storage ID only
UX_DEVICE_CLASS_PIMA_OC_GET_STORAGE_INFO	0x1005	Return information about the storage object such as max capacity and free space
UX_DEVICE_CLASS_PIMA_OC_GET_NUM_OBJECTS	0x1006	Return the number of objects contained in the storage device
UX_DEVICE_CLASS_PIMA_OC_GET_OBJECT_HANDLES	0x1007	Return an array of handles of the objects on the storage device
UX_DEVICE_CLASS_PIMA_OC_GET_OBJECT_INFO	0x1008	Return information about an object such as the name of the object, its creation date, modification date …
UX_DEVICE_CLASS_PIMA_OC_GET_OBJECT	0x1009	Return the data pertaining to a specific object.
UX_DEVICE_CLASS_PIMA_OC_GET_THUMB	0x100A	Send the thumbnail if available about an object
UX_DEVICE_CLASS_PIMA_OC_DELETE_OBJECT	0x100B	Delete an object on the media
UX_DEVICE_CLASS_PIMA_OC_SEND_OBJECT_INFO	0x100C	Send to the device information about an object for its creation on the media
UX_DEVICE_CLASS_PIMA_OC_SEND_OBJECT	0x100D	Send data for an object to the device
UX_DEVICE_CLASS_PIMA_OC_FORMAT_STORE	0x100F	Clean the device media
UX_DEVICE_CLASS_PIMA_OC_RESET_DEVICE	0x0110	Reset the target device

Operation Code	Value	Description
UX_DEVICE_CLASS_PIMA_EC_CANCEL_TRANSACTION	0x4001	Cancels the current transaction
UX_DEVICE_CLASS_PIMA_EC_OBJECT_ADDED	0x4002	An object has been added to the device media and can be retrieved by the host.
UX_DEVICE_CLASS_PIMA_EC_OBJECT_REMOVED	0x4003	An object has been deleted from the device media
UX_DEVICE_CLASS_PIMA_EC_STORE_ADDED	0x4004	A media has been added to the device
UX_DEVICE_CLASS_PIMA_EC_STORE_REMOVED	0x4005	A media has been deleted from the device
UX_DEVICE_CLASS_PIMA_EC_DEVICE_PROP_CHANGED	0x4006	Device properties have changed
UX_DEVICE_CLASS_PIMA_EC_OBJECT_INFO_CHANGED	0x4007	An object information has changed
UX_DEVICE_CLASS_PIMA_EC_DEVICE_INFO_CHANGE	0x4008	A device has changed
UX_DEVICE_CLASS_PIMA_EC_REQUEST_OBJECT_TRANSFER	0x4009	The device requests the transfer of an object from the host
UX_DEVICE_CLASS_PIMA_EC_STORE_FULL	0x400A	Device reports the media is full
UX_DEVICE_CLASS_PIMA_EC_DEVICE_RESET	0x400B	Device reports it was reset
UX_DEVICE_CLASS_PIMA_EC_STORAGE_INFO_CHANGED	0x400C	Storage information has changed on the device
UX_DEVICE_CLASS_PIMA_EC_CAPTURE_COMPLETE	0x400D	Capture is completed

The USBX PIMA device class uses a TX Thread to listen to PIMA commands from the host.

A PIMA command is composed of a command block, a data block and a status phase.

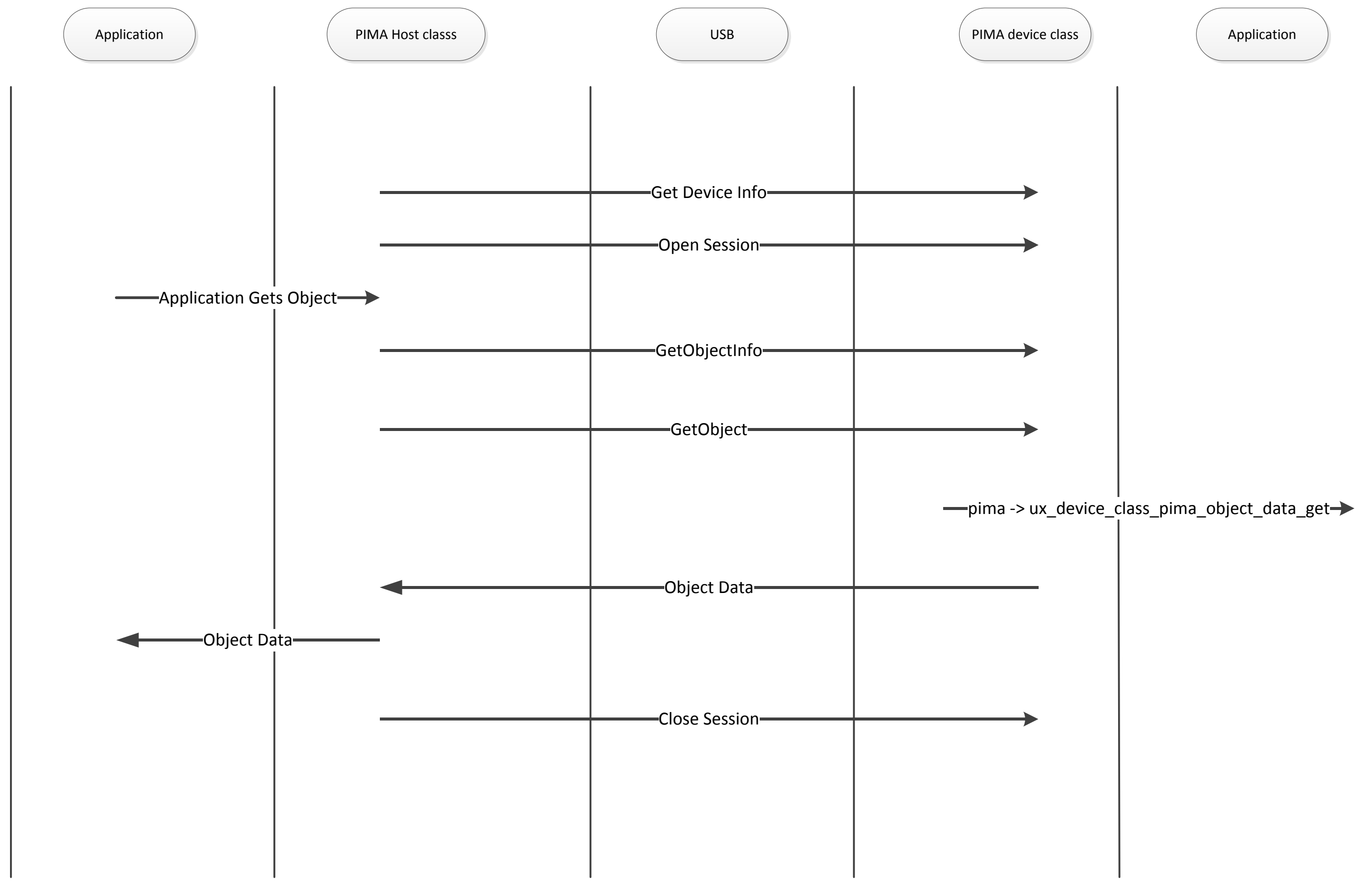
The function ux_device_class_pima_thread posts a request to the stack to receive a PIMA command from the host side. The PIMA command is decoded and verified for content. If the command block is valid, it branches to the appropriate command handler.

Most PIMA commands can only be executed when a session has been opened by the host. The only exception is the command `UX_DEVICE_CLASS_PIMA_OC_GET_DEVICE_INFO`. With USBX PIMA implementation, only one session can be opened between an Initiator and Responder at any time. All transactions within the single session are blocking and no new transaction can begin before the previous one completed.

PIMA transactions are composed of 3 phases, a command phase, an optional data phase and a response phase. If a data phase is present, it can only be in one direction.

The Initiator always determines the flow of the PIMA operations but the Responder can initiate events back to the Initiator to inform status changes that happened during a session.

The following diagram shows the transfer of a data object between the host and the PIMA device class:



Initialization of the PIMA device class

The PIMA device class needs some parameters supplied by the application during the initialization.

The following parameters describe the device and storage information:

- `ux_device_class_pima_manufacturer`
- `ux_device_class_pima_model`
- `ux_device_class_pima_device_version`
- `ux_device_class_pima_serial_number`

- `ux_device_class_pima_storage_id`

- ux_device_class_pima_storage_type
- ux_device_class_pima_storage_file_system_type
- ux_device_class_pima_storage_access_capability
- ux_device_class_pima_storage_max_capacity_low
- ux_device_class_pima_storage_max_capacity_high
- ux_device_class_pima_storage_free_space_low
- ux_device_class_pima_storage_free_space_high
- ux_device_class_pima_storage_free_space_image
- ux_device_class_pima_storage_description
- ux_device_class_pima_storage_volume_label

The PIMA class also requires the registration of callback into the application to inform the application of certain events or retrieve/store data from/to the local media. The callbacks are:

- ux_device_class_pima_object_number_get
- ux_device_class_pima_object_handles_get
- ux_device_class_pima_object_info_get
- ux_device_class_pima_object_data_get
- ux_device_class_pima_object_info_send
- ux_device_class_pima_object_data_send
- ux_device_class_pima_object_delete

The following example shows how to initialize the client side of PIMA. This example uses Pictbridge as a client for PIMA:

```
/* Initialize the first XML object valid in the pictbridge instance.
   Initialize the handle, type and file name.
   The storage handle and the object handle have a fixed value of 1 in our
   implementation. */
object_info = pictbridge -> ux_pictbridge_object_client;
object_info -> ux_device_class_pima_object_format =
                                      UX_DEVICE_CLASS_PIMA_OFC_SCRIPT;
object_info -> ux_device_class_pima_object_storage_id = 1;
object_info -> ux_device_class_pima_object_handle_id = 2;
ux_utility_string_to_unicode(_ux_pictbridge_ddiscovery_name,
                              object_info ->
                                ux_device_class_pima_object_filename);

/* Initialize the head and tail of the notification round robin buffers.
   At first, the head and tail are pointing to the beginning of the array.
*/
pictbridge -> ux_pictbridge_event_array_head =  pictbridge ->
                                      ux_pictbridge_event_array;
pictbridge -> ux_pictbridge_event_array_tail =  pictbridge ->
                                      ux_pictbridge_event_array;
pictbridge -> ux_pictbridge_event_array_end  =  pictbridge ->
                                      ux_pictbridge_event_array +
                                      UX_PICTBRIDGE_MAX_EVENT_NUMBER;

/* Initialialize the pima device parameter.  */
pictbridge -> ux_pictbridge_pima_parameter.
```

```
  ux_device_class_pima_parameter_manufacturer  = pictbridge ->
        ux_pictbridge_dpslocal.ux_pictbridge_devinfo_vendor_name;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_model = pictbridge ->
        ux_pictbridge_dpslocal.ux_pictbridge_devinfo_product_name;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_serial_number = pictbridge ->
        ux_pictbridge_dpslocal.ux_pictbridge_devinfo_serial_no;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_storage_id = 1;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_storage_type  =
  UX_DEVICE_CLASS_PIMA_STC_FIXED_RAM;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_storage_file_system_type =
  UX_DEVICE_CLASS_PIMA_FSTC_GENERIC_FLAT;
pictbridge -> ux_pictbridge_pima_parameter.
              ux_device_class_pima_parameter_storage_access_capability =
                    UX_DEVICE_CLASS_PIMA_AC_READ_WRITE;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_storage_max_capacity_low  =
  pictbridge -> ux_pictbridge_dpslocal.
  ux_pictbridge_devinfo_storage_size;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_storage_max_capacity_high     = 0;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_storage_free_space_low = pictbridge ->
  ux_pictbridge_dpslocal.ux_pictbridge_devinfo_storage_size;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_storage_free_space_high = 0;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_storage_free_space_image = 0;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_storage_description =
  _ux_pictbridge_volume_description;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_storage_volume_label =
  _ux_pictbridge_volume_label;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_object_number_get =
  ux_pictbridge_dpsclient_object_number_get;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_object_handles_get =
  ux_pictbridge_dpsclient_object_handles_get;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_object_info_get =
  ux_pictbridge_dpsclient_object_info_get;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_object_data_get =
  ux_pictbridge_dpsclient_object_data_get;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_object_info_send =
  ux_pictbridge_dpsclient_object_info_send;
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_object_data_send =
  ux_pictbridge_dpsclient_object_data_send;
pictbridge -> ux_pictbridge_pima_parameter.
```

```
  ux_device_class_pima_parameter_object_delete =
  ux_pictbridge_dpsclient_object_delete;

/* Store the instance owner.  */
pictbridge -> ux_pictbridge_pima_parameter.
  ux_device_class_pima_parameter_application = (VOID *) pictbridge;

/* Initialize the device pima class. The class is connected with interface
   0 */
status = ux_device_stack_class_register(_ux_system_slave_class_pima_name,
                                       ux_device_class_pima_entry, 1, 0,
                                       (VOID *)&pictbridge ->
                                          ux_pictbridge_pima_parameter);

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

ux_device_class_pima_object_number_get

Getting the object number from the application

Prototype

```
UINT  ux_device_class_pima_object_number_get(UX_SLAVE_CLASS_PIMA *pima,
                                             ULONG &object_numner)
```

Description

This function is called when the PIMA class needs to retrieve the number of objects in the local system and send it back to the host.

Parameters

pima	Pointer to the pima class instance.
object_number	Address of the number of objects to be returned.

Example

```
UINT  ux_pictbridge_dpsclient_object_number_get(UX_SLAVE_CLASS_PIMA *pima,
       ULONG *number_objects)
{

    /* We force the number of objects to be 1 only here. This will be the xml
       scripts. */
    *number_objects = 1;

    return(UX_SUCCESS);
}
```

ux_device_class_pima_object_handles_get

Return the object handle array

Prototype

```
UINT  ux_device_class_pima_object_handles_get(UX_SLAVE_CLASS_PIMA_STRUCT
                                *pima, ULONG object_handles_format_code,
                                ULONG object_handles_association,
                                ULONG *object_handles_array,
                                ULONG object_handles_max_number);
```

Description

This function is called when the PIMA class needs to retrieve the object handles array in the local system and send it back to the host.

Parameters

pima	Pointer to the pima class instance.
object_handles_format_code	Format code for the handles
object_handles_association	Object association code
object_handle_array	Address where to store the handles
object_handles_max_number	Maximum number of handles in the array

Example

```
UINT  ux_pictbridge_dpsclient_object_handles_get(UX_SLAVE_CLASS_PIMA *pima,
      ULONG object_handles_format_code, ULONG object_handles_association,
      ULONG *object_handles_array, ULONG object_handles_max_number)
{
UX_PICTBRIDGE                   *pictbridge;
UX_SLAVE_CLASS_PIMA_OBJECT      *object_info;

    /* Get the pointer to the Pictbridge instance.  */
    pictbridge = (UX_PICTBRIDGE *) pima -> ux_device_class_pima_application;

    /* Set the pima pointer to the pictbridge instance.  */
    pictbridge -> ux_pictbridge_pima = (VOID *) pima;

    /* We say we have one object but the caller might specify differnt format
       code and associations. */
    object_info = pictbridge -> ux_pictbridge_object_client;

   /* Insert in the array the number of found handles so far: 0.  */
    ux_utility_long_put((UCHAR *)object_handles_array, 0);

    /* Check the type demanded.  */
    if (object_handles_format_code == 0 || object_handles_format_code ==
                                  0xFFFFFFFF || object_info ->
                                  ux_device_class_pima_object_format ==
                                  object_handles_format_code)
```

```
    {

        /* Insert in the array the number of found handles.  This handle is
           for the client XML script. */
        ux_utility_long_put((UCHAR *)object_handles_array, 1);

         /* Adjust the array to point after the number of elements.  */
        object_handles_array++;

        /* We have a candicate.  Store the handle. */
        ux_utility_long_put((UCHAR *)object_handles_array, object_info ->
                                    ux_device_class_pima_object_handle_id);

    }

    return(UX_SUCCESS);
}
```

ux_device_class_pima_object_info_get

Return the object information

Prototype

```
UINT  ux_device_class_pima_object_info_get(struct
                     UX_SLAVE_CLASS_PIMA_STRUCT *pima, ULONG object_handle,
                     UX_SLAVE_CLASS_PIMA_OBJECT **object);
```

Description

This function is called when the PIMA class needs to retrieve the object handles array in the local system and send it back to the host.

Parameters

pima	Pointer to the pima class instance.
object_handles	Handle of the object
object	Object pointer address

Example

```
UINT  ux_pictbridge_dpsclient_object_info_get(UX_SLAVE_CLASS_PIMA *pima,
                  ULONG object_handle, UX_SLAVE_CLASS_PIMA_OBJECT **object)
{
UX_PICTBRIDGE                 *pictbridge;
UX_SLAVE_CLASS_PIMA_OBJECT    *object_info;

    /* Get the pointer to the Pictbridge instance.  */
    pictbridge = (UX_PICTBRIDGE *)pima -> ux_device_class_pima_application;

    /* Check the object handle. If this is handle 1 or 2 , we need to return
       the XML script object.
       If the handle is not 1 or 2, this is a JPEG picture or other object to
       be printed.  */
    if ((object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_HOST_RESPONSE) ||
              (object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_CLIENT_REQUEST))
    {
        /* Check what XML object is requested. It is either a request script
           or a response.  */
        if (object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_HOST_RESPONSE)
                  object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
                                                ux_pictbridge_object_host;
        else
            object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
                                              ux_pictbridge_object_client;
    }
    else
        /* Get the object info from the job info structure.  */
        object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
                          ux_pictbridge_jobinfo.ux_pictbridge_jobinfo_object;

    /* Return the pointer to this object.  */
    *object = object_info;
```

```
    /* We are done.  */
    return(UX_SUCCESS);
}
```

ux_device_class_pima_object_data_get

Return the object data

Prototype

```
UINT  ux_device_class_pima_object_info_get(UX_SLAVE_CLASS_PIMA *pima,
            ULONG object_handle, UCHAR *object_buffer, ULONG object_offset,
            ULONG object_length_requested, ULONG *object_actual_length)
```

Description

This function is called when the PIMA class needs to retrieve the object data in the local system and send it back to the host.

Parameters

pima	Pointer to the pima class instance.
object_handle	Handle of the object
object_buffer	Object buffer address
object_length_requested	Object data length requested by the client to the application
object_actual_length	Object data length returned by the application

Example

```
UINT  ux_pictbridge_dpsclient_object_data_get(UX_SLAVE_CLASS_PIMA *pima,
            ULONG object_handle, UCHAR *object_buffer, ULONG object_offset,
            ULONG object_length_requested, ULONG *object_actual_length)
{
UX_PICTBRIDGE                   *pictbridge;
UX_SLAVE_CLASS_PIMA_OBJECT      *object_info;
UCHAR                           *pima_object_buffer;
ULONG                           actual_length;
UINT                            status;

    /* Get the pointer to the Pictbridge instance.  */
    pictbridge = (UX_PICTBRIDGE *)pima -> ux_device_class_pima_application;

    /* Check the object handle. If this is handle 1 or 2 , we need to return
       the XML script object.
       If the handle is not 1 or 2, this is a JPEG picture or other object to
       be printed.  */
    if ((object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_HOST_RESPONSE) ||
              (object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_CLIENT_REQUEST))
    {

        /* Check what XML object is requested. It is either a request script
           or a response.  */
        if (object_handle == UX_PICTBRIDGE_OBJECT_HANDLE_HOST_RESPONSE)
            object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
                                                   ux_pictbridge_object_host;
```

```
        else
            object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
                                    ux_pictbridge_object_client;

        /* Is this the corrent handle ? */
        if (object_info ->  ux_device_class_pima_object_handle_id ==
                                                        object_handle)
        {

            /* Get the pointer to the object buffer.  */
            pima_object_buffer = object_info ->
                                  ux_device_class_pima_object_buffer;

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

            /* Update the length requested. for a demo, we do not do any
               checking.  */
            *object_actual_length = object_length_requested;

            /* What cycle are we in ? */
            if (pictbridge -> ux_pictbridge_host_client_state_machine &
                                UX_PICTBRIDGE_STATE_MACHINE_HOST_REQUEST)
            {
                /* Check if we are blocking for a client request.  */
                if (pictbridge -> ux_pictbridge_host_client_state_machine &
                        UX_PICTBRIDGE_STATE_MACHINE_CLIENT_REQUEST_PENDING)

                    /* Yes we are pending, send an event to release the
                       pending request.  */
                    ux_utility_event_flags_set(&pictbridge ->
                        ux_pictbridge_event_flags_group,
                        UX_PICTBRIDGE_EVENT_FLAG_STATE_MACHINE_READY, TX_OR);


                /* Since we are in host request, this indicates we are done
                   with the cycle.  */
                pictbridge -> ux_pictbridge_host_client_state_machine =
                                    UX_PICTBRIDGE_STATE_MACHINE_IDLE;

            }

            /* We have copied the requested data. Return OK.  */
            return(UX_SUCCESS);
        }
    }
    else
    {
        /* Get the object info from the job info structure.  */
        object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
                        ux_pictbridge_jobinfo.ux_pictbridge_jobinfo_object;

        /* Obtain the data from the application jobinfo callback.  */
        status = pictbridge ->
                        ux_pictbridge_jobinfo.
                        ux_pictbridge_jobinfo_object_data_read(pictbridge,
```

```
                        object_buffer, object_offset,
                        object_length_requested, &actual_length);

        /* Save the length returned.  */
        *object_actual_length =  actual_length;

        /* Return the application status.  */
        return(status);

    }
    /* Could not find the handle.  */
    return(UX_DEVICE_CLASS_PIMA_RC_INVALID_OBJECT_HANDLE);
}
```

ux_device_class_pima_object_info_send

Host sends the object information

Prototype

```
UINT  ux_device_class_pima_object_info_send(UX_SLAVE_CLASS_PIMA *pima,
                    UX_SLAVE_CLASS_PIMA_OBJECT *object, ULONG *object_handle)
```

Description

This function is called when the PIMA class needs to receive the object information in the local system for future storage.

Parameters

pima	Pointer to the pima class instance.
object	Pointer to the object
object_handle	Handle of the object

Example

```
UINT  ux_pictbridge_dpsclient_object_info_send(UX_SLAVE_CLASS_PIMA *pima,
            UX_SLAVE_CLASS_PIMA_OBJECT *object, ULONG *object_handle)
{
UX_PICTBRIDGE                   *pictbridge;
UX_SLAVE_CLASS_PIMA_OBJECT      *object_info;
UCHAR
string_discovery_name[UX_PICTBRIDGE_MAX_FILE_NAME_SIZE];

    /* Get the pointer to the Pictbridge instance.  */
    pictbridge = (UX_PICTBRIDGE *)pima -> ux_device_class_pima_application;

    /* We only have one object.  */
    object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
                                            ux_pictbridge_object_host;
    /* Copy the demanded object info set.  */
    ux_utility_memory_copy(object_info, object,
                        UX_SLAVE_CLASS_PIMA_OBJECT_DATA_LENGTH);

    /* Store the object handle.  In Pictbridge we only receive XML scripts so
       the handle is hardwired to 1.  */
    object_info -> ux_device_class_pima_object_handle_id = 1;
    *object_handle =  1;

    /* Check state machine. If we are in discovery pending mode, check file
       name of this object.  */
    if (pictbridge -> ux_pictbridge_discovery_state ==
                            UX_PICTBRIDGE_DPSCLIENT_DISCOVERY_PENDING)
    {

        /* We are in the discovery mode. Check for file name. It must match
           HDISCVRY.DPS in Unicode mode.  */
```

```
        /* Check if this is a script.  */
        if (object_info -> ux_device_class_pima_object_format ==
                        UX_DEVICE_CLASS_PIMA_OFC_SCRIPT)
        {

            /* Yes this is a script. We need to search for the HDISCVRY.DPS
               file name.  Get the file name in a ascii format.  */
            ux_utility_unicode_to_string(object_info ->
                                ux_device_class_pima_object_filename,
                                string_discovery_name);

            /* Now, compare it to the HDISCVRY.DPS file name.  Check length
               first.  */
            if (ux_utility_string_length_get(_ux_pictbridge_hdiscovery_name)
                  == ux_utility_string_length_get(string_discovery_name))
            {

                /* So far, the length of name of the files are the same.
                   Compare names now. */
                if(ux_utility_memory_compare(
                        _ux_pictbridge_hdiscovery_name,
                        string_discovery_name,
                        ux_utility_string_length_get(string_discovery_name))
                        ==  UX_SUCCESS)
                {

                    /* We are done with discovery of the printer. We can now
                       send notifications when the camera wants to print an
                       object.  */
                    pictbridge -> ux_pictbridge_discovery_state =
                                 UX_PICTBRIDGE_DPSCLIENT_DISCOVERY_COMPLETE;

                    /* Set an event flag if the application is listening.  */
                    ux_utility_event_flags_set(&pictbridge ->
                              ux_pictbridge_event_flags_group,
                              UX_PICTBRIDGE_EVENT_FLAG_DISCOVERY, TX_OR);

                    /* There is no object during th discovery cycle.  */
                    return(UX_SUCCESS);
                }
            }
        }
    }

    /* What cycle are we in ? */
    if (pictbridge -> ux_pictbridge_host_client_state_machine ==
                                   UX_PICTBRIDGE_STATE_MACHINE_IDLE)

        /* Since we are in idle state, we must have received a request from
           the host.  */
        pictbridge -> ux_pictbridge_host_client_state_machine =
                                UX_PICTBRIDGE_STATE_MACHINE_HOST_REQUEST;


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

ux_device_class_pima_object_data_send

Host sends the object data

Prototype

```
UINT  ux_device_class_pima_object_data_send(UX_SLAVE_CLASS_PIMA *pima,
                  ULONG object_handle, ULONG phase, UCHAR *object_buffer,
                  ULONG object_offset, ULONG object_length)
```

Description

This function is called when the PIMA class needs to receive the object data in the local system for storage.

Parameters

pima	Pointer to the pima class instance.
object_handle	Handle of the object
phase	phase of the transfer (active or complete)
object_buffer	Object buffer address
object_offset	Address of data
object_length	Object data length sent by application

Example

```
UINT  ux_pictbridge_dpsclient_object_data_send(UX_SLAVE_CLASS_PIMA *pima,
                                                ULONG object_handle,
                                                ULONG phase,
                                                UCHAR *object_buffer,
                                                ULONG object_offset,
                                                ULONG object_length)
{
UINT                            status;
UX_PICTBRIDGE                   *pictbridge;
UX_SLAVE_CLASS_PIMA_OBJECT      *object_info;
ULONG                           event_flag;
UCHAR                           *pima_object_buffer;


    /* Get the pointer to the Pictbridge instance.  */
    pictbridge = (UX_PICTBRIDGE *)pima -> ux_device_class_pima_application;

    /* Get the pointer to the pima object.  */
    object_info = (UX_SLAVE_CLASS_PIMA_OBJECT *) pictbridge ->
                                      ux_pictbridge_object_host;

    /* Is this the corrent handle ? */
    if (object_info -> ux_device_class_pima_object_handle_id ==
                                                  object_handle)
    {
```

```
        /* Get the pointer to the object buffer.  */
        pima_object_buffer = object_info ->
                                    ux_device_class_pima_object_buffer;

        /* Check the phase. We should wait for the object to be completed and
           the response sent back before parsing the object.  */
        if (phase == UX_DEVICE_CLASS_PIMA_OBJECT_TRANSFER_PHASE_ACTIVE)
        {

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

            /* Save the length of this object.  */
            object_info -> ux_device_class_pima_object_length =
                                                      object_length;

            /* We are not done yet.  */
            return(UX_SUCCESS);
        }
        else
        {

            /* Completion of transfer. We are done. */
            return(UX_SUCCESS);

        }
    }
}
```

ux_device_class_pima_object_delete

Delete a local object

Prototype

```
UINT  ux_device_class_pima_object_delete(UX_SLAVE_CLASS_PIMA *pima,
                                         ULONG object_handle)
```

Description

This function is called when the PIMA class needs to delete an object on the local storage.

Parameters

pima	Pointer to the pima class instance.
object_handle	Handle of the object

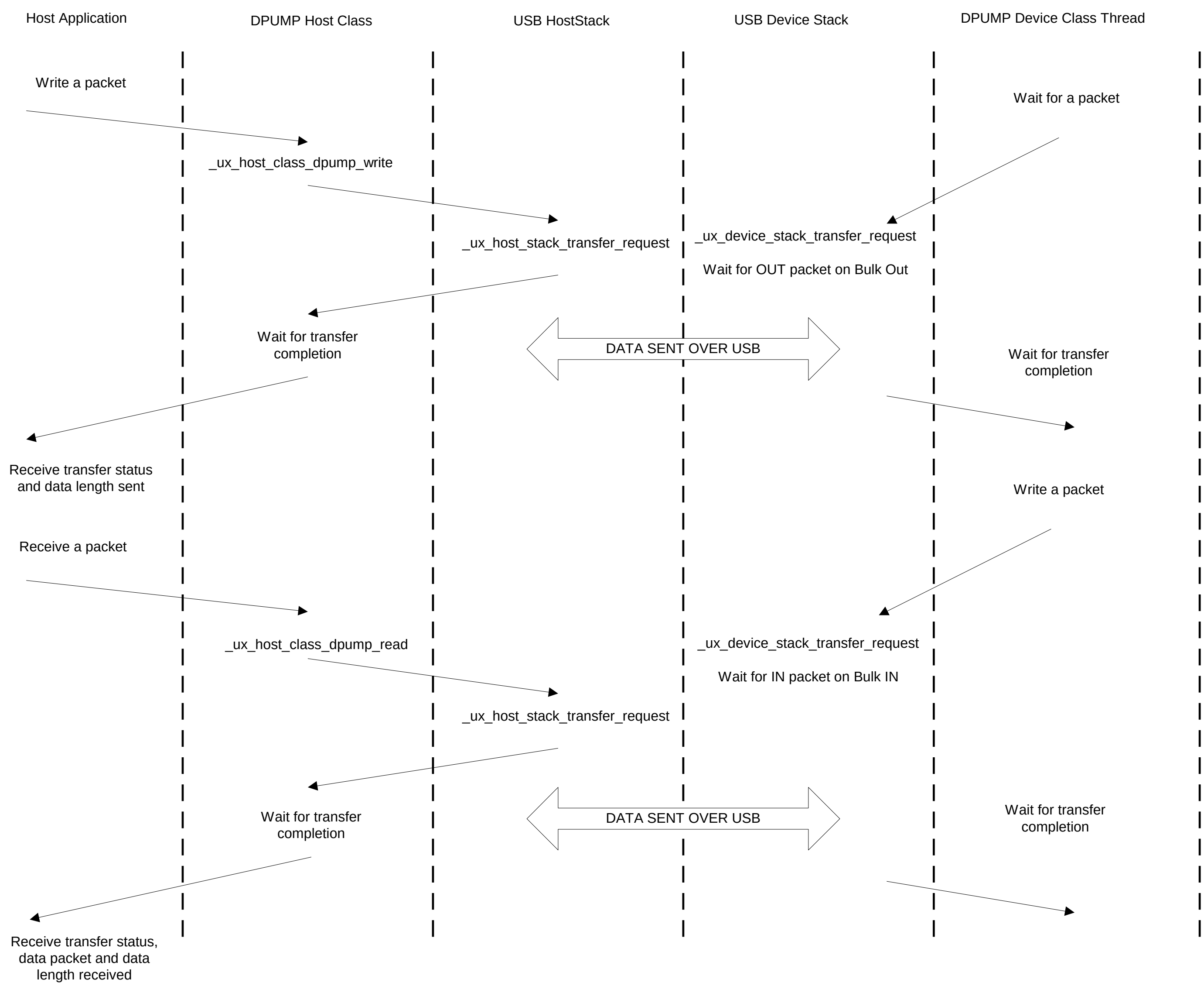
Example

```
UINT  ux_pictbridge_dpsclient_object_delete(UX_SLAVE_CLASS_PIMA *pima,
                                            ULONG object_handle)
{
      /* Delete the object pointer by the handle.  */
}
```

Chapter 6: USBX DPUMP Class Considerations

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

USB DPUMP flow chart:



USBX DPUMP Device Class

The device DPUMP class uses a thread which is started upon connection to the USB host. The thread waits for a packet coming on the Bulk Out endpoint. When a packet is received, it copies the content to the Bulk In endpoint buffer and posts a transaction on this endpoint, waiting for the host to issue a request to read from this endpoint. This provides a loopback mechanism between the Bulk Out and Bulk In endpoints.

Chapter 7: USBX Pictbridge implementation

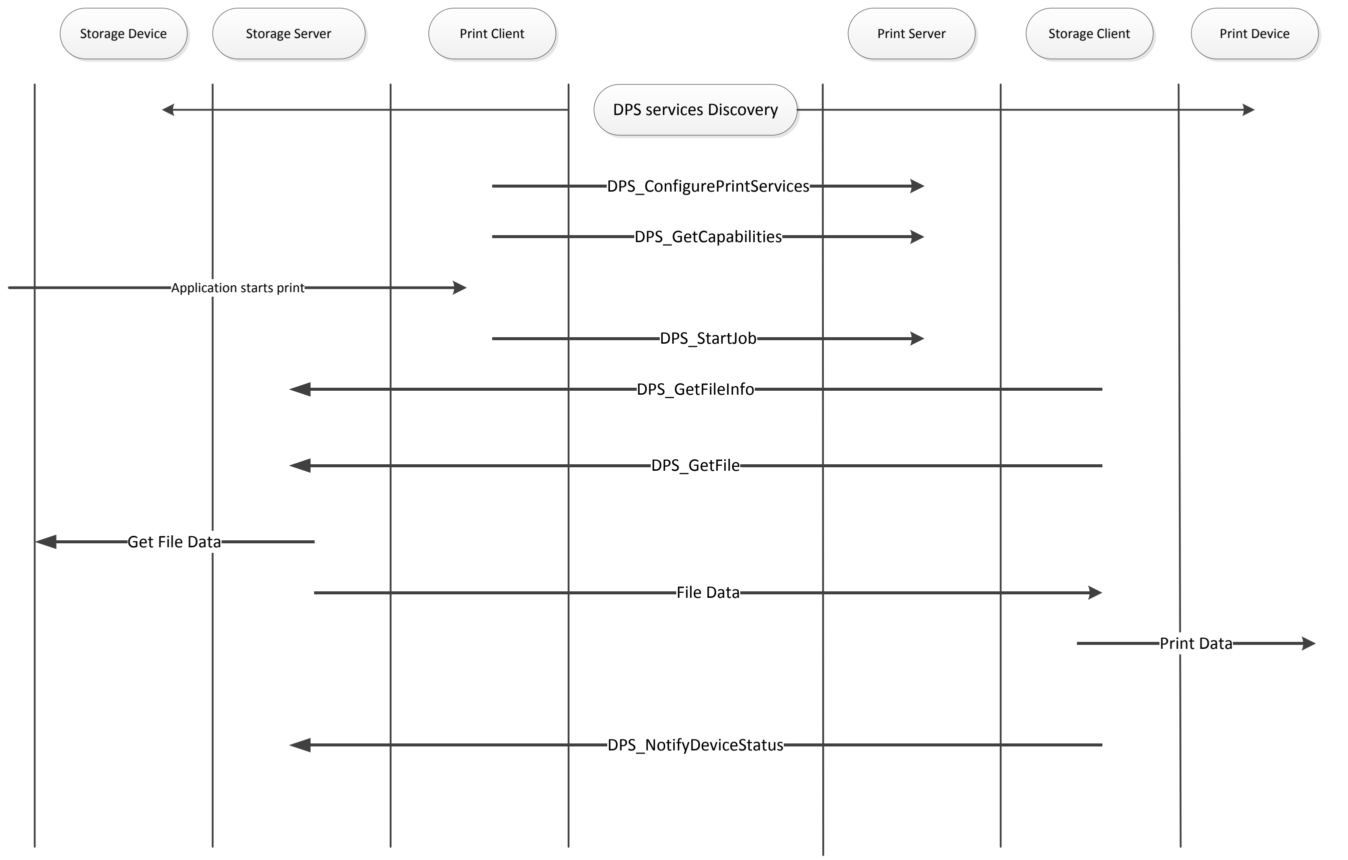
UBSX supports the full Pictbridge implementation both on the host and the device. Pictbridge sits on top of USBX PIMA class on both sides.

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

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

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

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



Pictbridge client implementation

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

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

```
UCHAR device_framework_full_speed[] =
{

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

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

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

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

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

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

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

};
```

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

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

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

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

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

The parameters passed to the pictbridge client are as follows:

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

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

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

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

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

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

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

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

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

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


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

/* Program the callback function for reading the object data.  */
```

```
jobinfo -> ux_pictbridge_jobinfo_object_data_read  =
                                                ux_demo_object_data_copy;

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

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

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

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

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

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

```
      jobinfo -> ux_pictbridge_jobinfo_object_data_read
```

The prototype of that function is defined as:

ux_pictbridge_jobinfo_object_data_read

Copying a block of data from user space for printing

Prototype

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

Description

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

Parameters

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

Return Value

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

Example

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

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

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

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

}
```

Pictbridge host implementation

The host implementation of Pictbridge is different from the client.

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

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

This class is the generic PTP layer sitting between the USB 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

Renesas Synergy^{TM} Platform

User’s Manual: Software

Publication Date: Rev.5.7 October, 2015

Renesas Synergy™ Platform

User's Manual: Software

Renesas Electronics Corporation