Microsoft

Azure RTOS
USBX Device 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 describes the functional components of the USBX device stack

Chapter 4 describes the USBX device stack services

Chapter 5 describes each USBX device class including their APIs

Chapter 1: Introduction to USBX

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

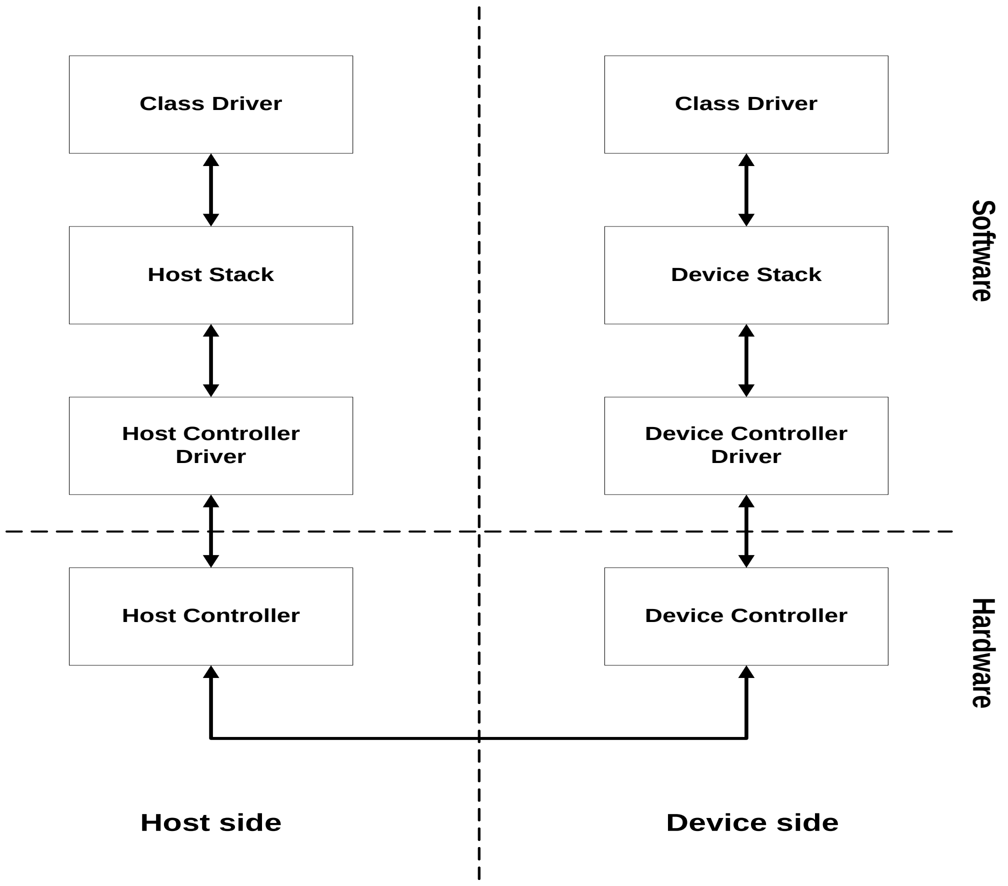
USBX features

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

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

- Controller layer
- Stack layer
- Class layer

The relationship between the USB layers is as follows:



Product Highlights

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

Powerful Services of USBX

.

Complete USB Device Framework Support

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

Easy-To-Use APIs

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

Chapter 2: USBX Installation

Host Considerations

Computer Type

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

Download Interfaces

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

Debugging Tools

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

Required Hard Disk Space

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

Target Considerations

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

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

Product Distribution

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

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

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

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

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

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

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

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

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

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

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

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

Configuration Options

There are several configuration options for building the USBX library. All options are located in the ***ux_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_THREAD_STACK_SIZE

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

UX_THREAD_PRIORITY_ENUM

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

UX_THREAD_PRIORITY_CLASS

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

UX_THREAD_PRIORITY_KEYBOARD

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

UX_THREAD_PRIORITY_DCD

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

UX_NO_TIME_SLICE

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

UX_MAX_SLAVE_CLASS_DRIVER

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

UX_MAX_SLAVE_LUN

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

UX_SLAVE_CLASS_STORAGE_INCLUDE_MMC

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

UX_DEVICE_CLASS_CDC_ECM_NX_PKPOOL_ENTRIES

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

UX_SLAVE_REQUEST_CONTROL_MAX_LENGTH

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

UX_DEVICE_CLASS_HID_EVENT_BUFFER_LENGTH

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

UX_DEVICE_CLASS_HID_MAX_EVENTS_QUEUE

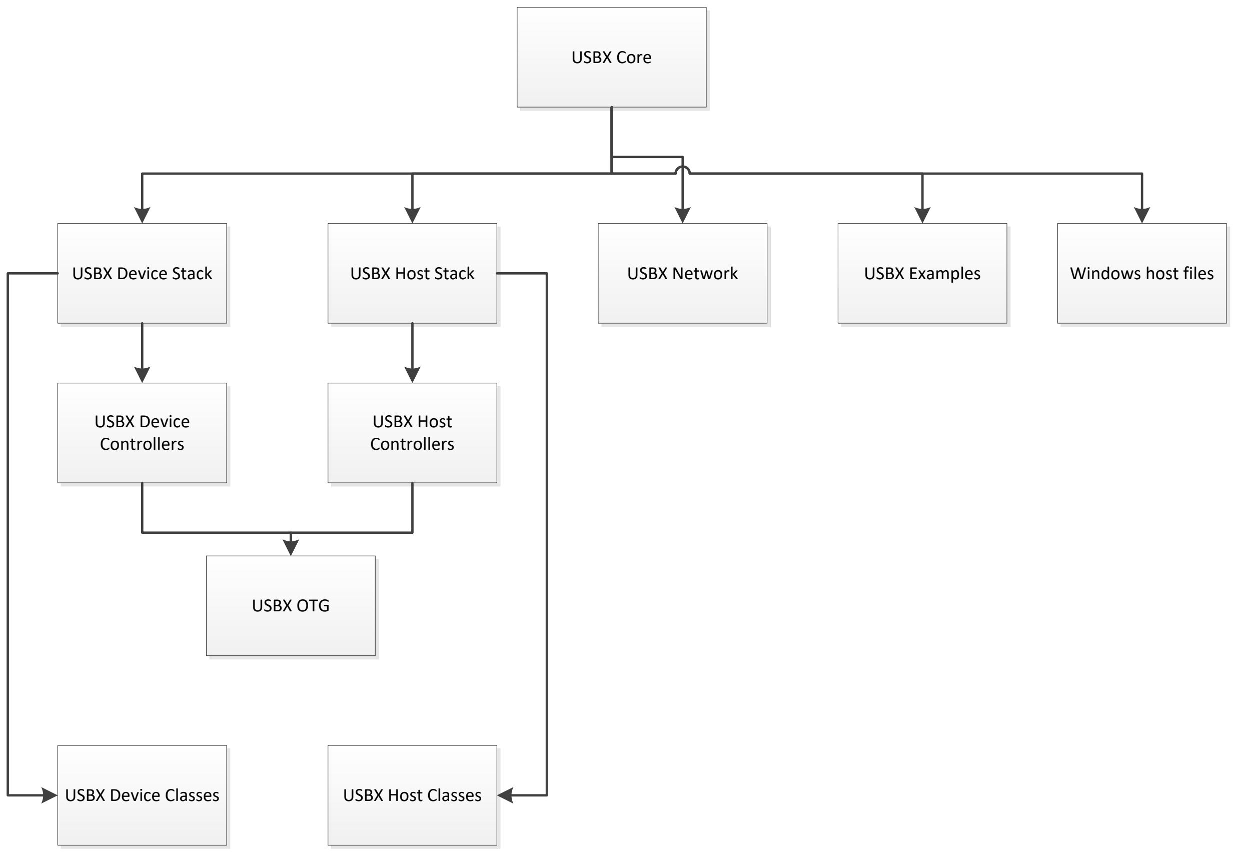
this value represents the the maximum number of HID reports that can be queued at once.

UX_SLAVE_REQUEST_DATA_MAX_LENGTH

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

Source Code Tree

The USBX files are provided in several directories.



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

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

Initialization of USBX resources

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

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

The prototype for the ux_system_initialize is as follows:

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

Input parameters:

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

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

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

Uninitialization of USBX resources

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

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

The prototype for the ux_system_initialize is as follows:

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

Definition of USB Device Controller

Only one USB device controller can be defined at any time to operate in device mode. The application initialization file should contain this definition.
The following line performs the definition of a generic usbcontroller:

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

The USB device initialization has the following prototype:

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

with the following parameters:

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

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

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

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

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

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

/* Initialize the storage class parameters for reading/writing to the Flash Disk.  */
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                  ux_slave_class_storage_media_last_lba = 0x1e6bfe;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                  ux_slave_class_storage_media_block_length =  512;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                  ux_slave_class_storage_media_type =  0;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                  ux_slave_class_storage_media_removable_flag =  0x80;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                  ux_slave_class_storage_media_read =
                                               tx_demo_thread_flash_media_read;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                  ux_slave_class_storage_media_write =
                                        tx_demo_thread_flash_media_write;
storage_parameter.ux_slave_class_storage_parameter_lun[0].
                                  ux_slave_class_storage_media_status =
                                        tx_demo_thread_flash_media_status;

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

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

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

Troubleshooting

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

USBX Version ID

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

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

Chapter 3: Functional Components of USBX Device Stack

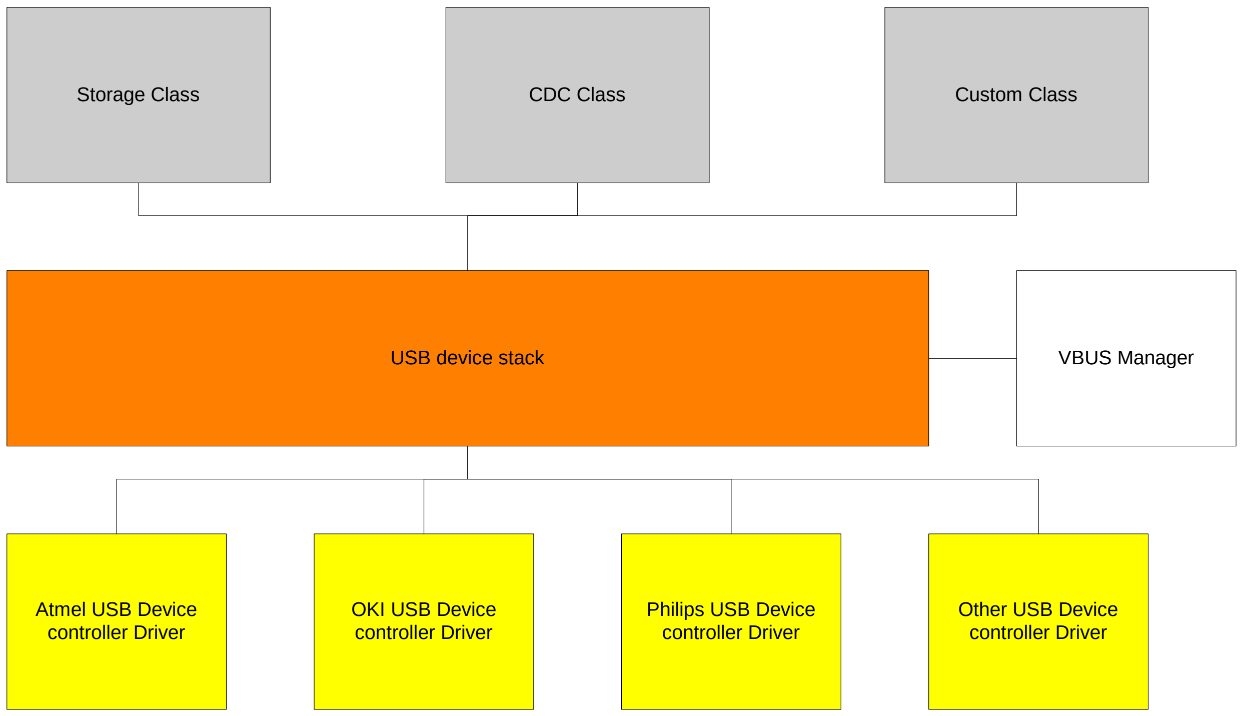
This chapter contains a description of the high performance USBX embedded USB device stack from a functional perspective.

Execution Overview:

USBX for the device is composed of several components:

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

The following diagram illustrates the USBX Device stack:



Storage Class
CDC Class
Custom Class
USB device stack
VBUS Manager
Atmel USB Device controller Driver
OKI USB Device controller Driver
Philips USB Device controller Driver
Other USB Device controller Driver

Initialization

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

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

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

Application Interface Calls

There are two levels of APIs in USBX:

USB Device Stack APIs
USB Device Class APIs

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

USB Device Stack APIs

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

USB Device Class APIs

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

Device Framework

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

Definition of the Components of the Device Framework

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

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

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

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

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

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

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

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

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

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

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

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

Definition of the Strings of the Device Framework

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

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

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

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

#define STRING_FRAMEWORK_LENGTH 38
UCHAR string_framework[] = {

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

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

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

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

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

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

Definition of the Languages Supported by the Device for each String

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

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

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

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

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

VBUS Manager

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

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

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

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

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

```
   ux_device_stack_disconnect()
```

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

Chapter 4: Description of USBX Device Services

ux_device_stack_alternate_setting_get

Get current alternate setting for an interface value

Prototype

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

Description

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

Input Parameter

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

Return Values

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

Example

```
ULONG   interface_value;
UINT    status;

/* The following example illustrates this service. */

status = ux_device_stack_alternate_setting_get(interface_value);

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

ux_device_stack_alternate_setting_set

Set current alternate setting for an interface value

Prototype

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

Description

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

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

Parameters

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

Return Values

UX_SUCCESS	(0x00)	The data transfer was completed.
UX_INTERFACE_HANDLE_UNKNOWN	(0x52)	No interface attached.
UX_FUNCTION_NOT_SUPPORTED	(0x54)	Device is not configured.
UX_ERROR	(0xFF)	Wrong interface value.

Example

```
ULONG   interface_value;
ULONG   alternate_setting_value;

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

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

ux_device_stack_class_register

Register a new USB device class

Prototype

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

Description

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

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

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

Parameters

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

Return Values

UX_SUCCESS	(0x00)	The class was registered
UX_MEMORY_INSUFFICIENT	(0x12)	No entries left in class table.
UX_THREAD_ERROR	(0x16)	Cannot create a class thread.

Example

```
UINT   status;

/* The following example illustrates this service. */

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

ux_device_stack_class_unregister

Unregister a USB device class

Prototype

```
UINT  ux_device_stack_class_unregister(UCHAR *class_name,
          UINT (*class_entry_function)(struct UX_SLAVE_CLASS_COMMAND_STRUCT *))
```

Description

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

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

Parameters

class_name	Class Name
class_entry_function	The entry function of the class.

Return Values

UX_SUCCESS	(0x00)	The class was unregistered.
UX_NO_CLASS_MATCH	(0x57)	The class isn't registered.

Example

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

/* Unitialize the device storage class. */
status =
ux_device_stack_class_unregister(_ux_system_slave_class_storage_name,
                                 ux_device_class_storage_entry);


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

ux_device_stack_configuration_get

Get the current configuration

Prototype

```
UINT  ux_device_stack_configuration_get(VOID)
```

Description

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

Input Parameter

None

Return Value

UX_SUCCESS	(0x00)	The data transfer was completed.

Example

```
UINT   status;

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

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

ux_device_stack_configuration_set

Set the current configuration

Prototype

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

Description

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

Input Parameter

configuration_value	The configuration value selected by the host.

Return Value

UX_SUCCESS	(0x00)	The configuration was successfully set.

Example

```
ULONG   configuration_value;
UINT    status;

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

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

ux_device_stack_descriptor_send

Send a descriptor to the host

Prototype

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

Description

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

Parameters

descriptor_type	The nature of the descriptor: UX_DEVICE_DESCRIPTOR_ITEM UX_CONFIGURATION_DESCRIPTOR_ITEM UX_STRING_DESCRIPTOR_ITEM UX_DEVICE_QUALIFIER_DESCRIPTOR_ITEM UX_OTHER_SPEED_DESCRIPTOR_ITEM
request_index	The index of the descriptor.
host_length	The length required by the host.

Return Values

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

Example

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

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

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

ux_device_stack_disconnect

Disconnect device stack

Prototype

```
UINT  ux_device_stack_disconnect(VOID)
```

Description

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

Input Parameter

None

Return Value

UX_SUCCESS	(0x00)	The device was disconnected.

Example

```
UINT   status;

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

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

ux_device_stack_endpoint_stall

Request endpoint Stall condition

Prototype

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

Description

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

Input Parameter

endpoint	The endpoint on which the Stall condition is requested.

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0xFF)	The device is in an invalid state.

Example

```
UINT   status;

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

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

ux_device_stack_host_wakeup

Wake up the host

Prototype

```
UINT  ux_device_stack_host_wakeup(VOID)
```

Description

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

Input Parameter

None

Return values

UX_SUCCESS	(0x00)	The call was successful.
UX_FUNCTION_NOT_SUPPORTED	(0x54)	The call failed (the device was probably not in the suspended mode).

Example

```
UINT   status;

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

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

ux_device_stack_initialize

Initialize USB device stack

Prototype

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

Description

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

Parameters

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

Return Values

UX_SUCCESS	(0x00)	This operation was successful.
UX_MEMORY_INSUFFICIENT	(0x12)	Not enough memory to initialize the stack.
UX_DESCRIPTOR_CORRUPTED	(0x42)	The descriptor is invalid.

Example

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

#define DEVICE_FRAMEWORK_LENGTH_FULL_SPEED 50
UCHAR device_framework_full_speed[] = {

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

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

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

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

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


#define DEVICE_FRAMEWORK_LENGTH_HIGH_SPEED 60
UCHAR device_framework_high_speed[] = {

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

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

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

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

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

      /* Endpoint descriptor (Bulk In) */
            0x07, 0x05, 0x82, 0x02, 0x00, 0x02, 0x00
      };
      /* String Device Framework:
      Byte 0 and 1: Word containing the language ID: 0x0904 for US
      Byte 2      : Byte containing the index of the descriptor
      Byte 3      : Byte containing the length of the descriptor string
      */

#define STRING_FRAMEWORK_LENGTH 38
UCHAR string_framework[] = {

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

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

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

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

      #define LANGUAGE_ID_FRAMEWORK_LENGTH 2
      UCHAR language_id_framework[] = {

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

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

UX_DEVICE_SUSPENDED
UX_DEVICE_RESUMED

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

```
      UINT   status;

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

      status = ux_device_stack_initialize(&device_framework_high_speed,
                                          DEVICE_FRAMEWORK_LENGTH_HIGH_SPEED,
                                &device_framework_full_speed,
                                DEVICE_FRAMEWORK_LENGTH_FULL_SPEED,
                                &string_framework,
                                STRING_FRAMEWORK_LENGTH,
                                &language_id_framework,
                                LANGUAGE_ID_FRAMEWORK_LENGTH,
                                UX_NULL);

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

ux_device_stack_interface_delete

Delete a stack interface

Prototype

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

Description

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

Input Parameter

interface	Pointer to the interface to remove.

Return Value

UX_SUCCESS	(0x00)	This operation was successful.

Example

```
UINT   status;

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

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

ux_device_stack_interface_get

Get the current interface value

Prototype

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

Description

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

Note: this function is deprecated. ux_device_stack_alternate_setting_get should be used instead.

Input Parameter

interface_value	Interface value to return.

Return Values

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

Example

```
ULONG   interface_value;
UINT    status;

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

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

ux_device_stack_interface_set

Change the alternate setting of the interface

Prototype

```
UINT  ux_device_stack_interface_set(UCHAR *device_framework,
                                    ULONG device_framework_length,
                                    ULONG alternate_setting_value)
```

Description

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

Parameters

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

Return Values

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

Example

```
UCHAR * device_framework
ULONG      device_framework_length;
ULONG      alternate_setting_value;
UINT       status;

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

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

ux_device_stack_interface_start

Start search for a class to own an interface instance

Prototype

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

Description

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

Input Parameter

interface	Pointer to the interface created.

Return Values

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

Example

```
UINT   status;

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

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

ux_device_stack_transfer_request

Request to transfer data to the host

Prototype

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

Description

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

Parameters

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

Return Values

UX_SUCCESS	(0x00)	This operation was successful.
UX_TRANSFER_NOT_READY	(0x25)	The device is in an invalid state; it must be ATTACHED, CONFIGURED, or ADDRESSED.
UX_ERROR	(0xFF)	Transport error.

Example

```
UINT   status;

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

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

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

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

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

ux_device_stack_transfer_abort

Cancel a transfer request

Prototype

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

Description

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

Parameters

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

Return Value

UX_SUCCESS	(0x00)	This operation was successful.

Example

```
UINT   status;

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

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

ux_device_stack_uninitialize

Unitialize stack

Prototype

```
UINT  ux_device_stack_uninitialize()
```

Description

This function is called when an application needs to unitialize the USBX device stack – all device stack resources are freed. This should be called after all classes have been unregistered via ux_device_stack_class_unregister.

Parameters

None

Return Value

UX_SUCCESS	(0x00)	This operation was successful.

Chapter 5: USBX Device Class Considerations

Device Class registration

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

```
  /* Set the parameters for callback when insertion/extraction of a HID
device.  */
    hid_parameter.ux_slave_class_hid_instance_activate   =
tx_demo_hid_instance_activate;
    hid_parameter.ux_slave_class_hid_instance_deactivate =
tx_demo_hid_instance_deactivate;

    /* Initialize the hid class parameters for the device.  */
    hid_parameter.ux_device_class_hid_parameter_report_address =
hid_device_report;
    hid_parameter.ux_device_class_hid_parameter_report_length  =
HID_DEVICE_REPORT_LENGTH;
    hid_parameter.ux_device_class_hid_parameter_report_id      = UX_TRUE;
    hid_parameter.ux_device_class_hid_parameter_callback       =
demo_thread_hid_callback;

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

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

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

```
VOID    tx_demo_hid_instance_activate(VOID *hid_instance)
{

    /* Save the HID instance.  */
    hid_slave = (UX_SLAVE_CLASS_HID *) hid_instance;
}

VOID    tx_demo_hid_instance_deactivate(VOID *hid_instance)
{

    /* Reset the HID instance.  */
    hid_slave = UX_NULL;
}
```

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

USB Device Storage Class

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

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

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

```
/* Initialize the storage class parameters to customize vendor strings.
*/
storage_parameter.ux_slave_class_storage_parameter_vendor_id =
  demo_ux_system_slave_class_storage_vendor_id;
storage_parameter.ux_slave_class_storage_parameter_product_id =
  demo_ux_system_slave_class_storage_product_id;
storage_parameter.ux_slave_class_storage_parameter_product_rev =
  demo_ux_system_slave_class_storage_product_rev;
storage_parameter.ux_slave_class_storage_parameter_product_serial =
  demo_ux_system_slave_class_storage_product_serial;

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

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

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

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_block_length  =  512;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_type  =  0;

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

storage_parameter.ux_slave_class_storage_parameter_lun[0].
   ux_slave_class_storage_media_read_only_flag  =  UX_FALSE;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_read  =  tx_demo_thread_flash_media_read;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_write  =
     tx_demo_thread_flash_media_write;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
```

```
    ux_slave_class_storage_media_status  =
       tx_demo_thread_flash_media_status;


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

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

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

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

The prototypes for the read and write functions are:

```
UINT  media_read(VOID *storage, ULONG lun, UCHAR *data_pointer,
                 ULONG number_blocks, ULONG lba, ULONG *media_status);
UINT  media_write(VOID *storage, ULONG lun, UCHAR *data_pointer,
                  ULONG number_blocks, ULONG lba, ULONG *media_status);
```

Where:

storage is the instance of the storage class.
lun is the LUN the command is directed to.
data_pointer is the address of the buffer to be used for reading or writing.
number_blocks is the number of sectors to read/write.
lba is the sector address to read.
media_status should be filled out exactly like the media status callback return value.

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

This function has the following prototype:

```
ULONG    media_status(VOID *storage, ULONG lun, ULONG media_id,
                      ULONG *media_status);
```

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

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

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

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

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

07	27	00	WRITE PROTECTED MEDIA
0B	4E	00	OVERLAPPED COMMAND ATTEMPTED

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

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

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

Where:

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

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

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

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

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

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

Where:

storage is the instance of the storage class.
lun parameter specifies which LUN the command is directed to.

number_blocks specifies the number of blocks to synchronize.
lba is the sector address of the first block to synchronize.
media_status should be filled out exactly like the media status callback return value.

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

Multiple SCSI LUN

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

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

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

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

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_block_length  =  512;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_type  =  0;

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

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_read  =  tx_demo_thread_flash_media_read;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_write  =
        tx_demo_thread_flash_media_write;

storage_parameter.ux_slave_class_storage_parameter_lun[0].
  ux_slave_class_storage_media_status  =
        tx_demo_thread_flash_media_status;

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

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

storage_parameter.ux_slave_class_storage_parameter_lun[1].
  ux_slave_class_storage_media_block_length  =  2048;

storage_parameter.ux_slave_class_storage_parameter_lun[1].
```

```
   ux_slave_class_storage_media_type  =  5;

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

storage_parameter.ux_slave_class_storage_parameter_lun[1].
  ux_slave_class_storage_media_read  =  tx_demo_thread_cdrom_media_read;

storage_parameter.ux_slave_class_storage_parameter_lun[1].
   ux_slave_class_storage_media_write  =
         tx_demo_thread_cdrom_media_write;

storage_parameter.ux_slave_class_storage_parameter_lun[1].
   ux_slave_class_storage_media_status  =
         tx_demo_thread_cdrom_media_status;


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

USB Device CDC-ACM Class

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

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

```
unsigned char device_framework_full_speed[] = {

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

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

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

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

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

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

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

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

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

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

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

    /* First alternate setting Endpoint 1 descriptor 7 bytes*/
    0x07, 0x05, 0x02,0x02,0x40, 0x00,0x00,
    /* Endpoint 2 descriptor 7 bytes */
    0x07, 0x05, 0x81,0x02,0x40, 0x00, 0x00,

};
```

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

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

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

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

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

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

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

The CDC-ACM is based on a USB-IF standard and is automatically recognized by MAC Os and Linux operating systems. On Windows platforms, this class requires a .inf file for Windows version prior to 10. Windows 10 does not require any .inf files. ExpressLogic supplies a template for the CDC-ACM class and it can be found in the usbx_windows_host_files directory. For more recent version of Windows the file CDC_ACM_Template_Win7_64bit.inf should be used (except Win10). This file needs to be modified to reflect the PID/VID used by the device. The PID/VID will be specific to the final customer when the company and the product are registered with the USB-IF. In the inf file, the fields to modify are located here:

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

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

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

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

The CDC-ACM class APIs are defined below:

ux_device_class_cdc_acm_ioctl

Perform IOCTL on the CDC-ACM interface

Prototype

```
UINT ux_device_class_cdc_acm_ioctl (UX_SLAVE_CLASS_CDC_ACM *cdc_acm, ULONG
ioctl_function, VOID *parameter)
```

Description

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

Parameters

cdc_acm	Pointer to the cdc class instance.
ioctl_function	Ioctl function to be performed.
parameter	Pointer to a parameter specific to the ioctl call

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0xFF)	Error from function

Example

```
/* Start cdc acm callback transmission.  */
status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START, &callback);

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

Ioctl functions :

```
UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_CODING                    1
UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING                    2
UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_STATE                     3
UX_SLAVE_CLASS_CDC_ACM_IOCTL_ABORT_PIPE                         4
UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_STATE                     5
UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START                 6
UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_STOP                  7
```

ux_device_class_cdc_acm_ioctl: UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_CODING

Perform IOCTL Set Line Coding on the CDC-ACM interface

Prototype

```
UINT ux_device_class_cdc_acm_ioctl (UX_SLAVE_CLASS_CDC_ACM *cdc_acm, ULONG
ioctl_function, VOID *parameter)
```

Description

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

Parameters

cdc_acm	Pointer to the cdc class instance.
ioctl_function	UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_CODING
parameter	Pointer to a line parameter structure:

```
typedef struct UX_SLAVE_CLASS_CDC_ACM_LINE_CODING_PARAMETER_STRUCT
{
    ULONG ux_slave_class_cdc_acm_parameter_baudrate;
    UCHAR ux_slave_class_cdc_acm_parameter_stop_bit;
    UCHAR ux_slave_class_cdc_acm_parameter_parity;
    UCHAR ux_slave_class_cdc_acm_parameter_data_bit;
} UX_SLAVE_CLASS_CDC_ACM_LINE_CODING_PARAMETER;
```

Return Value

UX_SUCCESS	(0x00)	This operation was successful.

Example

```
        /* Change the line coding values.  */
        line_coding.ux_slave_class_cdc_acm_line_coding_dter = 9600;
        line_coding.ux_slave_class_cdc_acm_line_coding_stop_bit =
UX_HOST_CLASS_CDC_ACM_LINE_CODING_STOP_BIT_15;
        line_coding.ux_slave_class_cdc_acm_line_coding_parity =
UX_HOST_CLASS_CDC_ACM_LINE_CODING_PARITY_EVEN;
        line_coding.ux_slave_class_cdc_acm_line_coding_data_bits = 5;

        status = _ux_slave_class_cdc_acm_ioctl(cdc_acm,
UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_CODING, &line_coding);

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

ux_device_class_cdc_acm_ioctl: UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING

Perform IOCTL Get Line Coding on the CDC-ACM interface

Prototype

```
device_class_cdc_acm_ioctl (UX_SLAVE_CLASS_CDC_ACM *cdc_acm, ULONG
ioctl_function, VOID *parameter)
```

Description

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

Parameters

cdc_acm	Pointer to the cdc class instance.
ioctl_function	ux_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING
parameter	Pointer to a line parameter structure:

```
typedef struct UX_SLAVE_CLASS_CDC_ACM_LINE_CODING_PARAMETER_STRUCT
{
    ULONG ux_slave_class_cdc_acm_parameter_baudrate;
    UCHAR ux_slave_class_cdc_acm_parameter_stop_bit;
    UCHAR ux_slave_class_cdc_acm_parameter_parity;
    UCHAR ux_slave_class_cdc_acm_parameter_data_bit;
} UX_SLAVE_CLASS_CDC_ACM_LINE_CODING_PARAMETER;
```

Return Value

UX_SUCCESS	(0x00)	This operation was successful.

Example

```
/* This is to retrieve BAUD rate.  */
status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_CODING, &line_coding);

/* Any error ? */
if (status == UX_SUCCESS)
{
    /* Decode BAUD rate. */
    switch (line_coding.ux_slave_class_cdc_acm_parameter_baudrate)
    {

        case 1200 :
            status = tx_demo_thread_slave_cdc_acm_response("1200", 4);
            break;
        case 2400 :
            status = tx_demo_thread_slave_cdc_acm_response("2400", 4);
            break;

        case 4800 :
            status = tx_demo_thread_slave_cdc_acm_response("4800", 4);
            break;

        case 9600 :
            status = tx_demo_thread_slave_cdc_acm_response("9600", 4);
            break;

        case 115200 :
            status = tx_demo_thread_slave_cdc_acm_response("115200", 6);
            break;

    }

}
```

ux_device_class_cdc_acm_ioctl: UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_STATE

Perform IOCTL Get Line State on the CDC-ACM interface

Prototype

```
UINT ux_device_class_cdc_acm_ioctl (UX_SLAVE_CLASS_CDC_ACM *cdc_acm, ULONG
ioctl_function, VOID *parameter)
```

Description

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

Parameters

cdc_acm	Pointer to the cdc class instance.
ioctl_function	ux_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_STATE
parameter	Pointer to a line parameter structure:

```
typedef struct UX_SLAVE_CLASS_CDC_ACM_LINE_STATE_PARAMETER_STRUCT
{
      UCHAR ux_slave_class_cdc_acm_parameter_rts;
      UCHAR ux_slave_class_cdc_acm_parameter_dtr;
} UX_SLAVE_CLASS_CDC_ACM_LINE_STATE_PARAMETER;
```

Return Value

UX_SUCCESS	(0x00)	This operation was successful.

Example

```
/* This is to retrieve RTS state.  */
status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
UX_SLAVE_CLASS_CDC_ACM_IOCTL_GET_LINE_STATE, &line_state);

/* Any error ? */
if (status == UX_SUCCESS)
{
    /* Check state.  */
    if (line_state.ux_slave_class_cdc_acm_parameter_rts == UX_TRUE)

        /* State is ON.  */
        status = tx_demo_thread_slave_cdc_acm_response("RTS ON", 6);

    else

        /* State is OFF.  */
        status = tx_demo_thread_slave_cdc_acm_response("RTS OFF", 7);
}
```

ux_device_class_cdc_acm_ioctl: UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_STATE

Perform IOCTL Set Line State on the CDC-ACM interface

Prototype

```
UINT ux_device_class_cdc_acm_ioctl (UX_SLAVE_CLASS_CDC_ACM *cdc_acm, ULONG
ioctl_function, VOID *parameter)
```

Description

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

Parameters

cdc_acm	Pointer to the cdc class instance.
ioctl_function	ux_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_STATE
parameter	Pointer to a line parameter structure:

```
typedef struct UX_SLAVE_CLASS_CDC_ACM_LINE_STATE_PARAMETER_STRUCT
{
UCHAR           ux_slave_class_cdc_acm_parameter_rts;
UCHAR           ux_slave_class_cdc_acm_parameter_dtr;

} UX_SLAVE_CLASS_CDC_ACM_LINE_STATE_PARAMETER;
```

Return Value

UX_SUCCESS	(0x00)	This operation was successful.

Example

```
/* This is to set RTS state.  */
line_state.ux_slave_class_cdc_acm_parameter_rts = UX_TRUE;
status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
UX_SLAVE_CLASS_CDC_ACM_IOCTL_SET_LINE_STATE, &line_state);

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

ux_device_class_cdc_acm_ioctl: UX_SLAVE_CLASS_CDC_ACM_IOCTL_ABORT_PIPE

Perform IOCTL ABORT PIPE on the CDC-ACM interface

Prototype

```
UINT ux_device_class_cdc_acm_ioctl (UX_SLAVE_CLASS_CDC_ACM *cdc_acm, ULONG
ioctl_function, VOID *parameter)
```

Description

This function is called when an application needs to abort a pipe. For example, to abort an ongoing write, UX_SLAVE_CLASS_CDC_ACM_ENDPOINT_XMIT should be passed as the parameter.

Parameters

cdc_acm	Pointer to the cdc class instance.
ioctl_function	UX_SLAVE_CLASS_CDC_ACM_IOCTL_ABORT_PIPE
parameter	The pipe direction: UX_SLAVE_CLASS_CDC_ACM_ENDPOINT_XMIT 1 UX_SLAVE_CLASS_CDC_ACM_ENDPOINT_RCV 2

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_ENDPOINT_HANDLE_UNKNOWN	(0x53)	Invalid pipe direction.

Example

```
/* This is to abort the Xmit pipe.  */
status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
UX_SLAVE_CLASS_CDC_ACM_IOCTL_ABORT_PIPE,
UX_SLAVE_CLASS_CDC_ACM_ENDPOINT_XMIT);

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

ux_device_class_cdc_acm_ioctl: UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START

Perform IOCTL Transmission Start on the CDC-ACM interface

Prototype

```
UINT ux_device_class_cdc_acm_ioctl (UX_SLAVE_CLASS_CDC_ACM *cdc_acm, ULONG
ioctl_function, VOID *parameter)
```

Description

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

Parameters

cdc_acm	Pointer to the cdc class instance.
ioctl_function	UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START
parameter	Pointer to the Start Transmission parameter structure:

```
typedef struct UX_SLAVE_CLASS_CDC_ACM_CALLBACK_PARAMETER_STRUCT
{
    UINT (*ux_device_class_cdc_acm_parameter_write_callback)(struct
UX_SLAVE_CLASS_CDC_ACM_STRUCT *cdc_acm, UINT status, ULONG length);

    UINT (*ux_device_class_cdc_acm_parameter_read_callback)(struct
UX_SLAVE_CLASS_CDC_ACM_STRUCT *cdc_acm, UINT status, UCHAR *data_pointer,
ULONG length);

} UX_SLAVE_CLASS_CDC_ACM_CALLBACK_PARAMETER;
```

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0xFF)	Transmission already started.
UX_MEMORY_INSUFFICIENT	(0x12)	A memory allocation failed.

Example

```
    /* Set the callback parameter. */
    callback.ux_device_class_cdc_acm_parameter_write_callback =
tx_demo_thread_slave_write_callback;
    callback.ux_device_class_cdc_acm_parameter_read_callback =
tx_demo_thread_slave_read_callback;

    /* Program the start of transmission.  */
    status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START, &callback);

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

ux_device_class_cdc_acm_ioctl: UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_STOP

Perform IOCTL Transmission Stop on the CDC-ACM interface

Prototype

```
UINT ux_device_class_cdc_acm_ioctl (UX_SLAVE_CLASS_CDC_ACM *cdc_acm, ULONG
ioctl_function, VOID *parameter)
```

Description

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

Parameters

cdc_acm	Pointer to the cdc class instance.
ioctl_function	UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_STOP
parameter	Not used

Return Value

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

Example

```
  /* Program the stop of transmission.  */
    status = _ux_device_class_cdc_acm_ioctl(cdc_acm_slave,
UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_STOP, UX_NULL);

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

ux_device_class_cdc_acm_read

Read from CDC-ACM pipe

Prototype

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

Description

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

Parameters

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

Return Value

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

Example

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

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

ux_device_class_cdc_acm_write

Write to a CDC-ACM pipe

Prototype

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

Description

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

Parameters

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

Return Value

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

Example

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

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

ux_device_class_cdc_acm_write_with_callback

Writing to a CDC-ACM pipe with callback

Prototype

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

Description

This function is called when an application needs to write to the IN data pipe (IN from the host, OUT from the device). This function is non-blocking and the completion will be done through a callback set in UX_SLAVE_CLASS_CDC_ACM_IOCTL_TRANSMISSION_START.

Parameters

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

Return Value

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

Example

```
/* Write to the CDC class bulk in pipe non blocking mode.  */
status = ux_device_class_cdc_acm_write_with_callback(cdc, buffer,
UX_DEMO_BUFFER_SIZE);

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

USB Device CDC-ECM Class

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

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

```
unsigned char device_framework_full_speed[] = {

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

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

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

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

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

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

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

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

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

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

    /* Data Class Interface Descriptor Alternate Setting 1, 2 endpoints. 9
           bytes */
    0x09, 0x04, 0x01, 0x01, 0x02, 0x0A, 0x00, 0x00,0x00,
    /* First alternate setting Endpoint 1 descriptor 7 bytes */
    0x07, 0x05, 0x02, 0x02, 0x40, 0x00, 0x00,

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

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

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

```
unsigned char string_framework[] = {

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

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

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

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

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

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

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


    /* Define a NODE ID.  */
    cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[0] =
                                                                      0x00;
    cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[1] =
                                                                      0x1e;
    cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[2] =
                                                                      0x58;
```

```
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[3] =
                                                                  0x41;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[4] =
                                                                  0xb8;
cdc_ecm_parameter.ux_slave_class_cdc_ecm_parameter_local_node_id[5] =
                                                                  0x78;

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

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

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

The next parameters are for the definition of the node IDs. 2 Nodes are necessary for the CDC-ECM, a local node and a remote node. The local node specifies the MAC address of the device, while the remote node specifies the MAC address of the host. The remote node must be the same one as the one declared in the device framework string descriptor.

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

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

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

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

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

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

USB Device HID Class

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

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

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

```
UCHAR device_framework_full_speed[] = {

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

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

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

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

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

    };
```

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

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

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

```
    /* Initialize the hid class parameters for a keyboard.  */
    hid_parameter.ux_device_class_hid_parameter_report_address =
                                            hid_keyboard_report;
    hid_parameter.ux_device_class_hid_parameter_report_length  =
                                            HID_KEYBOARD_REPORT_LENGTH;
    hid_parameter.ux_device_class_hid_parameter_callback  =
                                            tx_demo_thread_hid_callback;
    hid_parameter.ux_device_class_hid_parameter_report_id      = 0;

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

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

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

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

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

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

The HID class can send data back to the host on the interrupt endpoint by using the ux_device_class_hid_event_set function.

ux_device_class_hid_event_set

Setting an event to the HID class

Prototype

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

Description

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

Parameters

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

Return Value

UX_SUCCESS	(0x00)	This operation was successful.
UX_ERROR	(0xFF)	Error no space available in circular queue.

Example

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

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

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

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

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

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

hid_callback

Getting an event from the HID class

Prototype

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

Description

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

Parameters

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

Example

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

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

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

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

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

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

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

    else

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

Index