Microsoft

Azure RTOS
USBX Host Stack
User Guide

Published: February 2020

For the latest information, please see
azure.com/rtos

Part Number: 000-1010

Revision 6.0

Contents

About This Guide

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

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

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

Organization

Chapter 1 contains an introduction to USBX

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

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

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

Chapter 5 describes the APIs of the USBX Host classes

Chapter 6 describes the USBX CDC-ECM class

Chapter 1: Introduction to USBX

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

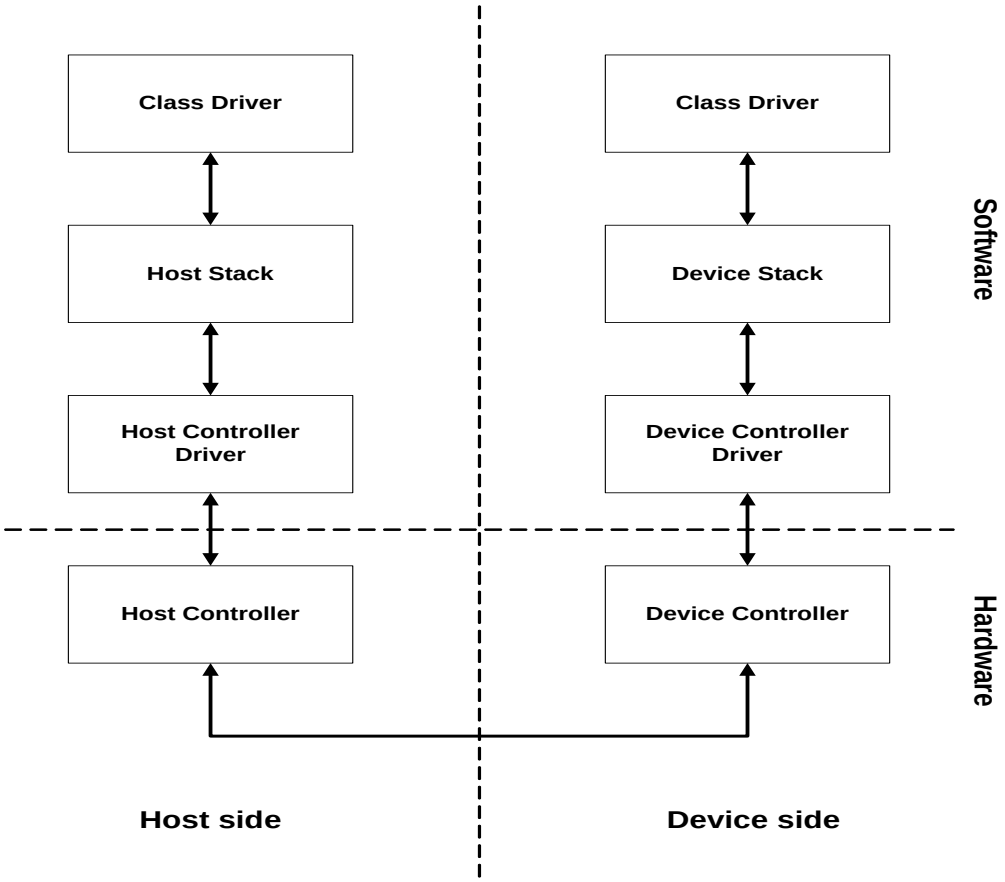
USBX features

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

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

- Controller layer
- Stack layer
- Class layer

The relationship between the USB layers is as follows:



Product Highlights

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

Powerful Services of USBX

Multiple Host Controller Support

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

USB Software Scheduler

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

Complete USB Device Framework Support

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

Easy-To-Use APIs

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

Chapter 2: USBX Installation

Host Considerations

Computer Type

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

Download Interfaces

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

Debugging Tools

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

Required Hard Disk Space

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

Target Considerations

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

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

Product Distribution

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

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

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

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

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

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

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

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

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

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

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

Step 8: Compile application source code and link with the USBX and ThreadX run time libraries (FileX and/or Netx may also be required if the USB storage

class and/or USB network classes are to be compiled in), ux.a (or ux.lib) and tx.a (or tx.lib). The resulting can be downloaded to the target and executed!

Configuration Options

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

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

UX_PERIODIC_RATE

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

UX_MAX_CLASS_DRIVER

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

UX_MAX_HCD

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

UX_MAX_DEVICES

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

UX_MAX_ED

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

UX_MAX_TD and UX_MAX_ISO_TD

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

UX_THREAD_STACK_SIZE

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

UX_HOST_ENUM_THREAD_STACK_SIZE

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

UX_HOST_HCD_THREAD_STACK_SIZE

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

UX_THREAD_PRIORITY_ENUM

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

UX_THREAD_PRIORITY_CLASS

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

UX_THREAD_PRIORITY_KEYBOARD

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

UX_THREAD_PRIORITY_HCD

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

UX_NO_TIME_SLICE

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

UX_MAX_HOST_LUN

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

UX_HOST_CLASS_STORAGE_INCLUDE_LEGACY_PROTOCOL_SUPPORT

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

UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE

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

UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE_REPORT_KEY_DOWN_ONLY

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

UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE_REPORT_LOCK_KEYS

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

UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE_REPORT_MODIFIER_KEYS

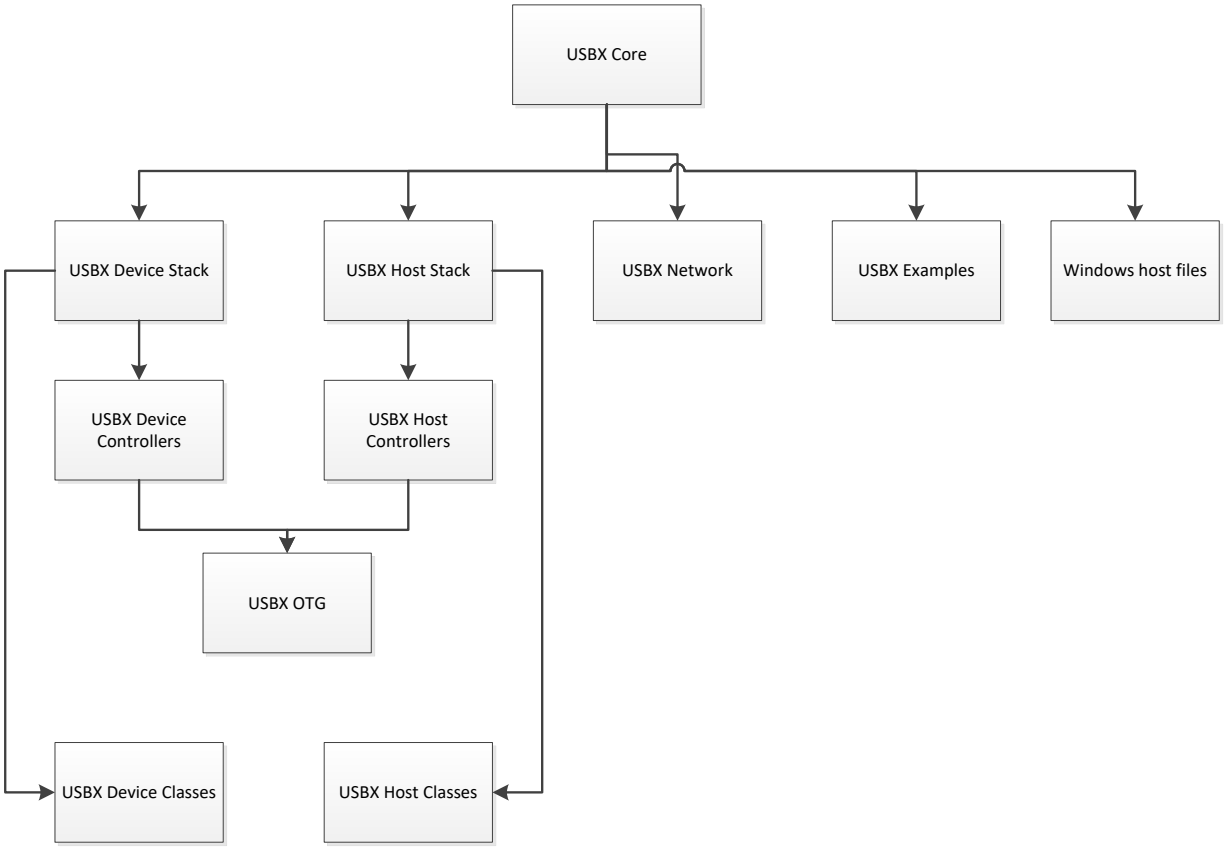
Only used if UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE is defined. If defined, causes ux_host_class_hid_keyboard_key_get to report modifier key (Ctrl/Alt/Shift/GUI) changes.

UX_HOST_CLASS_CDC_ECM_NX_PKPOOL_ENTRIES

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

Source Code Tree

The USBX files are provided in several directories.



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

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

Initialization of USBX resources

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

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

The prototype for the ux_system_initialize is as follows:

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

Input parameters:

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

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

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

Uninitialization of USBX resources

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

```
/* Unitialize USBX Resources */
ux_system_uninitialize();
```

The prototype for the ux_system_initialize is as follows:

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

Definition of USB Host Controllers

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

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

The ux_host_stack_hcd_register has the following prototype:

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

The ux_host_stack_hcd_register function has the following parameters:

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

In our previous example:

"ux_hcd_controller" is the name of the controller,
ux_hcd_controller_initialize is the initialization routine for the host controller,

0xd0000 is the address at which the host controller registers are visible in memory, and 0x0a is the IRQ used by the host controller.

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

```
UINT  status;

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

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

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

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

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

status = ux_host_class_register("ux_host_class_storage",
                                ux_host_class_storage_entry);

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

status = ux_host_class_register("ux_host_class_printer",
                                ux_host_class_printer_entry);

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

status = ux_host_class_register("ux_host_class_audio",
                                ux_host_class_audio_entry);

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

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

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

Definition of Host Classes

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

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

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

The function ux_host_class_register has the following prototype:

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

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

In the example of the HUB class initialization:

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

Troubleshooting

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

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

USBX Version ID

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

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

Chapter 3: Functional Components of USBX Host Stack

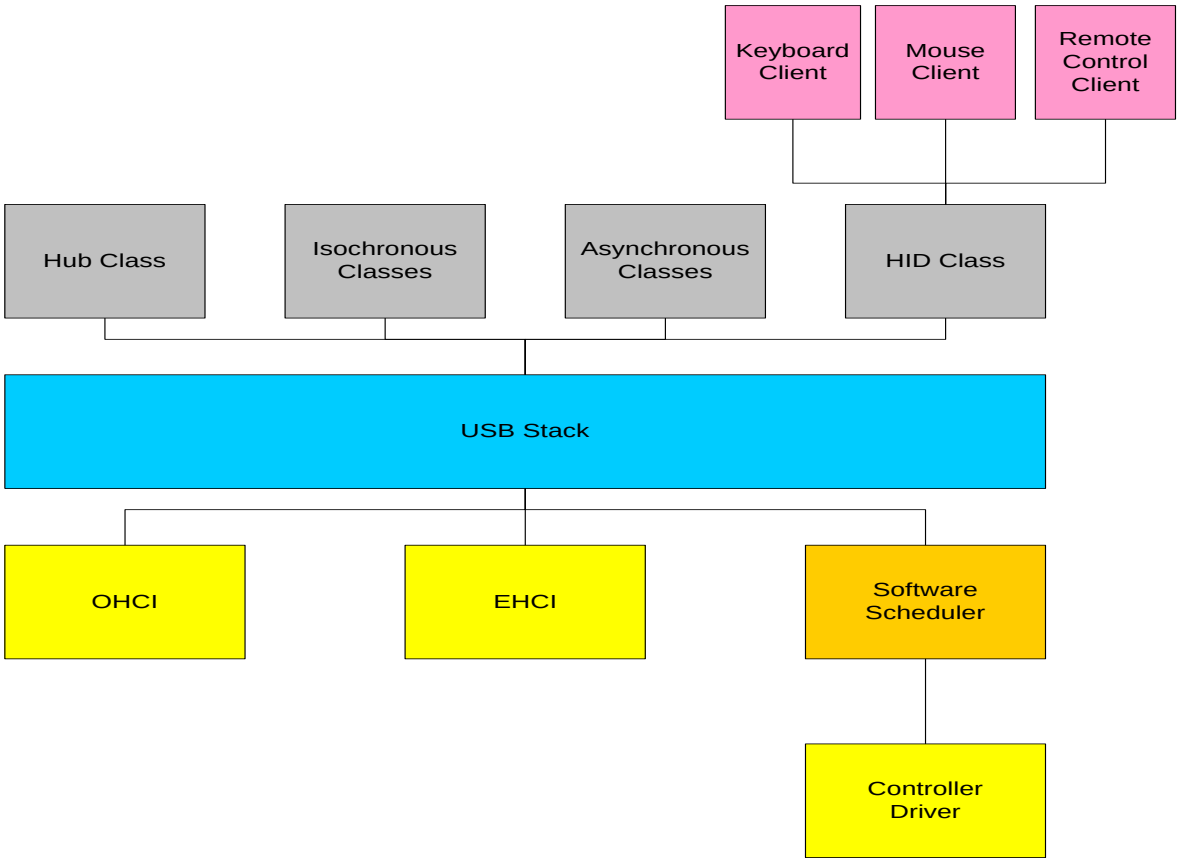
This chapter contains a description of the high performance USBX embedded USB host stack from a functional perspective.

Execution Overview:

USBX is composed of several components:

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

The following diagram illustrates the USBX host stack:



Initialization

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

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

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

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

Application Interface Calls

There are two levels of APIs in USBX:

USB Host Stack APIs
USB Host Class APIs

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

USB Host Stack APIs

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

USB Host Class APIs

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

Root Hub

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

Hub Class

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

USB Host Stack

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

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

Topology Manager

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

USB Class Binding

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

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

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

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

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

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

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

USBX APIs

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

Host Controller

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

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

The host controller is responsible for managing the following:

- Root Hub
- Power Management
- Endpoints
- Transfers

Root Hub

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

Power Management

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

Endpoints

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

Transfers

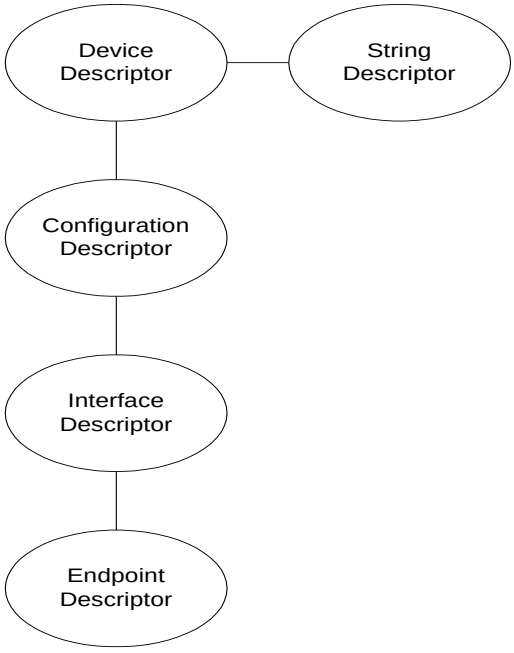
Transfer management provides for a class to perform a transaction on each of the endpoints that have been created. Each logical endpoint contains a component called TRANSFER REQUEST for USB transfer requests. The TRANSFER REQUEST is used by the stack to describe the transaction. This TRANSFER REQUEST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

USB Device Framework

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

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

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



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

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

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

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

Device Descriptors

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

Offset	Field	Size	Value	Description
0	BLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	DEVICE Descriptor Type
2	bcdUSB	2	BCD	USB Specification Release Number in Binary-Coded Decimal Example: 2.10 is equivalent to 0x210. This field identifies the release of the USB Specification that the device and its descriptors are compliant with.
4	bDeviceClass	1	Class	Class code (assigned by USB-IF). If this field is reset to 0, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and 0xFE, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to 0xFF, the device class is vendor specific.
5	bDeviceSubClass	1	SubClass	Subclass code (assigned by USB-IF). These codes are qualified by the value of the bDeviceClass field. If the bDeviceClass field is reset to 0, this field must also be reset to 0. If the bDeviceClass field is not set to 0xFF, all values are reserved for assignment by USB.
6	bDeviceProtocol	1	Protocol	Protocol code (assigned by USB-IF). These codes are qualified by the value of the bDeviceClass and the bDeviceSubClass fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to 0, the device does not use class specific protocols on a device basis. However, it may use class specific protocols on an interface basis. If this field is set to 0xFF, the device uses a vendor specific protocol on a device basis.
7	bMaxPacketSize0	1	Number	Maximum packet size for endpoint zero (only byte sizes of 8, 16, 32, or 64 are valid)
8	idVendor	2	ID	Vendor ID (assigned by USB-IF)

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

USBX defines a USB device descriptor as follows:

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

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

```
typedef struct UX_DEVICE_STRUCT
{

    ULONG           ux_device_handle;
    ULONG           ux_device_type;
    ULONG           ux_device_state;
    ULONG           ux_device_address;
    ULONG           ux_device_speed;
    ULONG           ux_device_port_location;
    ULONG           ux_device_max_power;
    ULONG           ux_device_power_source;
    UINT            ux_device_current_configuration;
    TX_SEMAPHORE    ux_device_protection_semaphore;
    struct UX_DEVICE_STRUCT     *ux_device_parent;
    struct UX_HOST_CLASS_STRUCT
                    *ux_device_class;
    VOID            *ux_device_class_instance;
    struct UX_HCD_STRUCT
                    *ux_device_hcd;
    struct UX_CONFIGURATION_STRUCT
                    *ux_device_first_configuration;
    struct UX_DEVICE_STRUCT
                    *ux_device_next_device;
    struct UX_DEVICE_DESCRIPTOR_STRUCT
                    ux_device_descriptor;
```

```
    struct UX_ENDPOINT_STRUCT
                    ux_device_control_endpoint;
    struct UX_HUB_TT_STRUCT
                    ux_device_hub_tt[UX_MAX_TT];
} UX_DEVICE;
```

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

ux_device_parent	Device container pointer of the parent of this device. If the pointer is null, the parent is the root hub of the controller.
ux_device_class	Pointer to the class type that owns this device.
ux_device_class_instance	Pointer to the instance of the class that owns this device.
ux_device_hcd	USB Host Controller Instance where this device is attached.
ux_device_first_configuration	Pointer to the first configuration container for this device.
ux_device_next_device	Pointer to the next device in the list of device on any of the buses detected by USBX.
ux_device_descriptor	USB device descriptor.
ux_device_control_endpoint	Descriptor of the default control endpoint used by this device.
ux_device_hub_tt	Array of Hub TTs for the device

Configuration Descriptors

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

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

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

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	1	Constant	CONFIGURATION
2	wTotalLength	2	Number	Total length of data returned for this configuration. Includes the combined length of all descriptors (configuration, interface, endpoint, and class or vendor specific) returned for this configuration.
4	bNumInterfaces	1	Number	Number of interfaces supported by this configuration.
5	bConfigurationValue	1	Number	Value to use as an argument to Set Configuration to select this configuration.
6	iConfiguration	1	Index	Index of string descriptor describing this configuration.
7	bMAttributes	1	Bitmap	Configuration characteristics D7 Bus Powered D6 Self Powered D5 Remote Wakeup D4..0 Reserved (reset to 0) A device configuration that uses power from the bus and a local source sets both D7 and D6. The actual power source at runtime may be determined using the Get Status device request. If a device configuration supports remote wakeup, D5 is set to 1.
8	MaxPower	1	mA	Maximum power consumption of USB device from the bus in this specific configuration when the device is fully operational. Expressed in 2 mA units (e.g., 50 = 100 mA). Note: A device configuration reports whether the configuration is bus-powered or self-powered. Device status reports whether the device is currently self-powered. If a device is disconnected from its external power source, it updates device status to indicate that it is no longer self-powered.

USBX defines a USB configuration descriptor as follows:

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

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

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

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

Interface Descriptors

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

descriptor in the data returned by the GET_DESCRIPTOR request for the specified configuration.

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

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

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

If a device configuration contains a single interface with two alternate settings, the GET_DESCRIPTOR request for the configuration would return the configuration descriptor, then the interface descriptor with the *bInterfaceNumber* and *bAlternateSetting* fields set to zero and then the endpoint descriptors for that setting, followed by another interface descriptor and its associated endpoint descriptors. The second interface descriptor's *bInterfaceNumber* field would also be set to zero, but the *bAlternateSetting* field of the second interface descriptor would be set to 1 indicating that this alternate setting belongs to the first interface.

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

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

The USB descriptor for the interface is as follows:

Offset	Field	Size	Value	Descriptor
0	bLength	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	1	Constant	INTERFACE Descriptor Type
2	bInterfaceNumber	1	Number	Number of interface. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration.
3	bAltenateSetting	1	Number	Value used to select alternate setting for the interface identified in the prior field.

4	bNumEndpoints	1	Number	Number of endpoints used by this interface (excluding endpoint zero). If this value is 0, this interface only uses endpoint zero.
5	bInterfaceClass	1	Class	Class code (assigned by USB) If this field is reset to 0, the interface does not belong to any USB specified device class. If this field is set to 0xFF, the interface class is vendor specific. All other values are reserved for assignment by USB.
6	bInterfaceSubClass	1	SubClass	Subclass code (assigned by USB). These codes are qualified by the value of the bInterfaceClass field. If the bInterfaceClass field is reset to 0, this field must also be reset to 0. If the bInterfaceClass field is not set to 0xFF, all values are reserved for assignment by USB.
7	bInterfaceProtocol	1	Protocol	Protocol code (assigned by USB). These codes are qualified by the value of the bInterfaceClass and the bInterfaceSubClass fields. If an interface supports class-specific requests, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to 0, the device does not use a class specific protocol on this interface. If this field is set to 0xFF, the device uses a vendor specific protocol for this interface.
8	iInterface	1	Index	Index of string descriptor describing this interface.

USBX defines a USB interface descriptor as follows:

```
typedef struct UX_INTERFACE_DESCRIPTOR_STRUCT
{

      UINT        bLength;
      UINT        bDescriptorType;
      UINT        bInterfaceNumber;
      UINT        bAlternateSetting;
      UINT        bNumEndpoints;
      UINT        bInterfaceClass;
      UINT        bInterfaceSubClass;
      UINT        bInterfaceProtocol;
      UINT        iInterface;
} UX_INTERFACE_DESCRIPTOR;
```

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

```
typedef struct UX_INTERFACE_STRUCT
{

    ULONG                                   ux_interface_handle;
    ULONG                                   ux_interface_state;
    ULONG                                   ux_interface_current_alternate_setting;
    struct UX_INTERFACE_DESCRIPTOR_STRUCT   ux_interface_descriptor;
    struct UX_HOST_CLASS_STRUCT             *ux_interface_class;
    VOID                                    *ux_interface_class_instance;
    struct UX_ENDPOINT_STRUCT               *ux_interface_first_endpoint;
    struct UX_INTERFACE_STRUCT              *ux_interface_next_interface;
    struct UX_CONFIGURATION_STRUCT          *ux_interface_configuration;

} UX_INTERFACE;
```

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

Endpoint Descriptors

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

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

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

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

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes.
1	bDescriptorType	1	Constant	ENDPOINT Descriptor Type.
2	bEndpointAddress	1	Endpoint	The address of the endpoint on the USB device described by this descriptor. The address is encoded as follows: Bit 3...0: The endpoint number Bit 6...4: Reserved, reset to zero Bit 7: Direction, ignored for control endpoints 0 = OUT endpoint 1 = IN endpoint
3	bmAttributes	1	Bitmap	This field describes the endpoint's attributes when it is configured using the bConfigurationValue. Bits 1..0: Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt If not an isochronous endpoint, bits 5..2 are reserved and must be set to zero. If isochronous, they are defined as follows: Bits 3..2: Synchronization Type 00 = No Synchronization 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4: Usage Type 00 = Data endpoint 01 = Feedback endpoint 10 = Implicit feedback data endpoint 11 = Reserved
4	wMaxPacketSize	2	Number	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. For isochronous endpoints, this value is used to reserve the bus time in the schedule, required for the per-(micro)frame data payloads. The pipe may, on an ongoing basis, actually use less bandwidth than that reserved. The device reports, if necessary, the actual bandwidth used via its normal, non-USB defined mechanisms. For all endpoints, bits 10..0 specify the maximum packet size (in bytes).

				For high-speed isochronous and interrupt endpoints: Bits 12..11 specify the number of additional transaction opportunities per microframe: 00 = None (1 transaction per microframe) 01 = 1 additional (2 per microframe) 10 = 2 additional (3 per microframe) 11 = Reserved Bits 15..13 are reserved and must be set to zero.
6	bInterval	1	Number	Number interval for polling endpoint for data transfers. Expressed in frames or microframes depending on the device operating speed (i.e., either 1 millisecond or 125 µs units). For full-/high-speed isochronous endpoints, this value must be in the range from 1 to 16. The *bInterval* value is used as the exponent for a 2bInterval-1 value; e.g., a *bInterval* of 4 means a period of 8 (24-1). For full-/low-speed interrupt endpoints, the value of this field may be from 1 to 255. For high-speed interrupt endpoints, the *bInterval* value is used as the exponent for a 2bInterval-1 value; e.g., a *bInterval* of 4 means a period of 8 (24-1). This value must be from 1 to 16. For high-speed bulk/control OUT endpoints, the *bInterval* must specify the maximum NAK rate of the endpoint. A value of 0 indicates the endpoint never NAKs. Other values indicate at most one NAK each *bInterval* number of microframes. This value must be in the range from 0 to 255.

USBX defines a USB endpoint descriptor as follows:

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

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

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

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

String descriptors

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

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

first byte of the descriptor.

The USB string descriptor 0 is encoded as follows:

Offset	Field	Size	Value	Description
0	bLength	1	N+2	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	STRING Descriptor Type
2	wLANGID[0]	2	Number	LANGID code 0
..	…]	..	…	…
N	wLANGID[n]	2	Number	LANGID code n

Other USB string descriptors are encoded as follows:

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	STRING Descriptor Type
2	bString	n	Number	UNICODE encoded string

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

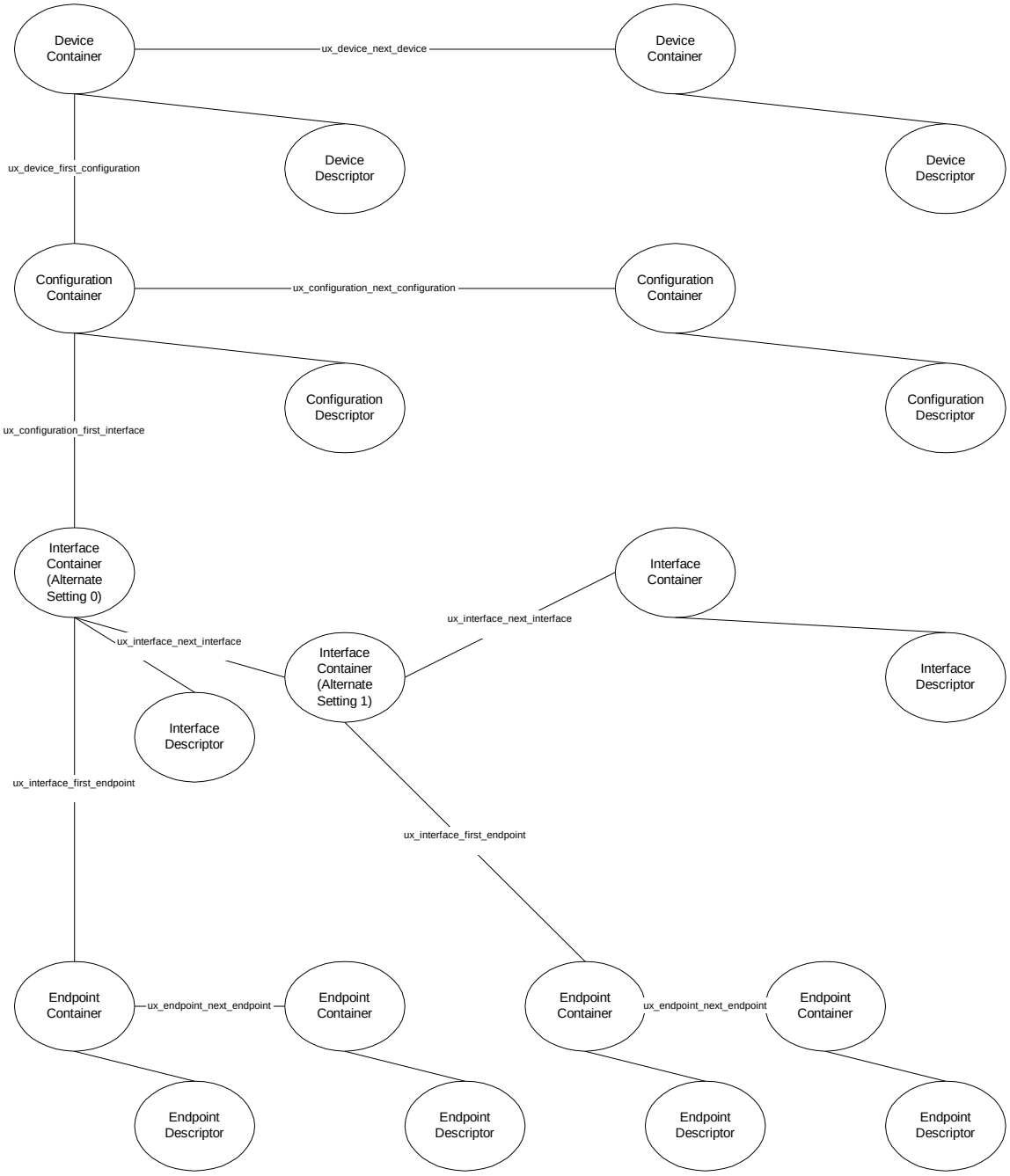
```
typedef struct UX_STRING_DESCRIPTOR_STRUCT
{

      UINT        bLength;
      UINT        bDescriptorType;
      USHORT      bString[1];
} UX_STRING_DESCRIPTOR;
```

Functional Descriptors

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

USBX Device Descriptor Framework in Memory

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


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

Chapter 4: Description of USBX Host Services

ux_host_stack_initialize

Initialize USBX for host operation

Prototype

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

Description

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

Input Parameter

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

Return Value

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

Example

```
UINT   status;

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

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

ux_host_stack_endpoint_transfer_abort

Abort all transactions attached to a transfer request for an endpoint

Prototype

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

Description

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

Input Parameter

endpoint	Pointer to an endpoint.

Return Values

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

Example

```
UX_HOST_CLASS_PRINTER   *printer;
UINT   status;

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

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

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

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

ux_host_stack_class_get

Get the pointer to a class container

Prototype

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

Description

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

Note: The C string of class_name must be NULL-terminated and the length of it (without the NULL-terminator itself) must be no larger than UX_MAX_CLASS_NAME_LENGTH.

Parameters

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

Return Values

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

Example

```
UX_HOST_CLASS   *printer_container;
UINT            status;

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

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

ux_host_stack_class_register

Register a USB class to the USB stack

Prototype

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

Description

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

UX_HOST_CLASS_COMMAND_QUERY
UX_HOST_CLASS_COMMAND_ACTIVATE
UX_HOST_CLASS_COMMAND_DESTROY

Note: The C string of class_name must be NULL-terminated and the length of it (without the NULL-terminator itself) must be no larger than UX_MAX_CLASS_NAME_LENGTH.

Parameters

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

Return Values

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

Example:

```
UINT   status;

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

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

ux_host_stack_class_instance_create

Create a new class instance for a class container

Prototype

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

Description

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

Parameters

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

Return Value

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

Example

```
UINT                    status;
UX_HOST_CLASS_PRINTER   *printer;

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

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

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

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

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

ux_host_stack_class_instance_destroy

Destroy a class instance for a class container

Prototype

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

Description

This function destroys a class instance for a class container.

Parameters

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

Return Values

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

Example

```
UINT                    status;
UX_HOST_CLASS_PRINTER   *printer;

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

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

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

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

ux_host_stack_class_instance_get

Get a class instance pointer for a specific class

Prototype

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

Description

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

Parameters

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

Return Values

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

Example

```
UINT                    status;
UX_HOST_CLASS_PRINTER   *printer;

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

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

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

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

ux_host_stack_device_configuration_get

Get a pointer to a configuration container

Prototype

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

Description

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

Parameters

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

Return Values

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

Example

```
UINT                    status;
UX_HOST_CLASS_PRINTER   *printer;

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

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

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

ux_host_stack_device_configuration_select

Select a specific configuration for a device

Prototype

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

Description

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

Parameters

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

Return Values

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

Example

```
UINT                    status;
UX_HOST_CLASS_PRINTER   *printer;

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

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

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

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

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

ux_host_stack_device_get

Get a pointer to a device container

Prototype

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

Description

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

Parameters

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

Return Values

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

Example

```
UINT   status;

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

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

ux_host_stack_interface_endpoint_get

Get an endpoint container

Prototype

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

Description

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

Parameters

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

Return Values

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

Example

```
UINT                    status;
UX_HOST_CLASS_PRINTER   *printer;

for(endpoint_index = 0;
    endpoint_index < printer -> printer_interface ->
                                  ux_interface_descriptor.bNumEndpoints;
    endpoint_index++)
{

      status = ux_host_stack_interface_endpoint_get
               (printer ->printer_interface, endpoint_index, &endpoint);

      if (status == UX_SUCCESS)
      {

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

ux_host_stack_hcd_register

Register a USB controller to the USB stack

Prototype

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

Description

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

Parameters

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

Return Values

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

Example

```
UINT   status;

/* Initialize a host controller mapped at address 0xd0000 and using
      IRQ 10. */
status = ux_host_stack_hcd_register("ux_hcd_controller",
                                    ux_hcd_controller_initialize,
                                    0xd0000, 0x0a);

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

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

ux_host_stack_configuration_interface_get

Get an interface container pointer

Prototype

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

Description

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

Parameters

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

Return Values

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

Example

```
UINT   status;

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

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

ux_host_stack_interface_setting_select

Select an alternate setting for an interface

Prototype

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

Description

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

Input Parameter

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

Return Values

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

Example

```
UINT   status;

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

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

ux_host_stack_transfer_request_abort

Abort a pending transfer request

Prototype

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

Description

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

Parameters

transfer request	Pointer to the transfer request to be aborted.

Return Values

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

Example

```
UINT   status;

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

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

ux_host_stack_transfer_request

Request a USB transfer

Prototype

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

Description

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

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

Input Parameter

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

Return Values

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

Example:

```
UINT   status;

/* Create a transfer request for the SET_CONFIGURATION request.
   No data for this request. */
transfer_request -> ux_transfer_request_requested_length = 0;
transfer_request -> ux_transfer_request_function =
```

```
                                                UX_SET_CONFIGURATION;
transfer_request -> ux_transfer_request_type =
                                               UX_REQUEST_OUT |
                                               UX_REQUEST_TYPE_STANDARD |
                                               UX_REQUEST_TARGET_DEVICE;
transfer_request -> ux_transfer_request_value =
                      (USHORT) configuration ->
                       ux_configuration_descriptor.bConfigurationValue;
transfer_request -> ux_transfer_request_index = 0;

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

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

Chapter 5: USBX Host Classes API

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

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

ux_host_class_hid_client_register

Register a HID client to the HID class

Prototype

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

Description

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

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

Parameters

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

Return Values

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

Example

```
UINT   status;

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

status =
ux_host_class_hid_client_register("ux_host_class_hid_client_mouse",
                                  ux_host_class_hid_mouse_entry);

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

ux_host_class_hid_report_callback_register

Register a callback from the HID class

Prototype

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

Description

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

Parameters

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

Return values

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

Example

```
UINT   status;

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

call_back.ux_host_class_hid_report_callback_id = 0;
call_back.ux_host_class_hid_report_callback_function =
                              ux_host_class_hid_mouse_callback;
call_back.ux_host_class_hid_report_callback_buffer = UX_NULL;
call_back.ux_host_class_hid_report_callback_flags =
                              UX_HOST_CLASS_HID_REPORT_INDIVIDUAL_USAGE;
call_back.ux_host_class_hid_report_callback_length = 0;

status = ux_host_class_hid_report_callback_register(hid, &call_back);

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

ux_host_class_hid_periodic_report_start

Start the periodic endpoint for a HID class instance

Prototype

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

Description

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

Input Parameter

hid Pointer to the HID class instance.

Return Values

UX_SUCCESS (0x00) Periodic reporting successfully started.

UX_HOST_CLASS_HID_PERIODIC_REPORT_ERROR (0x7A) Error in the periodic report.

UX_HOST_CLASS_INSTANCE_UNKNOWN (0x5b) HID class instance does not exist.

Example

```
UINT   status;

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

status = ux_host_class_hid_periodic_report_start(hid);

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

ux_host_class_hid_periodic_report_stop

Stop the periodic endpoint for a HID class instance

Prototype

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

Description

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

Input Parameter

hid Pointer to the HID class instance.

Return Values

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

.**Example**

```
UINT   status;

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

status = ux_host_class_hid_periodic_report_stop(hid);

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

ux_host_class_hid_report_get

Get a report from a HID class instance

Prototype

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

Description

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

Parameters

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

Return Values

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

.

Example

```
UX_HOST_CLASS_HID_CLIENT_REPORT    input_report;
UINT   status;

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

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

status = ux_host_class_hid_report_get(hid, &input_report);

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

ux_host_class_hid_report_set

Send a report

Prototype

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

Description

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

Parameters

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

Return Values

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

Example

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

UX_HOST_CLASS_HID_CLIENT_REPORT    input_report;

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

status = ux_host_class_hid_report_set(hid, &input_report);

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

ux_host_class_hid_mouse_buttons_get

Get mouse buttons

Prototype

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

Description

This function is used to get the mouse buttons

Parameters

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

Return Values

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

Example

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

UX_HOST_CLASS_HID_MOUSE *mouse_instance;
ULONG mouse_buttons;

status = ux_host_class_hid_mouse_button_get(mouse_instance,
&mouse_buttons);

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

ux_host_class_hid_mouse_position_get

Get mouse position

Prototype

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

Description

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

Parameters

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

Return Values

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

Example

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

UX_HOST_CLASS_HID_MOUSE *mouse_instance;
SLONG mouse_x_position;
SLONG mouse_y_position;

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

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

ux_host_class_hid_keyboard_key_get

Get keyboard key and state

Prototype

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

Description

This function is used to get the keyboard key and state

Parameters

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

Return Values

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

```
The keyboard state can have the following values:

UX_HID_KEYBOARD_STATE_KEY_UP                        0x10000

UX_HID_KEYBOARD_STATE_NUM_LOCK                      0x0001
UX_HID_KEYBOARD_STATE_CAPS_LOCK                     0x0002
UX_HID_KEYBOARD_STATE_SCROLL_LOCK                   0x0004
UX_HID_KEYBOARD_STATE_MASK_LOCK                     0x0007

UX_HID_KEYBOARD_STATE_LEFT_SHIFT                    0x0100
UX_HID_KEYBOARD_STATE_RIGHT_SHIFT                   0x0200
UX_HID_KEYBOARD_STATE_SHIFT                         0x0300

UX_HID_KEYBOARD_STATE_LEFT_ALT                      0x0400
UX_HID_KEYBOARD_STATE_RIGHT_ALT                     0x0800
UX_HID_KEYBOARD_STATE_ALT                           0x0a00

UX_HID_KEYBOARD_STATE_LEFT_CTRL                     0x1000
UX_HID_KEYBOARD_STATE_RIGHT_CTRL                    0x2000
UX_HID_KEYBOARD_STATE_CTRL                          0x3000

UX_HID_KEYBOARD_STATE_LEFT_GUI                      0x4000
UX_HID_KEYBOARD_STATE_RIGHT_GUI                     0x8000
```

UX_HID_KEYBOARD_STATE_GUI	0xa000

Example

```
    while (1)
    {
        /* Get a key/state from the keyboard.  */
        status = ux_host_class_hid_keyboard_key_get(keyboard,
&keyboard_char, &keyboard_state);

        /* Check if there is something.  */
        if (status == UX_SUCCESS)
        {
#ifdef UX_HOST_CLASS_HID_KEYBOARD_EVENTS_KEY_CHANGES_MODE
            if (keyboard_state & UX_HID_KEYBOARD_STATE_KEY_UP)
            {
                  /* The key was released. */
            }
            else
            {
                  /* The key was pressed. */
            }
#endif

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

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

        }

        tx_thread_sleep(10);
    }
```

ux_host_class_hid_keyboard_ioctl

Perform an IOCTL function to the HID keyboard

Prototype

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

Description

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

Parameters

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

Return Values

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

IOCTL functions

UX_HID_KEYBOARD_IOCTL_SET_LAYOUT
UX_HID_KEYBOARD_IOCTL_KEY_DECODING_ENABLE
UX_HID_KEYBOARD_IOCTL_KEY_DECODING_DISABLE

Example – change keyboard layout

```
UINT   status;

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

/* When the shift condition is not present and the raw key value is not
within the keypad value range, this array will be used to decode the
raw key value.  */
static UCHAR keyboard_layout_raw_to_unshifted_map[] =
{
   0,0,0,0,
   'a','b','c','d','e','f','g',
   'h','i','j','k','l','m','n',
   'o','p','q','r','s','t',
   'u','v','w','x','y','z',
   '1','2','3','4','5','6','7','8','9','0',
   0x0d,0x1b,0x08,0x07,0x20,'-','=','[',']',
   '\\','#',';',0x27,'`',',','.','/',0xf0,
   0xbb,0xbc,0xbd,0xbe,0xbf,0xc0,0xc1,0xc2,0xc3,0xc4,0xc5,0xc6,

0x00,0xf1,0x00,0xd2,0xc7,0xc9,0xd3,0xcf,0xd1,0xcd,0xcd,0xd0,0xc8,0xf2,
   '/','*','-','+',
   0x0d,'1','2','3','4','5','6','7','8','9','0','.','\\',0x00,0x00,'=',
   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
};

/* When the shift condition is present and the raw key value is not
within the keypad value range, this array will be used to decode the
raw key value.  */
static UCHAR keyboard_layout_raw_to_shifted_map[] =
{
   0,0,0,0,
   'A','B','C','D','E','F','G',
   'H','I','J','K','L','M','N',
   'O','P','Q','R','S','T',
   'U','V','W','X','Y','Z',
   '!','@','#','$','%','^','&','*','(',')',
   0x0d,0x1b,0x08,0x07,0x20,'_','+','{','}',
   '|','~',':','"','~','<','>','?',0xf0,
   0xbb,0xbc,0xbd,0xbe,0xbf,0xc0,0xc1,0xc2,0xc3,0xc4,0xc5,0xc6,

0x00,0xf1,0x00,0xd2,0xc7,0xc9,0xd3,0xcf,0xd1,0xcd,0xcd,0xd0,0xc8,0xf2,
   '/','*','-','+',
   0x0d,'1','2','3','4','5','6','7','8','9','0','.','\\',0x00,0x00,'=',
   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
};

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

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

/* When numlock is off and the raw key value is within the keypad value
range, this array will be used to decode the raw key value.  */
static UCHAR keyboard_layout_raw_to_numlock_off_map[] =
{
   '/','*','-','+',

0x0d,0xcf,0xd0,0xd1,0xcb,'5',0xcd,0xc7,0xc8,0xc9,0xd2,0xd3,'\\',0x00,0x
00,'=',
};

/* Specify the keyboard layout for USBX usage.  */
static UX_HOST_CLASS_HID_KEYBOARD_LAYOUT keyboard_layout =
{
    keyboard_layout_raw_to_shifted_map,
    keyboard_layout_raw_to_unshifted_map,
    keyboard_layout_raw_to_numlock_on_map,
    keyboard_layout_raw_to_numlock_off_map,

    /* The maximum raw key value. Values larger than this are
discarded. */
    UX_HID_KEYBOARD_KEYS_UPPER_RANGE,

    /* The raw key value for the letter 'a'.  */
    UX_HID_KEYBOARD_KEY_LETTER_A,

    /* The raw key value for the letter 'z'.  */
    UX_HID_KEYBOARD_KEY_LETTER_Z,

    /* The lower range raw key value for keypad keys - inclusive.  */
    UX_HID_KEYBOARD_KEYS_KEYPAD_LOWER_RANGE,

    /* The upper range raw key value for keypad keys.  */
    UX_HID_KEYBOARD_KEYS_KEYPAD_UPPER_RANGE
};


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

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

Example – disable keyboard key decode

```
UINT   status;

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

```
                UX_HID_KEYBOARD_IOCTL_DISABLE_KEYS_DECODE, UX_NULL);

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

ux_host_class_hid_remote_control_usage_get

Get remote control usage

Prototype

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

Description

This function is used to get the remote control usages.

Parameters

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

Return Values

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

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

Example

```
    /* Read usages and values as the user changes the vol/bass/treble
buttons on the speaker */
    while (remote_control != UX_NULL)
    {

        status =
ux_host_class_hid_remote_control_usage_get(remote_control, &usage,
&value);
        if (status == UX_SUCCESS)
        {

            /* We have something coming from the HID remote control, we
filter the usage here and only allow the volume usage which can be
VOLUME, VOLUME_INCREMENT or VOLUME_DECREMENT */
```

```
            switch(usage)
            {
                case UX_HOST_CLASS_HID_CONSUMER_VOLUME              :
                case UX_HOST_CLASS_HID_CONSUMER_VOLUME_INCREMENT    :
                case UX_HOST_CLASS_HID_CONSUMER_VOLUME_DECREMENT    :

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

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

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

                break;
            }
        }
        tx_thread_sleep(10);
    }
```

ux_host_class_cdc_acm_read

Read from the cdc_acm interface

Prototype

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

Description

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

Parameters

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

Return Values

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

Example

```
UINT   status;

/* The following example illustrates this service. */

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

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

ux_host_class_cdc_acm_write

Write to the cdc_acm interface

Prototype

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

Description

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

Parameters

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

Return Values

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

Example

```
UINT   status;

/* The following example illustrates this service. */

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

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

ux_host_class_cdc_acm_ioctl

Perform an IOCTL function to the cdc_acm interface

Prototype

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

Description

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

Parameters

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

Return Value

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

IOCTL functions:

UX_HOST_CLASS_CDC_ACM_IOCTL_SET_LINE_CODING
UX_HOST_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING
UX_HOST_CLASS_CDC_ACM_IOCTL_SET_LINE_STATE
UX_HOST_CLASS_CDC_ACM_IOCTL_SEND_BREAK
UX_HOST_CLASS_CDC_ACM_IOCTL_ABORT_IN_PIPE
UX_HOST_CLASS_CDC_ACM_IOCTL_ABORT_OUT_PIPE
UX_HOST_CLASS_CDC_ACM_IOCTL_NOTIFICATION_CALLBACK
UX_HOST_CLASS_CDC_ACM_IOCTL_GET_DEVICE_STATUS

Example

```
UINT   status;

/* The following example illustrates this service. */
status = ux_host_class_cdc_acm_ioctl(cdc_acm,
                            UX_HOST_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING,
                            (VOID *)&line_coding);

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

ux_host_class_cdc_acm_reception_start

Begins background reception of data from the device.

Prototype

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

Description

This function causes USBX to continuously read data from the device in the background. Upon completion of each transaction, the callback specified in **cdc_acm_reception** is invoked so the application may perform further processing of the transaction's data. Note that **ux_host_class_cdc_acm_read** must not be used while background reception is in use.

Parameters

cdc_acm Pointer to the cdc_acm class instance.

cdc_acm_reception Pointer to parameter that contains values defining behavior of background reception. The layout of this parameter follows:

```
typedef struct UX_HOST_CLASS_CDC_ACM_RECEPTION_STRUCT
{

    ULONG           ux_host_class_cdc_acm_reception_state;
    ULONG           ux_host_class_cdc_acm_reception_block_size;
    UCHAR           *ux_host_class_cdc_acm_reception_data_buffer;
    ULONG           ux_host_class_cdc_acm_reception_data_buffer_size;
    UCHAR           *ux_host_class_cdc_acm_reception_data_head;
    UCHAR           *ux_host_class_cdc_acm_reception_data_tail;
    VOID            (*ux_host_class_cdc_acm_reception_callback)(struct
                              UX_HOST_CLASS_CDC_ACM_STRUCT *cdc_acm,
                              UINT status, UCHAR *reception_buffer,
                              ULONG reception_size);

} UX_HOST_CLASS_CDC_ACM_RECEPTION;
```

Return Value

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

Example

```
UINT                                    status;
UX_HOST_CLASS_CDC_ACM_RECEPTION     cdc_acm_reception;


/* Setup the background reception parameter.  */

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

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

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

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

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

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

ux_host_class_cdc_acm_reception_stop

Stops background reception of packets.

Prototype

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

Description

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

Parameters

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

```
typedef struct UX_HOST_CLASS_CDC_ACM_RECEPTION_STRUCT
{

    ULONG          ux_host_class_cdc_acm_reception_state;
    ULONG          ux_host_class_cdc_acm_reception_block_size;
    UCHAR          *ux_host_class_cdc_acm_reception_data_buffer;
    ULONG          ux_host_class_cdc_acm_reception_data_buffer_size;
    UCHAR          *ux_host_class_cdc_acm_reception_data_head;
    UCHAR          *ux_host_class_cdc_acm_reception_data_tail;
    VOID           (*ux_host_class_cdc_acm_reception_callback)(struct
                              UX_HOST_CLASS_CDC_ACM_STRUCT *cdc_acm,
                              UINT status, UCHAR *reception_buffer,
                              ULONG reception_size);

} UX_HOST_CLASS_CDC_ACM_RECEPTION;
```

Return Value

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

Example

```
UINT                                status;
UX_HOST_CLASS_CDC_ACM_RECEPTION     cdc_acm_reception;


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

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

Chapter 6: USBX CDC-ECM Class Usage

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

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

```
 UINT  status;

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

    /* Initialize USBX. */
    ux_system_initialize(memory_pointer, UX_USBX_MEMORY_SIZE, UX_NULL, 0);

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

    /* Register cdc_ecm class.  */
    status = ux_host_stack_class_register(_ux_system_host_class_cdc_ecm_name,
ux_host_class_cdc_ecm_entry);

    /* Perform the initialization of the network driver.  */
    _ux_network_driver_init();

    /* Initialize NetX - refer to the NetX user guide. */
    ...

    /* Register the platform-specific USB controller.  */
    status =  ux_host_stack_hcd_register("controller_name", controller_entry,
param1, param2);

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

      /* Now wait for the link to be up.  */
    while (cdc_ecm -> ux_host_class_cdc_ecm_link_state !=
UX_HOST_CLASS_CDC_ECM_LINK_STATE_UP)
        tx_thread_sleep(10);

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

Index