TivaWare™ USB Library

USER'S GUIDE

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
www.ti.com/tiva-c

Revision Information

This is version 2.2.0.295 of this document, last updated on April 02, 2020.

Table of Contents

1 Introduction

The Texas Instruments® USB Library is a set of data types and functions for creating USB device, host or On-The-Go (OTG) applications. The contents of the USB library and its associated header files fall into four main groups: general purpose functions, device mode specific functions, host mode specific functions, and mode detection and control functions.

The set of general purpose functions are those that are to be used in device, host, and dual mode applications. These include functions to parse USB descriptors and configure features of the USB library. The device specific functions provide the class-independent features required by all USB device applications such as host connection signaling and responding to standard descriptor requests. The USB library also contains a set of modules that handle the class specific requests from the USB host as well as a layer to interact with an application. The USB library provides a set of lower level class-independent host functions required by all USB host applications such as device detection and enumeration, and endpoint management. This lower layer remains mostly invisible to the application, but can be exposed to the application via class specific USB host modules. Like the device mode layer, the host mode class specific modules provide an interface to allow the lower level USB library code to communicate directly over the USB bus and also has a higher level interface that interacts with the application. The USB library also provides functions to configure how the USB controller handles switching between host and device mode operation. The modes and the functions that control them are described in the next section.

1.1 Operating Modes

There are five modes that the USB Library can function in and they are set by application. The run time operating mode of the USB controller is set by the way the USB controller is configured to detect the USB connection to another device. This mode detection can be automatic by using full OTG operation, manual by using dual mode operation, or fixed to either host or device mode. In all cases these modes control how the USB controller interacts with the USB VBUS and USB ID pins.

When an application only needs to run in USB host mode it can choose how the device powers VBUS, the detection of over current, the automatic monitoring of VBUS. These are controlled at initialization time by calls to USBStackModeSet() with one of the eUSBMode∗ values defined in the tUSBMode type definition. The application uses the **eUSBModeForceHost** option if the application has no need to monitor VBUS. This still provides the ability to control VBUS power and the over current detection but does not allow for monitoring VBUS. The application uses the **eUSBMode-Host** setting when the application requires the ability to be able to monitor VBUS. This setting also requires that the ID pin be externally tied low as it does not fully disable the OTG mode of operation for mode detection.

If an application only needs to run in device mode there are two options to control how device mode is entered. The USBStackModeSet() function is still used to control device mode operation. The application uses **eUSBModeForceDevice** when the application needs to free up the VBUS and ID pins or does not want VBUS to be monitored. The impact of this is that the application is not informed of USB disconnection events because it can no longer monitor VBUS. This is only a problem for self powered applications and can be handled by monitoring VBUS on a separate pin. If the application needs to receive disconnect events it uses the **eUSBModeDevice** setting and must connect the VBUS pin to the USB connector and leave the ID pin unconnected.

Some application also need to run as either host or device and the USB library provides two methods to handle switching modes on the fly. The first is to use the normal USB OTG signaling to

control mode switching which requires both the ID and VBUS pins to be connected to the USB connector and using the **eUSBModeOTG** setting with USBStackModeSet(). This is designed to work with a single USB AB connector. See the USB OTG section of this document for further details on operating in OTG mode. Another method of switching operating mode is to allow the application to choose the operating mode of the USB controller manually. This is more useful when the application is using a host and a device connector and can detect when it needs to manually switch the USB operating mode. The APIs and further description is in the dual mode section of this document.

1.2   File Structure and Tools

The following tool chains are supported:

- Keil™ RealView® Microcontroller Development Kit
- Mentor Graphics® Sourcery™ CodeBench
- IAR Embedded Workbench®
- Texas Instruments Code Composer Studio™

Directory Structure Overview

The following is an overview of the organization of the USB library source code, along with references to where each portion is described in detail.

`usblib`	The main directory under DriverLib contains the Makefile used to build the library and also source files and headers for functions and data types which are of general use to device and host applications. The contents of this directory are described in chapter 5.
`usblib/device/`	This directory contains source code and header files relating to operation as a USB device. The contents of this directory are described in chapter 2.
`usblib/host/`	This directory contains source code and header files relating to operation as a USB host. The contents of this directory are described in chapter 3.

2 Device Functions

2.1 Introduction

This chapter describes the various API layers within the USB library that offer support for applications wishing to present themselves as USB devices. Several programming interfaces are provided ranging from the thinnest layer which merely abstracts the underlying USB controller hardware to high level interfaces offering simple APIs supporting specific devices.

Source Code Overview

Source code and headers for the device specific USB functions can be found in the device directory of the USB library tree, typically `DriverLib/usblib/device`.

`usbdevice.h`	The header file containing device mode function prototypes and data types offered by the library. This file is the main header file defining the USB Device API.
`usbdbulk.h`	The header file defining the USB generic bulk device class driver API.
`usbdcdc.h`	The header file defining the USB Communication Device Class (CDC) device class driver API.

`usbdhid.h`	The header file defining the USB Human Interface Device (HID) device class driver API.
`usbdhidkeyb.h`	The header file defining the USB HID keyboard device class API.
`usbdhidmouse.h`	The header file defining the USB HID keyboard device class API.
`usbdenum.c`	The source code for the USB device enumeration functions offered by the library.
`usbdhandler.c`	The source code for the USB device interrupt handler.
`usbdconfig.c`	The source code for the USB device configuration functions.
`usbdcdesc.c`	The source code for functions used to parse configuration descriptors defined in terms of an array of sections (as used with the USB Device API).
`usbdbulk.c`	The source code for the USB generic bulk device class driver.
`usbdcdc.c`	The source code for the USB Communication Device Class (CDC) device class driver.
`usbdhid.c`	The source code for the USB Human Interface Device (HID) device class driver.
`usbdhidkeyb.c`	The source code for the USB HID keyboard device class.
`usbdhidmouse.c`	The source code for the USB HID keyboard device class.
`usbdevicepriv.h`	The private header file containing definitions shared between various source files in the device directory. Applications must not include this header.

2.2   API choices for USB devices

The USB library contains four API layers relevant to the development of USB device applications. Moving down the stack, each API layer offers greater flexibility to an application but this is balanced by the greater effort required to use the lower layers. The available programming interfaces, starting at the highest level and working downwards, are:

- Device Class APIs
- Device Class Driver APIs
- The USB Device API
- The USB DriverLib API
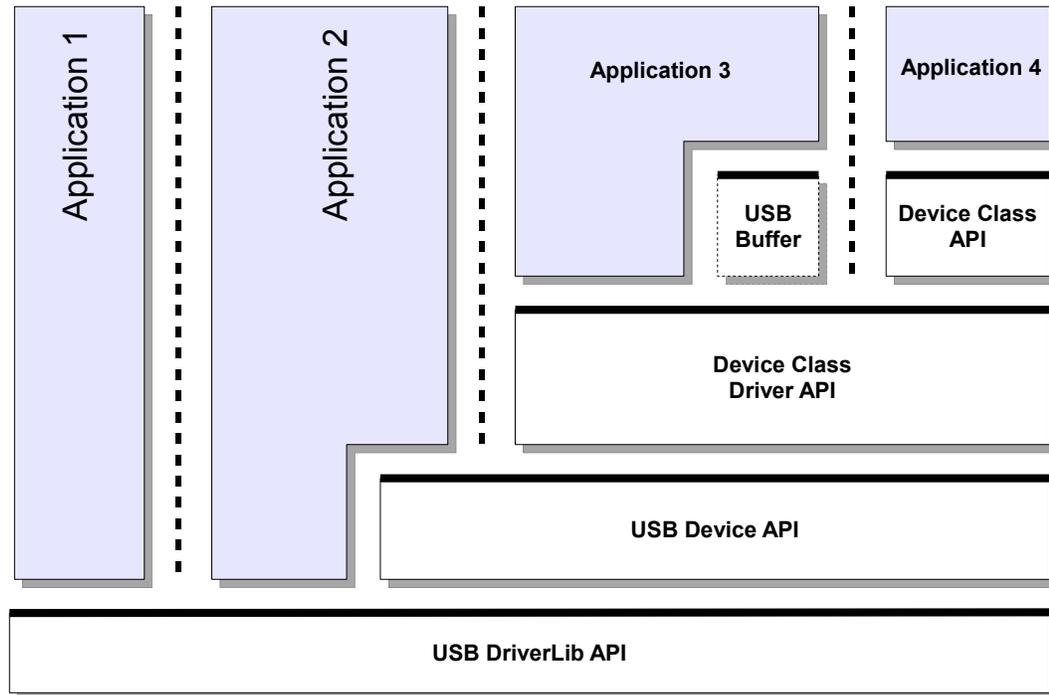
In the above diagram, bold horizontal lines represent APIs that are available for application use. Four possible applications are shown, each using a different programming interface to implement their USB functionality. The following sections provide an overview of the features and limitations of each layer and indicate the kinds of application which may choose to use that layer.

2.2.1 USB DriverLib API

The lowest layer in the USB device stack is the USB driver which can be found within the Driver Library (DriverLib) with source code in `usb.c` and header file `usb.h`. "Application 1" in the previous diagram offers device functionality by writing directly to this API.

Due to the fact that this API is a very thin layer above the USB controller's hardware registers and, hence, does not offer any higher level USB transaction support (such as endpoint zero transaction processing, standard descriptor and request processing, etc.), applications would not typically use this API as the only way to access USB functionality. This driver would, however, be a suitable interface to use if developing, for example, a third-party USB stack.

2.2.2 USB Library Device API

The USB Library Device API offers a group of functions specifically intended to allow development of fully-featured USB device applications with as much of the class-independent code as possible contained in the USB Library. The API supports device enumeration via standard requests from the host and handles the endpoint zero state machine on behalf of the application.

An application using this interface provides the descriptors that it wishes to publish to the host during initialization and these provide the information that the USB Device API requires to configure the hardware. Asynchronous events relating to the USB device are notified to the application by means of a collection of callback functions also provided to the USB Device API on initialization.

This API is used in the development of USB device class drivers and can also be used directly by applications which want to provide USB functionality not supported by an existing class driver. Examples of such devices would be those requiring complex alternate interface settings.

The USB Device API can be thought of as a set of high level device extensions to the USB DriverLib API rather than a wrapper over it. When developing to the USB Device API, some calls to the underlying USB DriverLib API are still necessary.

The header file for the USB Device API is `device/usbdevice.h`.

2.2.2.1 USB Device Feature API

The USB library provides a method to configure global settings for a USB device. The USBDCD-FeatureSet() function allows applications to modify the libraries configurable features options by setting features before starting up the USB library. This allows for run-time customizations of the USB library without requiring rebuilding of the library. Applications use these configurable options to control features like power settings and clocking options.

2.2.2.2 USB Device PLL Feature

On some device classes, like the TM4C129x, the USB library must know what the application is using for the main PLL since the USB controller uses this to generate the USB clock. The USB library defaults to a 480 MHz PLL frequency, however if this is not the case then the application must call the USBDCDFeatureSet() function with the **USBLIB_FEATURE_USBPLL** option to inform the USB library of the non-default PLL frequency. The following is an example of configuring the USB library to operate when the application is using a 320 MHz PLL.

Example: Using 320MHz PLL setting

```
uint32_t ui32PLLFrequency;

ui32PLLFrequency = 320000000;

//
// Inform the USB library that the system is running using a 320 MHz PLL.
//
USBDCDFeatureSet(0, USBLIB_FEATURE_USBPLL, &ui32PLLFrequency);
```

2.2.2.3 USB Device ULPI Feature

The TM4C129x class devices also supports using an external ULPI USB phy to allow operating the USB controller in high speed mode. This feature is enabled by setting the **USBLIB_FEATURE_USBULPI** option combined with the desired speed. From the applications perspective this has no affect to normal USB operation other than the necessity to properly enable the USB external phy pins. The possible options are the following:

- **USBLIB_FEATURE_ULPI_NONE** - Disable ULPI and use the internal phy(default).

- **USBLIB_FEATURE_ULPI_HS** - Use an externally connected ULPI phy at high-speed.
- **USBLIB_FEATURE_ULPI_FS** - Use an externally connected ULPI phy at full-speed.

The following is an example of configuring the USB library to use and external phy operating in high speed mode.

Example: Enable high speed external ULPI phy

```
uint32_t ui32ULPI;

ui32ULPI = USBLIB_FEATURE_ULPI_HS;

//
// Enable the use of an external USB ULPI connected phy.
//
USBDCDFeatureSet(0, USBLIB_FEATURE_USBULPI, &ui32ULPI);
```

2.2.2.4 USB Device Power Feature

Because many USB devices need to determine their own power configuration options, the USB library provides a feature settings for power as well. The power options are set using the **US-BLIB_FEATURE_POWER** value with the following options:

- **USBLIB_FEATURE_POWER_BUS** - USB device mode is bus powered(default).
- **USBLIB_FEATURE_POWER_SELF** - USB device mode is self powered.
- **USBLIB_FEATURE_REMOTE_WAKE** - Enable USB remote wake feature.

The following is an example of configuring the USB library to be self-powered with remote wake enabled.

Example: Configure as self powered with remote wake

```
uint32_t ui32Power;

ui32ULPI = USBLIB_FEATURE_POWER_SELF | USBLIB_FEATURE_REMOTE_WAKE;

//
// Configure as self powered with remote wake capabilities.
//
USBDCDFeatureSet(0, USBLIB_FEATURE_POWER, &ui32Power);
```

2.2.2.5 USB Device LPM Feature

The TM4C129 class devices added the ability to use the USB LPM features. This feature is not enabled by default and therefore must be enabled by the application if it wishes to use the LPM feature. These options also set using using the USBDCDFeatureSet() API and have the following options:

- **USBLIB_FEATURE_LPM_DIS** - Disable LPM transactions (default).
- **USBLIB_FEATURE_LPM_EN** - Enable receiving LPM transactions.
- **USBLIB_FEATURE_LPM_RMT_WAKE** - Enable support for LPM remote wake feature.

Example: Enable LPM transactions with remote wake.

The following is an example of configuring the USB library enable LPM mode and enable the device as an LPM remote wake device.

```
uint32_t ui32LPMFeature;

ui32LPMFeature = USBLIB_FEATURE_LPM_EN | USBLIB_FEATURE_LPM_RMT_WAKE;

//
// Enable LPM transactions and remote LPM wake.
//
USBDCDFeatureSet(0, USBLIB_FEATURE_LPM, &ui32LPMFeature);
```

2.2.2.6 USB Device Power APIs

When an application has enabled certain power related features like remote wake and LPM it also needs the ability to trigger run-time events using the USB library. The only USB device mode run-time power functions are the USBDCDRemoteWakeupRequest() and USBDCDRemoteWakeLPM() functions which are both used to trigger remote wake requests to the USB host controller. The USBDCDRemoteWakeupRequest() is used when a device has received a normal USB suspend request and the application needs to wake the USB host. The USBDCDRemoteWakeLPM() is used when a device entered a suspended state due to an LPM suspend event and needs to wake the USB host.

2.2.3 USB Device Class Driver APIs

Device Class Drivers offer high level USB function to applications wishing to offer particular USB features without having to deal with most of the USB transaction handling and connection management that would otherwise be required. These drivers provide high level APIs for several commonly-used USB device classes with the following features.

- Extremely easy to use. Device setup involves creating a set of static data structures and calling a single initialization API.
- Configurable VID/PID, power parameters and string table to allow easy customization of the device without the need to modify any library code.
- Consistent interfaces. All device class drivers use similar APIs making it very straightforward to move between them.
- Minimal application overhead. The vast majority of USB handling is performed within the class driver and lower layers leaving the application to deal only with reading and writing data.
- May be used with optional USB buffer objects to further simplify data transmission and reception. Using USB buffers, interaction with the device class driver can become as simple as a read/write API with no state machine required to ensure that data is transmitted or received at the correct time.
- Device Class Driver APIs completely wrap the underlying USB Device and USB Driver APIs so only a single API interface is used by the application.

Balancing these advantages, application developers should note the following restrictions that apply when using the Device Class Driver APIs.

- No calls may be made to any other USB layer while the device class driver API is in use.

■ Alternate configurations are not supported by the supplied device class drivers.

Device class drivers are currently provided to allow creation of a generic bulk device, a Communication Device Class (virtual serial port) device and a Human Interface Device class device (mouse, keyboard, joystick, etc.). A special class driver for composite devices is also included. This acts as a wrapper allowing multiple device class drivers to be used in a single device. Detailed information on each of these classes can be found later in this document.

2.2.4   USB Device Class APIs

In some cases, a standard device class may offer the possibility of creating a great number of different and varied devices using the same class and in these cases an additional API layer can be provided to further specialize the device operation and simplify the interface to the application.

The Human Interface Device (HID) class is one such class. It is used to support a wide variety of devices including keyboards, joysticks, mice and game controllers but the interface is specified in such a way that it could be used for a huge number of vendor-specific devices offering data gathering capability. As a result, the HID device class driver is extremely general to allow support for as wide a range of devices as possible. To simplify the use of the interface, specific APIs are provided to support BIOS-compatible keyboard and mouse operation. Using the mouse class API instead of the base HID class driver API, an application can make itself visible to the USB host as a mouse using an extremely simple interface consisting of an initialization call and a call to inform the host of mouse movement or button presses. Similarly, using the keyboard device class API, the application can use a single API to send key make and break information to the host without having to be aware of the underlying HID structures and USB protocols.

Example applications `usb_dev_mouse` and `usb_dev_keyboard` make use of the HID mouse and keyboard device class APIs respectively.

2.3   Audio Device Class Driver

The USB audio device class provides a playback interface which allows an application to act as a generic USB audio device to any host operating systems that natively supports USB audio devices. Most operating systems provide native support for generic audio class devices which means that no operating system specific drivers are required. The USB audio device class provides a single 16 bit stereo playback interface at 48kHz sample rate and also provides volume and mute controls.

USB Audio Device Model
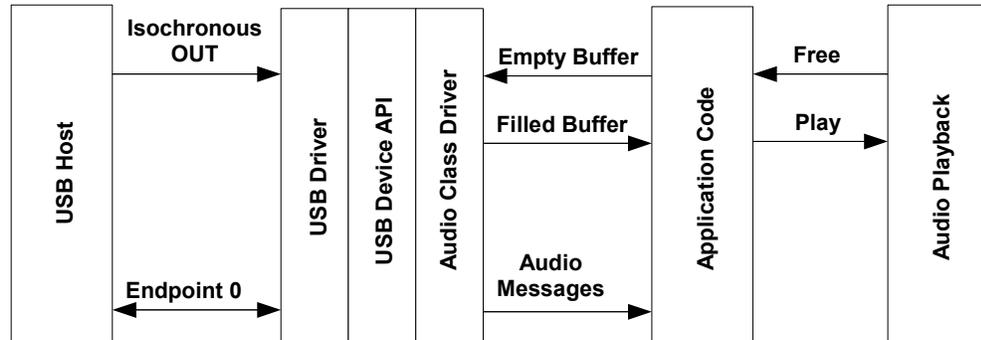
2.3.1 Handling Audio Playback

The audio playback path is handled by the application passing buffers to be filled to the USB audio class and receiving them back with audio data from the USB host controller. The USB audio class only holds one buffer at a time and returns it to the application when it is full. Because the USB audio class only holds one buffer, it is important to pass in a new buffer to the USB audio class as soon as possible once a buffer is returned to the application to prevent underflow from the USB host controller. Since most audio playback methods uses at least two buffers, one that is playing and one that is being filled, the single buffer in the USB audio class allows for minimal buffering and eliminates copying of data between buffers. When the application has an audio buffer that needs to be filled it passes it to the USB audio class using the USBAudioBufferOut() function. The USB audio class returns the buffer to the application via the audio message handler that the application provided in the pfnHandler member of the tUSBDAudioDevice structure. As soon as the audio device is active the application can provide a buffer to the USB audio class with a call to USBAudioBufferOut(). This call only fails if the USB audio class already has a buffer, at this point the application must wait for a previous buffer to be returned with a USBD_AUDIO_EVENT_DATAOUT message. Once the USBD_AUDIO_EVENT_DATAOUT message is received, the buffer has been filled and can be played. The buffer provided may not be completely full so the application should only play the portion of the buffer indicated by the message. To prevent underflow the application should always be sure that the audio device class has an empty buffer to fill as soon as a filled buffer is returned. The USB audio class does not provide a way to stop playing audio because typically when the USB host controller stops playing audio the host simply stops providing data to the USB audio device and playback stops. This does not result in any notification to the application other than USBD_AUDIO_EVENT_DATAOUT messages stop being received. If the application needs to stop receiving data, it can simply stop providing buffers to the USB audio class and the audio class ignores any incoming data from the USB host controller.

2.3.2 Handling Other Audio Messages

The USB audio class also provides a few other notification messages to the application. These are the USBD_AUDIO_EVENT_VOLUME and USBD_AUDIO_EVENT_MUTE messages which are both inform the application of volume and mute changes in the playback stream. The USBD_AUDIO_EVENT_VOLUME message returns a value that ranges from 0 - 100 in percentage for the playback volume. The USBD_AUDIO_EVENT_MUTE is either zero indicating that the playback path is not muted or 1 indicating that the playback path is muted. The application should always take care to defer any lenghty processing of messages to its non-callback routines to prevent underflow/overflow conditions from occuring.

2.3.3 Using the Generic Audio Device Class

To add USB Audio data playback capability to your application via the Audio Device Class Driver, take the following steps.

- Add the following header files to the source file(s) which are to support USB:

```
#include "src/usb.h"
#include "usblib/usblib.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdaudio.h"
```

- Define the six entry string descriptor table which is used to describe various features of your new device to the host system. The following is the string table taken from the `usb_dev_audio` example application. Edit the actual strings to suit your application and take care to ensure that you also update the length field (the first byte) of each descriptor to correctly reflect the length of the string and descriptor header. The number of string descriptors you include must be $(1 + (5 * num\ languages))$ where the number of languages agrees with the list published in string descriptor 0, *g_pLangDescriptor*. The strings for each language must be grouped together with all the language 1 strings before all the language 2 strings and so on.

```
//*****************************************************************************
//
// The languages supported by this device.
//
//*****************************************************************************
const uint8_t g_pui8LangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};

//*****************************************************************************
//
// The manufacturer string.
//
//*****************************************************************************
const uint8_t g_pui8ManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};
```

```c
//*****************************************************************************
//
// The product string.
//
//*****************************************************************************
const uint8_t g_pui8ProductString[] =
{
    (13 + 1) * 2,
    USB_DTYPE_STRING,
    'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 0, ' ', 0, 'E', 0, 'x', 0, 'a', 0,
    'm', 0, 'p', 0, 'l', 0, 'e', 0
};

//*****************************************************************************
//
// The serial number string.
//
//*****************************************************************************
const uint8_t g_pui8SerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

//*****************************************************************************
//
// The interface description string.
//
//*****************************************************************************
const uint8_t g_pui8HIDInterfaceString[] =
{
    (15 + 1) * 2,
    USB_DTYPE_STRING,
    'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 0, ' ', 0, 'I', 0, 'n', 0,
    't', 0, 'e', 0, 'r', 0, 'f', 0, 'a', 0, 'c', 0, 'e', 0
};

//*****************************************************************************
//
// The configuration description string.
//
//*****************************************************************************
const uint8_t g_pui8ConfigString[] =
{
    (20 + 1) * 2,
    USB_DTYPE_STRING,
    'A', 0, 'u', 0, 'd', 0, 'i', 0, 'o', 0, ' ', 0, ' ', 0, 'C', 0,
    'o', 0, 'n', 0, 'f', 0, 'i', 0, 'g', 0, 'u', 0, 'r', 0, 'a', 0,
    't', 0, 'i', 0, 'o', 0, 'n', 0
};

//*****************************************************************************
//
// The descriptor string table.
//
//*****************************************************************************
const uint8_t * const g_ppui8StringDescriptors[] =
{
    g_pui8LangDescriptor,
    g_pui8ManufacturerString,
    g_pui8ProductString,
    g_pui8SerialNumberString,
    g_pui8HIDInterfaceString,
    g_pui8ConfigString
```

```
    };

    #define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors) /              \
                                    sizeof(uint8_t *))
```

- Define a *tUSBDAudioDevice* structure and initialize all fields as required for your application.

```
    const tUSBDAudioDevice g_sAudioDevice =
    {
        //
        // The Vendor ID you have been assigned by USB-IF.
        //
        USB_VID_YOUR_VENDOR_ID,

        //
        // The product ID you have assigned for this device.
        //
        USB_PID_YOUR_PRODUCT_ID,

        //
        // The vendor string for your device (8 chars).
        //
        USB_YOUR_VENDOR_STRING,

        //
        // The product string for your device (16 chars).
        //
        USB_YOUR_PRODUCT_STRING,

        //
        // The revision string for your device (4 chars BCD).
        //
        USB_YOUR_REVISION_STRING,

        //
        // The power consumption of your device in milliamps.
        //
        POWER_CONSUMPTION_MA,

        //
        // The value to be passed to the host in the USB configuration descriptor's
        // bmAttributes field.
        //
        USB_CONF_ATTR_SELF_PWR,

        //
        // A pointer to your control callback event handler.
        //
        YourUSBAudioMessageHandler,

        //
        // A pointer to your string table.
        //
        g_ppui8StringDescriptors,

        //
        // The number of entries in your string table.
        //
        NUM_STRING_DESCRIPTORS,

        //
        // Maximum volume setting expressed as and 8.8 signed fixed point number.
        //
        VOLUME_MAX,

        //
```

```
                        // Minimum volume setting expressed as and 8.8 signed fixed point number.
                        //
                        VOLUME_MIN,

                        //
                        // Minimum volume step expressed as and 8.8 signed fixed point number.
                        //
                        VOLUME_STEP
            };
```

■ From your main initialization function call the audio device class driver initialization function to configure the USB controller and place the device on the bus.

```
            pvDevice = USBDAudioInit(0, &g_sAudioDevice);
```

■ Assuming *pvDevice* returned is not NULL, your device is now ready to communicate with a USB host.

■ Once the host connects, the audio message handler is sent the **USB_EVENT_CONNECTED** event.

■ Once the host is configured to use the Audio device the audio message handler is called with a **USBD_AUDIO_EVENT_ACTIVE** event.

2.3.4   Using the Audio Device Class in a Composite Device

When using the audio device class in a composite device, the configuration of the device is very similar to how it is configured as a non-composite device. Follow all of the configuration steps in the previous section with the exception of calling USBDAudioCompositeInit() instead of USBDAudioInit(). This prepares an instance of the audio device class to be enumerated as part of a composite device. The USBDAudioCompositeInit() function takes the audio device structure and a pointer to a tCompositeEntry value so that it can properly initialize the audio device and the composite entry that is passed to the USBDCompositeInit() funtion. The code example below provides an example of how to initialize an audio device to be a part of a composite device.

```
//
// These should be initialized with valid values for each class.
//
extern tUSBDAudioDevice g_sAudioDevice;
void *pvAudioDevice;

//
// The array of composite device entries.
//
tCompositeEntry psCompEntries[2];

//
// Allocate the device data for the top level composite device class.
//
tUSBDCompositeDevice g_sCompDevice =
{
    //
    // Texas Intruments C-Series VID.
    //
    USB_VID_TI_1CBE,

    //
    // Texas Intruments C-Series PID for composite serial device.
    //
    USB_PID_YOUR_COMPOSITE_PID,
```

```
    //
    // This is in 2mA increments so 500mA.
    //
    250,

    //
    // Bus powered device.
    //
    USB_CONF_ATTR_BUS_PWR,

    //
    // Composite event handler.
    //
    EventHandler,

    //
    // The string table.
    //
    g_pui8StringDescriptors,
    NUM_STRING_DESCRIPTORS,

    //
    // The Composite device array.
    //
    2,
    g_psCompEntries
};

//
// The OTHER_SIZES here are the sizes of the descriptor data for other classes
// that are part of the composite device.
//
#define DESCRIPTOR_DATA_SIZE    (COMPOSITE_DAUDIO_SIZE + OTHER_SIZES)
uint8_t g_pui8DescriptorData[DESCRIPTOR_DATA_SIZE];

//
// Initialize the audio device and its composite entry which is entry 0.
//
pvAudioDevice = USBDAudioCompositeInit(0, &g_sAudioDevice, &psCompEntries[0]);

//
// Initialize other devices to add to the composite device.
//

...

USBDCompositeInit(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
                  g_pui8DescriptorData);
```

All other API calls to the USB audio device class should use the value returned by USBDAudio-CompositeInit() when the APIs call for a pvInstance pointer. Also when using the audio device in a composite device the **COMPOSITE_DAUDIO_SIZE** value should be added to the size of the *g_pui8DescriptorData* array as shown in the example above.

2.3.5   Audio Device Class Events

The audio device class driver sends the following events to the application callback functions:

- USBD_AUDIO_EVENT_IDLE - Audio interface is idle.
- USBD_AUDIO_EVENT_ACTIVE - Audio interface is active.

- USBD_AUDIO_EVENT_DATAOUT - Audio playback buffer released.
- USBD_AUDIO_EVENT_VOLUME - Audio playback volume change.
- USBD_AUDIO_EVENT_MUTE - Audio mute change.

2.4   Audio Device Class Driver Definitions

Data Structures

- tUSBDAudioDevice

Defines

- COMPOSITE_DAUDIO_SIZE
- USBD_AUDIO_EVENT_ACTIVE
- USBD_AUDIO_EVENT_DATAOUT
- USBD_AUDIO_EVENT_IDLE
- USBD_AUDIO_EVENT_MUTE
- USBD_AUDIO_EVENT_VOLUME

Functions

- int32_t USBAudioBufferOut (void ∗pvAudioDevice, void ∗pvBuffer, uint32_t ui32Size, tUSBAudioBufferCallback pfnCallback)
- void ∗ USBDAudioCompositeInit (uint32_t ui32Index, tUSBDAudioDevice ∗psAudioDevice, tCompositeEntry ∗psCompEntry)
- void ∗ USBDAudioInit (uint32_t ui32Index, tUSBDAudioDevice ∗psAudioDevice)
- void USBDAudioTerm (void ∗pvAudioDevice)

2.4.1   Detailed Description

The macros and functions defined in this section can be found in header file `device/usbdaudio.h`.

2.4.2   Data Structure Documentation

2.4.2.1   tUSBDAudioDevice

Definition:
```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
```

```
        const char pcVendor[8];
        const char pcProduct[16];
        const char pcVersion[4];
        const uint16_t ui16MaxPowermA;
        const uint8_t ui8PwrAttributes;
        const tUSBCallback pfnCallback;
        const uint8_t *const *ppui8StringDescriptors;
        const uint32_t ui32NumStringDescriptors;
        const int16_t i16VolumeMax;
        const int16_t i16VolumeMin;
        const int16_t i16VolumeStep;
        tAudioInstance sPrivateData;
    }
    tUSBDAudioDevice
```

Members:

>   ***ui16VID*** The vendor ID that this device is to present in the device descriptor.
>
>   ***ui16PID*** The product ID that this device is to present in the device descriptor.
>
>   ***pcVendor*** 8 byte vendor string.
>
>   ***pcProduct*** 16 byte vendor string.
>
>   ***pcVersion*** 4 byte vendor string.
>
>   ***ui16MaxPowermA*** The maximum power consumption of the device, expressed in mA.
>
>   ***ui8PwrAttributes*** Indicates whether the device is self or bus-powered and whether or not it supports remote wake up. Valid values are USB_CONF_ATTR_SELF_PWR or USB_CONF_ATTR_BUS_PWR, optionally ORed with USB_CONF_ATTR_RWAKE.
>
>   ***pfnCallback*** A pointer to the callback function which will be called to notify the application of events relating to the operation of the audio device.
>
>   ***ppui8StringDescriptors*** A pointer to the string descriptor array for this device. This array must contain the following string descriptor pointers in this order. Language descriptor, Manufacturer name string (language 1), Product name string (language 1), Serial number string (language 1), Audio Interface description string (language 1), Configuration description string (language 1).
>   If supporting more than 1 language, the descriptor block (except for string descriptor 0) must be repeated for each language defined in the language descriptor.
>
>   ***ui32NumStringDescriptors*** The number of descriptors provided in the ppStringDescriptors array. This must be 1 + ((5 + (number of strings)) * (number of languages)).
>
>   ***i16VolumeMax*** The maximum volume expressed as an 8.8 signed value.
>
>   ***i16VolumeMin*** The minimum volume expressed as an 8.8 signed value.
>
>   ***i16VolumeStep*** The minimum volume step expressed as an 8.8 signed value.
>
>   ***sPrivateData*** The private instance data for the audio device.

Description:

>   The structure used by the application to define operating parameters for the device audio class.

2.4.3   Define Documentation

2.4.3.1   COMPOSITE_DAUDIO_SIZE

Definition:

```
#define COMPOSITE_DAUDIO_SIZE
```

Description:
> The size of the memory that should be allocated to create a configuration descriptor for a single instance of the USB Audio Device. This does not include the configuration descriptor which is automatically ignored by the composite device class.

2.4.3.2   USBD_AUDIO_EVENT_ACTIVE

Definition:
```
#define USBD_AUDIO_EVENT_ACTIVE
```

Description:
> This USB audio event indicates that the device is connected and is now active.

2.4.3.3   USBD_AUDIO_EVENT_DATAOUT

Definition:
```
#define USBD_AUDIO_EVENT_DATAOUT
```

Description:
> This USB audio event indicates that the device is returning a data buffer provided by the USBAudioBufferOut() function back to the application with valid audio data received from the USB host controller. The *pvBuffer* parameter holds the pointer to the buffer with the new audio data and the *ui32Param* value holds the amount of valid data in bytes that are contained in the *pvBuffer* parameter.

2.4.3.4   USBD_AUDIO_EVENT_IDLE

Definition:
```
#define USBD_AUDIO_EVENT_IDLE
```

Description:
> This USB audio event indicates that the device is connected but not active.

2.4.3.5   USBD_AUDIO_EVENT_MUTE

Definition:
```
#define USBD_AUDIO_EVENT_MUTE
```

Description:
> This USB audio event indicates that a mute request has occurred. The *ui32Param* value will either be a 1 to indicate that the audio is now muted, and a value of 0 indicates that the audio has been unmuted.

2.4.3.6 USBD_AUDIO_EVENT_VOLUME

Definition:
```
#define USBD_AUDIO_EVENT_VOLUME
```

Description:
This USB audio event indicates that a volume change has occurred. The *ui32Param* value contains a signed 8.8 fixed point value that represents the current volume gain/attenuation in decibels(dB). The provided message handler should be prepared to handle negative and positive values with the value 0x8000 indicating maximum attenuation. The *pvBuffer* parameter should be ignored.

2.4.4   Function Documentation

2.4.4.1   USBAudioBufferOut

This function is used to supply buffers to the audio class to be filled from the USB host device.

Prototype:
```
int32_t
USBAudioBufferOut(void *pvAudioDevice,
                  void *pvBuffer,
                  uint32_t ui32Size,
                  tUSBAudioBufferCallback pfnCallback)
```

Parameters:
pvAudioDevice is the pointer to the device instance structure as returned by USBDAudioInit() or USBDAudioCompositeInit().
pvBuffer is a pointer to the buffer to fill with audio data.
ui32Size is the size in bytes of the buffer pointed to by the *pvBuffer* parameter.
pfnCallback is a callback that will provide notification when this buffer has valid data.

Description:
This function fills the buffer pointed to by the *pvBuffer* parameter with at most *ui32Size* one packet of data from the host controller. The *ui32Size* has a minimum value of **ISOC_OUT_EP_MAX_SIZE** since each USB packet can be at most **ISOC_OUT_EP_MAX_SIZE** bytes in size. Since the audio data may not be received in amounts that evenly fit in the buffer provided, the buffer may not be completely filled. The *pfnCallback* function will provide the amount of valid data that was actually stored in the buffer provided. The function will return zero if the buffer could be scheduled to be filled, otherwise the function will return a non-zero value if there was some reason that the buffer could not be added.

Returns:
Returns 0 to indicate success any other value indicates that the buffer will not be filled.

2.4.4.2   USBDAudioCompositeInit

This function should be called once for the audio class device to initialized basic operation and prepare for enumeration.

Prototype:
```
void *
USBDAudioCompositeInit(uint32_t ui32Index,
                       tUSBDAudioDevice *psAudioDevice,
                       tCompositeEntry *psCompEntry)
```

Parameters:
> ***ui32Index*** is the index of the USB controller to initialize for audio class device operation.
>
> ***psAudioDevice*** points to a structure containing parameters customizing the operation of the audio device.
>
> ***psCompEntry*** is the composite device entry to initialize when creating a composite device.

Description:
> In order for an application to initialize the USB audio device class, it must first call this function with the a valid audio device class structure in the *psAudioDevice* parameter. This allows this function to initialize the USB controller and device code to be prepared to enumerate and function as a USB audio device. When this audio device is part of a composite device, then the *psCompEntry* should point to the composite device entry to initialize. This is part of the array that is passed to the USBDCompositeInit() function.
>
> This function returns a void pointer that must be passed in to all other APIs used by the audio class.
>
> See the documentation on the tUSBDAudioDevice structure for more information on how to properly fill the structure members.

Returns:
> Returns zero on failure or a non-zero instance value that should be used with the remaining USB audio APIs.

2.4.4.3   USBDAudioInit

This function should be called once for the audio class device to initialized basic operation and prepare for enumeration.

Prototype:
```
void *
USBDAudioInit(uint32_t ui32Index,
              tUSBDAudioDevice *psAudioDevice)
```

Parameters:
> ***ui32Index*** is the index of the USB controller to initialize for audio class device operation.
>
> ***psAudioDevice*** points to a structure containing parameters customizing the operation of the audio device.

Description:
> In order for an application to initialize the USB audio device class, it must first call this function with the a valid audio device class structure in the *psAudioDevice* parameter. This allows this function to initialize the USB controller and device code to be prepared to enumerate and function as a USB audio device.
>
> This function returns a void pointer that must be passed in to all other APIs used by the audio class.

See the documentation on the tUSBDAudioDevice structure for more information on how to properly fill the structure members.

Returns:
Returns 0 on failure or a non-zero void pointer on success.

2.4.4.4 USBDAudioTerm

Shuts down the audio device.

Prototype:
```
void
USBDAudioTerm(void *pvAudioDevice)
```

Parameters:
pvAudioDevice is the pointer to the device instance structure as returned by USBDAudioInit().

Description:
This function terminates audio interface for the instance supplied. This function should not be called if the audio device is part of a composite device and instead the USBDCompositeTerm() function should be called for the full composite device. Following this call, the *pvAudioDevice* instance should not me used in any other calls.

Returns:
None.

2.5 Bulk Device Class Driver

Although not offering support for a particular standard device class, the generic bulk device class driver offers a very simple method for an application to set up USB communication with a paired application running on the USB host system. The class driver offers a single bulk receive channel and a single bulk transmit channel and, when coupled with USB buffers on each channel, provides a straightforward read/write interface to the application.

The device supports a single interface containing bulk IN and bulk OUT endpoints. The configuration and interface descriptors published by the device contain vendor specific class identifiers so an application on the host has to communicate with the device using either a custom driver or a subsystem such as WinUSB or libusb-win32 on Windows to allow the device to be accessed. An example of this is provided in the `usb_dev_bulk` application.

This class driver is particularly useful for applications which intend passing high volumes of data via USB and where host-side application code is being developed in partnership with the device.

USB Generic Bulk Device Model
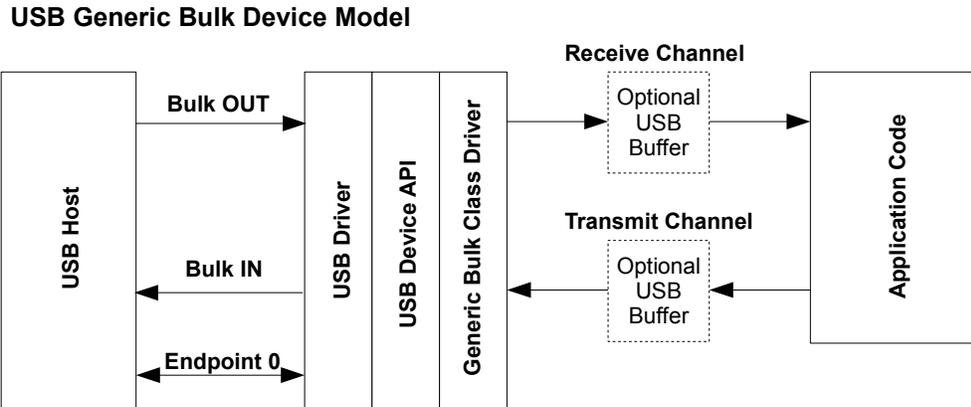
The `usb_dev_bulk` example application makes use of this device class driver.

2.5.1 Bulk Device Class Events

The bulk device class driver sends the following events to the application callback functions:

2.5.1.1 Receive Channel Events

- **USB_EVENT_RX_AVAILABLE**
- **USB_EVENT_ERROR**
- **USB_EVENT_CONNECTED**
- **USB_EVENT_DISCONNECTED**
- **USB_EVENT_SUSPEND**
- **USB_EVENT_RESUME**

Note: The **USB_EVENT_DISCONNECTED** event is not be reported to the application if the MCU's PB1/USB0VBUS pin is connected to a fixed +5 Volts rather than directly to the VBUS pin on the USB connector or if the USB controller is configured to force device mode.

2.5.1.2 Transmit Channel Events

- **USB_EVENT_TX_COMPLETE**

2.5.2 Using the Generic Bulk Device Class

To add USB bulk data transmit and receive capability to your application via the Generic Bulk Device Class Driver, take the following steps.

■ Add the following header files to the source file(s) which are to support USB:

```
#include "src/usb.h"
#include "usblib/usblib.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdbulk.h"
```

■ Define the 5 entry string table which is used to describe various features of your new device to the host system. The following is the string table taken from the `usb_dev_bulk` example application. Edit the actual strings to suit your application and take care to ensure that you also update the length field (the first byte) of each descriptor to correctly reflect the length of the string and descriptor header. The number of strings you include must be 5 ∗ (number of languages listed in string descriptor 0, *g_pLangDescriptor*, and the strings for each language must be grouped together with all the language 1 strings before all the language 2 strings and so on.

```
//*****************************************************************************
//
// The languages supported by this device.
//
//*****************************************************************************
const uint8_t g_pui8LangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};

//*****************************************************************************
//
// The manufacturer string.
//
//*****************************************************************************
const uint8_t g_pui8ManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};

//*****************************************************************************
//
// The product string.
//
//*****************************************************************************
const uint8_t g_pui8ProductString[] =
{
    (19 + 1) * 2,
    USB_DTYPE_STRING,
    'G', 0, 'e', 0, 'n', 0, 'e', 0, 'r', 0, 'i', 0, 'c', 0, ' ', 0, 'B', 0,
    'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'e', 0, 'v', 0, 'i', 0, 'c', 0,
    'e', 0
};

//*****************************************************************************
//
// The serial number string.
//
//*****************************************************************************
const uint8_t g_pui8SerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
```

```
};

//*****************************************************************************
//
// The data interface description string.
//
//*****************************************************************************
const uint8_t g_pui8DataInterfaceString[] =
{
    (19 + 1) * 2,
    USB_DTYPE_STRING,
    'B', 0, 'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'a', 0, 't', 0,
    'a', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0, 'r', 0, 'f', 0,
    'a', 0, 'c', 0, 'e', 0
};

//*****************************************************************************
//
// The configuration description string.
//
//*****************************************************************************
const uint8_t g_pui8ConfigString[] =
{
    (23 + 1) * 2,
    USB_DTYPE_STRING,
    'B', 0, 'u', 0, 'l', 0, 'k', 0, ' ', 0, 'D', 0, 'a', 0, 't', 0,
    'a', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 'f', 0, 'i', 0, 'g', 0,
    'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0, 'o', 0, 'n', 0
};

//*****************************************************************************
//
// The descriptor string table.
//
//*****************************************************************************
const uint8_t * const g_ppui8StringDescriptors[] =
{
    g_pui8LangDescriptor,
    g_pui8ManufacturerString,
    g_pui8ProductString,
    g_pui8SerialNumberString,
    g_pui8DataInterfaceString,
    g_pui8ConfigString
};

#define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors) /          \
                                sizeof(uint8_t *))
```

- Define a *tUSBDBulkDevice* structure and initialize all fields as required for your application.
  The following example illustrates a simple case where no USB buffers are in use. For an ex-
  ample using USB buffers, see the source file `usb_bulk_structs.c` in the `usb_dev_bulk`
  example application.

```
const tUSBDBulkDevice g_sBulkDevice =
{
    //
    // The Vendor ID you have been assigned by USB-IF.
    //
    USB_VID_YOUR_VENDOR_ID,

    //
    // The product ID you have assigned for this device.
    //
    USB_PID_YOUR_PRODUCT_ID,
```

```
    //
    // The power consumption of your device in milliamps.
    //
    POWER_CONSUMPTION_MA,

    //
    // The value to be passed to the host in the USB configuration descriptor's
    // bmAttributes field.
    //
    USB_CONF_ATTR_SELF_PWR,

    //
    // A pointer to your receive callback event handler.
    //
    YourUSBReceiveEventCallback,

    //
    // A value that you want passed to the receive callback alongside every
    // event.
    //
    (void *)&g_sYourInstanceData,

    //
    // A pointer to your transmit callback event handler.
    //
    YourUSBTransmitEventCallback,

    //
    // A value that you want passed to the transmit callback alongside every
    // event.
    //
    (void *)&g_sYourInstanceData,

    //
    // A pointer to your string table.
    //
    g_ppui8StringDescriptors,

    //
    // The number of entries in your string table.
    //
    NUM_STRING_DESCRIPTORS
};
```

- Add a receive event handler function, YourUSBReceiveEventCallback in the previous exam-ple, to your application taking care to handle all messages which require a particular response. For the generic bulk device class, only the **USB_EVENT_RX_AVAILABLE** MUST be handled by the receive event handler. In response to **USB_EVENT_RX_AVAILABLE**, your handler should check the amount of data received by calling USBDBulkRxPacketAvailable() then read it using a call to USBDBulkPacketRead(). This causes the newly received data to be ac-knowledged to the host and instructs the host that it may now transmit another packet. If you are unable to read the data immediately, return 0 from the callback handler and you are called back once again a few milliseconds later. Although no other events must be handled, **USB_EVENT_CONNECTED** and **USB_EVENT_DISCONNECTED** is typically required since these indicate when a host connects or disconnects and allow the application to flush any buffers or reset state as required. Attempts to send data when the host is disconnected result in an immediate failure.

- Add a transmit event handler function, YourUSBTransmitEventCallback in the previous exam-ple, to your application taking care to handle all messages which require a particular response. For the generic bulk device class, there are no events sent to the transmit callback which MUST be handled but applications usually want to note **USB_EVENT_TX_COMPLETE** since this is

an interlock message indicating that the previous packet sent has been acknowledged by the host and a new packet can now be sent.

- From your main initialization function call the generic bulk device class driver initialization function to configure the USB controller and place the device on the bus.

```
pvDevice = USBDBulkInit(0, &g_sBulkDevice);
```

- Assuming *pvDevice* returned is not NULL, your device is now ready to communicate with a USB host.

- Once the host connects, your receive event handler is sent **USB_EVENT_CONNECTED** and the first packet of data may be sent to the host using USBDBulkPacketWrite() with following packets transmitted as soon as **USB_EVENT_TX_COMPLETE** is received.

2.5.3   Using the Bulk Device Class in a Composite Device

When using the bulk device class in a composite device, the configuration of the device is very similar to how it is configured as a non-composite device. Follow all of the configuration steps in the previous section with the exception of calling USBDBulkCompositeInit() instead of USBDBulkInit(). This prepares an instance of the bulk device class to be enumerated as part of a composite device. The USBDBulkCompositeInit() function takes the bulk device structure and a pointer to a tCompositeEntry value so that it can properly initialize the bulk device and the composite entry that is passed to the USBDCompositeInit() funtion. The code example below provides an example of how to initialize a bulk device to be a part of a composite device.

```
//
// These should be initialized with valid values for each class.
//
extern tUSBDBulkDevice g_sBulkDevice;
void *pvBulkDevice;

//
// The array of composite devices.
//
tCompositeEntry psCompEntries[2];

//
// Allocate the device data for the top level composite device class.
//
tUSBDCompositeDevice g_sCompDevice =
{
    //
    // Texas Intruments C-Series VID.
    //
    USB_VID_TI_1CBE,

    //
    // Texas Intruments C-Series PID for composite serial device.
    //
    USB_PID_YOUR_COMPOSITE_PID,

    //
    // This is in 2mA increments so 500mA.
    //
    250,

    //
    // Bus powered device.
    //
```

```
        USB_CONF_ATTR_BUS_PWR,

        //
        // Composite event handler.
        //
        EventHandler,

        //
        // The string table.
        //
        g_pui8StringDescriptors,
        NUM_STRING_DESCRIPTORS,

        //
        // The Composite device array.
        //
        2,
        g_psCompEntries
};

//
// The OTHER_SIZES here are the sizes of the descriptor data for other classes
// that are part of the composite device.
//
#define DESCRIPTOR_DATA_SIZE    (COMPOSITE_DBULK_SIZE + OTHER_SIZES)
uint8_t g_pui8DescriptorData[DESCRIPTOR_DATA_SIZE];

//
// Initialize the bulk device and its composite entry.
//
pvBulkDevice = USBDBulkCompositeInit(0, &g_sBulkDevice, &psCompEntries[0]);

//
// Initialize other devices to add to the composite device.
//

...

//
// Initialize the USB controller as a composite device.
//
USBDCompositeInit(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
                  g_pui8DescriptorData);
```

All other API calls to the USB bulk device class should use the value returned by USBDBulk-CompositeInit() when the API calls for a pvInstance pointer. Also when using the bulk device in a composite device the **COMPOSITE_DBULK_SIZE** value should be added to the size of the *g_pui8DescriptorData* array as shown in the example above.

2.5.4 Windows Drivers for Generic Bulk Devices

Since generic bulk devices appear to a host operating system as vendor-specific devices, no device drivers on the host system is able to communicate with them without some help from the device developer. This help may involve writing a specific Windows kernel driver for the device or, if kernel driver programming is too daunting, steering Windows to use one of several possible generic kernel drivers that can manage the device on behalf of a user mode application.

Using this second model, a device developer need not write any Windows driver code but would need to write an application or DLL that interfaces with the device via the user-mode API offered by whichever USB subsystem they chose to manage their device. The developer is also responsible

for producing a suitable INF file to allow Windows to associate the device (identified via its VID/PID combination) with a particular driver.

A least two suitable USB subsystems are available for Windows - WinUSB from Microsoft or open-source project libusb-win32 available from SourceForge.

WinUSB supports WindowsXP, Windows Vista and Windows7 systems. Further information can be obtained from MSDN at `http://msdn.microsoft.com/en-us/library/aa476426.aspx`. To develop applications using the WinUSB interface, the Windows Driver Development Kit (DDK) must be installed on your build PC. This interface is currently used by the Windows example applications, "Oscilloscope" and "USB Bulk Example". These applications can be found in package "PDL-LM3S-win" which may be downloaded via a link on `http://www.ti.com/tivaware`.

libusb-win32 supports Windows98SE, Windows2000, WindowsNT and WindowsXP and can be downloaded from `http://libusb-win32.sourceforge.net/`. It offers a straightforward method of accessing the device and also provides a very helpful INF file generator.

2.5.4.1   Sample WinUSB INF file

This file illustrates how to build an INF to associate your device with the WinUSB subsystem on WindowsXP or Vista. Note that the driver package for the device must include not only this INF file but the Microsoft-supplied coinstallers listed in the files section. These can be found within the Windows Driver Development Kit (DDK).

```
; -------------------------------------------------------------------------
;
; USBLib Generic Bulk USB device driver installer
;
; This INF file may be used as a template when creating customized applications
; based on the USBLib generic bulk devices. Areas of the file requiring
; customization for a new device are commented with NOTEs.
;
; -------------------------------------------------------------------------

; NOTE: When you customize this INF for your own device, create a new class
; name (Class) and a new GUID (ClassGuid). GUIDs may be created using the
; guidgen tool from Windows Visual Studio.

[Version]
Signature = "$Windows NT$"
Class = USBLibBulkDeviceClass
ClassGuid={F5450C06-EB58-420e-8F98-A76C5D4AFB18}
Provider = %ProviderName%
CatalogFile=MyCatFile.cat

; ========== Manufacturer/Models sections ===========

[Manufacturer]
%ProviderName% = USBLibBulkDevice_WinUSB,NTx86,NTamd64

; NOTE: Replace the VID and PID in the following two sections with the
; correct values for your device.

[USBLibBulkDevice_WinUSB.NTx86]
%USB\USBLibBulkDevice.DeviceDesc% =USB_Install, USB\VID_1CBE&PID_0003

[USBLibBulkDevice_WinUSB.NTamd64]
%USB\USBLibBulkDevice.DeviceDesc% =USB_Install, USB\VID_1CBE&PID_0003

; =================== Installation ===================
```

```
[ClassInstall32]
AddReg=AddReg_ClassInstall

[AddReg_ClassInstall]
HKR,,,,"%DeviceClassDisplayName%"
HKR,,Icon,,"-20"

[USB_Install]
Include=winusb.inf
Needs=WINUSB.NT

[USB_Install.Services]
Include=winusb.inf
AddService=WinUSB,0x00000002,WinUSB_ServiceInstall

[WinUSB_ServiceInstall]
DisplayName     = %WinUSB_SvcDesc%
ServiceType     = 1
StartType       = 3
ErrorControl    = 1
ServiceBinary   = %12%\WinUSB.sys

[USB_Install.Wdf]
KmdfService=WINUSB, WinUsb_Install

[WinUSB_Install]
KmdfLibraryVersion=1.5

[USB_Install.HW]
AddReg=Dev_AddReg

; NOTE: Create a new GUID for your interface and replace the following one
; when customizing for a new device.

[Dev_AddReg]
HKR,,DeviceInterfaceGUIDs,0x10000,"{6E45736A-2B1B-4078-B772-B3AF2B6FDE1C}"

[USB_Install.CoInstallers]
AddReg=CoInstallers_AddReg
CopyFiles=CoInstallers_CopyFiles

[CoInstallers_AddReg]
HKR,,CoInstallers32,0x00010000,"WdfCoInstaller01005.dll,WdfCoInstaller","WinUSBCoInstaller.dll"

[CoInstallers_CopyFiles]
WinUSBCoInstaller.dll
WdfCoInstaller01005.dll

[DestinationDirs]
CoInstallers_CopyFiles=11

; ================ Source Media Section ====================

[SourceDisksNames]
1 = %DISK_NAME%,,,\i386
2 = %DISK_NAME%,,,\amd64

[SourceDisksFiles.x86]
WinUSBCoInstaller.dll=1
WdfCoInstaller01005.dll=1

[SourceDisksFiles.amd64]
WinUSBCoInstaller.dll=2
WdfCoInstaller01005.dll=2
```

```
; ================== Strings ==================

; Note: Replace these as appropriate to describe your device.

[Strings]
ProviderName="Texas Instruments"
USB\USBLibBulkDevice.DeviceDesc="Generic Bulk Device"
WinUSB_SvcDesc="WinUSB"
DISK_NAME="USBLib Install Disk"
DeviceClassDisplayName="USBLib Bulk Devices"
```

2.5.4.2   Sample libusb-win32 INF File

The following is an example of an INF file that can be used to associate the usb_dev_bulk ex-
ample device with the libusb-win32 subsystem on Windows systems and to install the necessary
drivers. This was created using the "INF Wizard" application which is included in the libusb-win32
download package.

```
[Version]
Signature = "$Chicago$"
provider  = %manufacturer%
DriverVer = 03/20/2007,0.1.12.1
CatalogFile = usb_dev_bulk_libusb.cat
CatalogFile.NT = usb_dev_bulk_libusb.cat
CatalogFile.NTAMD64 = usb_dev_bulk_libusb_x64.cat

Class = LibUsbDevices
ClassGUID = {EB781AAF-9C70-4523-A5DF-642A87ECA567}

[ClassInstall]
AddReg=libusb_class_install_add_reg

[ClassInstall32]
AddReg=libusb_class_install_add_reg

[libusb_class_install_add_reg]
HKR,,,,"LibUSB-Win32 Devices"
HKR,,Icon,,"-20"

[Manufacturer]
%manufacturer%=Devices,NT,NTAMD64


;--------------------------------------------------------------------------
; Files
;--------------------------------------------------------------------------

[SourceDisksNames]
1 = "Libusb-Win32 Driver Installation Disk",,

[SourceDisksFiles]
libusb0.sys = 1,,
libusb0.dll = 1,,
libusb0_x64.sys = 1,,
libusb0_x64.dll = 1,,

[DestinationDirs]
libusb_files_sys = 10,system32\drivers
libusb_files_sys_x64 = 10,system32\drivers
libusb_files_dll = 10,system32
libusb_files_dll_wow64 = 10,syswow64
libusb_files_dll_x64 = 10,system32
```

```
[libusb_files_sys]
libusb0.sys

[libusb_files_sys_x64]
libusb0.sys,libusb0_x64.sys

[libusb_files_dll]
libusb0.dll

[libusb_files_dll_wow64]
libusb0.dll

[libusb_files_dll_x64]
libusb0.dll,libusb0_x64.dll

;-------------------------------------------------------------------------
; Device driver
;-------------------------------------------------------------------------

[LIBUSB_DEV]
CopyFiles = libusb_files_sys, libusb_files_dll
AddReg    = libusb_add_reg

[LIBUSB_DEV.NT]
CopyFiles = libusb_files_sys, libusb_files_dll

[LIBUSB_DEV.NTAMD64]
CopyFiles = libusb_files_sys_x64, libusb_files_dll_wow64, libusb_files_dll_x64

[LIBUSB_DEV.HW]
DelReg = libusb_del_reg_hw
AddReg = libusb_add_reg_hw

[LIBUSB_DEV.NT.HW]
DelReg = libusb_del_reg_hw
AddReg = libusb_add_reg_hw

[LIBUSB_DEV.NTAMD64.HW]
DelReg = libusb_del_reg_hw
AddReg = libusb_add_reg_hw

[LIBUSB_DEV.NT.Services]
AddService = libusb0, 0x00000002, libusb_add_service

[LIBUSB_DEV.NTAMD64.Services]
AddService = libusb0, 0x00000002, libusb_add_service

[libusb_add_reg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,libusb0.sys

; Older versions of this .inf file installed filter drivers. They are not
; needed any more and must be removed
[libusb_del_reg_hw]
HKR,,LowerFilters
HKR,,UpperFilters

; Device properties
[libusb_add_reg_hw]
HKR,,SurpriseRemovalOK, 0x00010001, 1

;-------------------------------------------------------------------------
; Services
;-------------------------------------------------------------------------

[libusb_add_service]
```

```
DisplayName   = "LibUsb-Win32 - Kernel Driver 03/20/2007, 0.1.12.1"
ServiceType   = 1
StartType     = 3
ErrorControl  = 0
ServiceBinary = %12%\libusb0.sys


;---------------------------------------------------------------------------
; Devices
;---------------------------------------------------------------------------

[Devices]
"Generic Bulk Device"=LIBUSB_DEV, USB\VID_1cbe&PID_0003

[Devices.NT]
"Generic Bulk Device"=LIBUSB_DEV, USB\VID_1cbe&PID_0003

[Devices.NTAMD64]
"Generic Bulk Device"=LIBUSB_DEV, USB\VID_1cbe&PID_0003


;---------------------------------------------------------------------------
; Strings
;---------------------------------------------------------------------------

[Strings]
manufacturer = "Texas Instruments"
```

2.6 **Bulk Device Class Driver Definitions**

Data Structures

- tUSBDBulkDevice

Defines

- COMPOSITE_DBULK_SIZE

Functions

- void ∗ USBDBulkCompositeInit (uint32_t ui32Index, tUSBDBulkDevice ∗psBulkDevice, tCompositeEntry ∗psCompEntry)
- void ∗ USBDBulkInit (uint32_t ui32Index, tUSBDBulkDevice ∗psBulkDevice)
- uint32_t USBDBulkPacketRead (void ∗pvBulkDevice, uint8_t ∗pi8Data, uint32_t ui32Length, bool bLast)
- uint32_t USBDBulkPacketWrite (void ∗pvBulkDevice, uint8_t ∗pi8Data, uint32_t ui32Length, bool bLast)
- void USBDBulkPowerStatusSet (void ∗pvBulkDevice, uint8_t ui8Power)
- bool USBDBulkRemoteWakeupRequest (void ∗pvBulkDevice)
- uint32_t USBDBulkRxPacketAvailable (void ∗pvBulkDevice)
- void ∗ USBDBulkSetRxCBData (void ∗pvBulkDevice, void ∗pvCBData)
- void ∗ USBDBulkSetTxCBData (void ∗pvBulkDevice, void ∗pvCBData)

■ void USBDBulkTerm (void ∗pvBulkDevice)

■ uint32_t USBDBulkTxPacketAvailable (void ∗pvBulkDevice)

2.6.1 Detailed Description

The macros and functions defined in this section can be found in header file `device/usbdbulk.h`.

2.6.2 Data Structure Documentation

2.6.2.1 tUSBDBulkDevice

Definition:
```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const uint16_t ui16MaxPowermA;
    const uint8_t ui8PwrAttributes;
    const tUSBCallback pfnRxCallback;
    void *pvRxCBData;
    const tUSBCallback pfnTxCallback;
    void *pvTxCBData;
    const uint8_t *const *ppui8StringDescriptors;
    const uint32_t ui32NumStringDescriptors;
    tBulkInstance sPrivateData;
}
tUSBDBulkDevice
```

Members:

ui16VID The vendor ID that this device is to present in the device descriptor.

ui16PID The product ID that this device is to present in the device descriptor.

ui16MaxPowermA The maximum power consumption of the device, expressed in milliamps.

ui8PwrAttributes Indicates whether the device is self- or bus-powered and whether or not it supports remote wakeup. Valid values are USB_CONF_ATTR_SELF_PWR or USB_CONF_ATTR_BUS_PWR, optionally ORed with USB_CONF_ATTR_RWAKE.

pfnRxCallback A pointer to the callback function which will be called to notify the application of events related to the device's data receive channel.

pvRxCBData A client-supplied pointer which will be sent as the first parameter in all calls made to the receive channel callback, pfnRxCallback.

pfnTxCallback A pointer to the callback function which will be called to notify the application of events related to the device's data transmit channel.

pvTxCBData A client-supplied pointer which will be sent as the first parameter in all calls made to the transmit channel callback, pfnTxCallback.

ppui8StringDescriptors A pointer to the string descriptor array for this device. This array must contain pointers to the following string descriptors in this order. Language descriptor, Manufacturer name string (language 1), Product name string (language 1), Serial number string (language 1), Interface description string (language 1) and Configuration description string (language 1).

If supporting more than 1 language, the strings for indices 1 through 5 must be repeated for each of the other languages defined in the language descriptor.

ui32NumStringDescriptors The number of descriptors provided in the ppStringDescriptors array. This must be 1 + (5 * number of supported languages).

sPrivateData The private instance data for this device. This memory must not be modified by any code outside the bulk class driver.

Description:

The structure used by the application to define operating parameters for the bulk device.

2.6.3   Define Documentation

2.6.3.1   COMPOSITE_DBULK_SIZE

Definition:
```
#define COMPOSITE_DBULK_SIZE
```

Description:

The size of the memory that should be allocated to create a configuration descriptor for a single instance of the USB Bulk Device. This does not include the configuration descriptor which is automatically ignored by the composite device class.

2.6.4   Function Documentation

2.6.4.1   USBDBulkCompositeInit

Initializes bulk device operation for a given USB controller.

Prototype:
```
void *
USBDBulkCompositeInit(uint32_t ui32Index,
                      tUSBDBulkDevice *psBulkDevice,
                      tCompositeEntry *psCompEntry)
```

Parameters:

ui32Index is the index of the USB controller which is to be initialized for bulk device operation.

psBulkDevice points to a structure containing parameters customizing the operation of the bulk device.

psCompEntry is the composite device entry to initialize when creating a composite device.

Description:

This call is very similar to USBDBulkInit() except that it is used for initializing an instance of the bulk device for use in a composite device. When this bulk device is part of a composite device, then the *psCompEntry* should point to the composite device entry to initialize. This is part of the array that is passed to the USBDCompositeInit() function.

Returns:

Returns zero on failure or a non-zero value that should be used with the remaining USB Bulk APIs.

2.6.4.2 USBDBulkInit

Initializes bulk device operation for a given USB controller.

Prototype:
```
void *
USBDBulkInit(uint32_t ui32Index,
             tUSBDBulkDevice *psBulkDevice)
```

Parameters:

ui32Index is the index of the USB controller which is to be initialized for bulk device operation.

psBulkDevice points to a structure containing parameters customizing the operation of the bulk device.

Description:

An application wishing to make use of a USB bulk communication channel must call this function to initialize the USB controller and attach the device to the USB bus. This function performs all required USB initialization.

On successful completion, this function will return the *psBulkDevice* pointer passed to it. This must be passed on all future calls to the device driver related to this device.

The USBDBulk interface offers packet-based transmit and receive operation. If the application would rather use block based communication with transmit and receive buffers, USB buffers may be used above the bulk transmit and receive channels to offer this functionality.

Transmit Operation:

Calls to USBDBulkPacketWrite() must send no more than 64 bytes of data for FS USB and 512 bytes for HS USB at a time and may only be made when no other transmission is currently outstanding.

Once a packet of data has been acknowledged by the USB host, a **USB_EVENT_TX_COMPLETE** event is sent to the application callback to inform it that another packet may be transmitted.

Receive Operation:

An incoming USB data packet will result in a call to the application callback with event **USBD_EVENT_RX_AVAILABLE**. The application must then call USBDBulkPacketRead(), passing a buffer capable of holding 64 bytes for FS USB and 512 bytes for HS USB, to retrieve the data and acknowledge reception to the USB host.

Note:

The application must not make any calls to the low level USB Device API if interacting with USB via the USB bulk device class API. Doing so will cause unpredictable (though almost certainly unpleasant) behavior.

Returns:

Returns NULL on failure or void pointer that should be used with the remaining USB bulk class APSs.

2.6.4.3 USBDBulkPacketRead

Reads a packet of data received from the USB host via the bulk data interface.

Prototype:
```
uint32_t
USBDBulkPacketRead(void *pvBulkDevice,
                   uint8_t *pi8Data,
                   uint32_t ui32Length,
                   bool bLast)
```

Parameters:

    pvBulkDevice is the pointer to the device instance structure as returned by USBDBulkInit().

    pi8Data points to a buffer into which the received data will be written.

    ui32Length is the size of the buffer pointed to by pi8Data.

    bLast indicates whether the client will make a further call to read additional data from the packet.

Description:

    This function reads up to *ui32Length* bytes of data received from the USB host into the supplied application buffer. If the driver detects that the entire packet has been read, it is acknowledged to the host.

    The *bLast* parameter is ignored in this implementation since the end of a packet can be determined without relying upon the client to provide this information.

Returns:

    Returns the number of bytes of data read.

2.6.4.4   USBDBulkPacketWrite

Transmits a packet of data to the USB host via the bulk data interface.

Prototype:
```
uint32_t
USBDBulkPacketWrite(void *pvBulkDevice,
                    uint8_t *pi8Data,
                    uint32_t ui32Length,
                    bool bLast)
```

Parameters:

    pvBulkDevice is the pointer to the device instance structure as returned by USBDBulkInit().

    pi8Data points to the first byte of data which is to be transmitted.

    ui32Length is the number of bytes of data to transmit.

    bLast indicates whether more data is to be written before a packet should be scheduled for transmission. If **true**, the client will make a further call to this function. If **false**, no further call will be made and the driver should schedule transmission of a short packet.

Description:

    This function schedules the supplied data for transmission to the USB host in a single USB packet. If no transmission is currently ongoing, the data is immediately copied to the relevant USB endpoint FIFO for transmission. Whenever a USB packet is acknowledged by the host, a **USB_EVENT_TX_COMPLETE** event will be sent to the transmit channel callback indicating that more data can now be transmitted.

The maximum value for *ui32Length* is 64 bytes (the maximum USB packet size for the bulk endpoints in use by the device). Attempts to send more data than this will result in a return code of 0 indicating that the data cannot be sent.

The *bLast* parameter allows a client to make multiple calls to this function before scheduling transmission of the packet to the host. This can be helpful if, for example, constructing a packet on the fly or writing a packet which spans the wrap point in a ring buffer.

Returns:
Returns the number of bytes actually sent. At this level, this will either be the number of bytes passed (if less than or equal to the maximum packet size for the USB endpoint in use and no outstanding transmission ongoing) or 0 to indicate a failure.

2.6.4.5 USBDBulkPowerStatusSet

Reports the device power status (bus- or self-powered) to the USB library.

Prototype:
```
void
USBDBulkPowerStatusSet(void *pvBulkDevice,
                       uint8_t ui8Power)
```

Parameters:
pvBulkDevice is the pointer to the bulk device instance structure.
ui8Power indicates the current power status, either **USB_STATUS_SELF_PWR** or **USB_STATUS_BUS_PWR**.

Description:
Applications which support switching between bus- or self-powered operation should call this function whenever the power source changes to indicate the current power status to the USB library. This information is required by the USB library to allow correct responses to be provided when the host requests status from the device.

Returns:
None.

2.6.4.6 USBDBulkRemoteWakeupRequest

Requests a remote wake up to resume communication when in suspended state.

Prototype:
```
bool
USBDBulkRemoteWakeupRequest(void *pvBulkDevice)
```

Parameters:
pvBulkDevice is the pointer to the bulk device instance structure.

Description:
When the bus is suspended, an application which supports remote wake up (advertised to the host via the configuration descriptor) may call this function to initiate remote wake up signaling to the host. If the remote wake up feature has not been disabled by the host, this will cause

the bus to resume operation within 20mS. If the host has disabled remote wake up, **false** will be returned to indicate that the wake up request was not successful.

Returns:
Returns **true** if the remote wake up is not disabled and the signaling was started or **false** if remote wake up is disabled or if signaling is currently ongoing following a previous call to this function.

2.6.4.7 USBDBulkRxPacketAvailable

Determines whether a packet is available and, if so, the size of the buffer required to read it.

Prototype:
```
uint32_t
USBDBulkRxPacketAvailable(void *pvBulkDevice)
```

Parameters:
pvBulkDevice is the pointer to the device instance structure as returned by USBDBulkInit().

Description:
This function may be used to determine if a received packet remains to be read and allows the application to determine the buffer size needed to read the data.

Returns:
Returns 0 if no received packet remains unprocessed or the size of the packet if a packet is waiting to be read.

2.6.4.8 USBDBulkSetRxCBData

Sets the client-specific pointer parameter for the receive channel callback.

Prototype:
```
void *
USBDBulkSetRxCBData(void *pvBulkDevice,
                    void *pvCBData)
```

Parameters:
pvBulkDevice is the pointer to the device instance structure as returned by USBDBulkInit().
pvCBData is the pointer that client wishes to be provided on each event sent to the receive channel callback function.

Description:
The client uses this function to change the callback pointer passed in the first parameter on all callbacks to the *pfnRxCallback* function passed on USBDBulkInit().

If a client wants to make runtime changes in the callback pointer, it must ensure that the *pvBulkDevice* structure passed to USBDBulkInit() resides in RAM. If this structure is in flash, callback pointer changes are not possible.

Returns:
Returns the previous callback pointer that was being used for this instance's receive callback.

2.6.4.9 USBDBulkSetTxCBData

Sets the client-specific pointer parameter for the transmit callback.

Prototype:
```
void *
USBDBulkSetTxCBData(void *pvBulkDevice,
                    void *pvCBData)
```

Parameters:
>   ***pvBulkDevice*** is the pointer to the device instance structure as returned by USBDBulkInit().
>
>   ***pvCBData*** is the pointer that client wishes to be provided on each event sent to the transmit channel callback function.

Description:
>   The client uses this function to change the callback pointer passed in the first parameter on all callbacks to the *pfnTxCallback* function passed on USBDBulkInit().
>
>   If a client wants to make runtime changes in the callback pointer, it must ensure that the *pvBulkDevice* structure passed to USBDBulkInit() resides in RAM. If this structure is in flash, callback pointer changes are not possible.

Returns:
>   Returns the previous callback pointer that was being used for this instance's transmit callback.

2.6.4.10 USBDBulkTerm

Shut down the bulk device.

Prototype:
```
void
USBDBulkTerm(void *pvBulkDevice)
```

Parameters:
>   ***pvBulkDevice*** is the pointer to the device instance structure as returned by USBDBulkInit().

Description:
>   This function terminates device operation for the instance supplied and removes the device from the USB bus. This function should not be called if the bulk device is part of a composite device and instead the USBDCompositeTerm() function should be called for the full composite device.
>
>   Following this call, the *pvBulkDevice* instance should not me used in any other calls.

Returns:
>   None.

2.6.4.11 USBDBulkTxPacketAvailable

Returns the number of free bytes in the transmit buffer.

Prototype:
```
uint32_t
USBDBulkTxPacketAvailable(void *pvBulkDevice)
```

Parameters:
pvBulkDevice is the pointer to the device instance structure as returned by USBDBulkInit().

Description:
This function returns the maximum number of bytes that can be passed on a call to USBD-BulkPacketWrite() and accepted for transmission. The value returned will be the maximum USB packet size (64) if no transmission is currently outstanding or 0 if a transmission is in progress.

Returns:
Returns the number of bytes available in the transmit buffer.

2.7 CDC Device Class Driver

The USB Communication Device Class (CDC) class driver supports the CDC Abstract Control Model variant and allows a client application to be seen as a virtual serial port to the USB host system. The driver provides two channels, one transmit and one receive. The channels may be used in conjunction with USB buffers to provide a simple read/write interface for data transfer to and from the host. Additional APIs and events are used to support serial-link-specific operations such as notification of UART errors, sending break conditions and setting communication line parameters.

The data transmission capabilities of this device class driver are very similar to the generic bulk class but, since this is a standard device class, the host operating system is likely able to access the device without the need for any special additional device drivers. On Windows, for example, a simple INF file is all that is required to make the USB device appear as a COM port which can be accessed by any serial terminal application.

USB CDC Device Model
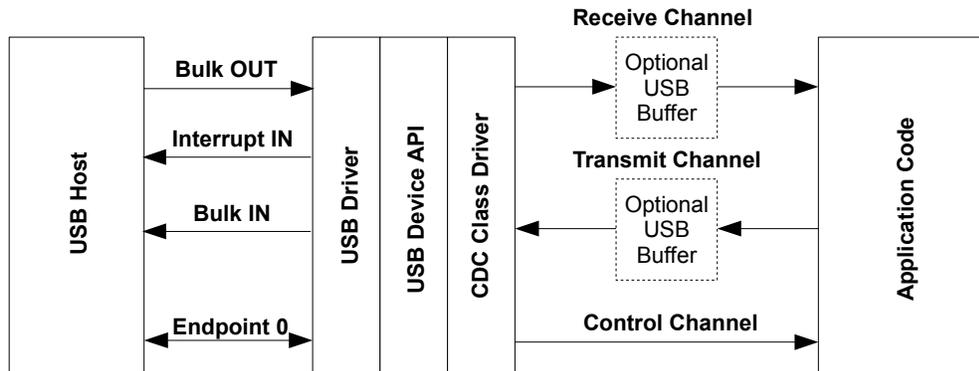
This device class uses three endpoints in addition to endpoint zero. Two bulk endpoints carry data to and from the host and an interrupt IN endpoint is used to signal any serial errors such as break, framing error or parity error detected by the device. Endpoint zero carries standard USB requests

and also CDC-specific requests which translate to events passed to the application via the control channel callback.

The `usb_dev_serial` example application makes use of this device class driver.

2.7.1 CDC Device Class Events

The CDC device class driver sends the following events to the application callback functions:

2.7.1.1 Receive Channel Events

- **USB_EVENT_RX_AVAILABLE**
- **USB_EVENT_DATA_REMAINING**
- **USB_EVENT_ERROR**

2.7.1.2 Transmit Channel Events

- **USB_EVENT_TX_COMPLETE**

2.7.1.3 Control Channel Events

- **USB_EVENT_CONNECTED**
- **USB_EVENT_DISCONNECTED**
- **USB_EVENT_SUSPEND**
- **USB_EVENT_RESUME**
- **USBD_CDC_EVENT_SEND_BREAK**
- **USBD_CDC_EVENT_CLEAR_BREAK**
- **USBD_CDC_EVENT_SET_LINE_CODING**
- **USBD_CDC_EVENT_GET_LINE_CODING**
- **USBD_CDC_EVENT_SET_CONTROL_LINE_STATE**

Note: The USB_EVENT_DISCONNECTED event is not reported to the application if the MCU's PB1/USB0VBUS pin is connected to a fixed +5 Volts rather than directly to the VBUS pin on the USB connector or if the USB controller is configured to force device mode.

2.7.2 Using the CDC Device Class Driver

To add USB CDC data transmit and receive capability to your application via the CDC Device Class Driver, take the following steps.

- Add the following header files to the source file(s) which are to support USB:

```
#include "src/usb.h"
#include "usblib/usblib.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdcdc.h"
```

■ Define the 6 entry string descriptor table which is used to describe various features of your new
device to the host system. The following is the string table taken from the `usb_dev_serial`
example application. Edit the actual strings to suit your application and take care to ensure
that you also update the length field (the first byte) of each descriptor to correctly reflect the
length of the string and descriptor header. The number of string descriptors you include must
be (1 + (5 ∗ num languages)) where the number of languages agrees with the list published
in string descriptor 0, *g_pLangDescriptor*. The strings for each language must be grouped
together with all the language 1 strings before all the language 2 strings and so on.

```
//*****************************************************************************
//
// The languages supported by this device.
//
//*****************************************************************************
const uint8_t g_pui8LangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};

//*****************************************************************************
//
// The manufacturer string.
//
//*****************************************************************************
const uint8_t g_pui8ManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};

//*****************************************************************************
//
// The product string.
//
//*****************************************************************************
const uint8_t g_pui8ProductString[] =
{
    2 + (16 * 2),
    USB_DTYPE_STRING,
    'V', 0, 'i', 0, 'r', 0, 't', 0, 'u', 0, 'a', 0, 'l', 0, ' ', 0,
    'C', 0, 'O', 0, 'M', 0, ' ', 0, 'P', 0, 'o', 0, 'r', 0, 't', 0
};

//*****************************************************************************
//
// The serial number string.
//
//*****************************************************************************
const uint8_t g_pui8SerialNumberString[] =
{
    2 + (8 * 2),
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

//*****************************************************************************
```

```
//
// The control interface description string.
//
//*****************************************************************************
const uint8_t g_pui8ControlInterfaceString[] =
{
    2 + (21 * 2),
    USB_DTYPE_STRING,
    'A', 0, 'C', 0, 'M', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 't', 0,
    'r', 0, 'o', 0, 'l', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0,
    'r', 0, 'f', 0, 'a', 0, 'c', 0, 'e', 0
};

//*****************************************************************************
//
// The configuration description string.
//
//*****************************************************************************
const uint8_t g_pui8ConfigString[] =
{
    2 + (26 * 2),
    USB_DTYPE_STRING,
    'S', 0, 'e', 0, 'l', 0, 'f', 0, ' ', 0, 'P', 0, 'o', 0, 'w', 0,
    'e', 0, 'r', 0, 'e', 0, 'd', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0,
    'f', 0, 'i', 0, 'g', 0, 'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0,
    'o', 0, 'n', 0
};

//*****************************************************************************
//
// The descriptor string table.
//
//*****************************************************************************
const uint8_t * const g_ppui8StringDescriptors[] =
{
    g_pui8LangDescriptor,
    g_pui8ManufacturerString,
    g_pui8ProductString,
    g_pui8SerialNumberString,
    g_pui8ControlInterfaceString,
    g_pui8ConfigString
};

#define NUM_STRING_DESCRIPTORS (sizeof(g_ppui8StringDescriptors) /          \
                                sizeof(uint8_t *))
```

■ Define a *tUSBDCDCDevice* structure and initialize all fields as required for your application. The following example illustrates a simple case where no USB buffers are in use. For an example using USB buffers, see the source file `usb_bulk_structs.c` in the `usb_dev_serial` example application.

```
const tUSBDCDCDevice g_sCDCDevice =
{
    //
    // The Vendor ID you have been assigned by USB-IF.
    //
    USB_VID_YOUR_VENDOR_ID,

    //
    // The product ID you have assigned for this device.
    //
    USB_PID_YOUR_PRODUCT_ID,

    //
    // The power consumption of your device in milliamps.
```

```
        //
        POWER_CONSUMPTION_MA,

        //
        // The value to be passed to the host in the USB configuration descriptor's
        // bmAttributes field.
        //
        USB_CONF_ATTR_SELF_PWR,

        //
        // A pointer to your control callback event handler.
        //
        YourUSBControlEventCallback,

        //
        // A value that you want passed to the control callback alongside every
        // event.
        //
        (void *)&g_sYourInstanceData,

        //
        // A pointer to your receive callback event handler.
        //
        YourUSBReceiveEventCallback,

        //
        // A value that you want passed to the receive callback alongside every
        // event.
        //
        (void *)&g_sYourInstanceData,

        //
        // A pointer to your transmit callback event handler.
        //
        YourUSBTransmitEventCallback,

        //
        // A value that you want passed to the transmit callback alongside every
        // event.
        //
        (void *)&g_sYourInstanceData,

        //
        // A pointer to your string table.
        //
        g_ppui8StringDescriptors,

        //
        // The number of entries in your string table.
        //
        NUM_STRING_DESCRIPTORS
    };
```

- Add a receive event handler function, YourUSBReceiveEventCallback in the previous example, to your application taking care to handle all messages which require a particular response. For the CDC device class, **USB_EVENT_RX_AVAILABLE** and **USB_EVENT_DATA_REMAINING** MUST be handled by the receive event handler.

  In response to **USB_EVENT_RX_AVAILABLE**, your handler should check the amount of data received by calling USBDCDCRxPacketAvailable() then read it using a call to USBDCDCPacketRead(). This causes the newly received data to be acknowledged to the host and instructs the host that it may now transmit another packet. If you are unable to read the data immediately, return 0 from the callback handler and you is called back once again a few milliseconds later.

On **USB_EVENT_DATA_REMAINING** the application should return the number of bytes of data it currently has buffered. This event controls timing of some incoming requests to, for example, send break conditions or change line transmission parameters. These requests are held off until all previously received data has been processed so it is important to ensure that this event returns 0 only once any application buffers are empty.

Although no other events must be handled, **USB_EVENT_CONNECTED** and **USB_EVENT_DISCONNECTED** is typically required since these indicate when a host connects or disconnects and allow the application to flush any buffers or reset state as required. Attempts to send data when the host is disconnected result in an immediate failure.

- Add a transmit event handler function, YourUSBTransmitEventCallback in the previous example, to your application taking care to handle all messages which require a particular response. For the CDC device class, there are no events sent to the transmit callback which MUST be handled but applications usually want to note **USB_EVENT_TX_COMPLETE** since this is an interlock message indicating that the previous packet sent has been acknowledged by the host and a new packet can now be sent.

- Add a control event handler function, YourUSBControlEventCallback in the previous example, to your application and ensure that you handle **USBD_CDC_EVENT_GET_LINE_CODING**, returning a valid line coding configuration even if your device is not actually driving a UART. Handle the other control events as required for your application.

- From your main initialization function call the CDC device class driver initialization function to configure the USB controller and place the device on the bus.

```
pvDevice = USBDCDCInit(0, &g_sCDCDevice);
```

- Assuming *pvDevice* returned is not NULL, your device is now ready to communicate with a USB host.

- Once the host connects, your control event handler is sent **USB_EVENT_CONNECTED** and the first packet of data may be sent to the host using USBDCDCPacketWrite() with following packets transmitted as soon as **USB_EVENT_TX_COMPLETE** is received via the transmit event handler.

2.7.3   Using the Composite CDC Serial Device Class

When using the CDC serial device class in a composite, the configuration of the device is very similar to how it is configured as a non-composite device. Follow all of the configuration steps in the previous section with the exception of calling USBDCDCCompositeInit() instead of USBDCDCInit(). This prepares an instance of the CDC serial device class to be enumerated as part of a composite device. The USBDCDCCompositeInit() function takes the CDC serial device structure and a pointer to a tCompositeEntry value so that it can properly initialize the CDC serial device and the composite entry that is passed to the USBDCompositeInit() funtion. The code example below provides an example of how to initialize an CDC serial device to be a part of a composite device.

```
//
// These should be initialized with valid values for each class.
//
extern tUSBDCDCDevice g_sCDCDevice;
void *pvCDCDevice;

//
// The array of composited devices.
//
tCompositeEntry psCompEntries[2];
```

```
//
// Allocate the device data for the top level composite device class.
//
tUSBDCompositeDevice g_sCompDevice =
{
    //
    // Texas Intruments C-Series VID.
    //
    USB_VID_TI_1CBE,

    //
    // Texas Intruments C-Series PID for composite serial device.
    //
    USB_PID_YOUR_COMPOSITE_PID,

    //
    // This is in 2mA increments so 500mA.
    //
    250,

    //
    // Bus powered device.
    //
    USB_CONF_ATTR_BUS_PWR,

    //
    // Composite event handler.
    //
    EventHandler,

    //
    // The string table.
    //
    g_pui8StringDescriptors,
    NUM_STRING_DESCRIPTORS,

    //
    // The Composite device array.
    //
    2,
    g_psCompEntries
};

//
// The OTHER_SIZES here are the sizes of the descriptor data for other classes
// that are part of the composite device.
//
#define DESCRIPTOR_DATA_SIZE    (COMPOSITE_DCDC_SIZE + OTHER_SIZES)
uint8_t g_pui8DescriptorData[DESCRIPTOR_DATA_SIZE];

//
// Save the instance data for this CDC serial device.
//
pvCDCDevice = USBDCDCCompositeInit(0, &g_sCDCDevice, &psCompEntries[0]);

...

//
// Initialize the USB controller as a composite device.
//
USBDCompositeInit(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
                    g_pui8DescriptorData);
```

All other API calls to the USB CDC serial device class should use the value returned by USBD-
CDCCompositeInit() when the API calls for a pvInstance pointer. Also when using the CDC serial

device in a composite device the **COMPOSITE_DCDC_SIZE** value should be added to the size of the *g_pui8DescriptorData* array as shown in the example above.

2.7.4   Windows Drivers for CDC Serial Devices

Making your CDC serial) device visible as a virtual COM port on a Windows system is very straight-forward since Windows already includes a device driver supporting USB CDC devices. The device developer must merely provide a single INF file to associate the VID and PID of the new device with the Windows USB CDC driver, usbser.sys. When using the serial device in a composite device it is important to remember to append &MI_xx value to the VID/PID entry as shown in the example below. The actual number used with the MI_∗ value is the interface number assigned to the serial device. An example INF file is provided below. Unlike the case for the generic bulk device class, no additional installation files are necessary since the CDC serial driver is already installed by default and does not, therefore, have to be redistributed by the device developer.

```
;
;    Texas Instruments USBLib USB CDC (serial) driver installation file.
;
[Version]
Signature="$Windows NT$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%MFGNAME%
LayoutFile=layout.inf
DriverVer=08/17/2001,5.1.2600.0

[Manufacturer]
%MFGNAME%=DeviceList

[DestinationDirs]
DefaultDestDir=12

[SourceDisksFiles]

[SourceDisksNames]

;
; NOTE: Change the VID and PID in the following section to match your device.
; The values with the &MI_xx values are for the composite serial devices
; examples.
;

[DeviceList]
;
; This entry is for the single serial port example usb_dev_serial.
;
%DESCRIPTION_0%=DriverInstall,USB\VID_1CBE&PID_0002

;
; These entries are for the dual serial port composite example usb_dev_cserial.
;
%DESCRIPTION_0%=DriverInstall,USB\VID_1CBE&PID_0007&MI_00
%DESCRIPTION_1%=DriverInstall,USB\VID_1CBE&PID_0007&MI_01

;
; This entry is for the composite hid/serial device usb_dev_chidcdc.  Notice
; that the value is MI_01 because the serial device is on interface 1.
;
%DESCRIPTION_1%=DriverInstall,USB\VID_1CBE&PID_0009&MI_01
```

```
;-------------------------------------------------------------------------------
;  Windows XP/2000 Sections
;-------------------------------------------------------------------------------

[DriverInstall.nt]
CopyFiles=DriverCopyFiles
AddReg=DriverInstall.nt.AddReg

[DriverCopyFiles]
usbser.sys,,,0x20

[DriverInstall.nt.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,usbser.sys
HKR,,EnumPropPages32,,"MsPorts.dll,SerialPortPropPageProvider"

[DriverInstall.nt.Services]
AddService=usbser, 0x00000002, DriverService

[DriverService]
DisplayName=%SERVICE%
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%12%\usbser.sys


;-------------------------------------------------------------------------------
;  String Definitions (change for your device)
;-------------------------------------------------------------------------------

[Strings]
MFGNAME       = "Texas Instruments"
DESCRIPTION_0 = "USB Serial Port"
DESCRIPTION_1 = "USB Serial Command Port"
SERVICE       = "USB CDC serial port"
```

2.8   CDC Device Class Driver Definitions

Data Structures

- tLineCoding
- tUSBDCDCDevice

Defines

- COMPOSITE_DCDC_SIZE
- USBD_CDC_EVENT_CLEAR_BREAK
- USBD_CDC_EVENT_GET_LINE_CODING
- USBD_CDC_EVENT_SEND_BREAK
- USBD_CDC_EVENT_SET_CONTROL_LINE_STATE
- USBD_CDC_EVENT_SET_LINE_CODING

Functions

- void ∗ USBDCDCCompositeInit (uint32_t ui32Index, tUSBDCDCDevice ∗psCDCDevice, tCompositeEntry ∗psCompEntry)
- void ∗ USBDCDCInit (uint32_t ui32Index, tUSBDCDCDevice ∗psCDCDevice)
- uint32_t USBDCDCPacketRead (void ∗pvCDCDevice, uint8_t ∗pi8Data, uint32_t ui32Length, bool bLast)
- uint32_t USBDCDCPacketWrite (void ∗pvCDCDevice, uint8_t ∗pi8Data, uint32_t ui32Length, bool bLast)
- void USBDCDCPowerStatusSet (void ∗pvCDCDevice, uint8_t ui8Power)
- bool USBDCDCRemoteWakeupRequest (void ∗pvCDCDevice)
- uint32_t USBDCDCRxPacketAvailable (void ∗pvCDCDevice)
- void USBDCDCSerialStateChange (void ∗pvCDCDevice, uint16_t ui16State)
- void ∗ USBDCDCSetControlCBData (void ∗pvCDCDevice, void ∗pvCBData)
- void ∗ USBDCDCSetRxCBData (void ∗pvCDCDevice, void ∗pvCBData)
- void ∗ USBDCDCSetTxCBData (void ∗pvCDCDevice, void ∗pvCBData)
- void USBDCDCTerm (void ∗pvCDCDevice)
- uint32_t USBDCDCTxPacketAvailable (void ∗pvCDCDevice)

2.8.1 Detailed Description

The macros and functions defined in this section can be found in header file `device/usbdcdc.h`. Users of the CDC device class driver also needs to include `usbcdc.h` which contains general CDC definitions required by both host and device implementations.

2.8.2 Data Structure Documentation

2.8.2.1 tLineCoding

Definition:
```
typedef struct
{
    uint32_t ui32Rate;
    uint8_t ui8Stop;
    uint8_t ui8Parity;
    uint8_t ui8Databits;
}
tLineCoding
```

Members:

ui32Rate The data terminal rate in bits per second.

ui8Stop The number of stop bits.   Valid values are USB_CDC_STOP_BITS_1, USB_CDC_STOP_BITS_1_5 or USB_CDC_STOP_BITS_2

ui8Parity The parity setting.   Valid values are USB_CDC_PARITY_NONE, USB_CDC_PARITY_ODD, USB_CDC_PARITY_EVEN, USB_CDC_PARITY_MARK and USB_CDC_PARITY_SPACE.

ui8Databits The number of data bits per character. Valid values are 5, 6, 7 and 8 in this implementation.

Description:
>   USB_CDC_GET/SET_LINE_CODING request-specific data.

2.8.2.2   tUSBDCDCDevice

Definition:
```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const uint16_t ui16MaxPowermA;
    const uint8_t ui8PwrAttributes;
    const tUSBCallback pfnControlCallback;
    void *pvControlCBData;
    const tUSBCallback pfnRxCallback;
    void *pvRxCBData;
    const tUSBCallback pfnTxCallback;
    void *pvTxCBData;
    const uint8_t *const *ppui8StringDescriptors;
    const uint32_t ui32NumStringDescriptors;
    tCDCSerInstance sPrivateData;
}
tUSBDCDCDevice
```

Members:
>   ***ui16VID*** The vendor ID that this device is to present in the device descriptor.
>
>   ***ui16PID*** The product ID that this device is to present in the device descriptor.
>
>   ***ui16MaxPowermA*** The maximum power consumption of the device, expressed in milliamps.
>
>   ***ui8PwrAttributes*** Indicates whether the device is self- or bus-powered and whether or not it supports remote wakeup. Valid values are USB_CONF_ATTR_SELF_PWR or USB_CONF_ATTR_BUS_PWR, optionally ORed with USB_CONF_ATTR_RWAKE.
>
>   ***pfnControlCallback*** A pointer to the callback function which will be called to notify the application of all asynchronous control events related to the operation of the device.
>
>   ***pvControlCBData*** A client-supplied pointer which will be sent as the first parameter in all calls made to the control channel callback, pfnControlCallback.
>
>   ***pfnRxCallback*** A pointer to the callback function which will be called to notify the application of events related to the device's data receive channel.
>
>   ***pvRxCBData*** A client-supplied pointer which will be sent as the first parameter in all calls made to the receive channel callback, pfnRxCallback.
>
>   ***pfnTxCallback*** A pointer to the callback function which will be called to notify the application of events related to the device's data transmit channel.
>
>   ***pvTxCBData*** A client-supplied pointer which will be sent as the first parameter in all calls made to the transmit channel callback, pfnTxCallback.
>
>   ***ppui8StringDescriptors*** A pointer to the string descriptor array for this device. This array must contain the following string descriptor pointers in this order. Language descriptor, Manufacturer name string (language 1), Product name string (language 1), Serial number string (language 1), Control interface description string (language 1), Configuration description string (language 1).
>   If supporting more than 1 language, the strings for indices 1 through 5 must be repeated for each of the other languages defined in the language descriptor.

ui32NumStringDescriptors The number of descriptors provided in the ppStringDescriptors array. This must be 1 + (5 * number of supported languages).

sPrivateData The private instance data for this device. This memory must remain accessible for as long as the CDC device is in use and must not be modified by any code outside the CDC class driver.

Description:

The structure used by the application to define operating parameters for the CDC device.

2.8.3   Define Documentation

2.8.3.1   COMPOSITE_DCDC_SIZE

Definition:

```
#define COMPOSITE_DCDC_SIZE
```

Description:

The size of the memory that should be allocated to create a configuration descriptor for a single instance of the USB Serial CDC Device. This does not include the configuration descriptor which is automatically ignored by the composite device class.

2.8.3.2   USBD_CDC_EVENT_CLEAR_BREAK

Definition:

```
#define USBD_CDC_EVENT_CLEAR_BREAK
```

Description:

The host requests that the device stop sending a BREAK condition on its serial communication channel.

2.8.3.3   USBD_CDC_EVENT_GET_LINE_CODING

Definition:

```
#define USBD_CDC_EVENT_GET_LINE_CODING
```

Description:

The host is querying the current RS232 communication parameters. The pvMsgData parameter points to a tLineCoding structure that the application must fill with the current settings prior to returning from the callback.

2.8.3.4   USBD_CDC_EVENT_SEND_BREAK

Definition:

```
#define USBD_CDC_EVENT_SEND_BREAK
```

Description:
>   The host requests that the device send a BREAK condition on its serial communication chan-
>   nel. The BREAK should remain active until a USBD_CDC_EVENT_CLEAR_BREAK event is
>   received.

2.8.3.5 USBD_CDC_EVENT_SET_CONTROL_LINE_STATE

Definition:
>   #define USBD_CDC_EVENT_SET_CONTROL_LINE_STATE

Description:
>   The host requests that the device set the RS232 signaling lines to a particular state. The
>   ui32MsgValue parameter contains the RTS and DTR control line states as defined in table 51
>   of the USB CDC class definition and is a combination of the following values:
>
>   (RTS) USB_CDC_DEACTIVATE_CARRIER or USB_CDC_ACTIVATE_CARRIER (DTR)
>   USB_CDC_DTE_NOT_PRESENT or USB_CDC_DTE_PRESENT

2.8.3.6 USBD_CDC_EVENT_SET_LINE_CODING

Definition:
>   #define USBD_CDC_EVENT_SET_LINE_CODING

Description:
>   The host requests that the device set the RS232 communication parameters. The pvMsgData
>   parameter points to a tLineCoding structure defining the required number of bits per character,
>   parity mode, number of stop bits and the baud rate.

2.8.4 Function Documentation

2.8.4.1 USBDCDCCompositeInit

Initializes CDC device operation when used with a composite device.

Prototype:
```
void *
USBDCDCCompositeInit(uint32_t ui32Index,
                     tUSBDCDCDevice *psCDCDevice,
                     tCompositeEntry *psCompEntry)
```

Parameters:
>   ***ui32Index*** is the index of the USB controller in use.
>
>   ***psCDCDevice*** points to a structure containing parameters customizing the operation of the
>   CDC device.
>
>   ***psCompEntry*** is the composite device entry to initialize when creating a composite device.

Description:
>   This call is very similar to USBDCDCInit() except that it is used for initializing an instance of the
>   serial device for use in a composite device. When this CDC serial device is part of a composite

device, then the *psCompEntry* should point to the composite device entry to initialize. This is part of the array that is passed to the USBDCompositeInit() function.

Returns:

Returns zero on failure or a non-zero instance value that should be used with the remaining USB CDC APIs.

2.8.4.2 USBDCDCInit

Initializes CDC device operation for a given USB controller.

Prototype:
```
void *
USBDCDCInit(uint32_t ui32Index,
            tUSBDCDCDevice *psCDCDevice)
```

Parameters:

ui32Index is the index of the USB controller which is to be initialized for CDC device operation.

psCDCDevice points to a structure containing parameters customizing the operation of the CDC device.

Description:

An application wishing to make use of a USB CDC communication channel and appear as a virtual serial port on the host system must call this function to initialize the USB controller and attach the device to the USB bus. This function performs all required USB initialization.

The value returned by this function is the *psCDCDevice* pointer passed to it if successful. This pointer must be passed to all later calls to the CDC class driver to identify the device instance.

The USB CDC device class driver offers packet-based transmit and receive operation. If the application would rather use block based communication with transmit and receive buffers, USB buffers on the transmit and receive channels may be used to offer this functionality.

Transmit Operation:

Calls to USBDCDCPacketWrite() must send no more than 64 bytes of data at a time and may only be made when no other transmission is currently outstanding.

Once a packet of data has been acknowledged by the USB host, a **USB_EVENT_TX_COMPLETE** event is sent to the application callback to inform it that another packet may be transmitted.

Receive Operation:

An incoming USB data packet will result in a call to the application callback with event **USB_EVENT_RX_AVAILABLE**. The application must then call USBDCDCPacketRead(), passing a buffer capable of holding the received packet to retrieve the data and acknowledge reception to the USB host. The size of the received packet may be queried by calling USBDCDCRxPacketAvailable().

Note:

The application must not make any calls to the low level USB Device API if interacting with USB via the CDC device class API. Doing so will cause unpredictable (though almost certainly unpleasant) behavior.

Returns:
>   Returns NULL on failure or the psCDCDevice pointer on success.

2.8.4.3   USBDCDCPacketRead

Reads a packet of data received from the USB host via the CDC data interface.

Prototype:
```
uint32_t
USBDCDCPacketRead(void *pvCDCDevice,
                  uint8_t *pi8Data,
                  uint32_t ui32Length,
                  bool bLast)
```

Parameters:
>   ***pvCDCDevice*** is the pointer to the device instance structure as returned by USBDCDCInit().
>   ***pi8Data*** points to a buffer into which the received data will be written.
>   ***ui32Length*** is the size of the buffer pointed to by *pi8Data*.
>   ***bLast*** indicates whether the client will make a further call to read additional data from the
>   packet.

Description:
>   This function reads up to ui32Length bytes of data received from the USB host into the supplied
>   application buffer.

Note:
>   The *bLast* parameter is ignored in this implementation since the end of a packet can be deter-
>   mined without relying upon the client to provide this information.

Returns:
>   Returns the number of bytes of data read.

2.8.4.4   USBDCDCPacketWrite

Transmits a packet of data to the USB host via the CDC data interface.

Prototype:
```
uint32_t
USBDCDCPacketWrite(void *pvCDCDevice,
                   uint8_t *pi8Data,
                   uint32_t ui32Length,
                   bool bLast)
```

Parameters:
>   ***pvCDCDevice*** is the pointer to the device instance structure as returned by USBDCDCInit().
>   ***pi8Data*** points to the first byte of data which is to be transmitted.
>   ***ui32Length*** is the number of bytes of data to transmit.
>   ***bLast*** indicates whether more data is to be written before a packet should be scheduled for
>   transmission. If **true**, the client will make a further call to this function. If **false**, no further
>   call will be made and the driver should schedule transmission of a short packet.

Description:

This function schedules the supplied data for transmission to the USB host in a single USB packet. If no transmission is currently ongoing the data is immediately copied to the relevant USB endpoint FIFO. If the *bLast* parameter is **true**, the newly written packet is then scheduled for transmission. Whenever a USB packet is acknowledged by the host, a **USB_EVENT_TX_COMPLETE** event will be sent to the application transmit callback indicating that more data can now be transmitted.

The maximum value for *ui32Length* is 64 bytes (the maximum USB packet size for the bulk endpoints in use by CDC). Attempts to send more data than this will result in a return code of 0 indicating that the data cannot be sent.

Returns:

Returns the number of bytes actually sent. At this level, this will either be the number of bytes passed (if less than or equal to the maximum packet size for the USB endpoint in use and no outstanding transmission ongoing) or 0 to indicate a failure.

2.8.4.5 USBDCDCPowerStatusSet

Reports the device power status (bus- or self-powered) to the USB library.

Prototype:
```
void
USBDCDCPowerStatusSet(void *pvCDCDevice,
                      uint8_t ui8Power)
```

Parameters:

pvCDCDevice is the pointer to the CDC device instance structure.

ui8Power indicates the current power status, either **USB_STATUS_SELF_PWR** or **USB_STATUS_BUS_PWR**.

Description:

Applications which support switching between bus- or self-powered operation should call this function whenever the power source changes to indicate the current power status to the USB library. This information is required by the USB library to allow correct responses to be provided when the host requests status from the device.

Returns:

None.

2.8.4.6 USBDCDCRemoteWakeupRequest

Requests a remote wakeup to resume communication when in suspended state.

Prototype:
```
bool
USBDCDCRemoteWakeupRequest(void *pvCDCDevice)
```

Parameters:

pvCDCDevice is the pointer to the CDC device instance structure.

Description:

When the bus is suspended, an application which supports remote wakeup (advertised to the host via the configuration descriptor) may call this function to initiate remote wakeup signaling to the host. If the remote wakeup feature has not been disabled by the host, this will cause the bus to resume operation within 20mS. If the host has disabled remote wakeup, **false** will be returned to indicate that the wakeup request was not successful.

Returns:

Returns **true** if the remote wakeup is not disabled and the signaling was started or **false** if remote wakeup is disabled or if signaling is currently ongoing following a previous call to this function.

2.8.4.7 USBDCDCRxPacketAvailable

Determines whether a packet is available and, if so, the size of the buffer required to read it.

Prototype:
```
uint32_t
USBDCDCRxPacketAvailable(void *pvCDCDevice)
```

Parameters:

pvCDCDevice is the pointer to the device instance structure as returned by USBDCDCInit().

Description:

This function may be used to determine if a received packet remains to be read and allows the application to determine the buffer size needed to read the data.

Returns:

Returns 0 if no received packet remains unprocessed or the size of the packet if a packet is waiting to be read.

2.8.4.8 USBDCDCSerialStateChange

Informs the CDC module of changes in the serial control line states or receive error conditions.

Prototype:
```
void
USBDCDCSerialStateChange(void *pvCDCDevice,
                         uint16_t ui16State)
```

Parameters:

pvCDCDevice is the pointer to the device instance structure as returned by USBDCDCInit().

ui16State indicates the states of the various control lines and any receive errors detected. Bit definitions are as for the USB CDC SerialState asynchronous notification and are defined in header file usbcdc.h.

Description:

The application should call this function whenever the state of any of the incoming RS232 handshake signals changes or in response to a receive error or break condition. The *ui16State* parameter is the ORed combination of the following flags with each flag indicating the presence of that condition.

- USB_CDC_SERIAL_STATE_OVERRUN
- USB_CDC_SERIAL_STATE_PARITY
- USB_CDC_SERIAL_STATE_FRAMING
- USB_CDC_SERIAL_STATE_RING_SIGNAL
- USB_CDC_SERIAL_STATE_BREAK
- USB_CDC_SERIAL_STATE_TXCARRIER
- USB_CDC_SERIAL_STATE_RXCARRIER

This function should be called only when the state of any flag changes.

Returns:
None.

2.8.4.9 USBDCDCSetControlCBData

Sets the client-specific pointer for the control callback.

Prototype:
```
void *
USBDCDCSetControlCBData(void *pvCDCDevice,
                        void *pvCBData)
```

Parameters:
pvCDCDevice is the pointer to the device instance structure as returned by USBDCDCInit().

pvCBData is the pointer that client wishes to be provided on each event sent to the control channel callback function.

Description:
The client uses this function to change the callback pointer passed in the first parameter on all callbacks to the *pfnControlCallback* function passed on USBDCDCInit().

If a client wants to make runtime changes in the callback pointer, it must ensure that the psCDCDevice structure passed to USBDCDCInit() resides in RAM. If this structure is in flash, callback pointer changes will not be possible.

Returns:
Returns the previous callback pointer that was being used for this instance's control callback.

2.8.4.10 USBDCDCSetRxCBData

Sets the client-specific data parameter for the receive channel callback.

Prototype:
```
void *
USBDCDCSetRxCBData(void *pvCDCDevice,
                   void *pvCBData)
```

Parameters:
pvCDCDevice is the pointer to the device instance structure as returned by USBDCDCInit().

pvCBData is the pointer that client wishes to be provided on each event sent to the receive channel callback function.

Description:

The client uses this function to change the callback pointer passed in the first parameter on all callbacks to the *pfnRxCallback* function passed on USBDCDCInit().

If a client wants to make runtime changes in the callback pointer, it must ensure that the psCDCDevice structure passed to USBDCDCInit() resides in RAM. If this structure is in flash, callback data changes will not be possible.

Returns:

Returns the previous callback pointer that was being used for this instance's receive callback.

2.8.4.11 USBDCDCSetTxCBData

Sets the client-specific data parameter for the transmit callback.

Prototype:

```
void *
USBDCDCSetTxCBData(void *pvCDCDevice,
                   void *pvCBData)
```

Parameters:

pvCDCDevice is the pointer to the device instance structure as returned by USBDCDCInit().

pvCBData is the pointer that client wishes to be provided on each event sent to the transmit channel callback function.

Description:

The client uses this function to change the callback pointer passed in the first parameter on all callbacks to the *pfnTxCallback* function passed on USBDCDCInit().

If a client wants to make runtime changes in the callback pointer, it must ensure that the psCDCDevice structure passed to USBDCDCInit() resides in RAM. If this structure is in flash, callback data changes will not be possible.

Returns:

Returns the previous callback pointer that was being used for this instance's transmit callback.

2.8.4.12 USBDCDCTerm

Shuts down the CDC device instance.

Prototype:

```
void
USBDCDCTerm(void *pvCDCDevice)
```

Parameters:

pvCDCDevice is the pointer to the device instance structure as returned by USBDCDCInit().

Description:

This function terminates CDC operation for the instance supplied and removes the device from the USB bus. This function should not be called if the CDC device is part of a composite device and instead the USBDCompositeTerm() function should be called for the full composite device.

Following this call, the *pvCDCDevice* instance should not me used in any other calls.

Returns:

None.

2.8.4.13 USBDCDCTxPacketAvailable

Returns the number of free bytes in the transmit buffer.

Prototype:

```
uint32_t
USBDCDCTxPacketAvailable(void *pvCDCDevice)
```

Parameters:

pvCDCDevice is the pointer to the device instance structure as returned by USBDCDCInit().

Description:

This function returns the maximum number of bytes that can be passed on a call to USBDCD-CPacketWrite() and accepted for transmission. The value returned will be the maximum USB packet size if no transmission is currently outstanding or 0 if a transmission is in progress.

Returns:

Returns the number of bytes available in the transmit buffer.

2.9 Composite Device Class Driver

The USB composite device class allows classes that are already defined in the USB library to be combined into a single composite device. The device configuration descriptors for the included device classes are merged at run time and returned to the USB host controller during device enumeration as a single composite USB device. Since each device class requires some unique initialization, the device classes provide a separate initialization API that does not touch the USB controller but does perform all other initialization. The initialization of the USB controller is deferred until the USB composite device is initialized and has merged the multiple device configuration descriptors into a single configuration descriptor so that it can properly initialize the USB controller. The endpoint numbers, interface numbers, and string indexes that are included in the device configuration descriptors are modified by the USB composite device class so that the values are valid in the composite device configuration descriptor.

2.9.1 Defining a Composite Device

The USB composite device class is defined at the top level in the tUSBDCompositeDevice structure which is used to describe the class to the USB library. In order for the USB composite device to enumerate and function properly, all members of this structure must be filled with valid information. The usVID and usPID values should have valid Vendor ID and Product ID values for the

composite device. The power requirements for the device as specified in the usMaxPowermA and ucPwrAttributes and should take into account the power requirements and settings for all devices classes that the composite device is using. The only truly optional member of the tUSBDCompositeDevice structure is the pfnCallback function which provides notifications to the application that are not handled by the individual device classes. The device specific strings should be included in the ppui8StringDescriptors and *ui32NumStringDescriptors* members. This list of strings should include the following three strings in the following order: Manufacturer, Product, and Product serial number. All other strings used by the classes are specified and are sourced from the included device classes. The psPrivateData should be set to point to a tCompositeInstance structure which provides the composite class with memory for its instance data.

Note: It is important to insure that the microcontroller has enough endpoints to satisfy the number of devices included in the composite class.

Example:

```
uint32_t g_pui32CompWorkspace[NUM_DEVICES];

tUSBDCompositeDevice g_sCompDevice =
{
    //
    // Vendor ID.
    //
    VENDOR_ID,

    //
    // Product ID.
    //
    VENDOR_PRODUCT_ID,

    //
    // This is in 2mA increments or 500mA.
    //
    250,

    //
    // Bus powered device.
    //
    USB_CONF_ATTR_BUS_PWR,

    //
    // Generic USB handler for the composite device.
    //
    CompositeHandler,

    //
    // The string table.
    //
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTORS,

    //
    // The number of device classes in the composite entry array.
    //
    NUM_DEVICES,
    g_psCompDevices
};
```

2.9.2   Allocating Memory

The USB composite device class requires three different types of memory allocated to properly enumerate and function with the included device classes. The main allocation is a block of memory that is used to build up the combined device configuration descriptor for the combination of the desired device classes. The individual device classes provides a size in a **COMPOSITE_∗_SIZE** macro that indicates the size in bytes required to hold the configuration descriptor for the device class. This allows the application to provide a large enough buffer to the USBDCompositeInit() function for merging the device descriptors.

2.9.2.1   Defining Device Class Instances

When defining a composite device the application must determine the size of the buffer that is passed into the USBDCompositeInit() function. For example, if a composite device is made up of two serial devices then a buffer of size (**COMPOSITE_DCDC_SIZE** ∗ 2) should be passed into the initialization routine and an array of that size should be declared in the application.

```
uint8_t pucDesciptorData[COMPOSITE_DCDC_SIZE*2];
```

The application must also allocate separate serial device structure for each instance of the devices that are included in a composite device. This is true even when including two devices classes of the same type are added so that the instances can be differentiated by the USB library. The USB composite device class can determine which instance to use based on the interface number that is accessed by the host controller. The application initializes the data in the array of tCompositeEntry structures passed into the composite initialization for the class.

Example: Two serial instances and the composite device array.

```
extern tUSBDCDCDevice g_sCDCDeviceA;
extern tUSBDCDCDevice g_sCDCDeviceB;

tCompositeEntry g_psDevices[2];
```

2.9.2.2   Interface Handling

The device class interfaces are merged into the composite device descriptor and the composite class modifies the default interface assignments to insure monotonically increasing indexes for all of the included interfaces. In the example above for the two serial ports, the first serial device would be interface 0 and the second would enumerate as interface 1.

2.9.2.3   String Handling

The device class strings are merged into the composite device descriptor which requires that the composite class modify the default string indexes. In doing this it always ignores the three default string indexes in the device descriptor. The remaining string indexes are modified to match in the configuration descriptor.

2.9.3   Example Composite Device

This section continues with the example above that used two USB device serial classes in a single device. This includes more detailed examples and code that demonstrate the configuration and setup needed for a composite serial device.

2.9.3.1   Composite Device Instance

The application must first allocate two serial device structures and pass them into the composite initialization function for the USB serial CDC device. The allocation and initialization are shown below:

```
//
// Buffers for serial device A.
//
const tUSBBuffer g_sTxBufferA;
const tUSBBuffer g_sRxBufferA;

//
// Buffers for serial device B.
//
const tUSBBuffer g_sTxBufferB;
const tUSBBuffer g_sRxBufferB;

//
// Device description for Serial Device A.
//
const tUSBDCDCDevice g_sCDCDeviceA =
{
    USB_VID_TI_1CBE,
    USB_PID_SERIAL,
    0,
    USB_CONF_ATTR_SELF_PWR,
    ControlHandler,
    (void *)&g_sCDCDeviceA,
    USBBufferEventCallback,
    (void *)&g_sRxBufferA,
    USBBufferEventCallback,
    (void *)&g_sTxBufferA,
    0,
    0
};

//
// Device description for Serial Device B.
//
const tUSBDCDCDevice g_sCDCDeviceB =
{
    USB_VID_TI_1CBE,
    USB_PID_SERIAL,
    0,
    USB_CONF_ATTR_SELF_PWR,
    ControlHandler,
    (void *)&g_sCDCDeviceB,
    USBBufferEventCallback,
    (void *)&g_sRxBufferB,
    USBBufferEventCallback,
    (void *)&g_sTxBufferB,
    0,
    0
};
```

Now the application must allocate the device array so that it is provided to the USB composite device class. The following code shows the allocation of the composite device array that holds the data for the two serial devices.

```
tCompositeEntry g_psDevices[2];
```

Once the array of devices has been allocated, this array is included in the USB composite device structure when the device structure is allocated and initialized. The code below shows this allocation:

```
//
// Initialize the USB composite device structure.
//
tUSBDCompositeDevice g_sCompDevice =
{
    //
    // TI USBLib VID.
    //
    USB_VID_TI_1CBE,

    //
    // PID for the composite serial device.
    //
    USB_PID_COMP_SERIAL,

    //
    // This is in 2mA increments so 500mA.
    //
    250,

    //
    // Bus powered device.
    //
    USB_CONF_ATTR_BUS_PWR,

    //
    // Generic USB handler for the composite device.
    //
    CompositeHandler,

    //
    // The string table.
    //
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTORS,

    //
    // Include the array of composite devices.
    //
    NUM_DEVICES,
    g_psCompDevices
};
```

The last bit of memory that needs to be allocated is the USB composite device descriptor workspace which is provided at Initialization time. The allocation for two serial devices is shown below:

```
uint8_t pucDesciptorData[COMPOSITE_DCDC_SIZE*2];
```

Once all of the memory has been initialized and the appropriate memory allocated, the application must call the initialization functions for each device instance. In the case of the serial ports, the USB buffers used must also first be initialized before completing initialization.

```
//
// Initialize the transmit and receive buffers.
//
USBBufferInit((tUSBBuffer *)&g_sTxBufferA);
USBBufferInit((tUSBBuffer *)&g_sRxBufferA);
USBBufferInit((tUSBBuffer *)&g_sTxBufferB);
USBBufferInit((tUSBBuffer *)&g_sRxBufferB);

//
// Initialize the two serial port instances that are part of this composite
// device.
//
pvSerialDeviceA =
    USBDCDCCompositeInit(0, &g_sCDCDeviceA, &g_psCompDevices[0]);
pvSerialDeviceB =
    USBDCDCCompositeInit(0, &g_sCDCDeviceB, &g_psCompDevices[1]);

//
// Pass the device information to the USB library and place the device
// on the bus.
//
USBDCompositeInit(0, &g_sCompDevice, COMPOSITE_DCDC_SIZE*2,
                  pucDesciptorData);
```

When calling the USB device classes that are included with a composite device, the instance data for that class should be passed into the API. In the composite serial example that is being described in this section, the USB serial device classes provide the same callback function, Control-Handler(). The callback information for this was the device class structure which was specified as g_sCDCDeviceA or g_sCDCDeviceB for the serial devices. Since the device instance is different for each serial device, the application can simply cast the pointer to a pointer of type tUSBDCDCDevice and use the data directly as shown below and only access the requested device:

```
uint32_t
ControlHandler(void *pvCBData, uint32_t ui32Event,
               uint32_t ui32MsgValue, void *pvMsgData)
{
    tUSBDCDCDevice pCDCDevice;

    pCDCDevice = (tUSBDCDCDevice *)pvCBData;

    //
    // Which event are we being asked to process?
    //
    switch(ui32Event)
    {
        ...
    }
}
```

2.10   Composite Device Class Driver Definitions

Data Structures

- tUSBDCompositeDevice

Functions

- void * [USBDCompositeInit](uint32_t ui32Index, [tUSBDCompositeDevice](*psDevice, uint32_t) ui32Size, uint8_t *pui8Data)
- void [USBDCompositeTerm](void *pvCompositeInstance)

2.10.1 Detailed Description

Definitions The macros and functions defined in this section can be found in header file `device/usbdcomp.h`.

2.10.2 Data Structure Documentation

2.10.2.1 tUSBDCompositeDevice

Definition:
```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const uint16_t ui16MaxPowermA;
    const uint8_t ui8PwrAttributes;
    const tUSBCallback pfnCallback;
    const uint8_t *const *ppui8StringDescriptors;
    const uint32_t ui32NumStringDescriptors;
    const uint32_t ui32NumDevices;
    tCompositeEntry *const psDevices;
    tCompositeInstance sPrivateData;
}
tUSBDCompositeDevice
```

Members:

ui16VID The vendor ID that this device is to present in the device descriptor.

ui16PID The product ID that this device is to present in the device descriptor.

ui16MaxPowermA The maximum power consumption of the device, expressed in mA.

ui8PwrAttributes Indicates whether the device is self or bus-powered and whether or not it supports remote wake up. Valid values are **USB_CONF_ATTR_SELF_PWR** or **USB_CONF_ATTR_BUS_PWR** , optionally ORed with **USB_CONF_ATTR_RWAKE** .

pfnCallback A pointer to the callback function which will be called to notify the application of events relating to the operation of the composite device.

ppui8StringDescriptors A pointer to the string descriptor array for this device. This array must contain the following string descriptor pointers in this order. Language descriptor, Manufacturer name string (language 1), Product name string (language 1), Serial number string (language 1), Composite device interface description string (language 1), Configuration description string (language 1).
If supporting more than 1 language, the descriptor block (except for string descriptor 0) must be repeated for each language defined in the language descriptor.

ui32NumStringDescriptors The number of descriptors provided in the ppStringDescriptors array. This must be 1 + ((5 + (number of strings)) * (number of languages)).

ui32NumDevices The number of devices in the psDevices array.

psDevices This application supplied array holds the the top level device class information as well as the Instance data for that class.

sPrivateData The private data for this device instance. This memory must remain accessible for as long as the composite device is in use and must not be modified by any code outside the composite class driver.

Description:

The structure used by the application to define operating parameters for the composite device class.

2.10.3   Function Documentation

2.10.3.1   USBDCompositeInit

This function should be called once for the composite class device to initialize basic operation and prepare for enumeration.

Prototype:
```
void *
USBDCompositeInit(uint32_t ui32Index,
                  tUSBDCompositeDevice *psDevice,
                  uint32_t ui32Size,
                  uint8_t *pui8Data)
```

Parameters:

ui32Index is the index of the USB controller to initialize for composite device operation.

psDevice points to a structure containing parameters customizing the operation of the composite device.

ui32Size is the size in bytes of the data pointed to by the *pui8Data* parameter.

pui8Data is the data area that the composite class can use to build up descriptors.

Description:

In order for an application to initialize the USB composite device class, it must first call this function with the a valid composite device class structure in the *psDevice* parameter. This allows this function to initialize the USB controller and device code to be prepared to enumerate and function as a USB composite device. The *ui32Size* and *pui8Data* parameters should be large enough to hold all of the class instances passed in via the *psDevice* structure. This is typically the full size of the configuration descriptor for a device minus its configuration header(9 bytes).

This function returns a void pointer that must be passed in to all other APIs used by the composite class.

See the documentation on the tUSBDCompositeDevice structure for more information on how to properly fill the structure members.

Returns:

This function returns 0 on failure or a non-zero void pointer on success.

2.10.3.2 USBDCompositeTerm

Shuts down the composite device.

Prototype:
```
void
USBDCompositeTerm(void *pvCompositeInstance)
```

Parameters:
> *pvCompositeInstance* is the pointer to the device instance structure as returned by USBD-CompositeInit().

Description:
> This function terminates composite device interface for the instance not me supplied. Following this call, the *pvCompositeInstance* instance should not be used in any other calls.

Returns:
> None.

2.11 Device Firmware Upgrade Device Class Driver

Although USB Device Firmware Upgrade functionality is provided primarily by the USB boot loader (`boot_usb`), applications which want to support DFU funcionality should publicize this in their configuration descriptor and be able to receive a request from the host indicating that they should switch into DFU mode to receive an upgrade.

The DFU device class supports this runtime DFU capability, providing a simple method for an application to indicate to the host that it is DFU-capable and to be signalled that a USB-based firmware upgrade is being requested. The device class is unusual in that it must be used as part of a composite device. Runtime DFU capability makes no sense on its own since it is basically only an indication that the DFU USB boot loader is present and usable.

The USB boot loader must also be used by any device supporting the DFU runtime device class since it implements all DFU mode operation and performs the actual upgrade operation. The runtime device class adds two sections to the configuration descriptor for the main application - a DFU Interface Descriptor and a DFU Functional Descriptor. Standard DFU DETACH requests sent to the DFU interface from the host result in a callback being made to the client application indicating that it must transfer control back to the USB boot loader (via the `USBDDFUUpdateBegin()` function. This function removes the application's existing device from the USB bus then reenters the boot loader which, in turn, publishes DFU mode descriptors and reconnects to the bus as a pure DFU device capable of downloading or uploading application images from the host.

2.11.1 Using the DFU Device Class

The `boot_demo_usb` application provides an example of an application using the Device Firmware Upgrade class. To support DFU in your application, do the following:

1. Ensure that your application is built to run alongside the USB boot loader (`boot_usb`). This requires that it is linked to run from the address defined in label **APP_START_ADDRESS** defined in the `bl_config.h` used to build the boot loader. Typically, for the USB boot loader, this is 0x1800.

2. Add a function to your application following the prototype:

```
uint32_t DFUDetachCallback(void *pvCBData, uint32_t ui32Event,
                           uint32_t ui32MsgData, void *pvMsgData)
```

This function need only check *ui32Event* and, if it is **USBD_DFU_EVENT_DETACH**, signal the application main loop that it should exit and pass control back to the USB boot loader in preparation for a firmware upgrade.

3. Add a mechanism in your application main loop to detect the signal set in your DFUDetach-Callback function and call the USBDDFUUpdateBegin() function. This function tidies up and passes control to the boot loader.

4. Define a structure of type *tDFUInstance*, ensuring that the linker places it in SRAM. This is used as workspace by the DFU class driver.

5. Define a structure of type *tUSBDFUDevice* and initialize it to contain a pointer to your callback function and instance data workspace.

```
//*****************************************************************************
//
// The DFU runtime interface initialization and customization structures
//
//*****************************************************************************
const tUSBDDFUDevice g_sDFUDevice =
{
    DFUDetachCallback,
    (void *)&g_sDFUDevice
};
```

6. Create an array of structures of type *tCompositeEntry* which define the individual device class instances that are to be used in the composite device. A structure defining a composite device containing a HID interface and the DFU runtime interface would look like the following:

```
//*****************************************************************************
//
// The number of device class instances that this composite device uses.
//
//*****************************************************************************
#define NUM_DEVICES         2

//*****************************************************************************
//
// The array of devices supported by this composite device.
//
//*****************************************************************************
tCompositeEntry g_psCompDevices[NUM_DEVICES];
```

7. Define the additional structures and storage required by the composite device class. For the HID/DFU device described above, this would look like:

```
//*****************************************************************************
//
// Allocate the Device Data for the top level composite device class.
//
//*****************************************************************************
tUSBDCompositeDevice g_sCompDevice =
{
    //
    // TI USBLib VID.
    //
    USB_VID_TI_1CBE,

    //
    // PID for composite HID/DFU device.
```

```
        //
        USB_PID_COMP_HID_DFU,

        //
        // This is in milliamps.
        //
        500,

        //
        // Bus powered device.
        //
        USB_CONF_ATTR_BUS_PWR,

        //
        // Device event handler function pointer (receives connect, disconnect
        // and other device-level notifications).
        //
        DeviceHandler,

        //
        // The string table.  This is the string table for the main device (no DFU
        // strings are required).
        //
        g_pStringDescriptors,
        NUM_STRING_DESCRIPTORS,

        //
        // The Composite device array.
        //
        NUM_DEVICES,
        g_psCompDevices
};

//*****************************************************************************
//
// A buffer into which the composite device can write the combined config
// descriptor.
//
//*****************************************************************************
uint8_t g_pui8cDescriptorBuffer[COMPOSITE_DDFU_SIZE + COMPOSITE_DHID_SIZE];
```

8. Rework the USB library initialization function calls to use the composite device. Again, following the HID/DFU example:

```
        //
        // Initialize each of the device instances that forms our composite
        // USB device.
        //
        pvMouseDevice =
            USBDHIDMouseCompositeInit(0, &g_sMouseDevice, &g_psCompDevices[0]);
        pvDFUDevice =
            USBDDFUCompositeInit(0, &g_sDFUDevice, &g_psCompDevices[1]);

        //
        // Pass the USB library our device information, initialize the USB
        // controller and connect the device to the bus.
        //
        USBDCompositeInit(0, &g_sCompDevice, DESCRIPTOR_BUFFER_SIZE,
                          g_pcDescriptorBuffer);
```

2.11.2 Windows Drivers for DFU Devices

Since DFU presents itself as another interface on the USB device, a device driver is required even though the interface is essentially static and does nothing other than listen for a single request while in runtime mode. No version of Microsoft Windows includes a generic Device Firmware Upgrade driver so any application supporting DFU must provide a suitable driver. For USBLib applications, a DFU driver is provided in the `SW-USB-windrivers-xxxx` package which may be downloaded via a link from `http://www.ti.com/tivaware`.

The DFU driver is common between the USB boot loader (`boot_usb`) and the runtime DFU interface provided by the DFU device class. It consists of two USBLib-specific DLLs along with the Microsoft-supplied co-installers for the WinUSB subsystem. Support is provided for WindowsXP, Windows Vista and Windows7 in both 32 bit and 64 bit flavors. The two DLLs are:

lmusbdll.dll A low level interface above WinUSB which provides simple packet transmission and reception and the ability to perform a control transaction on endpoint 0. Applications may be built using this interface without the need to download and install the Windows Device Driver Kit. The DLL itself requires the Windows DDK to build since WinUSB headers are not provided in the Windows SDK.

lmdfu.dll A high level API above the DFU function. This DLL allows DFU-capable devices to be found on the bus and application images to be downloaded to or uploaded from those devices.

To create a driver for your DFU-enabled device, either create a new .INF file from the `boot_usb.inf` example, substituting your device's VID, PID and DFU interface number for those in the original file or merge the content of this .inf (with the same changes) into a combined .inf for your composite device.

Note that the device publishs PID 0x00FF when in DFU mode so you must ensure that your driver .INF file contains an entry to install the driver for devices publishing this PID in addition to your composite device's standard PID.

2.11.3 Programming DFU Devices

The firmware on a DFU-capable USB device may be updated using either the LM Flash Programmer application or the command line `dfuprog` tool. Source code for `dfuprog` is included in the releases for all USB-enabled kits and can be found in C:\ti\TivaWare-for-C-Series\tools\dfuprog assuming you installed TivaWare in the default location. The executable for the application is installed as part of the "Windows-side examples for USB kits" package (SW-USB-win-xxxx) which can be downloaded via a link on `http://www.ti.com/tivaware`. By default, `dfuprog.exe` can be found in C:\Program Files\Texas Instruments\TivaWare\usb_examples.

LM Flash Programmer only lists connected DFU devices which are in DFU mode but the `dfuprog` tool also allows runtime DFU devices to be enumerated and switched into DFU mode ready for programming using either LM Flash Programmer or another call to `dfuprog`. The command

```
dfuprog -e
```

enumerates all DFU-capable devices attached to the host system and provide information on each. To switch a device from runtime mode to DFU mode, issue the command

```
dfuprog -i <index> -m
```

where <index> is the index of the device (as displayed during `dfuprog -e`) whose mode is to be switched.

When a device is switched from runtime mode to DFU mode, you will hear Windows' usual indication that a USB device has been disconnected followed immediately by a reconnection as the boot loader starts and publishes is DFU mode descriptors to the host. These descriptors use PID 0x00FF rather than the PID published by the composite device as suggested in the USB DFU specification. Once this mode switch has taken place, the device will be visible in LM Flash Programmer's device list shown when "Manual Configuration" is chosen along with the "USB DFU" interface. If LMFlash was running when the mode switch took place, press the "Refresh" button to search for and display the DFU device.

To program a device in DFU mode, either use LM Flash Programmer's GUI or `dfuprog -f <filename>`.

2.12   Device Firmware Upgrade Device Class Driver Definitions

Data Structures

- ◼ tUSBDDFUDevice

Defines

- ◼ COMPOSITE_DDFU_SIZE
- ◼ USBD_DFU_EVENT_DETACH

Functions

- ◼ void ∗ USBDDFUCompositeInit (uint32_t ui32Index, tUSBDDFUDevice ∗psDFUDevice, tCompositeEntry ∗psCompEntry)
- ◼ void USBDDFUCompositeTerm (void ∗pvDFUInstance)
- ◼ void USBDDFUUpdateBegin (void)

2.12.1   Detailed Description

Definitions The macros and functions defined in this section can be found in header file `device/usbddfu-rt.h`.

2.12.2   Data Structure Documentation

2.12.2.1   tUSBDDFUDevice

Definition:
```
typedef struct
{
    const tUSBCallback pfnCallback;
    void *const pvCBData;
    tDFUInstance sPrivateData;
}
tUSBDDFUDevice
```

Members:

pfnCallback   A pointer to the callback function which will be called to notify the application of DETACH requests.

pvCBData   A client-supplied pointer which will be sent as the first parameter in all calls made to the pfnCallback function.

sPrivateData   The private instance data for this device class. This memory must remain accessible for as long as the DFU device is in use and must not be modified by any code outside the DFU class driver.

Description:

The structure used by the application to define operating parameters for the DFU device. Note that, unlike all other devices, this structure does not contain any fields which configure the device descriptor sent back to the host. The DFU runtime device class must be used as part of a composite device since all it provides is the capability to signal the device to switch into DFU mode in preparation for a firmware upgrade. Creating a device with nothing but DFU runtime mode capability is rather pointless so this is not supported.

2.12.3   Define Documentation

2.12.3.1   COMPOSITE_DDFU_SIZE

Definition:
```
#define COMPOSITE_DDFU_SIZE
```

Description:

The size of the memory that should be allocated to create a configuration descriptor for a single instance of the DFU runtime device. This does not include the configuration descriptor which is automatically ignored by the composite device class.

This label is used to compute the value which will be passed to the USBDCompositeInit function in the ui32Size parameter.

2.12.3.2   USBD_DFU_EVENT_DETACH

Definition:
```
#define USBD_DFU_EVENT_DETACH
```

Description:
> This value is passed to the client via the callback function provided in the tUSBDDFUDevice structure and indicates that the host has sent a DETACH request to the DFU interface. This request indicates that the device detach from the USB bus and reattach in DFU mode in preparation for a firmware upgrade. Currently, this is the only event that the DFU runtime class reports to the client.
>
> When this event is received, the client should call USBDDFUUpdateBegin() from a non-interrupt context at its earliest opportunity.

2.12.4 Function Documentation

2.12.4.1 USBDDFUCompositeInit

Initializes DFU device operation for a given USB controller.

Prototype:
```
void *
USBDDFUCompositeInit(uint32_t ui32Index,
                     tUSBDDFUDevice *psDFUDevice,
                     tCompositeEntry *psCompEntry)
```

Parameters:
> ***ui32Index*** is the index of the USB controller which is to be initialized for DFU runtime device operation.
> ***psDFUDevice*** points to a structure containing parameters customizing the operation of the DFU device.
> ***psCompEntry*** is the composite device entry to initialize when creating a composite device.

Description:
> The *psCompEntry* should point to the composite device entry to initialize. This is part of the array that is passed to the USBDCompositeInit() function.

Returns:
> Returns zero on failure or a non-zero instance value that should be used with the remaining USB DFU APIs.

2.12.4.2 USBDDFUCompositeTerm

Shuts down the DFU device.

Prototype:
```
void
USBDDFUCompositeTerm(void *pvDFUInstance)
```

Parameters:
> ***pvDFUInstance*** is the pointer to the device instance structure as returned by USBDDFUCompositeInit().

Description:
This function terminates DFU operation for the instance supplied and removes the device from the USB bus.

Following this call, the *pvDFUInstance* instance should not me used in any other calls.

Returns:
None.

2.12.4.3 USBDDFUUpdateBegin

Removes the current USB device from the bus and transfers control to the DFU boot loader.

Prototype:
```
void
USBDDFUUpdateBegin(void)
```

Description:
This function should be called from the application's main loop (i.e. not in interrupt context) following a callback to the USB DFU callback function notifying the application of a DETACH request from the host. The function will prepare the system to switch to DFU mode and transfer control to the boot loader in preparation for a firmware upgrade from the host.

The application must ensure that it has completed all necessary shutdown activities (saved any required data, etc.) before making this call since the function will not return.

Returns:
This function does not return.

2.13 HID Device Class Driver

The USB Human Interface Class device class is an enormously versatile architecture for supporting a wide variety of input/output devices regardless of whether or not they actually deal with "Human Interfaces". Although typically thought of in the context of keyboards, mice and joysticks, the specification can cover practically any device offering user controls or data gathering capabilities.

Communication between the HID device and host is via a collection of "report" structures which are defined by the device in HID report descriptors which the host can query. Reports are defined both for communication of device input to the host and for output and feature selection from the host.

In addition to the flexibility offered by the basic architecture, HID devices also benefit from excellent operating system support for the class, meaning that no driver writing is necessary and, in the case of standard devices such as keyboards and joysticks, the device can connect to and operate with the host system without any new host software having to be written. Even in the case of a non-standard or vendor-specific HID device, the operating system support makes writing the host-side software very much more straightforward than developing the device using a vendor-specific class.

Despite these advantages, there is one downside to using HID. The interface is limited in the amount of data that can be transferred so is not suitable for use by devices which expect to use a high percentage of the USB bus bandwidth. Devices are limited to a maximum of 64KB of data per second for each report they support. Multiple reports can be used if necessary but high bandwidth devices

may be better implemented using a class which supports bulk rather than interrupt endpoints (such as CDC or the generic bulk device class).

USB HID Device Model
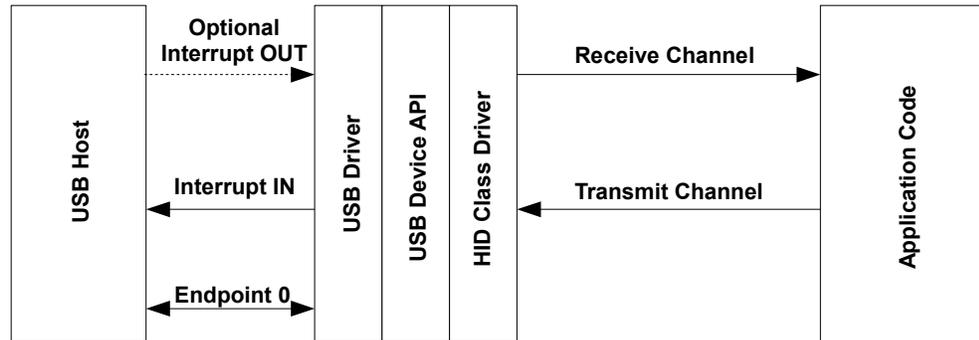
This device class uses one or, optionally, two endpoints in addition to endpoint zero. One interrupt IN endpoint carries HID input reports from the device to the host. Output and Feature reports from the host to the device are typically carried via endpoint zero but devices which expect high host-to-device data rates can select to offer an independent interrupt OUT endpoint to carry these. Endpoint zero carries standard USB requests and also HID-specific descriptor requests.

The HID mouse and keyboard device APIs described later in this document are both implemented above the HID Device Class Driver API.

2.13.1 HID Device Class Events

The HID device class driver sends the following events to the application callback functions:

2.13.1.1 Receive Channel Events

- **USB_EVENT_CONNECTED**
- **USB_EVENT_DISCONNECTED**
- **USB_EVENT_RX_AVAILABLE**
- **USB_EVENT_ERROR**
- **USB_EVENT_SUSPEND**
- **USB_EVENT_RESUME**
- **USBD_HID_EVENT_IDLE_TIMEOUT**
- **USBD_HID_EVENT_GET_REPORT_BUFFER**
- **USBD_HID_EVENT_GET_REPORT**
- **USBD_HID_EVENT_SET_PROTOCOL**
- **USBD_HID_EVENT_GET_PROTOCOL**

- **USBD_HID_EVENT_SET_REPORT**

- **USBD_HID_EVENT_REPORT_SENT**

Note: The USB_EVENT_DISCONNECTED event is not be reported to the application if the MCU's PB1/USB0VBUS pin is connected to a fixed +5 Volts rather than directly to the VBUS pin on the USB connector or if the USB controller is configured to force device mode.

2.13.1.2 Transmit Channel Events

- **USB_EVENT_TX_COMPLETE**

2.13.2   Using the HID Device Class Driver

To add a USB HID interface to your application using the HID Device Class Driver, take the following steps.

- Add the following header files to the source file(s) which are to support USB:

```
#include "src/usb.h"
#include "usblib/usblib.h"
#include "usblib/usbhid.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdhid.h"
```

- Define the string table which is used to describe various features of your new device to the host system. The following is the string table taken from the `usb_dev_mouse` example application. Edit the actual strings to suit your application and take care to ensure that you also update the length field (the first byte) of each descriptor to correctly reflect the length of the string and descriptor header. The number of strings included varies depending upon the device but must be at least 5. HID report descriptors may refer to string IDs and, if the descriptor for your device includes these, additional strings are required. Also, if multiple languages are reported in string descriptor 0, you must ensure that you have strings available for each language with all language 1 strings occurring in order in a block before all language 2 strings and so on.

```
//*****************************************************************************
//
// The languages supported by this device.
//
//*****************************************************************************
const uint8_t g_pui8LangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};

//*****************************************************************************
//
// The manufacturer string.
//
//*****************************************************************************
const uint8_t g_pui8ManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
```

```
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};

//*****************************************************************************
//
// The product string.
//
//*****************************************************************************
const uint8_t g_pui8ProductString[] =
{
    (13 + 1) * 2,
    USB_DTYPE_STRING,
    'M', 0, 'o', 0, 'u', 0, 's', 0, 'e', 0, ' ', 0, 'E', 0, 'x', 0, 'a', 0,
    'm', 0, 'p', 0, 'l', 0, 'e', 0
};

//*****************************************************************************
//
// The serial number string.
//
//*****************************************************************************
const uint8_t g_pui8SerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

//*****************************************************************************
//
// The interface description string.
//
//*****************************************************************************
const uint8_t g_pui8HIDInterfaceString[] =
{
    (19 + 1) * 2,
    USB_DTYPE_STRING,
    'H', 0, 'I', 0, 'D', 0, ' ', 0, 'M', 0, 'o', 0, 'u', 0, 's', 0,
    'e', 0, ' ', 0, 'I', 0, 'n', 0, 't', 0, 'e', 0, 'r', 0, 'f', 0,
    'a', 0, 'c', 0, 'e', 0
};

//*****************************************************************************
//
// The configuration description string.
//
//*****************************************************************************
const uint8_t g_pui8ConfigString[] =
{
    (23 + 1) * 2,
    USB_DTYPE_STRING,
    'H', 0, 'I', 0, 'D', 0, ' ', 0, 'M', 0, 'o', 0, 'u', 0, 's', 0,
    'e', 0, ' ', 0, 'C', 0, 'o', 0, 'n', 0, 'f', 0, 'i', 0, 'g', 0,
    'u', 0, 'r', 0, 'a', 0, 't', 0, 'i', 0, 'o', 0, 'n', 0
};

//*****************************************************************************
//
// The descriptor string table.
//
//*****************************************************************************
const uint8_t * const g_pStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
```

```
        g_pSerialNumberString,
        g_pHIDInterfaceString,
        g_pConfigString
};

#define NUM_STRING_DESCRIPTORS (sizeof(g_pStringDescriptors) /              \
                                sizeof(uint8_t *))
```

■ Develop the HID report descriptors and, if required, physical descriptors for your device and, from these, the HID descriptor itself. Details of how to do this are beyond the scope of this document other than to say that macros in header file `usbdhid.h` are included to help add the various tags required in the descriptor. For information on how these descriptors are constructed, please see the "USB Device Class Definition for Human Interface Devices, version 1.11" which can be downloaded from `http://www.usb.org/developers/devclass_docs/HID1_11.pdf`. The required structures for a BIOS-compatible HID mouse are:

```
//*****************************************************************************
//
// The report descriptor for the BIOS mouse class device.
//
//*****************************************************************************
static const uint8_t g_pui8ucMouseReportDescriptor[]=
{
    UsagePage(USB_HID_GENERIC_DESKTOP),
    Usage(USB_HID_MOUSE),
    Collection(USB_HID_APPLICATION),
        Usage(USB_HID_POINTER),
        Collection(USB_HID_PHYSICAL),

            //
            // The buttons.
            //
            UsagePage(USB_HID_BUTTONS),
            UsageMinimum(1),
            UsageMaximum(3),
            LogicalMinimum(0),
            LogicalMaximum(1),

            //
            // 3 - 1 bit values for the buttons.
            //
            ReportSize(1),
            ReportCount(3),
            Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
                  USB_HID_INPUT_ABS),

            //
            // 1 - 5 bit unused constant value to fill the 8 bits.
            //
            ReportSize(5),
            ReportCount(1),
            Input(USB_HID_INPUT_CONSTANT | USB_HID_INPUT_ARRAY |
                  USB_HID_INPUT_ABS),

            //
            // The X and Y axis.
            //
            UsagePage(USB_HID_GENERIC_DESKTOP),
            Usage(USB_HID_X),
            Usage(USB_HID_Y),
            LogicalMinimum(-127),
            LogicalMaximum(127),
```

```
            //
            // 2 - 8 bit Values for x and y.
            //
            ReportSize(8),
            ReportCount(2),
            Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
                USB_HID_INPUT_RELATIVE),
        EndCollection,
    EndCollection,
};

//*****************************************************************************
//
// The HID class descriptor table. For the mouse class, we have only a single
// report descriptor.
//
//*****************************************************************************
static const uint8_t * const g_pMouseClassDescriptors[] =
{
    g_pucMouseReportDescriptor
};

//*****************************************************************************
//
// The HID descriptor for the mouse device.
//
//*****************************************************************************
static const tHIDDescriptor g_sMouseHIDDescriptor =
{
    9,                              // bLength
    USB_HID_DTYPE_HID,              // bDescriptorType
    0x111,                          // bcdHID (version 1.11 compliant)
    0,                              // bCountryCode (not localized)
    1,                              // bNumDescriptors
    USB_HID_DTYPE_REPORT,           // Report descriptor
    sizeof(g_pucMouseReportDescriptor) // Size of report descriptor
};
```

- Define an array of *tHIDReportIdle* structures in RAM with one entry for each input report your device supports. Initialize the `ucDuration4mS` and *ucReportID* fields in each of the entries to set the default idle report time for each input report. Note that *ucDuration4mS* defines the idle time in 4mS increments as used in the USB HID Set_Idle and Get_Idle requests. The times defined in these structures are used to determine how often a given input report is resent to the host in the absence of any device state change. For example, a device supporting two input reports with IDs 1 and 2 may initialize the array as follows:

```
tHIDReportIdle g_psReportIdle[2] =
{
    { 125, 1, 0, 0 }, // Report 1 polled every 500mS (4 * 125).
    {   0, 2, 0, 0}   // Report 2 is not polled (0mS timeout)
};
```

- Define a *tUSBDHIDDevice* structure and initialize all fields as required for your application. The following example shows a structure suitable for a BIOS-compatible mouse device which publishes a single input report.

```
const tUSBDHIDDevice g_sHIDMouseDevice =
{
    //
    // The Vendor ID you have been assigned by USB-IF.
    //
    USB_VID_YOUR_VENDOR_ID,

    //
```

```
// The product ID you have assigned for this device.
//
USB_PID_YOUR_PRODUCT_ID,

//
// The power consumption of your device in milliamps.
//
POWER_CONSUMPTION_MA,

//
// The value to be passed to the host in the USB configuration descriptor's
// bmAttributes field.
//
USB_CONF_ATTR_BUS_PWR,

//
// This mouse supports the boot subclass.
//
USB_HID_SCLASS_BOOT,

//
// This device supports the BIOS mouse report protocol.
//
USB_HID_PROTOCOL_MOUSE,

//
// The device has a single input report.
//
1,

//
// A pointer to our array of tHIDReportIdle structures. For this device,
// the array must have 1 element (matching the value of the previous field).
//
g_psMouseReportIdle,

//
// A pointer to your receive callback event handler.
//
YourUSBReceiveEventCallback,

//
// A value that you want passed to the receive callback alongside every
// event.
//
(void *)&g_sYourInstanceData,

//
// A pointer to your transmit callback event handler.
//
YourUSBTransmitEventCallback,

//
// A value that you want passed to the transmit callback alongside every
// event.
//
(void *)&g_sYourInstanceData,

//
// This device does not want to use a dedicated interrupt OUT endpoint
// since there are no output or feature reports required.
//
false,

//
// A pointer to the HID descriptor for the device.
```

```
            //
            &g_sMouseHIDDescriptor,

            //
            // A pointer to the array of HID class descriptor pointers for this device.
            // The number of elements in this array and their order must match the
            // information in the HID descriptor provided above.
            //
            g_pMouseClassDescriptors,

            //
            // A pointer to your string table.
            //
            g_pStringDescriptors,

            //
            // The number of entries in your string table. This must equal
            // (1 + (5 + (num HID strings)) * (num languages)).
            //
            NUM_STRING_DESCRIPTORS
        };
```

- Add a receive event handler function, YourUSBReceiveEventCallback in the previous example, to your application taking care to handle all messages which require a particular response. For the HID device class the following receive callback events MUST be handled by the application:

    - **USB_EVENT_RX_AVAILABLE**
    - **USBD_HID_EVENT_IDLE_TIMEOUT**
    - **USBD_HID_EVENT_GET_REPORT_BUFFER**
    - **USBD_HID_EVENT_GET_REPORT**
    - **USBD_HID_EVENT_SET_PROTOCOL** (for BIOS protocol devices)
    - **USBD_HID_EVENT_GET_PROTOCOL** (for BIOS protocol devices)
    - **USBD_HID_EVENT_SET_REPORT**

    Although no other events must be handled, **USB_EVENT_CONNECTED** and **USB_EVENT_DISCONNECTED** is typically be required since these indicate when a host connects or disconnects and allow the application to flush any buffers or reset state as required. Attempts to send data when the host is disconnected are ignored and result in an error.

- Add a transmit event handler function, YourUSBTransmitEventCallback in the previous example, to your application and use **USB_EVENT_TX_COMPLETE** to indicate when a new report may be scheduled for transmission. While a report is being transmitted, attempts to send another report via USBDHIDReportWrite() are ignored and results in an error.

- From your main initialization function call the HID device class driver initialization function to configure the USB controller and place the device on the bus.

    ```
    pvDevice = USBDHIDMouseInit(0, &g_sHIDMouseDevice);
    ```

- Assuming *pvDevice* returned is not NULL, your device is now ready to communicate with a USB host.

- Once the host connects, your control event handler is sent USB_EVENT_CONNECTED and the first input report may be sent to the host using USBDHIDReportWrite() with following packets transmitted as soon as **USB_EVENT_TX_COMPLETE** is received via the transmit event handler.

2.13.3   Using the Composite HID Mouse Device Class

When using the HID mouse device class in a composite device, the configuration of the device is very similar to how it is configured as a non-composite device. Follow all of the configuration steps in the previous section with the exception of calling USBDHIDMouseCompositeInit() instead of USBDHIDMouseInit(). This prepares an instance of the HID mouse device class to be enumerated as part of a composite device. The USBDHIDMouseCompositeInit() function takes the mouse device structure and a pointer to a tCompositeEntry value so that it can properly initialize the mouse device and the composite entry that is later be passed to the USBDCompositeInit() function. The code example below provides an example of how to initialize a mouse device to be a part of a composite device.

```
//
// These should be initialized with valid values for each class.
//
extern tUSBDHIDMouseDevice g_sHIDMouseDevice;

//
// The array of composited devices.
//
tCompositeEntry psCompEntries[2];

//*****************************************************************************
//
// Allocate the Device Data for the top level composite device class.
//
//*****************************************************************************
tUSBDCompositeDevice g_sCompDevice =
{
    //
    // TI USBLib VID.
    //
    USB_VID_TI_1CBE,

    //
    // PID for composite HID/DFU device.
    //
    USB_PID_COMP_HID_DFU,

    //
    // This is in milliamps.
    //
    500,

    //
    // Bus powered device.
    //
    USB_CONF_ATTR_BUS_PWR,

    //
    // Device event handler function pointer (receives connect, disconnect
    // and other device-level notifications).
    //
    DeviceHandler,

    //
    // The string table.  This is the string table for the main device (no DFU
    // strings are required).
    //
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTORS,

    //
```

```
    // The Composite device array.
    //
    NUM_DEVICES,
    g_psCompDevices
};

//
// The OTHER_SIZES here are the sizes of the descriptor data for other classes
// that are part of the composite device.
//
#define DESCRIPTOR_DATA_SIZE    (COMPOSITE_DHID_SIZE + OTHER_SIZES)
uint8_t g_pui8DescriptorData[DESCRIPTOR_DATA_SIZE];

//
// Initialize the HID mouse and its composite entry.
//
pvMouseDevice = USBDHIDMouseCompositeInit(0, &g_sHIDMouseDevice,
                                          &psCompEntries[0]);

...

USBDCompositeInit(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
                  g_pui8DescriptorData);
```

All other API calls to the USB HID mose device class should use the value returned by USBD-HIDMouseCompositeInit() when the API calls for a pvInstance pointer. Also when using the audio device in a composite device the **COMPOSITE_DHID_SIZE** value should be added to the size of the g_pui8DescriptorData array as shown in the example above.

2.13.4 Handling HID Reports

Communication between a HID device and host takes place using data structures known as "reports".

Input reports are sent from the device to the host in response to device state changes, queries from the host or a configurable timeout. In the case of a state change, the device sends a new copy of the relevant input report to the host via the interrupt IN endpoint. This is accomplished by calling USBDHIDReportWrite(). Whereas other USB device class drivers require that the application send no more than 1 packet of data in each call to the driver's "PacketWrite" function, the HID device class driver allows a complete report to be sent. If the report passed is longer than the maximum packet size for the endpoint, the class driver handles the process of breaking it up into multiple USB packets. Once a full report has been transmitted to the host and acknowledged, the application's transmit event handler receives **USB_EVENT_TX_COMPLETE** indicating that the application is free to send another report.

The host may also poll for the latest version of an input report. This procedure involves a request on endpoint zero and results in a sequence of events that the application must respond to. On receipt of the Get_Report request, the HID device class driver sends **USBD_HID_EVENT_GET_REPORT** to the application receive callback. The application must respond to this by returning a pointer to the latest version of the requested report and the size of the report in bytes. This data is then returned to the host via endpoint zero and successful completion of the transmission is notified to the application using **USBD_HID_EVENT_REPORT_SENT** passed to the receive callback.

One other condition may cause an input report to be sent. Each input report has a timeout associated with it and, when this time interval expires, the report must be returned to the host regardless of whether or not the device state has changed. The timeout is set using a Set_Idle request from

the host and may be completely disabled (as is typically done for mice and keyboards when communicating with a Windows PC, for example) by setting the timeout to 0.

The HID device class driver internally tracks the required timeouts for each input report. When a timer expires, indicating that the report must be resent, **USBD_HID_EVENT_IDLE_TIMEOUT** is sent to the application receive callback. As in the previous case, the application must respond with a pointer to the appropriate report and its length in bytes. In this case, the returned report is transmitted to the host using the interrupt IN endpoint and the successful completion of the transmission is notified to the application using **USB_EVENT_TX_COMPLETE** sent to the transmit callback. Note that the application returns information on the location and size of the report and MUST NOT call USBDHIDReportWrite() in response to this event.

Output and Feature reports are sent from the host to the device to instruct it to set various parameters and options. A device can chose whether all host-to-device report communication takes place via endpoint zero or whether a dedicated interrupt OUT endpoint is used. Typically host-to-device traffic is low bandwidth and endpoint zero communication can be used but, if a dedicated endpoint is required, the field *bUseOutEndpoint* in the *tUSBDHIDDevice* structure for the device should be set to **true**.

If using a dedicated endpoint for output and feature reports, the application receive callback is called with **USB_EVENT_RX_AVAILABLE** whenever a report packet is available. During this callback, the application can call USBDHIDPacketRead() to retrieve the packet. If it is not possible to read the packet immediately, the HID device class driver calls the application back later to give it another opportunity. Until the packet is read, NAK is sent to the host preventing more data from being sent.

In the more typical case where endpoint zero is used to transfer output and feature reports, the application can expect the following sequence of events on the receive callback.

- **USBD_HID_EVENT_GET_REPORT_BUFFER** indicates that a Set_Report request has been received from the host and the device class driver is requesting a buffer into which the received report can be written. The application must return a pointer to a buffer which is at least as large as required to store the report.
- **USBD_HID_EVENT_SET_REPORT** follows next once the report data has been read from endpoint zero into the buffer supplied on the earlier **USBD_HID_EVENT_GET_REPORT_BUFFER** callback. The device class driver does not access the report buffer after this event is sent and the application may handle the memory as it wishes following this point.

2.14 HID Device Class Driver Definitions

Data Structures

- tHIDClassDescriptorInfo
- tHIDDescriptor
- tHIDKeyboardUsageTable
- tHIDReportIdle
- tUSBDHIDDevice

Defines

- Collection(ui8Value)
- COMPOSITE_DHID_SIZE
- EndCollection
- Feature(ui8Value)
- Feature2(ui16Value)
- Input(ui8Value)
- Input2(ui16Value)
- LogicalMaximum(i8Value)
- LogicalMinimum(i8Value)
- Output(ui8Value)
- Output2(ui16Value)
- PhysicalMaximum(i16Value)
- PhysicalMinimum(i16Value)
- ReportCount(ui8Value)
- ReportID(ui8Value)
- ReportSize(ui8Value)
- Unit(ui32Value)
- UnitAccelerationSI
- UnitAngAccelerationSI
- UnitCurrent_A
- UnitDistance_cm
- UnitDistance_i
- UnitEnergySI
- UnitExponent(i8Value)
- UnitForceSI
- UnitMass_g
- UnitMomentumSI
- UnitRotation_deg
- UnitRotation_rad
- UnitTemp_F
- UnitTemp_K
- UnitTime_s
- UnitVelocitySI
- UnitVoltage
- Usage(ui8Value)
- UsageMaximum(ui8Value)
- UsageMinimum(ui8Value)
- UsagePage(ui8Value)
- UsagePageVendor(ui16Value)
- UsageVendor(ui16Value)
- USBD_HID_EVENT_GET_PROTOCOL
- USBD_HID_EVENT_GET_REPORT
- USBD_HID_EVENT_GET_REPORT_BUFFER
- USBD_HID_EVENT_IDLE_TIMEOUT
- USBD_HID_EVENT_REPORT_SENT
- USBD_HID_EVENT_SET_PROTOCOL
- USBD_HID_EVENT_SET_REPORT

Functions

- void * USBDHIDCompositeInit (uint32_t ui32Index, tUSBDHIDDevice *psHIDDevice, tCompositeEntry *psCompEntry)
- void * USBDHIDInit (uint32_t ui32Index, tUSBDHIDDevice *psHIDDevice)
- uint32_t USBDHIDPacketRead (void *pvHIDInstance, uint8_t *pi8Data, uint32_t ui32Length, bool bLast)
- void USBDHIDPowerStatusSet (void *pvHIDInstance, uint8_t ui8Power)
- bool USBDHIDRemoteWakeupRequest (void *pvHIDInstance)
- uint32_t USBDHIDReportWrite (void *pvHIDInstance, uint8_t *pi8Data, uint32_t ui32Length, bool bLast)
- uint32_t USBDHIDRxPacketAvailable (void *pvHIDInstance)
- void * USBDHIDSetRxCBData (void *pvHIDInstance, void *pvCBData)
- void * USBDHIDSetTxCBData (void *pvHIDInstance, void *pvCBData)
- void USBDHIDTerm (void *pvHIDInstance)
- uint32_t USBDHIDTxPacketAvailable (void *pvHIDInstance)

2.14.1 Detailed Description

The macros and functions defined in this section can be found in header file `device/usbdhid.h`. Users of the HID device class driver also need to include `usbhid.h` which includes HID-related definitions required by both host and device implementations.

2.14.2 Data Structure Documentation

2.14.2.1 tHIDClassDescriptorInfo

Definition:
```
typedef struct
{
    uint8_t bDescriptorType;
    uint16_t wDescriptorLength;
}
tHIDClassDescriptorInfo
```

Members:
> *bDescriptorType* The type of HID class descriptor. This will be **USB_HID_DTYPE_REPORT** or **USB_HID_DTYPE_PHYSICAL** .
> *wDescriptorLength* The total length of the HID class descriptor.

Description:
> The class descriptor information structure is used to announce the presence of HID-specific class descriptors within the HID descriptor.

2.14.2.2 tHIDDescriptor

Definition:
```
typedef struct
{
    uint8_t bLength;
    uint8_t bDescriptorType;
    uint16_t bcdHID;
    uint8_t bCountryCode;
    uint8_t bNumDescriptors;
    tHIDClassDescriptorInfo sClassDescriptor[1];
}
tHIDDescriptor
```

Members:

bLength The length of this descriptor in bytes.

bDescriptorType The type of the descriptor. For a HID descriptor, this will be **USB_HID_DTYPE_HID** .

bcdHID A BCD value identifying the HID Class specification release supported by the device. For version 1.11, for example, this value would be 0x0111.

bCountryCode The country code for which this hardware is localized or 0 if no localization has been performed. Valid country (or language) codes are in labels of the form **USB_HID_COUNTRY_xxx** .

bNumDescriptors The number of class-specific descriptors that exist for this device. This indicates the number of class descriptor information structures that are appended to this structure and must be at least 1 (since all HID devices must publish at least 1 report descriptor).

sClassDescriptor A table announcing each of the class-specific descriptors that this device publishes. The actual number of entries in the array is given by the bNumDescriptors field.

Description:

The HID descriptor is inserted following the interface descriptor and before the endpoint descriptors for a HID class device.

2.14.2.3 tHIDKeyboardUsageTable

Definition:
```
typedef struct
{
    uint8_t ui8BytesPerChar;
    uint32_t pui32CapsLock[USBH_HID_CAPS_ARRAY_SZ];
    void *pvCharMapping;
}
tHIDKeyboardUsageTable
```

Members:

ui8BytesPerChar Number of bytes per character in the pCharMapping table of this structure.

pui32CapsLock This is a packed bitmasked structure with a one bit flags that indicates if the corresponding Usage ID is affected by the Caps Lock key.

pvCharMapping This is the indexed table of Usage ID to character value. It must be at least ui8BytesPerChar * 2 * **USBH_HID_MAX_USAGE** bytes in size as it is treated as a double indexed array.

Description:
This structure defines the mapping of USB usage identifiers to printable characters. The structure has three members that hold this information. The ui8BytesPerChar, indicates the number of bytes per character in the table. The pui32CapsLock array holds a packed bit array of usage identifiers that can be modified by the Caps Lock key. The pCharMapping array is treated as a double indexed array with two "columns". In the case of a single byte character it is treated as pairs of 8 bit values for unshifted and shifted values. In the case of a double byte characters it is treated as pairs of 16 bit values.

2.14.2.4   tHIDReportIdle

Definition:
```
typedef struct
{
    uint8_t ui8Duration4mS;
    uint8_t ui8ReportID;
    uint16_t ui16TimeTillNextmS;
    uint32_t ui32TimeSinceReportmS;
}
tHIDReportIdle
```

Members:
ui8Duration4mS The idle duration for the report expressed in units of 4mS. 0 indicates infinite and informs the class driver not to send the report unless a state change occurs.
ui8ReportID The ID of the report which this structure applies to. This is the report ID as specified using a ReportID tag in the report descriptor rather than the index of the report in the HID class descriptor array. If only a single Input report is supported and, thus, no ReportID tag is present, this field should be set to 0.
ui16TimeTillNextmS The number of milliseconds before we need to send a copy of a given report back to the host. This field is updated by the HID driver and used to time sending of **USBD_HID_EVENT_IDLE_TIMEOUT** .
ui32TimeSinceReportmS The number of milliseconds that have passed since the last time this report was sent. The HID class driver needs to track this since Set_Idle requests are required to take effect as if issued immediately after the last transmission of the report to which they refer.

Description:
The structure used to track idle time for reports. An array of these structures is passed to the HID device class driver during USBDHIDInit and is used to track automatic resending of each report (if not disabled by the host).

2.14.2.5   tUSBDHIDDevice

Definition:
```
typedef struct
{
    uint16_t ui16VID;
    uint16_t ui16PID;
    uint16_t ui16MaxPowermA;
    uint8_t ui8PwrAttributes;
    uint8_t ui8Subclass;
```

```
            uint8_t ui8Protocol;
            uint8_t ui8NumInputReports;
            tHIDReportIdle *psReportIdle;
            tUSBCallback pfnRxCallback;
            void *pvRxCBData;
            tUSBCallback pfnTxCallback;
            void *pvTxCBData;
            bool bUseOutEndpoint;
            const tHIDDescriptor *psHIDDescriptor;
            const uint8_t *const *ppui8ClassDescriptors;
            const uint8_t *const *ppui8StringDescriptors;
            uint32_t ui32NumStringDescriptors;
            const tConfigHeader *const *ppsConfigDescriptor;
            tHIDInstance sPrivateData;
      }
      tUSBDHIDDevice
```

Members:

    ui16VID The vendor ID that this device is to present in the device descriptor.

    ui16PID The product ID that this device is to present in the device descriptor.

    ui16MaxPowermA The maximum power consumption of the device, expressed in milliamps.

    ui8PwrAttributes Indicates whether the device is self- or bus-powered and whether or not it supports remote wakeup. Valid values are **USB_CONF_ATTR_SELF_PWR** or **USB_CONF_ATTR_BUS_PWR** , optionally ORed with **USB_CONF_ATTR_RWAKE** .

    ui8Subclass The interface subclass to publish to the server for this HID device.

    ui8Protocol The interface protocol to publish to the server for this HID device.

    ui8NumInputReports The number of Input reports that this device supports. This field must equal the number of reports published in the HID class descriptors for the device and also the number of entries in the array whose first element is pointed to by field *pi16ReportIdle* below.

    psReportIdle A pointer to the first element in an array of structures used to track idle time for each Input report. When is called, the ui8Duration4mS and ui8ReportID fields of each of these array members should be initialized to indicate the default idle timeout for each input report. This array must be in RAM since the HID device class driver updates values in it in response to requests from the host and to track elapsed time. The number of elements in the array must match the number supplied in the ui8NumInputReports field above.

    pfnRxCallback A pointer to the callback function which is called to notify the application of general events, events related to report transfers on endpoint zero and events related to reception of Output and Feature reports via the (optional) interrupt OUT endpoint.

    pvRxCBData A client-supplied pointer which is sent as the first parameter in all calls made to the receive channel callback, pfnRxCallback.

    pfnTxCallback A pointer to the callback function which is called to notify the application of events related to transmission of Input reports via the interrupt IN endpoint.

    pvTxCBData A client-supplied pointer which is sent as the first parameter in all calls made to the transmit channel callback, pfnTxCallback.

    bUseOutEndpoint If set to true, this field indicates that the device should use a dedicated interrupt OUT endpoint to receive reports from the host. In this case, reports from the host are passed to the application via the receive callback using **USB_EVENT_RX_AVAILABLE** events. If false, reports from the host are received via endpoint zero and passed to the application via **USBD_HID_EVENT_REPORT_SENT** events.

    psHIDDescriptor The HID descriptor that the device is to publish (following the standard interface descriptor and prior to the endpoint descriptors for the interface).

> ***ppui8ClassDescriptors*** The HID class descriptors offered by the device are defined in an array of byte pointers and this field points to that array. The order and number of elements in the array must match the associated information provided in the HID descriptor in field by *pi16HIDDescriptor* .

> ***ppui8StringDescriptors*** A pointer to the string descriptor array for this device. This array must contain the following string descriptor pointers in this order. Language descriptor, Manufacturer name string (language 1), Product name string (language 1), Serial number string (language 1), HID Interface description string (language 1), Configuration description string (language 1), (optionally) First HID device-specific string (language 1), (optionally) Second HID device-specific string (language 1), etc.

> If supporting more than 1 language, the descriptor block (except for string descriptor 0) must be repeated for each language defined in the language descriptor.

> The number of HID device-specific strings is dependent upon the content of the report descriptor passed to the interface and is, thus, application controlled.

> ***ui32NumStringDescriptors*** The number of descriptors provided in the *ppStringDescriptors* array. This must be 1 + ((5 + (num HID strings)) * (num languages)).

> ***ppsConfigDescriptor***

> ***sPrivateData*** The private instance data for this device instance. This memory must remain accessible for as long as the HID device is in use and must not be modified by any code outside the HID class driver.

Description:
> The structure used by the application to define operating parameters for the HID device.

2.14.3 Define Documentation

2.14.3.1 Collection

This is a macro to assist adding Collection entries in HID report descriptors.

Definition:
```
#define Collection(ui8Value)
```

Parameters:
> ***ui8Value*** is the type of Collection.

Description:
> This macro takes a value and prepares it to be placed as a Collection entry into a HID report structure. This is the type of values that are being grouped together, for instance input, output or features can be grouped together as a collection.

Returns:
> Not a function.

2.14.3.2 COMPOSITE_DHID_SIZE

Definition:
```
#define COMPOSITE_DHID_SIZE
```

Description:
> The size of the memory that should be allocated to create a configuration descriptor for a single instance of the USB HID Device. This does not include the configuration descriptor which is automatically ignored by the composite device class.

2.14.3.3 EndCollection

Definition:
> ```
> #define EndCollection
> ```

Description:
> This is a macro to assist adding End Collection entries in HID report descriptors.
>
> This macro can be used to place an End Collection entry into a HID report structure. This is a tag to indicate that a collection of entries has ended in the HID report structure. This terminates a previous Collection() entry.

> **Returns:**
> > Not a function.

2.14.3.4 Feature

This is a macro to assist adding Feature entries in HID report descriptors.

Definition:
> ```
> #define Feature(ui8Value)
> ```

Parameters:
> ***ui8Value*** is bit mask to specify the type of a set of feature report items. Note that if the **USB_HID_FEATURE_BITF** flag is required, the Feature2 macro (which uses a 2 byte version of the Feature item tag) must be used instead of this macro.

Description:
> This macro takes a value and prepares it to be placed as a Feature entry into a HID report structure. This specifies the type of a feature item in a report structure. These refer to a bit mask of flags that indicate the type of feature for a set of items.

> **Returns:**
> > Not a function.

2.14.3.5 Feature2

This is a macro to assist adding Feature entries in HID report descriptors.

Definition:
> ```
> #define Feature2(ui16Value)
> ```

Parameters:
> ***ui16Value*** is bit mask to specify the type of a set of feature report items. Note that this macro uses a version of the Feature item tag with a two byte payload and allows any of the 8 possible data bits for the tag to be used. If **USB_HID_FEATURE_BITF** (bit 8) is not required, the Feature macro may be used instead.

Description:
    This macro takes a value and prepares it to be placed as a Feature entry into a HID report structure. This specifies the type of a feature item in a report structure. These refer to a bit mask of flags that indicate the type of feature for a set of items.

Returns:
    Not a function.

2.14.3.6 Input

This is a macro to assist adding Input entries in HID report descriptors.

Definition:
```
#define Input(ui8Value)
```

Parameters:
    ui8Value is bit mask to specify the type of a set of input report items. Note that if the USB_HID_INPUT_BITF flag is required, the Input2 macro (which uses a 2 byte version of the Input item tag) must be used instead of this macro.

Description:
    This macro takes a value and prepares it to be placed as an Input entry into a HID report structure. This specifies the type of an input item in a report structure. These refer to a bit mask of flags that indicate the type of input for a set of items.

Returns:
    Not a function.

2.14.3.7 Input2

This is a macro to assist adding Input entries in HID report descriptors.

Definition:
```
#define Input2(ui16Value)
```

Parameters:
    ui16Value is bit mask to specify the type of a set of input report items. Note that this macro uses a version of the Input item tag with a two byte payload and allows any of the 8 possible data bits for the tag to be used. If USB_HID_INPUT_BITF (bit 8) is not required, the Input macro may be used instead.

Description:
    This macro takes a value and prepares it to be placed as an Input entry into a HID report structure. This specifies the type of an input item in a report structure. These refer to a bit mask of flags that indicate the type of input for a set of items.

Returns:
    Not a function.

2.14.3.8 LogicalMaximum

This is a macro to assist adding Logical Maximum entries in HID report descriptors.

Definition:
```
#define LogicalMaximum(i8Value)
```

Parameters:
i8Value is the Logical Maximum value.

Description:
This macro takes a value and prepares it to be placed as a Logical Maximum entry into a HID report structure. This is the actual maximum value for a range of values associated with a field.

Returns:
Not a function.

2.14.3.9 LogicalMinimum

This is a macro to assist adding Logical Minimum entries in HID report descriptors.

Definition:
```
#define LogicalMinimum(i8Value)
```

Parameters:
i8Value is the Logical Minimum value.

Description:
This macro takes a value and prepares it to be placed as a Logical Minimum entry into a HID report structure. This is the actual minimum value for a range of values associated with a field.

Returns:
Not a function.

2.14.3.10 Output

This is a macro to assist adding Output entries in HID report descriptors.

Definition:
```
#define Output(ui8Value)
```

Parameters:
ui8Value is bit mask to specify the type of a set of output report items. Note that if the **USB_HID_OUTPUT_BITF** flag is required, the Output2 macro (which uses a 2 byte version of the Output item tag) must be used instead of this macro.

Description:
This macro takes a value and prepares it to be placed as an Output entry into a HID report structure. This specifies the type of an output item in a report structure. These refer to a bit mask of flags that indicate the type of output for a set of items.

Returns:
Not a function.

2.14.3.11 Output2

This is a macro to assist adding Output entries in HID report descriptors.

Definition:
```
#define Output2(ui16Value)
```

Parameters:
>*ui16Value* is bit mask to specify the type of a set of output report items. Note that this macro uses a version of the Output item tag with a two byte payload and allows any of the 8 possible data bits for the tag to be used. If **USB_HID_OUTPUT_BITF** is not required, the Output macro may be used instead.

Description:
>This macro takes a value and prepares it to be placed as an Output entry into a HID report structure. This specifies the type of an output item in a report structure. These refer to a bit mask of flags that indicate the type of output for a set of items.

Returns:
>Not a function.

2.14.3.12 PhysicalMaximum

This is a macro to assist adding Physical Maximum entries in HID report descriptors.

Definition:
```
#define PhysicalMaximum(i16Value)
```

Parameters:
>*i16Value* is the Physical Maximum value. It is a signed, 16 bit number.

Description:
>This macro takes a value and prepares it to be placed as a Physical Maximum entry into a HID report structure. This is value is used in conversion of the control logical value, as returned to the host in the relevant report, to a physical measurement in the appropriate units.

Returns:
>Not a function.

2.14.3.13 PhysicalMinimum

This is a macro to assist adding Physical Minimum entries in HID report descriptors.

Definition:
```
#define PhysicalMinimum(i16Value)
```

Parameters:
>*i16Value* is the Physical Minimum value. It is a signed, 16 bit number.

Description:
This macro takes a value and prepares it to be placed as a Physical Minimum entry into a HID report structure. This is value is used in conversion of the control logical value, as returned to the host in the relevant report, to a physical measurement in the appropriate units.

Returns:
Not a function.

2.14.3.14 ReportCount

This is a macro to assist adding Report Count entries in HID report descriptors.

Definition:
```
#define ReportCount(ui8Value)
```

Parameters:
ui8Value is the number of items in a report item.

Description:
This macro takes a value and prepares it to be placed as a Report Count entry into a HID report structure. This is number of entries of Report Size for a given item.

Returns:
Not a function.

2.14.3.15 ReportID

This is a macro to assist adding Report ID entries in HID report descriptors.

Definition:
```
#define ReportID(ui8Value)
```

Parameters:
ui8Value is the identifier prefix for the current report.

Description:
This macro takes a value and prepares it to be placed as a Report ID entry into a HID report structure. This value is used as a 1 byte prefix for the report it is contained within.

Returns:
Not a function.

2.14.3.16 ReportSize

This is a macro to assist adding Report Size entries in HID report descriptors.

Definition:
```
#define ReportSize(ui8Value)
```

Parameters:
  ui8Value is the size, in bits, of items in a report item.

Description:
  This macro takes a value and prepares it to be placed as a Report Size entry into a HID report structure. This is size in bits of the entries of of a report entry. The Report Count specifies how many entries of Report Size are in a given item. These can be individual bits or bit fields.

Returns:
  Not a function.

2.14.3.17 Unit

This is a macro to assist adding Unit entries for uncommon units in HID report descriptors.

Definition:
```
#define Unit(ui32Value)
```

Parameters:
  ui32Value is the definition of the unit required as defined in section 6.2.2.7 of the USB HID device class definition document.

Description:
  This macro takes a value and prepares it to be placed as a Unit entry into a HID report structure. Note that individual macros are defined for common units and this macro is intended for use when a complex or uncommon unit is needed. It allows entry of a 5 nibble unit definition into the report descriptor.

Returns:
  Not a function.

2.14.3.18 UnitAccelerationSI

Definition:
```
#define UnitAccelerationSI
```

Description:
  This macro inserts a Unit entry for acceleration in cm/s∗∗2 into a report descriptor.

2.14.3.19 UnitAngAccelerationSI

Definition:
```
#define UnitAngAccelerationSI
```

Description:
  This macro inserts a Unit entry for angular acceleration in degrees/(s^2) into a report descriptor.

2.14.3.20 UnitCurrent_A

Definition:
```
#define UnitCurrent_A
```

Description:
This macro inserts a Unit entry for voltage into a a report descriptor.

2.14.3.21 UnitDistance_cm

Definition:
```
#define UnitDistance_cm
```

Description:
This macro inserts a Unit entry for centimeters into a report descriptor.

2.14.3.22 UnitDistance_i

Definition:
```
#define UnitDistance_i
```

Description:
This macro inserts a Unit entry for inches into a report descriptor.

2.14.3.23 UnitEnergySI

Definition:
```
#define UnitEnergySI
```

Description:
This macro inserts a Unit entry for energy in (grams $*$ cm^2)/(s^2) into a report descriptor.

2.14.3.24 UnitExponent

This is a macro to assist adding Unit Exponent entries in HID report descriptors.

Definition:
```
#define UnitExponent(i8Value)
```

Parameters:
i8Value is the required exponent in the range [-8, 7].

Description:
This macro takes a value and prepares it to be placed as a Unit Exponent entry into a HID report structure. This is the exponent applied to PhysicalMinimum and PhysicalMaximum when scaling and converting control values to "real" units.

Returns:
Not a function.

2.14.3.25 UnitForceSI

Definition:
```
#define UnitForceSI
```

Description:
This macro inserts a Unit entry for force in (cm $*$ grams)/s$**$2 into a report descriptor.

2.14.3.26 UnitMass_g

Definition:
```
#define UnitMass_g
```

Description:
This macro inserts a Unit entry for grams into a report descriptor.

2.14.3.27 UnitMomentumSI

Definition:
```
#define UnitMomentumSI
```

Description:
This macro inserts a Unit entry for momentum in (grams $*$ cm)/s into a report descriptor.

2.14.3.28 UnitRotation_deg

Definition:
```
#define UnitRotation_deg
```

Description:
This macro inserts a Unit entry for degrees into a report descriptor.

2.14.3.29 UnitRotation_rad

Definition:
```
#define UnitRotation_rad
```

Description:
This macro inserts a Unit entry for radians into a report descriptor.

2.14.3.30 UnitTemp_F

Definition:
```
#define UnitTemp_F
```

Description:
This macro inserts a Unit entry for temperature in Fahrenheit into a report descriptor.

2.14.3.31 UnitTemp_K

Definition:
```
#define UnitTemp_K
```

Description:
This macro inserts a Unit entry for temperature in Kelvin into a report descriptor.

2.14.3.32 UnitTime_s

Definition:
```
#define UnitTime_s
```

Description:
This macro inserts a Unit entry for seconds into a report descriptor.

2.14.3.33 UnitVelocitySI

Definition:
```
#define UnitVelocitySI
```

Description:
This macro inserts a Unit entry for velocity in cm/s into a report descriptor.

2.14.3.34 UnitVoltage

Definition:
```
#define UnitVoltage
```

Description:
This macro inserts a Unit entry for voltage into a a report descriptor.

2.14.3.35 Usage

This is a macro to assist adding Usage entries in HID report descriptors.

Definition:
```
#define Usage(ui8Value)
```

Parameters:
ui8Value is the Usage value.

Description:
This macro takes a value and prepares it to be placed as a Usage entry into a HID report structure. These are defined by the USB HID specification.

Returns:
Not a function.

2.14.3.36 UsageMaximum

This is a macro to assist adding Usage Maximum entries in HID report descriptors.

Definition:
```
#define UsageMaximum(ui8Value)
```

Parameters:
ui8Value is the Usage Maximum value.

Description:
This macro takes a value and prepares it to be placed as a Usage Maximum entry into a HID report structure. This is the last or maximum value associated with a usage value.

Returns:
Not a function.

2.14.3.37 UsageMinimum

This is a macro to assist adding Usage Minimum entries in HID report descriptors.

Definition:
```
#define UsageMinimum(ui8Value)
```

Parameters:
ui8Value is the Usage Minimum value.

Description:
This macro takes a value and prepares it to be placed as a Usage Minimum entry into a HID report structure. This is the first or minimum value associated with a usage value.

Returns:
Not a function.

2.14.3.38 UsagePage

This is a macro to assist adding Usage Page entries in HID report descriptors.

Definition:
```
#define UsagePage(ui8Value)
```

Parameters:
ui8Value is the Usage Page value.

Description:
This macro takes a value and prepares it to be placed as a Usage Page entry into a HID report structure. These are defined by the USB HID specification.

Returns:
Not a function.

2.14.3.39 UsagePageVendor

This is a macro to assist adding Usage Page entries in HID report descriptors when a vendor-specific value is to be used.

Definition:
```
#define UsagePageVendor(ui16Value)
```

Parameters:
ui16Value is the Usage Page value.

Description:
This macro takes a value and prepares it to be placed as a Usage Page entry into a HID report structure. These are defined by the USB HID specification. Vendor-specific values must lie in the range 0xFF00 to 0xFFFF inclusive.

Returns:
Not a function.

2.14.3.40 UsageVendor

This is a macro to assist adding vendor-specific Usage entries in HID report descriptors.

Definition:
```
#define UsageVendor(ui16Value)
```

Parameters:
ui16Value is the vendor-specific Usage value in the range 0xFF00 to 0xFFFF.

Description:
This macro takes a value and prepares it to be placed as a Usage entry into a HID report structure. These are defined by the USB HID specification.

Returns:
Not a function.

2.14.3.41 USBD_HID_EVENT_GET_PROTOCOL

Definition:
```
#define USBD_HID_EVENT_GET_PROTOCOL
```

Description:
This event is sent in response to a Get_Protocol request from the host. The callback should provide the current protocol via the return code, **USB_HID_PROTOCOL_BOOT** or **USB_HID_PROTOCOL_REPORT**.

2.14.3.42 USBD_HID_EVENT_GET_REPORT

Definition:
```
#define USBD_HID_EVENT_GET_REPORT
```

Description:
This event indicates that the host is requesting a particular report be returned via endpoint 0, the control endpoint. The ui32MsgValue parameter contains the requested report type in the high byte and report ID in the low byte (as passed in the wValue field of the USB request structure). The pvMsgData parameter contains a pointer which must be written with the address of the first byte of the requested report. The callback must return the size in bytes of the report pointed to by ∗pvMsgData. The memory returned in response to this event must remain unaltered until **USBD_HID_EVENT_REPORT_SENT** is sent.

2.14.3.43 USBD_HID_EVENT_GET_REPORT_BUFFER

Definition:
```
#define USBD_HID_EVENT_GET_REPORT_BUFFER
```

Description:
This event indicates that the host has sent a Set_Report request to the device and requests that the device provide a buffer into which the report can be written. The ui32MsgValue parameter contains the received report type in the high byte and report ID in the low byte (as passed in the wValue field of the USB request structure). The pvMsgData parameter contains the length of buffer requested. Note that this is the actual length value cast to a "void ∗" type and not a pointer in this case. The callback must return a pointer to a suitable buffer (cast to the standard "uint32_t" return type for the callback).

2.14.3.44 USBD_HID_EVENT_IDLE_TIMEOUT

Definition:
```
#define USBD_HID_EVENT_IDLE_TIMEOUT
```

Description:
This event indicates to an application that a report idle timeout has occurred and requests a pointer to the report that must be sent back to the host. The ui32MsgData value contains the requested report ID and pvMsgData contains a pointer that must be written with a pointer to the report data that is to be sent. The callback must return the number of bytes in the report pointed to by ∗pvMsgData.

2.14.3.45 USBD_HID_EVENT_REPORT_SENT

Definition:
```
#define USBD_HID_EVENT_REPORT_SENT
```

Description:
This event indicates that a report previously requested via a **USBD_HID_EVENT_GET_REPORT** has been successfully transmitted to the host. The application may now free or reuse the report memory passed on the previous event. Although

this would seem to be an event that would be passed to the transmit channel callback, it is actually passed to the receive channel callback. This ensures that all events related to the request and transmission of reports via endpoint zero can be handled in a single function.

2.14.3.46 USBD_HID_EVENT_SET_PROTOCOL

Definition:
```
#define USBD_HID_EVENT_SET_PROTOCOL
```

Description:
This event is sent in response to a Set_Protocol request from the host. The ui32MsgData value contains the requested protocol, **USB_HID_PROTOCOL_BOOT** or **USB_HID_PROTOCOL_REPORT**.

2.14.3.47 USBD_HID_EVENT_SET_REPORT

Definition:
```
#define USBD_HID_EVENT_SET_REPORT
```

Description:
This event indicates that the host has sent the device a report via endpoint 0, the control endpoint. The ui32MsgValue field indicates the size of the report and pvMsgData points to the first byte of the report. The report buffer was previously returned in response to an earlier **USBD_HID_EVENT_GET_REPORT_BUFFER** callback. The HID device class driver does not access the memory pointed to by pvMsgData after this callback is made so the application is free to reuse or free it at this point.

2.14.4 Function Documentation

2.14.4.1 USBDHIDCompositeInit

Initializes HID device operation for a given USB controller.

Prototype:
```
void *
USBDHIDCompositeInit(uint32_t ui32Index,
                     tUSBDHIDDevice *psHIDDevice,
                     tCompositeEntry *psCompEntry)
```

Parameters:
> ***ui32Index*** is the index of the USB controller which is to be initialized for HID device operation.
> ***psHIDDevice*** points to a structure containing parameters customizing the operation of the HID device.
> ***psCompEntry*** is the composite device entry to initialize when creating a composite device.

Description:
USB HID device classes call this function to initialize the lower level HID interface in the USB controller. If this HID device device is part of a composite device, then the *psCompEntry* should

point to the composite device entry to initialize. This is part of the array that is passed to the USBDCompositeInit() function.

Returns:
Returns zero on failure or a non-zero instance value that should be used with the remaining USB HID APIs.

2.14.4.2 USBDHIDInit

Initializes HID device operation for a given USB controller.

Prototype:
```
void *
USBDHIDInit(uint32_t ui32Index,
            tUSBDHIDDevice *psHIDDevice)
```

Parameters:
ui32Index is the index of the USB controller which is to be initialized for HID device operation.
psHIDDevice points to a structure containing parameters customizing the operation of the HID device.

Description:
An application wishing to offer a USB HID interface to a host system must call this function to initialize the USB controller and attach the device to the USB bus. This function performs all required USB initialization.

On successful completion, this function will return the *psHIDDevice* pointer passed to it. This must be passed on all future calls from the application to the HID device class driver.

The USB HID device class API offers the application a report-based transmit interface for Input reports. Output reports may be received via the control endpoint or via a dedicated Interrupt OUT endpoint. If using the dedicated endpoint, report data is delivered to the application packet-by- packet. If the application uses reports longer than **USBDHID_MAX_PACKET** bytes and would rather receive full reports, it may use a USB buffer above the receive channel to allow full reports to be read.

Transmit Operation:

Calls to USBDHIDReportWrite() pass complete reports to the driver for transmission. These will be transmitted to the host using as many USB packets as are necessary to complete the transmission.

Once a full Input report has been acknowledged by the USB host, a **USB_EVENT_TX_COMPLETE** event is sent to the application transmit callback to inform it that another report may be transmitted.

Receive Operation (when using a dedicated interrupt OUT endpoint):

An incoming USB data packet will result in a call to the application callback with event **USB_EVENT_RX_AVAILABLE**. The application must then call USBDHIDPacketRead(), passing a buffer capable of holding the received packet. The size of the packet may be determined by calling function USBDHIDRxPacketAvailable() prior to reading the packet.

Receive Operation (when not using a dedicated OUT endpoint):

If no dedicated OUT endpoint is used, Output and Feature reports are sent from the host using the control endpoint, endpoint zero. When such a report is received,

USBD_HID_EVENT_GET_REPORT_BUFFER is sent to the application which must respond with a buffer large enough to hold the report. The device class driver will then copy the received report into the supplied buffer before sending **USBD_HID_EVENT_SET_REPORT** to indicate that the report is now available.

Note:
The application must not make any calls to the low level USB device interface if interacting with USB via the USB HID device class API. Doing so will cause unpredictable (though almost certainly unpleasant) behavior.

Returns:
Returns NULL on failure or the *psHIDDevice* pointer on success.

2.14.4.3 USBDHIDPacketRead

Reads a packet of data received from the USB host via the interrupt OUT endpoint (if in use).

Prototype:
```
uint32_t
USBDHIDPacketRead(void *pvHIDInstance,
                  uint8_t *pi8Data,
                  uint32_t ui32Length,
                  bool bLast)
```

Parameters:
pvHIDInstance is the pointer to the device instance structure as returned by USBDHIDInit().
pi8Data points to a buffer into which the received data will be written.
ui32Length is the size of the buffer pointed to by pi8Data.
bLast indicates whether the client will make a further call to read additional data from the packet.

Description:
This function reads up to *ui32Length* bytes of data received from the USB host into the supplied application buffer. If the driver detects that the entire packet has been read, it is acknowledged to the host.

The *bLast* parameter is ignored in this implementation since the end of a packet can be determined without relying upon the client to provide this information.

Returns:
Returns the number of bytes of data read.

2.14.4.4 USBDHIDPowerStatusSet

Reports the device power status (bus- or self-powered) to the USB library.

Prototype:
```
void
USBDHIDPowerStatusSet(void *pvHIDInstance,
                      uint8_t ui8Power)
```

Parameters:
>   ***pvHIDInstance*** is the pointer to the HID device instance structure.
>
>   ***ui8Power*** indicates the current power status, either **USB_STATUS_SELF_PWR** or
>   **USB_STATUS_BUS_PWR**.

Description:
>   Applications which support switching between bus- or self-powered operation should call this
>   function whenever the power source changes to indicate the current power status to the USB
>   library. This information is required by the USB library to allow correct responses to be provided
>   when the host requests status from the device.

Returns:
>   None.

2.14.4.5   USBDHIDRemoteWakeupRequest

Requests a remote wake up to resume communication when in suspended state.

Prototype:
```
bool
USBDHIDRemoteWakeupRequest(void *pvHIDInstance)
```

Parameters:
>   ***pvHIDInstance*** is the pointer to the HID device instance structure.

Description:
>   When the bus is suspended, an application which supports remote wake up (advertised to the
>   host via the configuration descriptor) may call this function to initiate remote wake up signaling
>   to the host. If the remote wake up feature has not been disabled by the host, this will cause
>   the bus to resume operation within 20mS. If the host has disabled remote wake up, **false** will
>   be returned to indicate that the wake up request was not successful.

Returns:
>   Returns **true** if the remote wake up is not disabled and the signaling was started or **false** if
>   remote wake up is disabled or if signaling is currently ongoing following a previous call to this
>   function.

2.14.4.6   USBDHIDReportWrite

Transmits a HID device report to the USB host via the HID interrupt IN endpoint.

Prototype:
```
uint32_t
USBDHIDReportWrite(void *pvHIDInstance,
                   uint8_t *pi8Data,
                   uint32_t ui32Length,
                   bool bLast)
```

Parameters:
>   ***pvHIDInstance*** is the pointer to the device instance structure as returned by USBDHIDInit().
>
>   ***pi8Data*** points to the first byte of data which is to be transmitted.

ui32Length is the number of bytes of data to transmit.

bLast is ignored in this implementation. This parameter is required to ensure compatibility with other device class drivers and USB buffers.

Description:

This function schedules the supplied data for transmission to the USB host in a single USB transaction using as many packets as it takes to send all the data in the report. If no transmission is currently ongoing, the first packet of data is immediately copied to the relevant USB endpoint FIFO for transmission. Whenever all the report data has been acknowledged by the host, a **USB_EVENT_TX_COMPLETE** event will be sent to the application transmit callback indicating that another report can now be transmitted.

The caller must ensure that the data pointed to by *pui8Data* remains accessible and unaltered until the **USB_EVENT_TX_COMPLETE** is received.

Returns:

Returns the number of bytes actually scheduled for transmission. At this level, this will either be the number of bytes passed or 0 to indicate a failure.

2.14.4.7 USBDHIDRxPacketAvailable

Determines whether a packet is available and, if so, the size of the buffer required to read it.

Prototype:

```
uint32_t
USBDHIDRxPacketAvailable(void *pvHIDInstance)
```

Parameters:

pvHIDInstance is the pointer to the device instance structure as returned by USBDHIDInit().

Description:

This function may be used to determine if a received packet remains to be read and allows the application to determine the buffer size needed to read the data.

Returns:

Returns 0 if no received packet remains unprocessed or the size of the packet if a packet is waiting to be read.

2.14.4.8 USBDHIDSetRxCBData

Sets the client-specific pointer parameter for the receive channel callback.

Prototype:

```
void *
USBDHIDSetRxCBData(void *pvHIDInstance,
                   void *pvCBData)
```

Parameters:

pvHIDInstance is the pointer to the device instance structure as returned by USBDHIDInit().

pvCBData is the pointer that client wishes to be provided on each event sent to the receive channel callback function.

Description:

The client uses this function to change the callback pointer passed in the first parameter on all callbacks to the *pfnRxCallback* function passed on USBDHIDInit().

If a client wants to make runtime changes in the callback pointer, it must ensure that the pvHIDInstance structure passed to USBDHIDInit() resides in RAM. If this structure is in flash, callback data changes will not be possible.

Returns:

Returns the previous callback pointer that was being used for this instance's receive callback.

2.14.4.9 USBDHIDSetTxCBData

Sets the client-specific data pointer for the transmit callback.

Prototype:

```
void *
USBDHIDSetTxCBData(void *pvHIDInstance,
                   void *pvCBData)
```

Parameters:

pvHIDInstance is the pointer to the device instance structure as returned by USBDHIDInit().

pvCBData is the pointer that client wishes to be provided on each event sent to the transmit channel callback function.

Description:

The client uses this function to change the callback data pointer passed in the first parameter on all callbacks to the *pfnTxCallback* function passed on USBDHIDInit().

If a client wants to make runtime changes in the callback data, it must ensure that the pvHIDInstance structure passed to USBDHIDInit() resides in RAM. If this structure is in flash, callback data changes will not be possible.

Returns:

Returns the previous callback data pointer that was being used for this instance's transmit callback.

2.14.4.10 USBDHIDTerm

Shuts down the HID device.

Prototype:

```
void
USBDHIDTerm(void *pvHIDInstance)
```

Parameters:

pvHIDInstance is the pointer to the device instance structure as returned by USBDHIDInit().

Description:

This function terminates HID operation for the instance supplied and removes the device from the USB bus. This function should not be called if the HID device is part of a composite device and instead the USBDCompositeTerm() function should be called for the full composite device.

Following this call, the *pvHIDInstance* instance should not me used in any other calls.

Returns:
> None.

2.14.4.11 USBDHIDTxPacketAvailable

Returns the number of free bytes in the transmit buffer.

Prototype:
```
uint32_t
USBDHIDTxPacketAvailable(void *pvHIDInstance)
```

Parameters:
> ***pvHIDInstance*** is the pointer to the device instance structure as returned by USBDHIDInit().

Description:
> This function indicates to the caller whether or not it is safe to send a new report using a call to USBDHIDReportWrite(). The value returned will be the maximum USB packet size (**USBDHID_MAX_PACKET**) if no transmission is currently outstanding or 0 if a transmission is in progress. Since the function USBDHIDReportWrite() can accept full reports longer than a single USB packet, the caller should be aware that the returned value from this class driver, unlike others, does not indicate the maximum size of report that can be written but is merely an indication that another report can be written.

Returns:
> Returns 0 if an outgoing report is still being transmitted or **USBDHID_MAX_PACKET** if no transmission is currently in progress.

2.15 HID Mouse Device Class API

The USB HID device class is extremely versatile but somewhat daunting. For applications which want to offer a mouse-like appearance to a USB host, however, the HID Mouse Device Class API may be used without the need to develop any HID-specific software. This high-level interface completely encapsulates the USB stack and USB HID device class driver and allows an application to simply instantiate a USB mouse device and call a single function to notify the USB host of mouse movement and button presses.

The USB mouse device uses the BIOS mouse subclass and protocol so is recognized by the vast majority of host operating systems and BIOSs without the need for additional host-side software. The mouse provides two axis movement (reported to the host in terms of relative position changes) and up to three buttons which may be either pressed or released.

USB HID Mouse Device Model
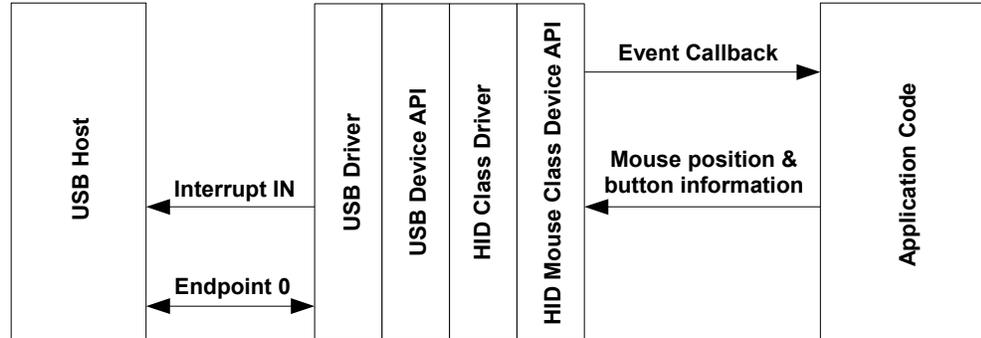
The `usb_dev_mouse` example application makes use of this device class API.

2.15.1   HID Mouse Device API Events

The HID mouse device API sends the following events to the application callback function:

- **USB_EVENT_CONNECTED**
- **USB_EVENT_DISCONNECTED**
- **USB_EVENT_TX_COMPLETE**
- **USB_EVENT_ERROR**
- **USB_EVENT_SUSPEND**
- **USB_EVENT_RESUME**

Note: The **USB_EVENT_DISCONNECTED** event is not be reported to the application if the MCU's USB0VBUS pin is connected to a fixed +5 Volts rather than directly to the VBUS pin on the USB connector or if the USB controller is configured to force device mode.

2.15.2   Using the HID Mouse Device Class API

To add a USB HID mouse interface to your application using the HID Mouse Device Class API, take the following steps.

- Add the following header files to the source file(s) which are to support USB:

```
#include "src/usb.h"
#include "usblib/usblib.h"
#include "usblib/device/usbdhidmouse.h"
```

- Define the string table which is used to describe various features of your new device to the host system. An example of a suitable string table for a mouse device can be found in Using the HID Device Class Driver. This table must include a minimum of 6 entries - string descriptor 0 defining the language(s) available and 5 strings for each supported language.

■ Define a *tUSBDHIDMouseDevice* structure and initialize all fields as required for your application.

```
const tUSBDHIDMouseDevice g_sMouseDevice =
{
    //
    // The Vendor ID you have been assigned by USB-IF.
    //
    USB_VID_YOUR_VENDOR_ID,

    //
    // The product ID you have assigned for this device.
    //
    USB_PID_YOUR_PRODUCT_ID,

    //
    // The power consumption of your device in milliamps.
    //
    POWER_CONSUMPTION_MA,

    //
    // The value to be passed to the host in the USB configuration descriptor's
    // bmAttributes field.
    //
    USB_CONF_ATTR_SELF_PWR,

    //
    // A pointer to your mouse callback event handler.
    //
    YourMouseHandler,

    //
    // A value that you want passed to the callback alongside every event.
    //
    (void *)&g_sYourInstanceData,

    //
    // A pointer to your string table.
    //
    g_pStringDescriptors,

    //
    // The number of entries in your string table. This must equal
    // (1 + (5 * (num languages))).
    //
    NUM_STRING_DESCRIPTORS
};
```

■ Add a mouse event handler function, YourMouseHandler in the previous example, to your application. A minimal implementation can ignore all events though **USB_EVENT_TX_COMPLETE** can be used to ensure that mouse messages are not sent when a previous report is still in transit to the host. Attempts to send a new mouse report when the previous report has not yet been acknowledged results in return code **MOUSE_ERR_TX_ERROR** from USBDHIDMouseStateChange().

■ From your main initialization function call the HID mouse device API initialization function to configure the USB controller and place the device on the bus.

```
        pvDevice = USBDHIDMouseInit(0, &g_sMouseDevice);
```

■ Assuming *pvDevice* returned is not NULL, your mouse device is now ready to communicate with a USB host.

■ Once the host connects, your mouse event handler is sent **USB_EVENT_CONNECTED** after which calls can be made to USBDHIDMouseStateChange()to inform the host of mouse position and button state changes.

2.16 HID Mouse Device Class API Definitions

Data Structures

- tUSBDHIDMouseDevice

Defines

- MOUSE_ERR_NOT_CONFIGURED
- MOUSE_ERR_TX_ERROR
- MOUSE_REPORT_BUTTON_1
- MOUSE_REPORT_BUTTON_2
- MOUSE_REPORT_BUTTON_3
- MOUSE_SUCCESS

Functions

- void ∗ USBDHIDMouseCompositeInit (uint32_t ui32Index, tUSBDHIDMouseDevice ∗psMouseDevice, tCompositeEntry ∗psCompEntry)
- void ∗ USBDHIDMouseInit (uint32_t ui32Index, tUSBDHIDMouseDevice ∗psMouseDevice)
- void USBDHIDMousePowerStatusSet (void ∗pvMouseDevice, uint8_t ui8Power)
- bool USBDHIDMouseRemoteWakeupRequest (void ∗pvMouseDevice)
- void ∗ USBDHIDMouseSetCBData (void ∗pvMouseDevice, void ∗pvCBData)
- uint32_t USBDHIDMouseStateChange (void ∗pvMouseDevice, int8_t i8DeltaX, int8_t i8DeltaY, uint8_t ui8Buttons)
- void USBDHIDMouseTerm (void ∗pvMouseDevice)

2.16.1 Detailed Description

The macros and functions defined in this section can be found in header file `device/usbdhidmouse.h`.

2.16.2 Data Structure Documentation

2.16.2.1 tUSBDHIDMouseDevice

Definition:
```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const uint16_t ui16MaxPowermA;
    const uint8_t ui8PwrAttributes;
    const tUSBCallback pfnCallback;
```

```
        void *pvCBData;
        const uint8_t *const *ppui8StringDescriptors;
        const uint32_t ui32NumStringDescriptors;
        tHIDMouseInstance sPrivateData;
    }
    tUSBDHIDMouseDevice
```

Members:

>   ***ui16VID*** The vendor ID that this device is to present in the device descriptor.

>   ***ui16PID*** The product ID that this device is to present in the device descriptor.

>   ***ui16MaxPowermA*** The maximum power consumption of the device, expressed in milliamps.

>   ***ui8PwrAttributes*** Indicates whether the device is self- or bus-powered and whether or not it supports remote wakeup. Valid values are USB_CONF_ATTR_SELF_PWR or USB_CONF_ATTR_BUS_PWR, optionally ORed with USB_CONF_ATTR_RWAKE.

>   ***pfnCallback*** A pointer to the callback function which is called to notify the application of events relating to the operation of the mouse.

>   ***pvCBData*** A client-supplied pointer which is sent as the first parameter in all calls made to the mouse callback, pfnCallback.

>   ***ppui8StringDescriptors*** A pointer to the string descriptor array for this device. This array must contain the following string descriptor pointers in this order. Language descriptor, Manufacturer name string (language 1), Product name string (language 1), Serial number string (language 1), HID Interface description string (language 1), Configuration description string (language 1).
>
>   If supporting more than 1 language, the descriptor block (except for string descriptor 0) must be repeated for each language defined in the language descriptor.

>   ***ui32NumStringDescriptors*** The number of descriptors provided in the ppStringDescriptors array. This must be (1 + (5 * (num languages))).

>   ***sPrivateData*** The private instance data for this device. This memory must remain accessible for as long as the mouse device is in use and must not be modified by any code outside the HID mouse driver.

Description:

>   This structure is used by the application to define operating parameters for the HID mouse device.

2.16.3 Define Documentation

2.16.3.1 MOUSE_ERR_NOT_CONFIGURED

Definition:

>   `#define MOUSE_ERR_NOT_CONFIGURED`

Description:

>   USBDHIDMouseStateChange() returns this value if it is called before the USB host has connected and configured the device. All mouse state information passed on the call is been ignored.

2.16.3.2 MOUSE_ERR_TX_ERROR

Definition:

```
#define MOUSE_ERR_TX_ERROR
```

Description:

This return code from USBDHIDMouseStateChange() indicates that an error was reported while attempting to send a report to the host. A client should assume that the host has disconnected if this return code is seen.

2.16.3.3 MOUSE_REPORT_BUTTON_1

Definition:

```
#define MOUSE_REPORT_BUTTON_1
```

Description:

Setting this bit in the ui8Buttons parameter to USBDHIDMouseStateChange() indicates to the USB host that button 1 on the mouse is pressed.

2.16.3.4 MOUSE_REPORT_BUTTON_2

Definition:

```
#define MOUSE_REPORT_BUTTON_2
```

Description:

Setting this bit in the ui8Buttons parameter to USBDHIDMouseStateChange() indicates to the USB host that button 2 on the mouse is pressed.

2.16.3.5 MOUSE_REPORT_BUTTON_3

Definition:

```
#define MOUSE_REPORT_BUTTON_3
```

Description:

Setting this bit in the ui8Buttons parameter to USBDHIDMouseStateChange() indicates to the USB host that button 3 on the mouse is pressed.

2.16.3.6 MOUSE_SUCCESS

Definition:

```
#define MOUSE_SUCCESS
```

Description:

This return code from USBDHIDMouseStateChange() indicates success.

2.16.4 Function Documentation

2.16.4.1 USBDHIDMouseCompositeInit

Initializes HID mouse device operation for a given USB controller.

Prototype:
```
void *
USBDHIDMouseCompositeInit(uint32_t ui32Index,
                          tUSBDHIDMouseDevice *psMouseDevice,
                          tCompositeEntry *psCompEntry)
```

Parameters:
>*ui32Index* is the index of the USB controller which is to be initialized for HID mouse device operation.
>
>*psMouseDevice* points to a structure containing parameters customizing the operation of the HID mouse device.
>
>*psCompEntry* is the composite device entry to initialize when creating a composite device.

Description:
>This call is very similar to USBDHIDMouseInit() except that it is used for initializing an instance of the HID mouse device for use in a composite device. If this HID mouse is part of a composite device, then the *psCompEntry* should point to the composite device entry to initialize. This is part of the array that is passed to the USBDCompositeInit() function.

Returns:
>Returns zero on failure or a non-zero instance value that should be used with the remaining USB HID Mouse APIs.

2.16.4.2 USBDHIDMouseInit

Initializes HID mouse device operation for a given USB controller.

Prototype:
```
void *
USBDHIDMouseInit(uint32_t ui32Index,
                 tUSBDHIDMouseDevice *psMouseDevice)
```

Parameters:
>*ui32Index* is the index of the USB controller which is to be initialized for HID mouse device operation.
>
>*psMouseDevice* points to a structure containing parameters customizing the operation of the HID mouse device.

Description:
>An application wishing to offer a USB HID mouse interface to a USB host must call this function to initialize the USB controller and attach the mouse device to the USB bus. This function performs all required USB initialization.
>
>On successful completion, this function returns the *psMouseDevice* pointer passed to it. This must be passed on all future calls to the HID mouse device driver.

When a host connects and configures the device, the application callback receives **USB_EVENT_CONNECTED** after which calls can be made to USBDHIDMouseStateChange() to report pointer movement and button presses to the host.

Note:
The application must not make any calls to the lower level USB device interfaces if interacting with USB via the USB HID mouse device API. Doing so causes unpredictable (though almost certainly unpleasant) behavior.

Returns:
Returns NULL on failure or the psMouseDevice pointer on success.

2.16.4.3 USBDHIDMousePowerStatusSet

Reports the device power status (bus- or self-powered) to the USB library.

Prototype:
```
void
USBDHIDMousePowerStatusSet(void *pvMouseDevice,
                           uint8_t ui8Power)
```

Parameters:
pvMouseDevice is the pointer to the mouse device instance structure.
ui8Power indicates the current power status, either **USB_STATUS_SELF_PWR** or **USB_STATUS_BUS_PWR**.

Description:
Applications which support switching between bus- or self-powered operation should call this function whenever the power source changes to indicate the current power status to the USB library. This information is required by the USB library to allow correct responses to be provided when the host requests status from the device.

Returns:
None.

2.16.4.4 USBDHIDMouseRemoteWakeupRequest

Requests a remote wake up to resume communication when in suspended state.

Prototype:
```
bool
USBDHIDMouseRemoteWakeupRequest(void *pvMouseDevice)
```

Parameters:
pvMouseDevice is the pointer to the mouse device instance structure.

Description:
When the bus is suspended, an application which supports remote wake up (advertised to the host via the configuration descriptor) may call this function to initiate remote wake up signaling to the host. If the remote wake up feature has not been disabled by the host, this causes the bus to resume operation within 20mS. If the host has disabled remote wake up, **false** is returned to indicate that the wake up request was not successful.

Returns:
> Returns **true** if the remote wake up is not disabled and the signaling was started or **false** if remote wake up is disabled or if signaling is currently ongoing following a previous call to this function.

2.16.4.5 USBDHIDMouseSetCBData

Sets the client-specific pointer parameter for the mouse callback.

Prototype:
```
void *
USBDHIDMouseSetCBData(void *pvMouseDevice,
                      void *pvCBData)
```

Parameters:
> *pvMouseDevice* is the pointer to the mouse device instance structure.
>
> *pvCBData* is the pointer that client wishes to be provided on each event sent to the mouse callback function.

Description:
> The client uses this function to change the callback pointer passed in the first parameter on all callbacks to the *pfnCallback* function passed on USBDHIDMouseInit().
>
> If a client wants to make runtime changes in the callback pointer, it must ensure that the pvMouseDevice structure passed to USBDHIDMouseInit() resides in RAM. If this structure is in flash, callback data changes are not possible.

Returns:
> Returns the previous callback pointer that was set for this instance.

2.16.4.6 USBDHIDMouseStateChange

Reports a mouse state change, pointer movement or button press, to the USB host.

Prototype:
```
uint32_t
USBDHIDMouseStateChange(void *pvMouseDevice,
                        int8_t i8DeltaX,
                        int8_t i8DeltaY,
                        uint8_t ui8Buttons)
```

Parameters:
> *pvMouseDevice* is the pointer to the mouse device instance structure.
>
> *i8DeltaX* is the relative horizontal pointer movement that the application wishes to report. Valid values are in the range [-127, 127] with positive values indicating movement to the right.
>
> *i8DeltaY* is the relative vertical pointer movement that the application wishes to report. Valid values are in the range [-127, 127] with positive values indicating downward movement.
>
> *ui8Buttons* is a bit mask indicating which (if any) of the three mouse buttons is pressed. Valid values are logical OR combinations of **MOUSE_REPORT_BUTTON_1**, **MOUSE_REPORT_BUTTON_2** and **MOUSE_REPORT_BUTTON_3**.

Description:

This function is called to report changes in the mouse state to the USB host. These changes can be movement of the pointer, reported relative to its previous position, or changes in the states of up to 3 buttons that the mouse may support. The return code indicates whether or not the mouse report could be sent to the host. In cases where a previous report is still being transmitted, **MOUSE_ERR_TX_ERROR** is returned and the state change is ignored.

Returns:

Returns **MOUSE_SUCCESS** on success, **MOUSE_ERR_TX_ERROR** if an error occurred while attempting to schedule transmission of the mouse report to the host (typically due to a previous report which has not yet completed transmission or due to disconnection of the host) or **MOUSE_ERR_NOT_CONFIGURED** if called before a host has connected to and configured the device.

2.16.4.7   USBDHIDMouseTerm

Shuts down the HID mouse device.

Prototype:
```
void
USBDHIDMouseTerm(void *pvMouseDevice)
```

Parameters:

pvMouseDevice is the pointer to the device instance structure.

Description:

This function terminates HID mouse operation for the instance supplied and removes the device from the USB bus. Following this call, the *pvMouseDevice* instance may not me used in any other call to the HID mouse device other than USBDHIDMouseInit().

Returns:

None.

2.17   HID Keyboard Device Class API

As with the HID Mouse Device Class API described above, the HID Keyboard Device Class API provides an easy-to-use high-level interface for applications wishing to appear to the USB host as a BIOS-compatible keyboard. The keyboard supports up to 6 simultaneously pressed, non-modifier keys and up to 5 state indication LEDs.

Key press and release notifications along with the state of the modifier keys (Shift, Ctrl, Alt, etc.) are passed to the API in a single API call and a callback informs the application whenever the host requests that the LED states be changed.

Keys are identified to the API by means of USB HID key usage codes.   A subset of these are defined in the header file `usbhid.h` and the full set can be found in the document "Universal Serial Bus (USB) HID Usage Tables" which can be downloaded from `http://www.usb.org/developers/devclass_docs/Hut1_12.pdf`.

USB HID Keyboard Device Model
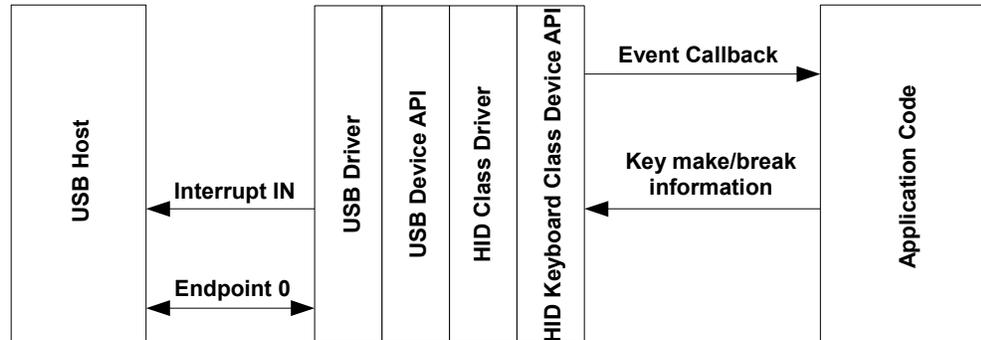
The `usb_dev_keyboard` example application makes use of this device class API.

2.17.1   HID Keyboard Device API Events

The HID keyboard device API sends the following events to the application callback function:

- **USB_EVENT_CONNECTED**
- **USB_EVENT_DISCONNECTED**
- **USB_EVENT_TX_COMPLETE**
- **USB_EVENT_ERROR**
- **USB_EVENT_SUSPEND**
- **USB_EVENT_RESUME**
- **USBD_HID_KEYB_EVENT_SET_LEDS**

Note: The **USB_EVENT_DISCONNECTED** event is not reported to the application if the MCU's PB1/USB0VBUS pin is connected to a fixed +5 Volts rather than directly to the VBUS pin on the USB connector or if the USB controller is configured to force device mode.

- Add the following header files to the source file(s) which are to support USB:

```
#include "src/usb.h"
#include "usblib/usblib.h"
#include "usblib/device/usbdhidkeyb.h"
```

- Define the string table which is used to describe various features of your new device to the host system. The string table found in Using the HID Device Class Driver illustrates the format required. This table must include a minimum of 6 entries - string descriptor 0 defining the language(s) available and 5 strings for each supported language.

- Define a *tUSBDHIDKeyboardDevice* structure and initialize all fields as required for your application.

```
const tUSBDHIDKeyboardDevice g_sKeyboardDevice =
{
    //
    // The Vendor ID you have been assigned by USB-IF.
    //
    USB_VID_YOUR_VENDOR_ID,

    //
    // The product ID you have assigned for this device.
    //
    USB_PID_YOUR_PRODUCT_ID,

    //
    // The power consumption of your device in milliamps.
    //
    POWER_CONSUMPTION_MA,

    //
    // The value to be passed to the host in the USB configuration descriptor's
    // bmAttributes field.
    //
    USB_CONF_ATTR_SELF_PWR,

    //
    // A pointer to your keyboard callback event handler.
    //
    YourKeyboardHandler,

    //
    // A value that you want passed to the callback alongside every event.
    //
    (void *)&g_sYourInstanceData,

    //
    // A pointer to your string table.
    //
    g_pStringDescriptors,

    //
    // The number of entries in your string table. This must equal
    // (1 + (5 * (num languages))).
    //
    NUM_STRING_DESCRIPTORS
};
```

- Add a keyboard event handler function, YourKeyboardHandler in the previous example, to your application. A minimal implementation can ignore all events since key information is buffered in the API and sent later if USBDHIDKeyboardKeyStateChange() is called while a previous report transmission remains unacknowledged.

- From your main initialization function call the HID keyboard device API initialization function to configure the USB controller and place the device on the bus.

```
pvDevice = USBDHIDKeyboardInit(0, &g_sKeyboardDevice);
```

- Assuming *pvDevice* returned is not NULL, your keyboard device is now ready to communicate with a USB host.

- Once the host connects, your keyboard event handler is sent **USB_EVENT_CONNECTED** after which calls can be made to USBDHIDKeyboardKeyStateChange() to inform the host of key press and release events.

2.18   HID Keyboard Device Class API Definitions

Data Structures

- tUSBDHIDKeyboardDevice

Defines

- KEYB_ERR_NOT_CONFIGURED
- KEYB_ERR_NOT_FOUND
- KEYB_ERR_TOO_MANY_KEYS
- KEYB_ERR_TX_ERROR
- KEYB_MAX_CHARS_PER_REPORT
- KEYB_SUCCESS
- USBD_HID_KEYB_EVENT_SET_LEDS

Functions

- void ∗ USBDHIDKeyboardCompositeInit (uint32_t ui32Index, tUSBDHIDKeyboardDevice ∗psHIDKbDevice, tCompositeEntry ∗psCompEntry)
- void ∗ USBDHIDKeyboardInit (uint32_t ui32Index, tUSBDHIDKeyboardDevice ∗psHIDKbDevice)
- uint32_t USBDHIDKeyboardKeyStateChange (void ∗pvKeyboardDevice, uint8_t ui8Modifiers, uint8_t ui8UsageCode, bool bPress)
- void USBDHIDKeyboardPowerStatusSet (void ∗pvKeyboardDevice, uint8_t ui8Power)
- bool USBDHIDKeyboardRemoteWakeupRequest (void ∗pvKeyboardDevice)
- void ∗ USBDHIDKeyboardSetCBData (void ∗pvKeyboardDevice, void ∗pvCBData)
- void USBDHIDKeyboardTerm (void ∗pvKeyboardDevice)

2.18.1   Detailed Description

The macros and functions defined in this section can be found in header file `device/usbdhidkeyb.h`.

2.18.2   Data Structure Documentation

2.18.2.1   tUSBDHIDKeyboardDevice

Definition:
```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const uint16_t ui16MaxPowermA;
```

```
                const uint8_t ui8PwrAttributes;
                const tUSBCallback pfnCallback;
                void *pvCBData;
                const uint8_t *const *ppui8StringDescriptors;
                const uint32_t ui32NumStringDescriptors;
                tHIDKeyboardInstance sPrivateData;
        }
        tUSBDHIDKeyboardDevice
```

Members:

ui16VID The vendor ID that this device is to present in the device descriptor.

ui16PID The product ID that this device is to present in the device descriptor.

ui16MaxPowermA The maximum power consumption of the device, expressed in milliamps.

ui8PwrAttributes Indicates whether the device is self- or bus-powered and whether or not it supports remote wakeup. Valid values are **USB_CONF_ATTR_SELF_PWR** or **USB_CONF_ATTR_BUS_PWR** , optionally ORed with **USB_CONF_ATTR_RWAKE** .

pfnCallback A pointer to the callback function which is called to notify the application of general events and those related to reception of Output and Feature reports via the (optional) interrupt OUT endpoint.

pvCBData A client-supplied pointer which is sent as the first parameter in all calls made to the keyboard callback, pfnCallback.

ppui8StringDescriptors A pointer to the string descriptor array for this device. This array must contain the following string descriptor pointers in this order. Language descriptor, Manufacturer name string (language 1), Product name string (language 1), Serial number string (language 1), HID Interface description string (language 1), Configuration description string (language 1).

If supporting more than 1 language, the descriptor block (except for string descriptor 0) must be repeated for each language defined in the language descriptor.

ui32NumStringDescriptors The number of descriptors provided in the ppStringDescriptors array. This must be (1 + (5 * (num languages))).

sPrivateData The private instance data for this device. This memory must remain accessible for as long as the keyboard device is in use and must not be modified by any code outside the HID keyboard driver.

Description:

This structure is used by the application to define operating parameters for the HID keyboard device.

2.18.3 Define Documentation

2.18.3.1 KEYB_ERR_NOT_CONFIGURED

Definition:

```
        #define KEYB_ERR_NOT_CONFIGURED
```

Description:

USBDHIDKeyboardKeyStateChange() returns this value if it is called before the USB host has connected and configured the device. Any key usage code passed is stored and passed to the host once configuration completes.

2.18.3.2 KEYB_ERR_NOT_FOUND

Definition:

```
#define KEYB_ERR_NOT_FOUND
```

Description:

USBDHIDKeyboardKeyStateChange() returns this value if it is called with the bPress parameter set to false but with a ui8UsageCode parameter which does does not indicate a key that is currently recorded as being pressed. This may occur if an attempt was previously made to report more than 6 pressed keys and the earlier pressed keys are released before the later ones. This condition is benign and should not be used to indicate a host disconnection or serious error.

2.18.3.3 KEYB_ERR_TOO_MANY_KEYS

Definition:

```
#define KEYB_ERR_TOO_MANY_KEYS
```

Description:

This return code from USBDHIDKeyboardKeyStateChange() indicates that an attempt has been made to record more than 6 simultaneously pressed, non-modifier keys. The USB HID BIOS keyboard protocol allows no more than 6 pressed keys to be reported at one time. Until at least one key is released, the device reports a roll over error to the host each time it is asked for the keyboard input report.

2.18.3.4 KEYB_ERR_TX_ERROR

Definition:

```
#define KEYB_ERR_TX_ERROR
```

Description:

This return code from USBDHIDKeyboardKeyStateChange() indicates that an error was reported while attempting to send a report to the host. A client should assume that the host has disconnected if this return code is seen.

2.18.3.5 KEYB_MAX_CHARS_PER_REPORT

Definition:

```
#define KEYB_MAX_CHARS_PER_REPORT
```

Description:

The maximum number of simultaneously-pressed, non-modifier keys that the HID BIOS keyboard protocol can send at once. Attempts to send more pressed keys than this results in a rollover error being reported to the host and KEYB_ERR_TOO_MANY_KEYS being returned from USBDHIDKeyboardKeyStateChange().

2.18.3.6 KEYB_SUCCESS

Definition:
```
#define KEYB_SUCCESS
```

Description:
This return code from USBDHIDKeyboardKeyStateChange() indicates success.

2.18.3.7 USBD_HID_KEYB_EVENT_SET_LEDS

Definition:
```
#define USBD_HID_KEYB_EVENT_SET_LEDS
```

Description:
This event indicates that the keyboard LED states are to be set. The ui32MsgValue parameter contains the requested state for each of the LEDs defined as a collection of ORed bits where a 1 indicates that the LED is to be turned on and a 0 indicates that it should be turned off. The individual LED bits are defined using labels **HID_KEYB_NUM_LOCK**, **HID_KEYB_CAPS_LOCK**, **HID_KEYB_SCROLL_LOCK**, **HID_KEYB_COMPOSE** and **HID_KEYB_KANA**.

2.18.4 Function Documentation

2.18.4.1 USBDHIDKeyboardCompositeInit

Initializes HID keyboard device operation for a given USB controller.

Prototype:
```
void *
USBDHIDKeyboardCompositeInit(uint32_t ui32Index,
                             tUSBDHIDKeyboardDevice *psHIDKbDevice,
                             tCompositeEntry *psCompEntry)
```

Parameters:
> ***ui32Index*** is the index of the USB controller which is to be initialized for HID keyboard device operation.
>
> ***psHIDKbDevice*** points to a structure containing parameters customizing the operation of the HID keyboard device.
>
> ***psCompEntry*** is the composite device entry to initialize when creating a composite device.

Description:
This call is very similar to USBDHIDKeyboardInit() except that it is used for initializing an instance of the HID keyboard device for use in a composite device. If this HID keyboard is part of a composite device, then the *psCompEntry* should point to the composite device entry to initialize. This is part of the array that is passed to the USBDCompositeInit() function.

Returns:
Returns zero on failure or a non-zero instance value that should be used with the remaining USB HID Keyboard APIs.

2.18.4.2 USBDHIDKeyboardInit

Initializes HID keyboard device operation for a given USB controller.

Prototype:
```
void *
USBDHIDKeyboardInit(uint32_t ui32Index,
                    tUSBDHIDKeyboardDevice *psHIDKbDevice)
```

Parameters:
>   ***ui32Index*** is the index of the USB controller which is to be initialized for HID keyboard device operation.

>   ***psHIDKbDevice*** points to a structure containing parameters customizing the operation of the HID keyboard device.

Description:
>   An application wishing to offer a USB HID keyboard interface to a USB host must call this function to initialize the USB controller and attach the keyboard device to the USB bus. This function performs all required USB initialization.

>   On successful completion, this function returns the *psHIDKbDevice* pointer passed to it. This must be passed on all future calls to the HID keyboard device driver.

>   When a host connects and configures the device, the application callback receives **USB_EVENT_CONNECTED** after which calls can be made to USBDHIDKeyboardKeyStateChange() to report key presses and releases to the USB host.

Note:
>   The application must not make any calls to the lower level USB device interfaces if interacting with USB via the USB HID keyboard device class API. Doing so causes unpredictable (though almost certainly unpleasant) behavior.

Returns:
>   Returns NULL on failure or the *psHIDKbDevice* pointer on success.

2.18.4.3 USBDHIDKeyboardKeyStateChange

Reports a key state change to the USB host.

Prototype:
```
uint32_t
USBDHIDKeyboardKeyStateChange(void *pvKeyboardDevice,
                              uint8_t ui8Modifiers,
                              uint8_t ui8UsageCode,
                              bool bPress)
```

Parameters:
>   ***pvKeyboardDevice*** is the pointer to the device instance structure as returned by USBDHIDKeyboardInit().

>   ***ui8Modifiers*** contains the states of each of the keyboard modifiers (left/right shift, ctrl, alt or GUI keys). Valid values are logical OR combinations of the labels **HID_KEYB_LEFT_CTRL**, **HID_KEYB_LEFT_SHIFT**, **HID_KEYB_LEFT_ALT**, **HID_KEYB_LEFT_GUI**, **HID_KEYB_RIGHT_CTRL**, **HID_KEYB_RIGHT_SHIFT**,

HID_KEYB_RIGHT_ALT and **HID_KEYB_RIGHT_GUI**. Presence of one of these bit flags indicates that the relevant modifier key is pressed and absence indicates that it is released.

ui8UsageCode is the usage code of the key whose state has changed. If only modifier keys have changed, **HID_KEYB_USAGE_RESERVED** should be passed in this parameter.

bPress is **true** if the key has been pressed or **false** if it has been released. If only modifier keys have changed state, this parameter is ignored.

Description:

This function adds or removes a key usage code from the list of keys currently pressed and schedules a report transmission to the host to inform it of the new keyboard state. If the maximum number of simultaneous key presses are already recorded, the report to the host contains the rollover error code, **HID_KEYB_USAGE_ROLLOVER** instead of key usage codes and the caller receives return code **KEYB_ERR_TOO_MANY_KEYS**.

Returns:

Returns **KEYB_SUCCESS** if the key usage code was added to or removed from the current list successfully. **KEYB_ERR_TOO_MANY_KEYS** is returned if an attempt is made to press a 7th key (the BIOS keyboard protocol can report no more than 6 simultaneously pressed keys). If called before the USB host has configured the device, **KEYB_ERR_NOT_CONFIGURED** is returned and, if an error is reported while attempting to transmit the report, **KEYB_ERR_TX_ERROR** is returned. If an attempt is made to remove a key from the pressed list (by setting parameter *bPressed* to **false**) but the key usage code is not found, **KEYB_ERR_NOT_FOUND** is returned.

2.18.4.4 USBDHIDKeyboardPowerStatusSet

Reports the device power status (bus or self powered) to the USB library.

Prototype:

```
void
USBDHIDKeyboardPowerStatusSet(void *pvKeyboardDevice,
                             uint8_t ui8Power)
```

Parameters:

pvKeyboardDevice is the pointer to the keyboard device instance structure.

ui8Power indicates the current power status, either **USB_STATUS_SELF_PWR** or **USB_STATUS_BUS_PWR**.

Description:

Applications which support switching between bus or self powered operation should call this function whenever the power source changes to indicate the current power status to the USB library. This information is required by the USB library to allow correct responses to be provided when the host requests status from the device.

Returns:

None.

2.18.4.5 USBDHIDKeyboardRemoteWakeupRequest

Requests a remote wake up to resume communication when in suspended state.

Prototype:
```
bool
USBDHIDKeyboardRemoteWakeupRequest(void *pvKeyboardDevice)
```

Parameters:
> ***pvKeyboardDevice*** is the pointer to the keyboard device instance structure.

Description:
> When the bus is suspended, an application which supports remote wake up (advertised to the host via the configuration descriptor) may call this function to initiate remote wake up signaling to the host. If the remote wake up feature has not been disabled by the host, this causes the bus to resume operation within 20mS. If the host has disabled remote wake up, **false** is returned to indicate that the wake up request was not successful.

Returns:
> Returns **true** if the remote wake up is not disabled and the signaling was started or **false** if remote wake up is disabled or if signaling is currently ongoing following a previous call to this function.

2.18.4.6 USBDHIDKeyboardSetCBData

Sets the client-specific pointer parameter for the keyboard callback.

Prototype:
```
void *
USBDHIDKeyboardSetCBData(void *pvKeyboardDevice,
                         void *pvCBData)
```

Parameters:
> ***pvKeyboardDevice*** is the pointer to the device instance structure as returned by USBDHID-KeyboardInit().
> ***pvCBData*** is the pointer that client wishes to be provided on each event sent to the keyboard callback function.

Description:
> The client uses this function to change the callback pointer passed in the first parameter on all callbacks to the *pfnCallback* function passed on USBDHIDKeyboardInit().

> If a client wants to make runtime changes in the callback pointer, it must ensure that the *pvKeyboardDevice* structure passed to USBDHIDKeyboardInit() resides in RAM. If this structure is in flash, callback data changes is not possible.

Returns:
> Returns the previous callback pointer that was set for this instance.

2.18.4.7 USBDHIDKeyboardTerm

Shuts down the HID keyboard device.

Prototype:
```
void
USBDHIDKeyboardTerm(void *pvKeyboardDevice)
```

Parameters:

pvKeyboardDevice is the pointer to the device instance structure as returned by USBDHID-
KeyboardInit().

Description:

This function terminates HID keyboard operation for the instance supplied and removes the
device from the USB bus. Following this call, the *pvKeyboardDevice* instance may not me
used in any other call to the HID keyboard device other than USBDHIDKeyboardInit().

Returns:

None.

2.19   HID Gamepad Device Class API

The HID Gamepad Device Class API provides an interface for applications to provide USB
gamepad support to a USB host. The default gamepad supports 3 8-bit signed axis values (X,
Y, and Z) and 8 buttons. The USB gamepad is configured to allow the host to poll at 1ms rate for
the values. The application calls the USBDHIDGamepadSendReport() function to provide data to
the USB library.

2.19.1   Defining a HID Gamepad Device

In order to properly configure an application to function as a HID Gamepad device, there are a few
structures that the application must properly declare and pass to the USB library. These include
declaring the strings that are passed back to the USB host, the basic configuration of the device
and an optional report descriptor to customize the data that is sent back to the USB host.

The following header files are needed to properly define all of the structures needed to implement
a USB gamepad device:

```
#include "usblib/usblib.h"
#include "usblib/usbhid.h"
#include "usblib/device/usbdevice.h"
#include "usblib/device/usbdhid.h"
#include "usblib/device/usbdhidgamepad.h"
```

The application must first define the string table that is used to describe various features of the
device to the host system. A detailed description of how to declare a string table is found in the
Using the HID Device Class Driver section of this document. This table must include a minimum of 6
entries with the following meanings: languages available, manufacturer name, product description,
serial number, HID interface description, and a configuration description.

The application must also provide a gamepad event handler function. This function responds to
the **USB_EVENTS_** parameter passed to the application when events occur. The HID gamepad
device API sends the following events to the application callback function:

- **USB_EVENT_CONNECTED** - USB connection event occurred.
- **USB_EVENT_DISCONNECTED** - USB disconnect event occurred.
- **USB_EVENT_TX_COMPLETE** - A previously scheduled transmit request completed.
- **USB_EVENT_ERROR** - An error prevented the call from completing successfully.

- **USB_EVENT_SUSPEND** - A USB suspend event occurred.

- **USB_EVENT_RESUME** - A USB resume event occurred.

- **USB_EVENT_LPM_RESUME** - A USB LPM resume event occurred.

- **USB_EVENT_LPM_SLEEP** - A USB LPM sleep event occurred.

- **USB_EVENT_LPM_ERROR** - A USB LPM error occurred.

Note: The **USB_EVENT_DISCONNECTED** event is not reported to the application if the MCU's USB0VBUS pin is connected to a fixed +5 Volts rather than directly to the VBUS pin on the USB connector or if the USB controller is configured to force device mode using the USBStackModeSet() function.

Note: The **USB_EVENT_LPM_**$*$ events are not supported on all microcontrollers. Please check the device's data sheet to see if USB LPM is supported on the device that is in use.

The string table and event handler function are used in the final required structure to complete the definition of the USB game device to the host. The application must define a *tUSBDHIDGamepadDevice* structure and initialize all fields so that the USB library can enumerate the device to the host.

Example: Gamepad Device Definition.

```
const tUSBDHIDGamepadDevice g_sGamepadDevice =
{
    //
    // The Vendor ID assigned by USB-IF.
    //
    USB_VID_YOUR_VENDOR_ID,

    //
    // The Product ID assigned to this device.
    //
    USB_PID_YOUR_PRODUCT_ID,

    //
    // The power consumption of the device in milliamps.
    //
    POWER_CONSUMPTION_MA,

    //
    // This value is passed to the host in the USB configuration descriptor's
    // bmAttributes field.
    //
    USB_CONF_ATTR_SELF_PWR,

    //
    // A pointer to the gamepad callback event handler.
    //
    YourGamepadHandler,

    //
    // A value that the application passes to the gamepad callback function.
    //
    (void *)&g_sYourInstanceData,

    //
    // A pointer to the string table for the device.
    //
    g_pStringDescriptors,

    //
    // The number of entries in your string table, which must equal
```

```
    // (1 + (5 * (number of languages))).
    //
    NUM_STRING_DESCRIPTORS,

    //
    // An optional override HID descriptor for the gamepad, the value of which
    // can be 0 if the application is using the default gamepad descriptor.
    //
    g_pui8GameReportDescriptor,

    //
    // The size of the optional override HID descriptor for the
    // gamepad, which must be 0 if the application is using the default
    // gamepad descriptor or the size of the custom report descriptor in bytes.
    //
    sizeof(g_pui8GameReportDescriptor)
};
```

Finally, to initialize a single gamepad device, the application must call the HID gamepad device API initialization function, USBDHIDGamepadInit(), to configure the USB controller and prepare the device to be connected to the host. Assuming the *psGamepad* returned is not NULL, the gamepad device is now ready to communicate with a USB host.

Example: Gamepad initialization.

```
    //
    // Initialize a gamepad HID instance.
    //
    psGamepad = USBDHIDGamepadInit(0, &g_sGamepadDevice);
```

Once the host connects, your gamepad event handler is sent the **USB_EVENT_CONNECTED** event after which calls can be made to USBDHIDGamepadSendReport() to provide up to date information from the application to the USB host. After a successful call to USBDHIDGamepad-SendReport(), the application must wait for a **USB_EVENT_TX_COMPLETE** event before sending more data.

Example: Sending a gamepad report.

```
    tGamepadReport sReport;

    ...

    //
    // Send a gamepad HID report.
    //
    USBDHIDGamepadSendReport(psGamepad, &sReport, sizeof(sReport);
```

2.19.2   Defining a Custom HID Gamepad Report

This section handles some of the considerations when defining an application-supplied report descriptor. HID report descriptors are infinitely configurable and can be confusing to define properly, so a good understanding of the USB HID specification is required. The USB HID class supported by the USB library is based on the USB HID 1.11 specification available from usb.org, which should be reviewed before attempting to declare a custom report descriptor. The definition of the structure is as similiar as possible to the USB specification to make it as easy as possible to properly define a structure similar to the USB specification.

The example below defines a more complicated 12-bit four-axis controller with 16 buttons. Each of the 4 12-bit values are padded with a single 4-bit constant so that the values are easily accessed by the application. This padding is not required, but does make sending the data much more convenient for the application by allowing a structure like the tCustomReport in the example below. The final value in the report is a single 16-bit value containing the state of all 16 buttons as individual bits.

Example: A 12-bit four-axis 8 button gamepad report

```
//*****************************************************************************
//
// The following is the HID report structure definition that is passed back
// to the host.
//
//*****************************************************************************
static const uint8_t g_pui8GameReportDescriptor[] =
{
    UsagePage(USB_HID_GENERIC_DESKTOP),
    Usage(USB_HID_JOYSTICK),
    Collection(USB_HID_APPLICATION),

        //
        // The axis for the controller.
        //
        UsagePage(USB_HID_GENERIC_DESKTOP),
        Usage (USB_HID_POINTER),
        Collection (USB_HID_PHYSICAL),

            //
            // The X, Y, RX, and RY values, which are specified as 8-bit
            // absolute position values.
            //
            Usage (USB_HID_X),

            //
            // 12-bit absolute X value.
            //
            ReportSize(12),
            ReportCount(1),
            Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
                  USB_HID_INPUT_ABS),

            //
            // 4-bit padding to 16 bits.
            //
            ReportCount(1),
            ReportSize(4),
            Input(USB_HID_INPUT_CONSTANT),

            //
            // 12-bit absolute Y value.
            //
            Usage (USB_HID_Y),
            ReportSize(12),
            ReportCount(1),
            Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
                  USB_HID_INPUT_ABS),

            //
            // 4-bit padding to 16 bits.
            //
            ReportCount(1),
            ReportSize(4),
            Input(USB_HID_INPUT_CONSTANT),
```

```
                    //
                    // 12-bit absolute RX value.
                    //
                    Usage (USB_HID_RX),
                    ReportSize(12),
                    ReportCount(1),
                    Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
                          USB_HID_INPUT_ABS),

                    //
                    // 4-bit padding to 16 bits.
                    //
                    ReportCount(1),
                    ReportSize(4),
                    Input(USB_HID_INPUT_CONSTANT),

                    //
                    // 12-bit absolute RY value.
                    //
                    Usage (USB_HID_RY),
                    ReportSize(12),
                    ReportCount(1),
                    Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
                          USB_HID_INPUT_ABS),

                    //
                    // 4-bit padding to 16 bits.
                    //
                    ReportCount(1),
                    ReportSize(4),
                    Input(USB_HID_INPUT_CONSTANT),

                    //
                    // The 16 buttons.
                    //
                    UsagePage(USB_HID_BUTTONS),
                    UsageMinimum(1),
                    UsageMaximum(16),
                    LogicalMinimum(0),
                    LogicalMaximum(1),
                    PhysicalMinimum(0),
                    PhysicalMaximum(1),

                    //
                    // 16 1-bit values for the buttons.
                    //
                    ReportSize(1),
                    ReportCount(16),
                    Input(USB_HID_INPUT_DATA | USB_HID_INPUT_VARIABLE |
                          USB_HID_INPUT_ABS),

            EndCollection,
        EndCollection
};

//*****************************************************************************
//
// The packed custom report structure that is sent to the host to match the
// report descriptor defined above.
//
//*****************************************************************************
typedef struct
{
    uint16_t i16XPos;
    uint16_t i16YPos;
    uint16_t i16RXPos;
```

```
        uint16_t i16RYPos;
        uint16_t ui16Buttons;
}
PACKED tCustomReport;
```

2.20 HID Gamepad Device Class API Definitions

Data Structures

- tGamepadReport
- tUSBDHIDGamepadDevice

Defines

- USBDGAMEPAD_NOT_CONFIGURED
- USBDGAMEPAD_SUCCESS
- USBDGAMEPAD_TX_ERROR

Functions

- tUSBDHIDGamepadDevice ∗ USBDHIDGamepadCompositeInit (uint32_t ui32Index, tUSBD-HIDGamepadDevice ∗psGamepad, tCompositeEntry ∗psCompEntry)
- tUSBDHIDGamepadDevice ∗ USBDHIDGamepadInit (uint32_t ui32Index, tUSBD-HIDGamepadDevice ∗psGamepad)
- uint32_t USBDHIDGamepadSendReport (tUSBDHIDGamepadDevice ∗psHIDGamepad, void ∗pvReport, uint32_t ui32Size)
- void USBDHIDGamepadTerm (tUSBDHIDGamepadDevice ∗psGamepad)

2.20.1 Detailed Description

The macros and functions defined in this section can be found in header file `device/usbdhidgamepad.h`.

2.20.2 Data Structure Documentation

2.20.2.1 tGamepadReport

Definition:
```
typedef struct
{
    int8_t i8XPos;
    int8_t i8YPos;
    int8_t i8ZPos;
    uint8_t ui8Buttons;
```

```
        }
        tGamepadReport
```

Members:

    i8XPos Signed 8-bit value (-128 to 127).

    i8YPos Signed 8-bit value (-128 to 127).

    i8ZPos Signed 8-bit value (-128 to 127).

    ui8Buttons 8-bit button mapping with button 1 in the LSB.

Description:

    This structure is the default packed report structure that is sent to the host. The application can provide its own structure if the default report descriptor is overridden by the application. This structure or an application-defined structure is passed to the USBDHIDGamepadSendReport function to send gamepad updates to the host.

2.20.2.2 tUSBDHIDGamepadDevice

Definition:

```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const uint16_t ui16MaxPowermA;
    const uint8_t ui8PwrAttributes;
    const tUSBCallback pfnCallback;
    void *pvCBData;
    const uint8_t *const *ppui8StringDescriptors;
    const uint32_t ui32NumStringDescriptors;
    const uint8_t *pui8ReportDescriptor;
    const uint32_t ui32ReportSize;
    tUSBDGamepadInstance sPrivateData;
}
tUSBDHIDGamepadDevice
```

Members:

    ui16VID The vendor ID that this device is to present in the device descriptor.

    ui16PID The product ID that this device is to present in the device descriptor.

    ui16MaxPowermA The maximum power consumption of the device, expressed in milliamps.

    ui8PwrAttributes Indicates whether the device is self- or bus-powered and whether or not it supports remote wake up. Valid values are **USB_CONF_ATTR_SELF_PWR** or **USB_CONF_ATTR_BUS_PWR** , optionally ORed with **USB_CONF_ATTR_RWAKE** .

    pfnCallback A pointer to the callback function that is called to notify the application of general events. This pointer must point to a valid function.

    pvCBData A client-supplied pointer that is sent as the first parameter in all calls made to the pfnCallback gamedevice callback function.

    ppui8StringDescriptors A pointer to the string descriptor array for this device. This array must contain the following string descriptor pointers in this order: Language descriptor, Manufacturer name string (language 1), Product name string (language 1), Serial number string (language 1), HID Interface description string (language 1), Configuration description string (language 1).

If supporting more than 1 language, the descriptor block (except for string descriptor 0) must be repeated for each language defined in the language descriptor.

ui32NumStringDescriptors The number of descriptors provided in the *ppStringDescriptors* array, which must be (1 + (5 * (number of languages))).

pui8ReportDescriptor Optional report descriptor if the application wants to use a custom descriptor.

ui32ReportSize The size of the optional report descriptor define in pui8ReportDescriptor.

sPrivateData The private instance data for this device. This memory must remain accessible for as long as the game device is in use and must not be modified by any code outside the HID game device driver.

Description:
This structure is used by the application to define operating parameters for the HID game device.

2.20.3 Define Documentation

2.20.3.1 USBDGAMEPAD_NOT_CONFIGURED

Definition:
```
#define USBDGAMEPAD_NOT_CONFIGURED
```

Description:
The device is not currently configured and cannot perform any operations.

2.20.3.2 USBDGAMEPAD_SUCCESS

Definition:
```
#define USBDGAMEPAD_SUCCESS
```

Description:
The USBDHIDGamepadSendReport() call successfully scheduled the report.

2.20.3.3 USBDGAMEPAD_TX_ERROR

Definition:
```
#define USBDGAMEPAD_TX_ERROR
```

Description:
The USBDHIDGamepadSendReport() function could not send the report at this time.

2.20.4 Function Documentation

2.20.4.1 USBDHIDGamepadCompositeInit

Initializes HID gamepad device operation for a given USB controller.

Prototype:
```
tUSBDHIDGamepadDevice *
USBDHIDGamepadCompositeInit(uint32_t ui32Index,
                           tUSBDHIDGamepadDevice *psGamepad,
                           tCompositeEntry *psCompEntry)
```

Parameters:
> ***ui32Index*** is the index of the USB controller that is to be initialized for HID gamepad device operation.
>
> ***psGamepad*** points to a structure containing parameters customizing the operation of the HID gamepad device.
>
> ***psCompEntry*** is the composite device entry to initialize when creating a composite device.

Description:
> This call is very similar to USBDHIDGamepadInit() except that it is used for initializing an instance of the HID gamepad device for use in a composite device. If this HID gamepad is part of a composite device, then the *psCompEntry* should point to the composite device entry to initialize. This entry is part of the array that is passed to the USBDCompositeInit() function to start up and complete configuration of a composite USB device.

Returns:
> Returns NULL on failure or the *psGamepad* value that should be used with the remaining USB HID gamepad APIs.

2.20.4.2 USBDHIDGamepadInit

Initializes HID gamepad device operation for a given USB controller.

Prototype:
```
tUSBDHIDGamepadDevice *
USBDHIDGamepadInit(uint32_t ui32Index,
                   tUSBDHIDGamepadDevice *psGamepad)
```

Parameters:
> ***ui32Index*** is the index of the USB controller that is to be initialized for HID gamepad device operation.
>
> ***psGamepad*** points to a structure containing parameters customizing the operation of the HID gamepad device.

Description:
> An application that enables a USB HID gamepad interface to a USB host must call this function to initialize the USB controller and attach the gamepad device to the USB bus. This function performs all required USB initialization, and the device is ready for operation on the function return.
>
> On successful completion, this function returns the modified *psGamepad* pointer passed to it or returns a NULL pointer if there was a problem. This pointer must be passed on all future calls to the HID gamepad device driver.
>
> When a host connects and configures the device, the application callback receives **USB_EVENT_CONNECTED**, after which calls can be made to USBDHIDGamepad-SendReport() to report changes to the gamepad interface to the USB host when it requests them.

Note:

The application must not make any calls to the lower level USB device interfaces if interacting with USB via the USB HID gamepad device class API.

Returns:

Returns NULL on failure or the *psGamepad* pointer on success.

2.20.4.3 USBDHIDGamepadSendReport

Schedules a report to be sent once the host requests more data.

Prototype:
```
uint32_t
USBDHIDGamepadSendReport(tUSBDHIDGamepadDevice *psHIDGamepad,
                         void *pvReport,
                         uint32_t ui32Size)
```

Parameters:

psHIDGamepad is the structure pointer that is returned from the USBDHIDGamepadCompositeInit() or USBDHIDGamepadInit() functions.

pvReport is the data to send to the host.

ui32Size is the number of bytes in the *pvReport* buffer.

Description:

This call is made by an application to schedule data to be sent to the host when the host requests an update from the device. The application must then wait for a **USB_EVENT_TX_COMPLETE** event in the function provided in the *pfnCallback* pointer in the tUSBDHIDGamepadDevice structure before being able to send more data with this function. The pointer passed in the *pvReport* can be updated once this call returns as the data has been copied from the buffer. The function returns **USBDGAMEPAD_SUCCESS** if the transmission was successfully scheduled or **USBDGAMEPAD_TX_ERROR** if the report could not be sent at this time. If the call is made before the device is connected or ready to communicate with the host, then the function can return **USBDGAMEPAD_NOT_CONFIGURED**.

Returns:

The function returns one of the **USBDGAMEPAD_∗** values.

2.20.4.4 USBDHIDGamepadTerm

Shuts down the HID gamepad device.

Prototype:
```
void
USBDHIDGamepadTerm(tUSBDHIDGamepadDevice *psGamepad)
```

Parameters:

psGamepad is the pointer to the device instance structure as returned by USBDHIDGamepadInit() or USBDHIDGamepadCompositeInit().

Description:
This function terminates HID gamepad operation for the instance supplied and removes the device from the USB bus. Following this call, the *psGamepad* instance may not me used in any other call to the HID gamepad device other than to reinitialize by calling USBDHIDGamepadInit() or USBDHIDGamepadCompositeInit().

Returns:
None.

2.21 Mass Storage Device Class API

The USB mass storage device class allows an application to act as a physical storage device for use by another USB application or for a host operating system. Because the type of storage can vary per application, the mass storage class abstracts the storage with a set of block based APIs that are provided by the application to the USB library. These APIs allow the USB mass storage class to call an external set of functions that actually perform the operations on the physical storage media. The storage APIs are given to the USB library's mass storage device class initialization function and are called by the USB library whenever it needs to access the physical media. The mass storage class implementation does not require any run time calls once it is initialized. This is because all interaction with the mass storage class occur through the callback function that is provided to the USB library's mass storage class interface. The callback function is used to notify the application when the device is being read, written or has gone idle. Depending on the operating system and how it accesses the mass storage device, these callbacks may occur at a very rapid rate (once per block) or slower (once per file). The USB library's mass storage class also provides an interface to inform the USB library when the media status has changed in case the physical media can be ejected. This is important for devices like SD cards that can be removed before the device is connected or even while the device is running.

Note:
Some care must be taken by the application if it wishes to share the media with the mass storage class. Proper access protection must be in place as the mass storage class may be using the media access functions during the USB interrupt.

2.21.1 Initialization

The USB library's mass storage class provides a simple interface to initialize the mass storage class and pass it the needed functions to access a device without having any knowledge of the physical media. The USBDMSCInit() function is the only initialization required by the mass storage class and it uses the structure tUSBDMSCDevice to hold all customizable values for the mass storage class.

2.21.2 Customization

The USB library's mass storage class provides the ability to customize how the device is reported to the USB host controller in the tUSBDMSCDevice structure. The members of this structure contain all of the customizable parameters.

2.21.2.1 VID and PID

The VID and PID values reported to the host controller are provided in the usVID and usPID members of the tUSBDMSCDevice structure and should be valid for the application and unique for all vendors and devices. The mass storage device class also reports some class specific strings to the operating system which can be customized in the pucVendor, pucProduct, and pucVersion structure members. These strings may or may not be requested by the USB host controller, however they are required. The pucVendor member is an 8 byte character string for the Vendor, and should be exactly 8 bytes padded with spaces(for example: "TI "). The pucProduct member is a 16 character string that indicates the product type and it too should be padded out with spaces to 16 bytes(for example: "Mass Storage "). The last customizable mass storage string is contained in pucVersion member and is a character based version which should be a 4 bytes padded with spaces (for example: "1.00").

2.21.2.2 Power Configuration

The power configuration for the mass storage class device is held in two of the tUSBDMSCDevice members variables, *usMaxPowermA* and *ucPwrAttributes*. The *usMaxPowermA* variable holds the maximum power consumption for the device and is expressed in milliamps. The power configuration is held in the ucPwrAttributes member variable and indicates whether the device is self or bus powered. Valid values are **USB_CONF_ATTR_SELF_PWR** or **USB_CONF_ATTR_BUS_PWR**.

2.21.2.3 Device Strings

The remaining USB strings are contained in the *ppui8StringDescriptors* and are just like the strings that are used in every other device class in the USB library. The *ppui8StringDescriptors* is a pointer to the string descriptor array for this device. This array must contain the following string descriptor pointers in this order. Language descriptor, Manufacturer name string (language 1), Product name string (language 1), Serial number string (language 1), MSC Interface description string (language 1), Configuration description string (language 1). If the device is supporting more than 1 language, the descriptor block (except for string descriptor 0) must be repeated for each language defined in the language descriptor. The number of descriptors provided in the *ppui8StringDescriptors* array must be $((5 + (\text{num MSC strings})) * (\text{num languages})) + 1$.

2.21.2.4 Media Access Functions

The media access functions are passed in to the USB mass storage device class in the sMediaFunctions member variable. This structure holds the access functions for the media used by this instance of the mass storage class device. All of the functions in this structure are required to be filled out with valid functions. These function is called by the USB mass storage device class whenever it needs to read or write the physical media and assumes fixed block sizes of 512 bytes for the media.

2.21.2.5 Event Callbacks

In some cases the application may need to be informed when the state of the mass storage device has changed. The *pfnEventCallback* member of the tUSBDMSCDevice structure

provides event notification to applications for the following events: **USBD_MSC_EVENT_IDLE**, **USBD_MSC_EVENT_READING**, and **USBD_MSC_WRITING**. When the function of type tUSB-Callback is called, only the first two parameters *pvCBData* and *ui32Event* parameters are valid. The *pvCBData* is the value that was returned when the application called USBDMSCInit() and can be used with other APIs. The ui32Event parameter is one of the **USBD_MSC_EVENT_∗** values.

2.21.3   the Mass Storage Device Class:

The following sections has some coding examples for the initialization of the mass storage class structures as well as how to call the initialization function itself.

Example: Device Structure

```
const tUSBDMSCDevice g_sMSCDevice =
{
    //
    // Vendor ID.
    //
    USB_VID_TI_1CBE,

    //
    // Product ID.
    //
    USB_PID_MSC,

    //
    // Vendor Information.
    //
    "TI      ",

    //
    // Product Identification.
    //
    "Mass Storage    ",

    //
    // Revision.
    //
    "1.00",

    //
    // 500mA.
    500,

    //
    // Bus Powered.
    //
    USB_CONF_ATTR_BUS_PWR,

    //
    // A list of string descriptors and the number of descriptors.
    //
    g_pStringDescriptors,
    NUM_STRING_DESCRIPTORS,

    //
    // The media access functions.
    //
    {
        USBDMSCStorageOpen,
        USBDMSCStorageClose,
```

```
        USBDMSCStorageRead,
        USBDMSCStorageWrite,
        USBDMSCStorageNumBlocks
    },

    //
    // The event notification call back function.
    //
    USBMSCDEventCallback,
};
```

The initialization and configuration calls for the USB library's mass storage device class are handled by a single call that passes in the [tUSBDMSCDevice](#) structure.

Example: Initialization code

```
    //
    // Pass our device information to the USB library and place the device
    // on the bus.
    //
    USBDMSCInit(0, (tUSBDMSCDevice *)&g_sMSCDevice);

    //
    // Drop into the main loop.
    //
    while(1)
    {
    }
```

The application's event call back function provides the application with notifications of changes in the USB mass storage class. The application can use this information to update it's own state. The events may occur in rapid succession and the application must be careful not to spend much time in this function as it is called from a interrupt handler. The application should expect many calls to this function during USB transfers.

Example: Callback handler.

```
uint32_t
USBDMSCEventCallback(void *pvCBData, uint32_t ui32Event,
                     uint32_t ui32MsgParam, void *pvMsgData)
{
    switch(ui32Event)
    {
        //
        // Writing to the device.
        //
        case USBD_MSC_EVENT_WRITING:
        {
            //
            //  Handle write case.
            //

            ...

            break;
        }

        //
        // Reading from the device.
        //
        case USBD_MSC_EVENT_READING:
        {
```

```
                    //
                    //  Handle read case.
                    //

                    ...

                    break;
                }
                case USBD_MSC_EVENT_IDLE:
                {
                    //
                    //  Handle idle case.
                    //

                    ...

                }
                default:
                {
                    break;
                }
        }
        return(0);
}
```

2.21.4 Using the Composite Mass Storage Device Class

When using the mass storage device class in a composite device, the configuration of the device is very similar to how it is configured as a non-composite device. Follow all of the configuration steps in the previous section with the exception of calling USBDMSCCompositeInit() instead of USBDMSCInit(). This prepares an instance of the mass storage device class to be enumerated as part of a composite device. The return value from the USBDMSCCompositeInit() function should be placed in the pvInstance member of the tCompositeEntry structure for the mass storage device. The code example below provides an example of how to initialize some of the structures necessary to use the USB mass storage device class in a composite device.

```
//
// These should be initialized with valid values for each class.
//
extern tUSBDMSCDevice g_sMSCDevice;

//
// The array of composite device entries.
//
tCompositeEntry psCompEntries[2];

//
// Allocate the device data for the top level composite device class.
//
tUSBDCompositeDevice g_sCompDevice =
{
    //
    // Texas Intruments C-Series VID.
    //
    USB_VID_TI_1CBE,

    //
    // Texas Intruments C-Series PID for composite serial device.
    //
    USB_PID_YOUR_COMPOSITE_PID,
```

```
        //
        // This is in 2mA increments so 500mA.
        //
        250,

        //
        // Bus powered device.
        //
        USB_CONF_ATTR_BUS_PWR,

        //
        // Composite event handler.
        //
        EventHandler,

        //
        // The string table.
        //
        g_pui8StringDescriptors,
        NUM_STRING_DESCRIPTORS,

        //
        // The Composite device array.
        //
        2,
        g_psCompEntries
    };

    //
    // The OTHER_SIZES here are the sizes of the descriptor data for other classes
    // that are part of the composite device.
    //
    #define DESCRIPTOR_DATA_SIZE    (COMPOSITE_DMSC_SIZE + OTHER_SIZES)
    uint8_t g_pui8DescriptorData[DESCRIPTOR_DATA_SIZE];

    //
    // Save the instance data for this mass storage device.
    //
    pvAudioDevice = USBDMSCCompositeInit(0, &g_sMSCDevice, &psCompEntries[0]);

    ...

    //
    // Initialize the USB controller as a composite device.
    //
    USBDCompositeInit(0, &g_sCompDevice, DESCRIPTOR_DATA_SIZE,
                      g_pui8DescriptorData);
```

All other API calls to the USB mass storage device class should use the value returned by US-BDAudioCompositeInit() when the API calls for a pvInstance pointer. Also when using the audio device in a composite device the **COMPOSITE_DMSC_SIZE** value should be added to the size of the *g_pui8DescriptorData* array as shown in the example above.

2.22 Mass Storage Device Class API Definitions

Data Structures

- tMSCDMedia
- tUSBDMSCDevice

Defines

- COMPOSITE_DMSC_SIZE
- USBD_MSC_EVENT_IDLE
- USBD_MSC_EVENT_READING
- USBD_MSC_EVENT_WRITING

Functions

- void ∗ USBDMSCCompositeInit (uint32_t ui32Index, tUSBDMSCDevice ∗psMSCDevice, tCompositeEntry ∗psCompEntry)
- void ∗ USBDMSCInit (uint32_t ui32Index, tUSBDMSCDevice ∗psMSCDevice)
- void USBDMSCMediaChange (void ∗pvMSCDevice, tUSBDMSCMediaStatus iMediaStatus)
- void USBDMSCTerm (void ∗pvMSCDevice)

2.22.1 Detailed Description

The macros and functions defined in this section can be found in header file `device/usbdmsc.h`.

2.22.2 Data Structure Documentation

2.22.2.1 tMSCDMedia

Definition:
```
typedef struct
{
    void * (*pfnOpen)(uint32_t ui32Drive);
    void (*pfnClose)(void *pvDrive);
    uint32_t (*pfnBlockRead)(void *pvDrive,
                             uint8_t *pui8Data,
                             uint32_t ui32Sector,
                             uint32_t ui32NumBlocks);
    uint32_t (*pfnBlockWrite)(void *pvDrive,
                              uint8_t *pui8Data,
                              uint32_t ui32Sector,
                              uint32_t ui32NumBlocks);
    uint32_t (*pfnNumBlocks)(void *pvDrive);
    uint32_t (*pfnBlockSize)(void *pvDrive);
}
tMSCDMedia
```

Members:
  pfnOpen This function is used to initialize and open the physical drive number associated with the parameter *ui32Drive* . The function returns zero if the drive could not be opened for some reason. In the case of removable device like an SD card this function must return zero if the SD card is not present. The function returns a pointer to data that should be passed to other APIs or returns 0 if no drive was found.

pfnClose This function closes the drive number in use by the mass storage class device. The *pvDrive* is the pointer that was returned from a call to *pfnOpen* . This function is used to close the physical drive number associated with the parameter *pvDrive* . This function returns 0 if the drive was closed successfully and any other value indicates a failure.

pfnBlockRead This function reads a block of data from a device opened by the *pfnOpen* call. The *pvDrive* parameter is the pointer that was returned from the original call to *pfnOpen* . The *pui8Data* parameter is the buffer that data will be written into. The data area pointed to by *pui8Data* must be at least *ui32NumBlocks* * Block Size bytes to prevent overwriting data. The *ui32Sector* is the block address to read and *ui32NumBlocks* is the number of blocks to read. This function returns the number of bytes that were read from the and placed into the *pui8Data* buffer..

pfnBlockWrite This function is use to write blocks to a physical device from the buffer pointed to by the *pui8Data* buffer. The *pvDrive* parameter is the pointer that was returned from the original call to *pfnOpen* . The *pui8Data* is the pointer to the data to write to the storage device and *ui32NumBlocks* is the number of blocks to write. The *ui32Sector* parameter is the sector number used to write the block. If the number of blocks is greater than one then the block address increments and writes to the next block until *ui32NumBlocks* * Block Size bytes are written. This function returns the number of bytes that were written to the device.

pfnNumBlocks This function returns the total number of blocks on a physical device based on the *pvDrive* parameter. The *pvDrive* parameter is the pointer that was returned from the original call to *pfnOpen* .

pfnBlockSize This function returns the block size for a physical device based on the *pvDrive* parameter. The *pvDrive* parameter is the pointer that was returned from the original call to *pfnOpen* .

Description:
Media Access functions.

2.22.2.2   tUSBDMSCDevice

Definition:
```
typedef struct
{
    const uint16_t ui16VID;
    const uint16_t ui16PID;
    const uint8_t pui8Vendor[8];
    const uint8_t pui8Product[16];
    const uint8_t pui8Version[4];
    const uint16_t ui16MaxPowermA;
    const uint8_t ui8PwrAttributes;
    const uint8_t *const *ppui8StringDescriptors;
    const uint32_t ui32NumStringDescriptors;
    const tMSCDMedia sMediaFunctions;
    const tUSBCallback pfnEventCallback;
    tMSCInstance sPrivateData;
}
tUSBDMSCDevice
```

Members:
ui16VID The vendor ID that this device is to present in the device descriptor.
ui16PID The product ID that this device is to present in the device descriptor.

pui8Vendor 8 byte vendor string.

pui8Product 16 byte vendor string.

pui8Version 4 byte vendor string.

ui16MaxPowermA The maximum power consumption of the device, expressed in milliamps.

ui8PwrAttributes Indicates whether the device is self or bus-powered and whether or not it supports remote wakeup. Valid values are **USB_CONF_ATTR_SELF_PWR** or **USB_CONF_ATTR_BUS_PWR** , optionally ORed with **USB_CONF_ATTR_RWAKE** .

ppui8StringDescriptors A pointer to the string descriptor array for this device. This array must contain the following string descriptor pointers in this order. Language descriptor, Manufacturer name string (language 1), Product name string (language 1), Serial number string (language 1), MSC Interface description string (language 1), Configuration description string (language 1).

If supporting more than 1 language, the descriptor block (except for string descriptor 0) must be repeated for each language defined in the language descriptor.

ui32NumStringDescriptors The number of descriptors provided in the *ppStringDescriptors* array. This must be 1 + ((5 + (num HID strings)) * (num languages)).

sMediaFunctions This structure holds the access functions for the media used by this instance of the mass storage class device. All of the functions in this structure are required to be filled out with valid functions.

pfnEventCallback This is the callback function for various events that occur during mass storage class operation.

sPrivateData The private instance data for this device. This memory must remain accessible for as long as the MSC device is in use and must not be modified by any code outside the MSC class driver.

Description:

The structure used by the application to define operating parameters for the mass storage device.

2.22.3 Define Documentation

2.22.3.1 COMPOSITE_DMSC_SIZE

Definition:

```
#define COMPOSITE_DMSC_SIZE
```

Description:

The size of the memory that should be allocated to create a configuration descriptor for a single instance of the USB Audio Device. This does not include the configuration descriptor which is automatically ignored by the composite device class.

2.22.3.2 USBD_MSC_EVENT_IDLE

Definition:

```
#define USBD_MSC_EVENT_IDLE
```

Description:

This event indicates that the host has completed other operations and is no longer accessing the device.

2.22.3.3 USBD_MSC_EVENT_READING

Definition:
```
#define USBD_MSC_EVENT_READING
```

Description:
This event indicates that the host is reading the storage media.

2.22.3.4 USBD_MSC_EVENT_WRITING

Definition:
```
#define USBD_MSC_EVENT_WRITING
```

Description:
This event indicates that the host is writing to the storage media.

2.22.4 Function Documentation

2.22.4.1 USBDMSCCompositeInit

This function should be called once for the mass storage class device to initialized basic operation and prepare for enumeration.

Prototype:
```
void *
USBDMSCCompositeInit(uint32_t ui32Index,
                     tUSBDMSCDevice *psMSCDevice,
                     tCompositeEntry *psCompEntry)
```

Parameters:
> **ui32Index** is the index of the USB controller to initialize for mass storage class device operation.
>
> **psMSCDevice** points to a structure containing parameters customizing the operation of the mass storage device.
>
> **psCompEntry** is the composite device entry to initialize when creating a composite device.

Description:
In order for an application to initialize the USB device mass storage class, it must first call this function with the a valid mass storage device class structure in the *psMSCDevice* parameter. This allows this function to initialize the USB controller and device code to be prepared to enumerate and function as a USB mass storage device. If this mass storage device is part of a composite device, then the *psCompEntry* should point to the composite device entry to initialize. This is part of the array that is passed to the USBDCompositeInit() function.

This function returns a void pointer that must be passed in to all other APIs used by the mass storage class.

See the documentation on the tUSBDMSCDevice structure for more information on how to properly fill the structure members.

Returns:

Returns zero on failure or a non-zero instance value that should be used with the remaining USB mass storage APIs.

2.22.4.2 USBDMSCInit

This function should be called once for the mass storage class device to initialized basic operation and prepare for enumeration.

Prototype:
```
void *
USBDMSCInit(uint32_t ui32Index,
            tUSBDMSCDevice *psMSCDevice)
```

Parameters:

ui32Index is the index of the USB controller to initialize for mass storage class device operation.

psMSCDevice points to a structure containing parameters customizing the operation of the mass storage device.

Description:

In order for an application to initialize the USB device mass storage class, it must first call this function with the a valid mass storage device class structure in the *psMSCDevice* parameter. This allows this function to initialize the USB controller and device code to be prepared to enumerate and function as a USB mass storage device.

This function returns a void pointer that must be passed in to all other APIs used by the mass storage class.

See the documentation on the tUSBDMSCDevice structure for more information on how to properly fill the structure members.

Returns:

Returns 0 on failure or a non-zero void pointer on success.

2.22.4.3 USBDMSCMediaChange

This function is used by an application if it can detect insertion or removal of the media.

Prototype:
```
void
USBDMSCMediaChange(void *pvMSCDevice,
                   tUSBDMSCMediaStatus iMediaStatus)
```

Parameters:

pvMSCDevice is the mass storage device instance that had a media change.

iMediaStatus is the updated status for the media.

Description:

This function should be called by an application when it detects a change in the status of the media in use by the USB mass storage class. The *iMediaStatus* parameter will indicate the

new status of the media and can also indicate that the application has no knowledge of the media state.

There are currently the three following values for the *iMediaStatus* parameter:

- **eUSBDMSCMediaPresent** indicates that the media is present or has been added.
- **eUSBDMSCMediaNotPresent** indicates that the media is not present or was removed.
- **eUSBDMSCMediaUnknown** indicates that the application has no knowledge of the media state and the USB mass storage class.

It will be left up to the application to call this function whenever it detects a change or simply call it once with **eUSBDMSCMediaUnknown** and allow the mass storage class to infer the state from the remaining device APIs.

Note:
It is recommended that the application use this function to inform the mass storage class of media state changes as it will lead to a more responsive system.

Returns:
None.

2.22.4.4 USBDMSCTerm

Shuts down the mass storage device.

Prototype:
```
void
USBDMSCTerm(void *pvMSCDevice)
```

Parameters:
pvMSCDevice is the pointer to the device instance structure as returned by USBDMSCInit() or USBDMSCCompositeInit().

Description:
This function terminates mass storage operation for the instance supplied and removes the device from the USB bus. Following this call, the *pvMSCDevice* instance may not me used in any other call to the mass storage device other than USBDMSCInit() or USBDMSCCompositeInit().

Returns:
None.

2.23   Using the USB Device API

If an existing USB Device Class Driver is not suitable for your application, you may choose to develop your device using the lower-level USB Device API instead. This offers greater flexibility but involves somewhat more work. Creating a device application using the USB Device API involves several steps:

- Build device, configuration, interface and endpoint descriptor structures to describe your device.

■ Write handlers for each of the USB events your device is interested in receiving from the USB library.

■ Call the USB Device API to connect the device to the bus and manage standard host interaction on your behalf.

The following sections walk through each of these steps offering code examples to illustrate the process. Working examples illustrating use of the library can also be found in the DriverLib release for your USB-capable evaluation kit.

The term "device code" used in the following sections describes all class specific code written above the USB Device API to implement a particular USB device application. This may be either application code or a USB device class driver.

2.23.1 Building Descriptors

The USB Device API manages all standard USB descriptors on behalf of the device. These descriptors are provided to the library via four fields in the *tDeviceInfo* structure which is passed on a call to USBDCDInit(). The relevant fields are:

■ *psDeviceDescriptor*
■ *ppui8ConfigDescriptors*
■ *ppui8StringDescriptors*
■ *ui32NumStringDescriptors*

All descriptors are provided as pointers to arrays of uint8_tacters where the contents of the individual descriptor arrays are USB 2.0-compliant descriptors of the appropriate type. For examples of particular descriptors, see the main source files for each of the USB device class drivers (for example `device/usbdbulk.c` for the generic bulk device class driver) or the file `usbdescriptors.c` in the `qs-scope` example application.

2.23.1.1 tDeviceInfo.pDeviceDescriptor

This array must hold the device descriptor that the USB Device API returns to the host in response to a GET_DESCRIPTOR(DEVICE) request. The following example contains the device descriptor provided by a USB HID keyboard device.

```
const uint8_t g_pui8DeviceDescriptor[] =
{
    18,                         // Size of this structure.
    USB_DTYPE_DEVICE,           // Type of this structure.
    USBShort(0x200),            // USB version 2.0.
    USB_CLASS_DEVICE,           // USB Device Class.
    0,                          // USB Device Sub-class.
    USB_HID_PROTOCOL_NONE,      // USB Device protocol.
    64,                         // Maximum packet size for default pipe.
    USBShort(USB_VID_TI_1CBE),  // Vendor ID (VID).
    USBShort(USB_PID_KEYBOARD), // Product ID (PID).
    USBShort(0x100),            // Device Version BCD.
    1,                          // Manufacturer string identifier.
    2,                          // Product string identifier.
    3,                          // Product serial number.
    1                           // Number of configurations.
};
```

Header file `usblib.h` contains macros and labels to help in the construction of descriptors and individual device class header files, such as `usbhid.h` and `device/usbdhid.h` for the Human Interface Device class, provide class specific values and labels.

2.23.1.2 tDeviceInfo.ppui8ConfigDescriptors

While only a single device descriptor is required, multiple configuration descriptors may be offered so the *ppui8ConfigDescriptors* field is an array of pointers to *tConfigHeader* structures, each defining the descriptor for a single configuration. The number of entries in this array must agree with the number of configurations specified in the final byte of the device descriptor provided in the *psDeviceDescriptor* field.

To allow flexibility when defining composite devices, individual configuration descriptors are also defined in terms of an array of structures. In this case, the *tConfigHeader* structure contains a count and a pointer to an array of *tConfigSection* structures each of which contains a pointer to a block of bytes and a size indicating the number of bytes in the section. The sections described in this array are concatenated to generate the full config descriptor published to the host.

Config descriptors are somewhat more complex than device descriptors due to the amount of additional information passed alongside the basic configuration descriptor. In addition to USB 2.0 standard descriptors for the configuration, interfaces and endpoints in use, additional, class specific, descriptors may also be included.

The USB Device API imposes one restriction on configuration descriptors that devices must be aware of. While the USB 2.0 specification does not restrict the values that can be specified in the *bConfigurationValue* field (byte 6) of the configuration descriptor, the USB Device API requires that individual configurations are numbered consecutively starting at 1 for the first configuration.

The following example contains the configuration descriptor structures provided for a USB HID keyboard. This example offers a single configuration containing one interface and using a single interrupt endpoint. In this case, in addition to the standard portions of the descriptor, a Human Interface Device (HID) class descriptor is also included. Due to the use of a standard format for descriptor headers, the USB Device API is capable of safely skipping device specific descriptors when parsing these structures.

In this example, we illustrate the use of multiple sections to build the configuration descriptor. The content of the config descriptor given here is, however, static so it could easily have been defined in terms of a single *tConfigSection* entry instead. The label *g_pucReportDescriptor* is assumed to be a pointer to a HID-specific report descriptor for the keyboard.

Note that the value used to initialize the *wTotalLength* field of the configuration descriptor is irrelevant since the USB library calculates this based on the content of the sections that are concatenated to build the final descriptor.

```
//*****************************************************************************
//
// HID keyboard device configuration descriptor.
//
// It is vital that the configuration descriptor bConfigurationValue field
// (byte 6) is 1 for the first configuration and increments by 1 for each
// additional configuration defined here.  This relationship is assumed in the
// device stack for simplicity even though the USB 2.0 specification imposes
// no such restriction on the bConfigurationValue values.
//
//*****************************************************************************
const uint8_t g_pui8KeyboardDescriptor[] =
{
```

```
    //
    // Configuration descriptor header.
    //
    9,                          // Size of the configuration descriptor.
    USB_DTYPE_CONFIGURATION,    // Type of this descriptor.
    USBShort(34),               // The total size of this full structure
                                // (Value is patched by the USB library so is
                                // not important here)
    1,                          // The number of interfaces in this
                                // configuration.
    1,                          // The unique value for this configuration.
    5,                          // The string identifier that describes this
                                // configuration.
    USB_CONF_ATTR_SELF_PWR,     // Bus Powered, Self Powered, remote wakeup.
    125,                        // The maximum power in 2mA increments.
};

//*****************************************************************************
//
// The interface and HID descriptors for the keyboard device.
//
//*****************************************************************************
uint8_t g_pui8HIDInterface[] =
{
    //
    // HID Device Class Interface Descriptor.
    //
    9,                          // Size of the interface descriptor.
    USB_DTYPE_INTERFACE,        // Type of this descriptor.
    0,                          // The index for this interface.
    0,                          // The alternate setting for this interface.
    1,                          // The number of endpoints used by this
                                // interface.
    USB_CLASS_HID,              // The interface class
    USB_HID_SCLASS_BOOT,        // The interface sub-class.
    USB_HID_PROTOCOL_KEYB,      // The interface protocol for the sub-class
                                // specified above.
    4,                          // The string index for this interface.


    //
    // HID Descriptor.
    //
    9,                          // Size of this HID descriptor.
    USB_HID_DTYPE_HID,          // HID descriptor type.
    USBShort(0x101),            // Version is 1.1.
    0,                          // Country code is not specified.
    1,                          // Number of descriptors.
    USB_HID_DTYPE_REPORT,       // Type of this descriptor.
    USBShort(sizeof(g_pucReportDescriptor)),
                                // Length of the Descriptor.
};

//*****************************************************************************
//
// The interrupt IN endpoint descriptor for the HID keyboard.
//
//*****************************************************************************
const uint8_t g_pui8HIDInEndpoint[] =
{
    //
    // Interrupt IN endpoint descriptor
    //
    7,                                  // The size of the endpoint descriptor.
    USB_DTYPE_ENDPOINT,                 // Descriptor type is an endpoint.
    USB_EP_DESC_IN | USB_EP_TO_INDEX(INT_IN_ENDPOINT),
    USB_EP_ATTR_INT,                    // Endpoint is an interrupt endpoint.
```

```
        USBShort(INT_IN_EP_MAX_SIZE),    // The maximum packet size.
        16,                              // The polling interval for this endpoint.
    };

    //*****************************************************************************
    //
    // The HID keyboard config descriptor is defined using three sections:
    //
    // 1. The 9 byte configuration descriptor.
    // 2. The interface and HID report descriptors.
    // 4. The mandatory interrupt IN endpoint descriptor (FLASH).
    //
    //*****************************************************************************
    const tConfigSection g_sKeyboardConfigSection =
    {
        sizeof(g_pKeyboardDescriptor),
        g_pKeyboardDescriptor
    };

    const tConfigSection g_sHIDInterfaceSection =
    {
        sizeof(g_pHIDInterface),
        g_pHIDInterface
    };

    const tConfigSection g_sHIDInEndpointSection =
    {
        sizeof(g_pHIDInEndpoint),
        g_pHIDInEndpoint
    };


    //*****************************************************************************
    //
    // This array lists all the sections that must be concatenated to make a
    // single, complete HID keyboard configuration descriptor.
    //
    //*****************************************************************************
    const tConfigSection *g_psKeyboardSections[] =
    {
        &g_sKeyboardConfigSection,
        &g_sHIDInterfaceSection,
        &g_sHIDInEndpointSection
    };

    #define NUM_KEYBOARD_SECTIONS (sizeof(g_psKeyboardSections) /              \
                                   sizeof(tConfigSection *))

    //*****************************************************************************
    //
    // The header for the single configuration we support. This is the root of
    // the data structure that defines all the bits and pieces that are pulled
    // together to generate the HID keyboard's config descriptor.  A pointer to
    // this structure is used to initialize the ppui8ConfigDescriptors field of
    // the tDeviceInfo structure passed to USBDCDInit().
    //
    //*****************************************************************************
    const tConfigHeader g_sKeyboardConfigHeader =
    {
        NUM_KEYBOARD_SECTIONS,
        g_psKeyboardSections
    };
```

2.23.1.3 tDeviceInfo.ppui8StringDescriptors and tDeviceInfo.ui32NumStringDescriptors

Descriptive strings referenced by device and configuration descriptors are provided to the USB Device API as an array of string descriptors containing the basic descriptor length and type header followed by a Unicode string. The various string identifiers passed in other descriptors are indexes into the *pStringDescriptor* array. The first entry of the string descriptor array has a special format and indicates the languages supported by the device.

The field *ui32NumStringDescriptors* indicates the number of individual string descriptors in the *ppui8StringDescriptors* array.

The string descriptor array provided to the USB Device API for a USB HID keyboard follows.

```
//*****************************************************************************
//
// The languages supported by this device.
//
//*****************************************************************************
const uint8_t g_pui8LangDescriptor[] =
{
    4,
    USB_DTYPE_STRING,
    USBShort(USB_LANG_EN_US)
};

//*****************************************************************************
//
// The manufacturer string.
//
//*****************************************************************************
const uint8_t g_pui8ManufacturerString[] =
{
    (17 + 1) * 2,
    USB_DTYPE_STRING,
    'T', 0, 'e', 0, 'x', 0, 'a', 0, 's', 0, ' ', 0, 'I', 0, 'n', 0, 's', 0,
    't', 0, 'r', 0, 'u', 0, 'm', 0, 'e', 0, 'n', 0, 't', 0, 's', 0,
};

//*****************************************************************************
//
// The product string.
//
//*****************************************************************************
const uint8_t g_pui8ProductString[] =
{
    (16 + 1) * 2,
    USB_DTYPE_STRING,
    'K', 0, 'e', 0, 'y', 0, 'b', 0, 'o', 0, 'a', 0, 'r', 0, 'd', 0, ' ', 0,
    'E', 0, 'x', 0, 'a', 0, 'm', 0, 'p', 0, 'l', 0, 'e', 0
};

//*****************************************************************************
//
// The serial number string.
//
//*****************************************************************************
const uint8_t g_pui8SerialNumberString[] =
{
    (8 + 1) * 2,
    USB_DTYPE_STRING,
    '1', 0, '2', 0, '3', 0, '4', 0, '5', 0, '6', 0, '7', 0, '8', 0
};

//*****************************************************************************
```

```
//
// The array of string descriptors needed by the enumeration code.
//
//*****************************************************************************
const uint8_t * const g_ppui8StringDescriptors[] =
{
    g_pui8LangDescriptor,
    g_pui8ManufacturerString,
    g_pui8ProductString,
    g_pui8SerialNumberString
};
```

In this example, the *ppui8StringDescriptors* member of the *tDeviceInfo* structure would be initialized with the value *g_ppui8StringDescriptors* and the *ui32NumStringDescriptors* member would be set to the number of elements in the *g_ppui8StringDescriptors* array.

2.23.2 USB Event Handlers

The majority of the work in a USB device application is carried out either in the context of, or in response to callbacks from the USB Device API. These callback functions are made available to the USB Device API in the *sCallbacks* field of the *tDeviceInfo* structure passed in a call to USBDC-DInit().

Field *sCallbacks* is a structure of type *tCustomHandlers* which contains a function pointer for each USB event. The application must populate the table with valid function pointers for each event that it wishes to be informed of. Setting any function pointer to NULL disables notification for that event.

The *tCustomHandlers* structure contains the following fields:

- pfnGetDescriptor
- pfnRequestHandler
- pfnInterfaceChange
- pfnConfigChange
- pfnDataReceived
- pfnDataSent
- pfnResetHandler
- pfnSuspendHandler
- pfnResumeHandler
- pfnDisconnectHandler
- pfnEndpointHandler
- pfnDeviceHandler

Note that all callbacks except the *pfnDeviceHandler* entry are made in interrupt context. It is, therefore, vital that handlers do not block or make calls to functions which cannot safely be made in an interrupt handler.

2.23.2.1 pfnGetDescriptor

Standard USB device, configuration and string descriptors are handled by the USB Device API internally but some device classes also define additional, class specific descriptors. In cases where

the host requests one of these non-standard descriptors, this callback is made to give the device code an opportunity to provide its own descriptor to satisfy the request.

If the device can satisfy the request, it must call USBDCDSendDataEP0() to provide the requested descriptor data to the host. If the request cannot be satisfied, the device should call USBDCD-StallEP0() to indicate that the descriptor request is not supported.

If this member of *sCallbacks* is set to NULL, the USB Device API stalls endpoint zero whenever it receives a request for a non-standard descriptor.

2.23.2.2 pfnRequestHandler

The USB Device API contains handlers for all standard USB requests (as defined in Table 9-3 of the USB 2.0 specification) where a standard request is indicated by bits 5 and 6 of the request structure *bmRequestType* field being clear. If a request is received with a non-standard request type, this callback is made to give the device code an opportunity to satisfy the request.

The callback function receives a pointer to a standard, 8 byte request structure of type *tUSBRequest* containing information on the request type, the request identifier and various request-specific parameters. The structure also contains a length field, *wLength*, which indicates how much (if any) data follows in the data stage of the USB transaction. Note that this data is not available at the time the callback is made and the device code is responsible for requesting it using a call to USBDCDRequestDataEP0() if required.

The sequence required when additional data is attached to the request is as follows:

- Parse the request to determine the request type and verify that it is handled by the device. If not, call USBDCDStallEP0() to indicate the problem.
- If the request is to be handled and *wLength* is non-zero, indicating that additional data is required, call USBDCDRequestDataEP0() passing a pointer to the buffer into which the data is to be written and the number of bytes of data to receive.
- Call USBDevEndpointDataAck() to acknowledge reception of the initial request transmission. This function is found in the Driver Library USB driver API.

Note that it is important to call USBDCDRequestDataEP0() prior to acknowledging the initial request since the acknowledgment frees the host to send the additional data. By making the calls in this order, the USB Device API is guaranteed to be in the correct state to receive the data when it arrives. Making the calls in the opposite order, creates a race condition which could result in loss of data.

Data received as a result of a call to USBDCDRequestDataEP0() is delivered asynchronously via the *pfnDataReceived* callback described below.

If this member of *sCallbacks* is set to NULL, the USB Device API stalls endpoint zero whenever it receives a non-standard request.

2.23.2.3 pfnInterfaceChange

Based on the configuration descriptor published by the device code, several different alternate interface settings may be supported. In cases where the host wishes to change from the default interface configuration and the USB library determines that the requested alternate setting is supported, this callback is made to inform the device code of the change. The parameters passed provide the new alternate interface (*ucAlternateSetting* and the interface number (*ucInterfaceNum*).

This callback is only made once the USB Device API has validated the requested alternate setting. If the requested setting is not available in the published configuration descriptor, the USB Device API stalls endpoint zero to indicate the error to the host and make no callback to the device code.

If this member of *sCallbacks* is set to NULL, the USB Device API notes the interface change internally but not report it to the device code.

2.23.2.4 pfnConfigChange

When the host enumerates a device, it ultimately selects the configuration that is to be used and send a SET_CONFIGURATION request to the device. When this occurs, the USB Device API validates the configuration number passed against the device code's published configuration descriptors then calls the `pfnConfigChange` callback to inform the device code of the configuration that is to be used.

If this member of *sCallbacks* is set to NULL, the USB Device API notes the configuration change internally but not report it to the device code.

2.23.2.5 pfnDataReceived

This callback informs the device code of the arrival of data following an earlier call to USBDCDRequestDataEP0(). On this callback, the received data has been written into the buffer provided to the USB Device API in the *pucData* parameter to USBDCDRequestDataEP0().

The callback handler does not need to acknowledge the data using a call to USBDevEndpointDataAck() in this case since this acknowledgment is performed within the USB Device API itself.

If this member of *sCallbacks* is set to NULL, the USB Device API reads endpoint zero data requested via USBDCDRequestDataEP0() but not report its availability to the device code. Devices making use of the USBDCDRequestDataEP0() call must, therefore, ensure that they supply a *pfnDataReceived* handler.

2.23.2.6 pfnDataSent

The USBDCDSendDataEP0() function allows device code to send an arbitrarily-sized block of data to the host via endpoint zero. The maximum packet size that can be sent via endpoint zero is, however, 64 bytes so larger blocks of data are sent in multiple packets. This callback function is used by the USB Device API to inform the device code when all data provided in the buffer passed to USBDCDSendDataEP0() has been consumed and scheduled for transmission to the host. On reception of this callback, the device code is free to reuse the outgoing data buffer if required.

If this member of *sCallbacks* is set to NULL, the USB Device API does not inform the device code when a block of EP0 data is sent.

2.23.2.7 pfnResetHandler

The *pfnResetHandler* callback is made by the USB Device API whenever a bus reset is detected. This typically occurs during enumeration. The device code may use this notification to perform any housekeeping required in preparation for a new configuration being set.

If this member of *sCallbacks* is set to NULL, the USB Device API does not inform the device code when a bus reset occurs.

2.23.2.8 pfnSuspendHandler

The *pfnSuspendHandler* callback is made whenever the USB Device API detects that suspend has been signaled on the bus. Device code may make use of this notification to, for example, set appropriate power saving modes.

If this member of *sCallbacks* is set to NULL, the USB Device API does not inform the device code when a bus suspend occurs.

2.23.2.9 pfnResumeHandler

The *pfnResumeHandler* callback is made whenever the USB Device API detects that resume has been signaled on the bus. Device code may make use of this notification to undo any changes made in response to an earlier call to the *pfnSuspendHandler* callback.

If this member of *sCallbacks* is set to NULL, the USB Device API does not inform the device code when a bus resume occurs.

2.23.2.10 pfnDisconnectHandler

The *pfnDisconnectHandler* callback is made whenever the USB Device API detects that the device has been disconnected from the bus.

If this member of *sCallbacks* is set to NULL, the USB Device API does not inform the device code when a disconnection event occurs.

Note: The **USB_EVENT_DISCONNECTED** event is not reported to the application if the MCU's PB1/USB0VBUS pin is connected to a fixed +5 Volts rather than directly to the VBUS pin on the USB connector or if the USB controller is configured to force device mode.

2.23.2.11 pfnEndpointHandler

While the use of endpoint zero is standardized and supported via several of the other callbacks already listed (*pfnDataSent*, *pfnDataReceived*, *pfnGetDescriptor*, *pfnRequestHandler*, *pfnInterfaceChange* and *pfnConfigChange*), the use of other endpoints is entirely dependent upon the device class being implemented. The *pfnEndpointHandler* callback is, therefore, made to notify the device code of all activity on any endpoint other than endpoint zero and it is the device code's responsibility to determine the correct action to take in response to each callback.

The *ulStatus* parameter passed to the handler provides information on the actual endpoint for which the callback is being made and allows the handler to determine if the event is due to transmission (if an IN endpoint event occurs) or reception (if an OUT endpoint event occurs) of data.

Having determined the endpoint sourcing the event, the device code can determine the actual event by calling USBEndpointStatus() for the appropriate endpoint then clear the status by calling USBDevEndpointStatusClear().

When incoming data is indicated by the flag **USB_DEV_RX_PKT_RDY** being set in the endpoint status, data can be received using a call to USBEndpointDataGet() followed by a call to USBDevEndpointDataAck() to acknowledge the reception to the host.

When an event relating to an IN endpoint (data transmitted from the device to the host) is received, the status read from USBEndpointStatus() indicates any errors in transmission. If the value read is 0, this implies that the data was successfully transmitted and acknowledged by the host.

Any device whose configuration descriptor indicates that it uses any endpoint (endpoint zero use is assumed) must populate the *pfnEndpointHandler* member of *tCustomHandlers*.

2.23.2.12 pfnDeviceHandler

Unlike the other calling functions *pfnDeviceHandler* specifies a generic input handler to the device class. Callers of this function should check to insure that the class supports this entry by seeing if the *pfnDeviceHandler* is non-zero This call is provided to allow requests based on a given instance to be passed into a device. This is commonly used by a top level composite device that is using multiple instances of the same class.

USB device classes that need to support being part of a composite device must implement this function as the composite device class needs to call this function to inform the class of interface, endpoint, and string index changes. See the documentation on the **USB_EVENT_COMP_IFACE_CHANGE**, **USB_EVENT_COMP_EP_CHANGE**, and **USB_EVENT_COMP_STR_CHANGE**.

2.23.3 Interrupt Vector Selection

An application using the USB Device API should normally ensure that the interrupt vector for the hardware USB controller is set to call function *USB0DeviceIntHandler*.

If the target application is intended to allow switching between USB device and USB host mode, however, this handler should be replaced with *USB0DualModeIntHandler* to allow the USB library to perform appropriate interrupt steering depending upon the current mode of operation. Hybrid applications must also call USBStackModeSet() to indicate the mode they wish to operate in.

2.23.4 Passing Control to the USB Device API

When all previous setup steps have been completed, control can be passed to the USB Device API. The library enables the appropriate interrupts and connect the device to the bus in preparation for enumeration by the USB host. This operation is initiated using a call to USBDCDInit() passing the completed *tDeviceInfo* structure which describes the device.

Following this call, your device code callback functions is called when USB events specific to your device are detected by the library.

```
//
// Pass the USB Device API our device information and connect the device to
// the bus.
//
USBDCDInit(0, &g_sMouseDeviceInfo);
```

2.24 USB Device API Definitions

Data Structures

- tCompositeEntry
- tConfigHeader
- tConfigSection
- tCustomHandlers
- tDeviceInfo

Defines

- USB_MAX_INTERFACES_PER_DEVICE

Typedefs

- tUSBDeviceHandler

Functions

- void USB0DeviceIntHandler (void)
- void USBDCDDeviceInfoInit (uint32_t ui32Index, tDeviceInfo ∗psDeviceInfo)
- bool USBDCDFeatureGet (uint32_t ui32Index, uint32_t ui32Feature, void ∗pvFeature)
- bool USBDCDFeatureSet (uint32_t ui32Index, uint32_t ui32Feature, void ∗pvFeature)
- void USBDCDInit (uint32_t ui32Index, tDeviceInfo ∗psDevice, void ∗pvDCDCBData)
- void USBDCDPowerStatusSet (uint32_t ui32Index, uint8_t ui8Power)
- bool USBDCDRemoteWakeLPM (uint32_t ui32Index)
- bool USBDCDRemoteWakeupRequest (uint32_t ui32Index)
- void USBDCDRequestDataEP0 (uint32_t ui32Index, uint8_t ∗pui8Data, uint32_t ui32Size)
- void USBDCDSendDataEP0 (uint32_t ui32Index, uint8_t ∗pui8Data, uint32_t ui32Size)
- void USBDCDSetDefaultConfiguration (uint32_t ui32Index, uint32_t ui32DefaultConfig)
- void USBDCDStallEP0 (uint32_t ui32Index)
- void USBDCDTerm (uint32_t ui32Index)
- bool USBDeviceConfig (tDCDInstance ∗psDevInst, const tConfigHeader ∗psConfig)
- bool USBDeviceConfigAlternate (tDCDInstance ∗psDevInst, const tConfigHeader ∗psConfig, uint8_t ui8InterfaceNum, uint8_t ui8AlternateSetting)

2.24.1 Data Structure Documentation

2.24.1.1 tCompositeEntry

Definition:
```
typedef struct
{
```

```
                const tDeviceInfo *psDevInfo;
                void *pvInstance;
                uint32_t ui32DeviceWorkspace;
            }
            tCompositeEntry
```

Members:

psDevInfo This is the top level device information structure.

pvInstance This is the instance data for the device structure.

ui32DeviceWorkspace A per-device workspace used by the composite device.

Description:

This type is used by an application to describe and instance of a device and an instance data pointer for that class. The psDevice pointer should be a pointer to a valid device class to include in the composite device. The pvInstance pointer should be a pointer to an instance pointer for the device in the psDevice pointer.

2.24.1.2 tConfigHeader

Definition:

```
            typedef struct
            {
                uint8_t ui8NumSections;
                const tConfigSection *const *psSections;
            }
            tConfigHeader
```

Members:

ui8NumSections The number of sections comprising the full descriptor for this configuration.

psSections A pointer to an array of ui8NumSections section pointers which must be concatenated to form the configuration descriptor.

Description:

This is the top level structure defining a USB device configuration descriptor. A configuration descriptor contains a collection of device- specific descriptors in addition to the basic config, interface and endpoint descriptors. To allow flexibility in constructing the configuration, the descriptor is described in terms of a list of data blocks. The first block must contain the configuration descriptor itself and the following blocks are appended to this in order to produce the full descriptor sent to the host in response to a GetDescriptor request for the configuration descriptor.

2.24.1.3 tConfigSection

Definition:

```
            typedef struct
            {
                uint16_t ui16Size;
                const uint8_t *pui8Data;
            }
            tConfigSection
```

Members:

ui16Size The number of bytes of descriptor data pointed to by pui8Data.

pui8Data A pointer to a block of data containing an integral number of USB descriptors which form part of a larger configuration descriptor.

Description:

This structure defines a contiguous block of data which contains a group of descriptors that form part of a configuration descriptor for a device. It is assumed that a config section contains only whole descriptors. It is not valid to split a single descriptor across multiple sections.

2.24.1.4 tCustomHandlers

Definition:
```
typedef struct
{
    tStdRequest pfnGetDescriptor;
    tStdRequest pfnRequestHandler;
    tInterfaceCallback pfnInterfaceChange;
    tInfoCallback pfnConfigChange;
    tInfoCallback pfnDataReceived;
    tInfoCallback pfnDataSent;
    tUSBIntHandler pfnResetHandler;
    tUSBIntHandler pfnSuspendHandler;
    tUSBIntHandler pfnResumeHandler;
    tUSBIntHandler pfnDisconnectHandler;
    tUSBEPIntHandler pfnEndpointHandler;
    tUSBDeviceHandler pfnDeviceHandler;
}
tCustomHandlers
```

Members:

pfnGetDescriptor This callback is made whenever the USB host requests a non-standard descriptor from the device.

pfnRequestHandler This callback is made whenever the USB host makes a non-standard request.

pfnInterfaceChange This callback is made in response to a SetInterface request from the host.

pfnConfigChange This callback is made in response to a SetConfiguration request from the host.

pfnDataReceived This callback is made when data has been received following to a call to USBDCDRequestDataEP0.

pfnDataSent This callback is made when data has been transmitted following a call to USB-DCDSendDataEP0.

pfnResetHandler This callback is made when a USB reset is detected.

pfnSuspendHandler This callback is made when the bus has been inactive long enough to trigger a suspend condition.

pfnResumeHandler This is called when resume signaling is detected.

pfnDisconnectHandler This callback is made when the device is disconnected from the USB bus.

pfnEndpointHandler This callback is made to inform the device of activity on all endpoints other than endpoint zero.

pfnDeviceHandler This generic handler is provided to allow requests based on a given instance to be passed into a device. This is commonly used by a top level composite device that is using multiple instances of a class.

Description:

USB event handler functions used during enumeration and operation of the device stack.

2.24.1.5 tDeviceInfo

Definition:

```
typedef struct
{
    const tCustomHandlers *psCallbacks;
    const uint8_t *pui8DeviceDescriptor;
    const tConfigHeader *const *ppsConfigDescriptors;
    const uint8_t *const *ppui8StringDescriptors;
    uint32_t ui32NumStringDescriptors;
}
tDeviceInfo
```

Members:

psCallbacks A pointer to a structure containing pointers to event handler functions provided by the client to support the operation of this device.

pui8DeviceDescriptor A pointer to the device descriptor for this device.

ppsConfigDescriptors A pointer to an array of configuration descriptor pointers. Each entry in the array corresponds to one configuration that the device may be set to use by the USB host. The number of entries in the array must match the bNumConfigurations value in the device descriptor array, *pui8DeviceDescriptor* .

ppui8StringDescriptors A pointer to the string descriptor array for this device. This array must be arranged as follows:

[0] - Standard descriptor containing supported language codes.

[1] - String 1 for the first language listed in descriptor 0.

[2] - String 2 for the first language listed in descriptor 0.

...

[n] - String n for the first language listed in descriptor 0.

[n+1] - String 1 for the second language listed in descriptor 0.

...

[2n] - String n for the second language listed in descriptor 0.

[2n+1]- String 1 for the third language listed in descriptor 0.

...

[3n] - String n for the third language listed in descriptor 0.

and so on.

ui32NumStringDescriptors The total number of descriptors provided in the ppStringDescriptors array.

Description:

This structure is passed to the USB library on a call to USBDCDInit and provides the library with information about the device that the application is implementing. It contains functions pointers for the various USB event handlers and pointers to each of the standard device descriptors.

2.24.2 Define Documentation

2.24.2.1 USB_MAX_INTERFACES_PER_DEVICE

Definition:
```
#define USB_MAX_INTERFACES_PER_DEVICE
```

Description:
The maximum number of independent interfaces that any single device implementation can support. Independent interfaces means interface descriptors with different *bInterfaceNumber* values - several interface descriptors offering different alternative settings but the same interface number count as a single interface.

2.24.3 Typedef Documentation

2.24.3.1 tUSBDeviceHandler

Definition:
```
typedef void(* tUSBDeviceHandler)(void *pvInstance,
                                  uint32_t ui32Request,
                                  void *pvRequestData)
```

Description:
Generic handler callbacks that are used when the callers needs to call into an instance of class.

2.24.4 Function Documentation

2.24.4.1 USB0DeviceIntHandler

The USB device interrupt handler.

Prototype:
```
void
USB0DeviceIntHandler(void)
```

Description:
This the main USB interrupt handler entry point for use in USB device applications. This top-level handler will branch the interrupt off to the appropriate application or stack handlers depending on the current status of the USB controller.

Applications which operate purely as USB devices (rather than dual mode applications which can operate in either device or host mode at different times) must ensure that a pointer to this function is installed in the interrupt vector table entry for the USB0 interrupt. For dual mode operation, the vector should be set to point to *USB0DualModeIntHandler()* instead.

Returns:
None.

2.24.4.2 USBDCDDeviceInfoInit

Initialize an instance of the tDeviceInfo structure.

Prototype:
```
void
USBDCDDeviceInfoInit(uint32_t ui32Index,
                     tDeviceInfo *psDeviceInfo)
```

Parameters:
>   *ui32Index* is the index of the USB controller which is to be initialized.
>   *psDeviceInfo* is a pointer to the tDeviceInfo structure that needs to be initialized.

Description:
>   This function must be called by a USB device class instance to initialize the basic tDeviceInfo
>   required for all USB device class modules. This is typically called in the initialization routine for
>   USB device class. For example in usbdaudio.c that supports USB device audio classes, this
>   function is called in the USBDAudioCompositeInit() function which is used for both composite
>   and non-composites instances of the USB audio class.

Note:
>   This function should not be called directly by applications.

Returns:
>   None.

2.24.4.3 USBDCDFeatureGet

This function is used to get the features of the USB library.

Prototype:
```
bool
USBDCDFeatureGet(uint32_t ui32Index,
                 uint32_t ui32Feature,
                 void *pvFeature)
```

Parameters:
>   *ui32Index* is the index of the USB controller whose device power status is being reported.
>   *ui32Feature* indicates which feature is being requested.
>   *pvFeature* holds the data that will be reported for the feature request.

Description:
>   Applications can query the support levels of some USB library features by calling this function
>   to get values of the features. This function should normally be called before class initialization
>   functions since the settings need to be in place during enumeration. This allows the USB
>   library to properly respond to all enumeration requests. The *ui32Feature* value is one of the
>   **USBLIB_FEATURE_**∗ defines which controls the type of request being made. The *pvFeature*
>   is a feature specific data structure that is determined by the value passed in the *ui32Feature*
>   parameter.

Returns:
>   Returns **true** if the feature was requested for ULPI and returns **false** if the feature was not for
>   ULPI interface.

2.24.4.4 USBDCDFeatureSet

This function is used to enable/disable features of the USB library.

Prototype:
```
bool
USBDCDFeatureSet(uint32_t ui32Index,
                 uint32_t ui32Feature,
                 void *pvFeature)
```

Parameters:
> ***ui32Index*** is the index of the USB controller whose device power status is being reported.
> ***ui32Feature*** indicates which feature is being changed.
> ***pvFeature*** holds the data that controls the feature request.

Description:
> Applications can change the support levels of some USB library features by calling this function to enable/disable certain features. This function should normally be called before class initialization functions since the settings need to be in place before enumeration starts. This allows the USB library to properly respond to all enumeration requests. The *ui32Feature* value is one of the **USBLIB_FEATURE_** * defines which controls the type of request being made. The *pvFeature* is a feature specific data structure that is determined by the value passed in the *ui32Feature* parameter.

Returns:
> Returns **true** if the feature was successfully changed and returns **false** if the feature was not able to be changed or is not supported.

2.24.4.5 USBDCDInit

Initialize the USB library device control driver for a given hardware controller.

Prototype:
```
void
USBDCDInit(uint32_t ui32Index,
           tDeviceInfo *psDevice,
           void *pvDCDCBData)
```

Parameters:
> ***ui32Index*** is the index of the USB controller which is to be initialized.
> ***psDevice*** is a pointer to a structure containing information that the USB library requires to support operation of this application's device. The structure contains event handler callbacks and pointers to the various standard descriptors that the device wishes to publish to the host.
> ***pvDCDCBData*** is the callback data for any device callbacks.

Description:
> This function must be called by a device class which wishes to operate as a USB device and is not typically called by an application. This function initializes the USB device control driver for the given controller and saves the device information for future use. Prior to returning from this function, the device is connected to the USB bus. Following return, the caller can expect

to receive a callback to the supplied `pfnResetHandler` function when a host connects to the device. The *pvDCDCBData* contains a pointer to data that is returned with the DCD calls back to the function in the psDevice->psCallbacks() functions.

The device information structure passed in *psDevice* must remain unchanged between this call and any matching call to USBDCDTerm() because it is not copied by the USB library.

The USBStackModeSet() function can be called with eUSBModeForceDevice in order to cause the USB library to force the USB operating mode to a device controller. This allows the application to used the USBVBUS and USBID pins as GPIOs on devices that support forcing OTG to operate as a device only controller. By default the USB library will assume that the USBVBUS and USBID pins are configured as USB pins and not GPIOs.

Returns:
> None.

2.24.4.6 USBDCDPowerStatusSet

Reports the device power status (bus- or self-powered) to the library.

Prototype:
```
void
USBDCDPowerStatusSet(uint32_t ui32Index,
                     uint8_t ui8Power)
```

Parameters:
> *ui32Index* is the index of the USB controller whose device power status is being reported.
> *ui8Power* indicates the current power status, either **USB_STATUS_SELF_PWR** or **USB_STATUS_BUS_PWR**.

Description:
> Applications which support switching between bus- or self-powered operation should call this function whenever the power source changes to indicate the current power status to the USB library. This information is required by the library to allow correct responses to be provided when the host requests status from the device.

Returns:
> None.

2.24.4.7 USBDCDRemoteWakeLPM

Requests an LPM remote wake up to resume communication when in an LPM sleep state.

Prototype:
```
bool
USBDCDRemoteWakeLPM(uint32_t ui32Index)
```

Parameters:
> *ui32Index* is the index of the USB controller that will request a bus wake up.

Description:
When the host controller puts the device into an LPM sleep state, the device can call this function to initiate LPM remote wake up signaling to the host. If the remote wake up feature has been enabled by the host, this will cause the host to respond to the LPM remote wake request and resume normal operation. If the host has disabled remote wake up, **false** is returned to indicate that the wake up request was not successful.

Returns:
Returns **true** if the remote wake up request has been sent or **false** if LPM remote wake up is disabled.

2.24.4.8 USBDCDRemoteWakeupRequest

Requests a remote wake up to resume communication when in suspended state.

Prototype:
```
bool
USBDCDRemoteWakeupRequest(uint32_t ui32Index)
```

Parameters:
ui32Index is the index of the USB controller that will request a bus wake up.

Description:
When the bus is suspended, an application which supports remote wake up (advertised to the host via the configuration descriptor) may call this function to initiate remote wake up signaling to the host. If the remote wake up feature has not been disabled by the host, this will cause the bus to resume operation within 20mS. If the host has disabled remote wake up, **false** will be returned to indicate that the wake up request was not successful.

Returns:
Returns **true** if the remote wake up is not disabled and the signaling was started or **false** if remote wake up is disabled or if signaling is currently ongoing following a previous call to this function.

2.24.4.9 USBDCDRequestDataEP0

This function starts the request for data from the host on endpoint zero.

Prototype:
```
void
USBDCDRequestDataEP0(uint32_t ui32Index,
                     uint8_t *pui8Data,
                     uint32_t ui32Size)
```

Parameters:
ui32Index is the index of the USB controller from which the data is being requested.
pui8Data is a pointer to the buffer to fill with data from the USB host.
ui32Size is the size of the buffer or data to return from the USB host.

Description:
This function handles retrieving data from the host when a custom command has been is-sued on endpoint zero. If the application needs notification when the data has been received, `psCallbacks->pfnDataReceived()` in the tDeviceInfo structure must contain valid func-tion pointer. In nearly all cases this is necessary because the caller of this function would likely need to know that the data requested was received.

Returns:
None.

2.24.4.10 USBDCDSendDataEP0

This function requests transfer of data to the host on endpoint zero.

Prototype:
```
void
USBDCDSendDataEP0(uint32_t ui32Index,
                  uint8_t *pui8Data,
                  uint32_t ui32Size)
```

Parameters:
ui32Index is the index of the USB controller which is to be used to send the data.
pui8Data is a pointer to the buffer to send via endpoint zero.
ui32Size is the amount of data to send in bytes.

Description:
This function handles sending data to the host when a custom command is issued or non-standard descriptor has been requested on endpoint zero. If the application needs notification when this is complete, `psCallbacks->pfnDataSent` in the tDeviceInfo structure must con-tain a valid function pointer. This callback could be used to free up the buffer passed into this function in the *pui8Data* parameter. The contents of the *pui8Data* buffer must remain unchanged until the `pfnDataSent` callback is received.

Returns:
None.

2.24.4.11 USBDCDSetDefaultConfiguration

This function sets the default configuration for the device.

Prototype:
```
void
USBDCDSetDefaultConfiguration(uint32_t ui32Index,
                              uint32_t ui32DefaultConfig)
```

Parameters:
ui32Index is the index of the USB controller whose default configuration is to be set.
ui32DefaultConfig is the configuration identifier (byte 6 of the standard configuration descrip-tor) which is to be presented to the host as the default configuration in cases where the configuration descriptor is queried prior to any specific configuration being set.

Description:
This function allows a device to override the default configuration descriptor that will be returned to a host whenever it is queried prior to a specific configuration having been set. The parameter passed must equal one of the configuration identifiers found in the `ppsConfigDescriptors` array for the device.

If this function is not called, the USB library will return the first configuration in the `ppsConfigDescriptors` array as the default configuration.

Note:
The USB device stack assumes that the configuration IDs (byte 6 of the configuration descriptor, `bConfigurationValue`) stored within the configuration descriptor array, `ppsConfigDescriptors`, are equal to the array index + 1. In other words, the first entry in the array must contain a descriptor with `bConfigurationValue` 1, the second must have `bConfigurationValue` 2 and so on.

Returns:
None.

2.24.4.12 USBDCDStallEP0

This function generates a stall condition on endpoint zero.

Prototype:
```
void
USBDCDStallEP0(uint32_t ui32Index)
```

Parameters:
ui32Index is the index of the USB controller whose endpoint zero is to be stalled.

Description:
This function is typically called to signal an error condition to the host when an unsupported request is received by the device. It should be called from within the callback itself (in interrupt context) and not deferred until later since it affects the operation of the endpoint zero state machine in the USB library.

Returns:
None.

2.24.4.13 USBDCDTerm

Free the USB library device control driver for a given hardware controller.

Prototype:
```
void
USBDCDTerm(uint32_t ui32Index)
```

Parameters:
ui32Index is the index of the USB controller which is to be freed.

Description:
This function should be called by an application if it no longer requires the use of a given USB controller to support its operation as a USB device. It frees the controller for use by another client.

It is the caller's responsibility to remove its device from the USB bus prior to calling this function.

Returns:
None.

2.24.4.14 USBDeviceConfig

Configure the USB controller appropriately for the device whose configuration descriptor is passed.

Prototype:
```
bool
USBDeviceConfig(tDCDInstance *psDevInst,
                const tConfigHeader *psConfig)
```

Parameters:
psDevInst is a pointer to the device instance being configured.
psConfig is a pointer to the configuration descriptor that the USB controller is to be set up to support.

Description:
This function may be used to initialize a USB controller to operate as the device whose configuration descriptor is passed. The function enables the USB controller, partitions the FIFO appropriately and configures each endpoint required by the configuration. If the supplied configuration supports multiple alternate settings for any interface, the USB FIFO is set up assuming the worst case use (largest packet size for a given endpoint in any alternate setting using that endpoint) to allow for on-the-fly alternate setting changes later. On return from this function, the USB controller is configured for correct operation of the default configuration of the device described by the descriptor passed.

Returns:
Returns **true** on success or **false** on failure.

2.24.4.15 USBDeviceConfigAlternate

Configure the affected USB endpoints appropriately for one alternate interface setting.

Prototype:
```
bool
USBDeviceConfigAlternate(tDCDInstance *psDevInst,
                         const tConfigHeader *psConfig,
                         uint8_t ui8InterfaceNum,
                         uint8_t ui8AlternateSetting)
```

Parameters:
psDevInst is a pointer to the device instance being configured.

psConfig is a pointer to the configuration descriptor that contains the interface whose alternate settings is to be configured.

ui8InterfaceNum is the number of the interface whose alternate setting is to be configured. This number corresponds to the bInterfaceNumber field in the desired interface descriptor.

ui8AlternateSetting is the alternate setting number for the desired interface. This number corresponds to the bAlternateSetting field in the desired interface descriptor.

Description:

This function may be used to reconfigure the endpoints of an interface for operation in one of the interface's alternate settings. Note that this function assumes that the endpoint FIFO settings will not need to change and only the endpoint mode is changed. This assumption is valid if the USB controller was initialized using a previous call to USBDCDConfig().

In reconfiguring the interface endpoints, any additional configuration bits set in the endpoint configuration other than the direction (**USB_EP_DEV_IN** or **USB_EP_DEV_OUT**) and mode (**USB_EP_MODE_MASK**) are preserved.

Returns:

Returns **true** on success or **false** on failure.

3   Host Functions

3.1   Introduction

This chapter covers the support provided by the USB library for the USB controller in host mode. In order to simplify the application and the addition of new devices and device classes, the USB library provides a layered interface to the USB host controller. At the top layer of the USB library there are application interfaces that provide easy access to the various types of peripherals that are supported by the USB library. Below this layer are the USB host controller's device interfaces that handle the specifics of each type of device and how to communicate with the USB host class driver. The USB host class drivers handle the basics of dealing with whole classes of devices like HID and Mass Storage Class devices. The USB host class driver layer communicates with the lowest level of the USB library which is the USB host controller driver. This lowest level directly accesses DriverLib functions to provide communications with the USB device that is connected. This communication is provided by callbacks or direct APIs that is discussed in the rest of this chapter. Much like the USB library's device programming interface, the host interface has the following layers:

- Device APIs (Mouse, Keyboard, Filesystem)
- USB Class Driver APIs (HID, Mass Storage, Hub)
- USB Host Controller APIs
- DriverLib USB Driver APIs
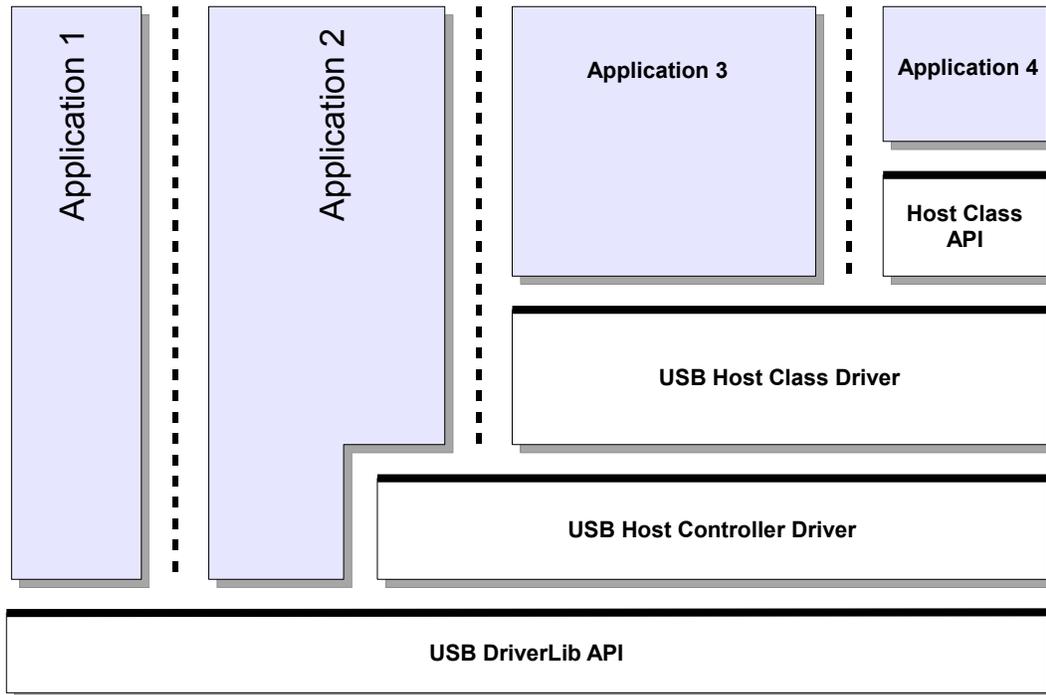
Source Code Overview

Source code and headers for the host specific USB functions can be found in the `host` directory of the USB library tree, typically `DriverLib/usblib/host`.

usbhost.h	The header file containing host mode function prototypes and data types offered by the USB library.
usbhostenum.c	The source code for the USB host enumeration functions offered by the library.
usbhaudio.c	The source code for the USB host Audio class driver.
usbhaudio.h	The header file containing Audio class definitions specific to hosts supporting this class of device.
usbhhid.c	The source code for the USB host HID class driver.
usbhhid.h	The header file containing the definitions needed to interact with the USB host HID class driver.
usbhhub.c	The source code for the USB host Hub class driver.
usbhhub.h	The header file containing the definitions needed to interact with the USB host Hub class driver.

`usbhhidkeyboard.c`

> The source code for the USB host HID class keyboard device.

`usbhhidkeyboard.h`

> The header file containing the definitions needed to interact with the USB host HID class keyboard device.

`usbhhidmouse.c`

> The source code for the USB host HID class mouse device.

`usbhhidmouse.h`

> The header file containing the definitions needed to interact with the USB host HID class mouse device.

`usbhmsc.c`   The source code for the USB host Mass Storage class driver.

`usbhmsc.h`   The header file containing Mass Storage class definitions specific to hosts supporting this class of device.

`usbhscsi.c`   The source code for a high level SCSI interface which calls the host Mass Storage class driver.

3.2   Host Controller Driver

The USB library host controller driver provides an interface to the host controller's hardware register interface. This is the lowest level of the driver interface and it interacts directly with the DriverLib USB APIs. The host controller driver provides all of the functionality necessary to provide enumeration of devices regardless of the type of device that is connected. This portion of the enumeration code only enumerates the device and allows the higher level drivers to actually handle normal device operations. To allow the application to conserve code and data memory, the host controller driver provides a method to allow applications to only include the host class drivers that are needed for each type of USB device. This allows an application to handle multiple classes of devices but only include the USB library code that the application needs to communicate with the devices that the application supports. While the host controller driver handles the enumeration of devices it relies on USB pipes, that are allocated by the higher level class drivers, as the direct communications method with a devices end points.

3.2.1   Enumeration

The USB host controller driver handles all of the details necessary to discover and enumerate any USB device. The USB host controller driver only performs enumeration and relies on the host class drivers to perform any other communications with USB devices including the allocation of the endpoints for the device. Most of the code used to enumerate devices is run in interrupt context and is contained in the enumeration handler. In order to complete the enumeration process, the host controller driver also requires that the application periodically call the USBHCDMain() function. When a host class driver or an application needs access to endpoint 0 of a device, it uses the USBHCD-ControlTransfer() interface to send data to the device or receive data from the device. During the

enumeration process the host controller driver searches a list of host class drivers provided by the application in the USBHCDRegisterDrivers() call. The details of this structure are covered in the host class drivers section of this document. If the host controller driver finds a host class driver that matches the class of the enumerated device, it calls the open function for that host class driver. If no host class driver is found the host controller driver ignores the device and there is no notification to the application. The host controller driver or the host class driver can provide callbacks up through the USB library to inform the application of enumeration events. The host class drivers are responsible for configuring the USB pipes based on the type of device that is discovered. The application is notified of events in two ways: one is from the host class driver for the connected device and the other is from a non-device specific event driver. The class specific events come from the host class driver, while the generic connection, power and other non-device class specific events come from the host event driver if it is included in the application. The section "USB Events Driver" covers the host events driver, the valid events and how to include the host events driver in an application.

3.2.2 USB Host Configurable Features

Like the USB device mode the USB host mode provides the application with the ability to set certain run-time global features of the USB library. The application uses the USBHCDFeatureSet() to set the configurable features of the USB host library. These features include USB clocking, enabling an external phy, and setting various power settings for the USB host library.

3.2.2.1 USB Host PLL Feature

One of the most important features used by applications informs the USB library of the of how the application has configured the main PLL. On TM4C129 class devices the USB clock is derived from the main PLL and the USB library sets up its own clock based on the value provided by the application. The **USBLIB_FEATURE_USBPLL** is only used on TM4C129 class devices and defaults to 480 MHz. If the PLL is operating at a value other than 480 MHz then the application is required to call the USBHCDFeatureSet() function with the **USBLIB_FEATURE_USBPLL** feature to provide the PLL frequency to the USB library. The following is an example of setting the PLL frequency to a value other than the default 480 MHz.

Example: Using a 320 MHz PLL setting

```
uint32_t ui32PLLFrequency;

ui32PLLFrequency = 320000000;

//
// Inform the USB library that the system is running using a 320 MHz PLL.
//
USBHCDFeatureSet(0, USBLIB_FEATURE_USBPLL, &ui32PLLFrequency);
```

3.2.2.2 USB Host System Clock Feature

The USB library needs to know the actual operating frequency of the system clock in order to provide some rough delays that the USB library uses for some of the USB signaling. The application provides the system clock frequency using the **USBLIB_FEATURE_CPUCLK** feature setting. The default for this value is 120 MHz for TM4C129 class devices and 80 MHz for TM4C123 class devices. If the processor is operating at something other than one of these frequencies then the

application must call the USBHCDFeatureSet() function with the **USBLIB_FEATURE_CPUCLK** feature to provide the actual system clock frequency.

Example: Using 60 MHz system clock

```
uint32_t ui32SysClock;

ui32SysClock = 60000000;

//
// Inform the USB library that the system is running at 60 MHz.
//
USBHCDFeatureSet(0, USBLIB_FEATURE_CPUCLK, &ui32SysClock);
```

3.2.2.3 USB Host ULPI Feature

The TM4C129x class devices support using an external ULPI USB phy to allow the host to communicate with high speed devices. This feature is enabled by setting the **US-BLIB_FEATURE_USBULPI** option combined with the desired speed. From the applications perspective this has no affect to normal USB operation other than the necessity to properly enable the USB external phy pins. The following are the possible configuration options when using ULPI:

- **USBLIB_FEATURE_ULPI_NONE** - Disable ULPI and use the internal phy (default).
- **USBLIB_FEATURE_ULPI_HS** - Use an externally connected ULPI phy at high-speed.
- **USBLIB_FEATURE_ULPI_FS** - Use an externally connected ULPI phy at full-speed.

The following is an example of configuring the USB library to use and external phy operating in high speed mode.

Example: Enable high speed external ULPI phy

```
uint32_t ui32ULPI;

ui32ULPI = USBLIB_FEATURE_ULPI_HS;

//
// Enable the use of an external USB ULPI connected phy.
//
USBHCDFeatureSet(0, USBLIB_FEATURE_USBULPI, &ui32ULPI);
```

3.2.2.4 USB HOST LPM Feature

The TM4C129 class devices added the ability to use the USB LPM feature. This feature is not enabled by default and therefore must be enabled by the application if it wishes to use any LPM features. Once LPM is enabled, the application can use the USB library to issue an LPM suspend transaction to a device by calling the USBHCDLPMSleep() function and wake devices using the USBHCDLPMResume() function. There are only two options for LPM which allow the feature to be enabled or disabled with the following values:

- **USBLIB_FEATURE_LPM_DIS** - Disable LPM transactions (default).
- **USBLIB_FEATURE_LPM_EN** - Enable sending and receiving LPM transactions.

Example: Enable LPM transactions.

The following is an example of configuring the USB library enable LPM mode.

```
uint32_t ui32LPMFeature;

ui32LPMFeature = USBLIB_FEATURE_LPM_EN;

//
// Enable LPM transactions.
//
USBHCDFeatureSet(0, USBLIB_FEATURE_LPM, &ui32LPMFeature);
```

3.2.3 USB Pipes

The host controller driver layer uses interfaces called USB pipes as the primary method of communications with USB devices. These USB pipes can be dynamically allocated or statically allocated by the USB class drivers during enumeration. The USB pipes are usually only used within the USB library or by host class drivers and are not usually directly accessed by applications. The USB pipes are allocated and freed by calling the USBHCDPipeAlloc() and USBHCDPipeFree() functions and are initially configured by calling the USBHCDPipeConfig(). The USBHCDPipeAlloc() and USBHCDPipeConfig() functions are used during USB device enumeration to allocate USB pipes to specific endpoints of the USB device. On disconnect, the USBHCDPipeFree() function is called to free up the USB pipe for use by a new USB device. While in use, the USB pipes perform read and write operations. Access to the USB pipes occurs through USBHCDPipeWrite() and USBHCDPipeRead() and the callback function provided when the USB pipe was allocated. These are used to read or write to endpoints on\ USB devices on endpoints other than the control endpoint on endpoint 0. Since endpoint 0 is shared with all devices, the host controller interface does not use USB pipes for communications over endpoint 0 and instead uses the USBHCDControlTransfer() function.

3.2.4 Control Transactions

All USB control transactions are handled through the USBHCDControlTransfer() function. This function is primarily used inside the host controller driver itself during enumeration, however some devices may require using control transactions through endpoint 0. The HID class drivers are a good example of a USB class driver that uses control transactions to send data to a USB device. The USBHCDControlTransfer() function should not be called from within interrupt context as control transfers are a blocking operation that relies on interrupts to proceed. Since most callbacks occur in interrupt context, any calls to USBHCDControlTransfer() should be deferred until running outside the callback event. The USB host HID keyboard example is a good example of performing a control transaction outside of a callback function.

3.2.5 Interrupt Handling

All interrupt handling is done by the USB library host controller driver and most callbacks are done in interrupt context and like interrupt handlers should defer any real processing of events to occur outside the interrupt context. The callbacks are used to notify the upper layers of events that occur during enumeration or during normal operation. Because most of enumeration code is handled by interrupt handlers the enumeration code does require that the application call the USBHCDMain() function in order to progress through the enumeration states without running all code in interrupt context.

3.3   Host Controller Driver Definitions

Data Structures

- tUSBHostClassDriver

Defines

- DECLARE_EVENT_DRIVER(VarName, pfnOpen, pfnClose, pfnEvent)

Functions

- void USB0HostIntHandler (void)
- uint32_t USBHCDControlTransfer (uint32_t ui32Index, tUSBRequest *psSetupPacket, tUSB-HostDevice *psDevice, uint8_t *pui8Data, uint32_t ui32Size, uint32_t ui32MaxPacketSize)
- uint8_t USBHCDDevAddress (uint32_t ui32Instance)
- uint8_t USBHCDDevClass (uint32_t ui32Instance, uint32_t ui32Interface)
- uint8_t USBHCDDevHubPort (uint32_t ui32Instance)
- uint8_t USBHCDDevProtocol (uint32_t ui32Instance, uint32_t ui32Interface)
- uint8_t USBHCDDevSubClass (uint32_t ui32Instance, uint32_t ui32Interface)
- int32_t USBHCDEventDisable (uint32_t ui32Index, void *pvEventDriver, uint32_t ui32Event)
- int32_t USBHCDEventEnable (uint32_t ui32Index, void *pvEventDriver, uint32_t ui32Event)
- bool USBHCDFeatureSet (uint32_t ui32Index, uint32_t ui32Feature, void *pvFeature)
- void USBHCDInit (uint32_t ui32Index, void *pvPool, uint32_t ui32PoolSize)
- void USBHCDLPMResume (uint32_t ui32Index)
- uint32_t USBHCDLPMSleep (tUSBHostDevice *psDevice)
- uint32_t USBHCDLPMStatus (tUSBHostDevice *psDevice)
- void USBHCDMain (void)
- uint32_t USBHCDPipeAlloc (uint32_t ui32Index, uint32_t ui32EndpointType, tUSBHostDevice *psDevice, tHCDPipeCallback pfnCallback)
- uint32_t USBHCDPipeAllocSize (uint32_t ui32Index, uint32_t ui32EndpointType, tUSBHost-Device *psDevice, uint32_t ui32Size, tHCDPipeCallback pfnCallback)
- uint32_t USBHCDPipeConfig (uint32_t ui32Pipe, uint32_t ui32MaxPayload, uint32_t ui32Interval, uint32_t ui32TargetEndpoint)
- void USBHCDPipeDataAck (uint32_t ui32Pipe)
- void USBHCDPipeFree (uint32_t ui32Pipe)
- uint32_t USBHCDPipeRead (uint32_t ui32Pipe, uint8_t *pui8Data, uint32_t ui32Size)
- uint32_t USBHCDPipeReadNonBlocking (uint32_t ui32Pipe, uint8_t *pui8Data, uint32_t ui32Size)
- uint32_t USBHCDPipeSchedule (uint32_t ui32Pipe, uint8_t *pui8Data, uint32_t ui32Size)
- uint32_t USBHCDPipeTransferSizeGet (uint32_t ui32Pipe)
- uint32_t USBHCDPipeWrite (uint32_t ui32Pipe, uint8_t *pui8Data, uint32_t ui32Size)
- uint32_t USBHCDPowerAutomatic (uint32_t ui32Index)
- uint32_t USBHCDPowerConfigGet (uint32_t ui32Index)
- void USBHCDPowerConfigInit (uint32_t ui32Index, uint32_t ui32PwrConfig)

- uint32_t USBHCDPowerConfigSet (uint32_t ui32Index, uint32_t ui32Config)
- void USBHCDRegisterDrivers (uint32_t ui32Index, const tUSBHostClassDriver *const *ppsHClassDrvs, uint32_t ui32NumDrivers)
- void USBHCDReset (uint32_t ui32Index)
- void USBHCDResume (uint32_t ui32Index)
- void USBHCDSetAddress (uint32_t ui32DevIndex, uint32_t ui32DevAddress)
- void USBHCDSetConfig (uint32_t ui32Index, uint32_t ui32Device, uint32_t ui32Configuration)
- void USBHCDSetInterface (uint32_t ui32Index, uint32_t ui32Device, uint32_t ui32Interface, uint32_t ui32AltSetting)
- uint32_t USBHCDStringDescriptorGet (tUSBHostDevice *psDevice, uint8_t *pui8Buffer, uint32_t ui32Size, uint32_t ui32LangID, uint32_t ui32StringIndex)
- void USBHCDSuspend (uint32_t ui32Index)
- void USBHCDTerm (uint32_t ui32Index)

3.3.1 Detailed Description

The macros and functions defined in this section can be found in header file `host/usbhost.h`.

3.3.2 Data Structure Documentation

3.3.2.1 tUSBHostClassDriver

Definition:
```
typedef struct
{
    uint32_t ui32InterfaceClass;
    void * (*pfnOpen)(tUSBHostDevice *psDevice);
    void (*pfnClose)(void *pvInstance);
    void (*pfnIntHandler)(void *pvInstance);
}
tUSBHostClassDriver
```

Members:
> *ui32InterfaceClass* The interface class that this device class driver supports.
>
> *pfnOpen* The function is called when this class of device has been detected.
>
> *pfnClose* The function is called when the device, originally opened with a call to the pfnOpen function, is disconnected.
>
> *pfnIntHandler* This is the optional interrupt handler that will be called when an endpoint associated with this device instance generates an interrupt.

Description:
> This structure defines a USB host class driver interface, it is parsed to find a USB class driver once a USB device is enumerated.

3.3.3   Define Documentation

3.3.3.1   DECLARE_EVENT_DRIVER

This macro is used to declare an instance of an Event driver for the USB library.

Definition:
```
#define DECLARE_EVENT_DRIVER(VarName,
                             pfnOpen,
                             pfnClose,
                             pfnEvent)
```

Parameters:
>   ***VarName*** is the name of the variable.
>   ***pfnOpen*** is the callback for the Open call to this driver. This value is currently reserved and
>   should be set to 0.
>   ***pfnClose*** is the callback for the Close call to this driver. This value is currently reserved and
>   should be set to 0.
>   ***pfnEvent*** is the callback that will be called for various USB events.

Description:
>   The first parameter is the actual name of the variable that will be declared by this macro. The
>   second and third parameter are reserved for future functionality and are unused and should be
>   set to zero. The last parameter is the actual callback function and is specified as a function
>   pointer of the type:
>
>   ```
>   void (*pfnEvent)(void *pvData);
>   ```
>
>   When the *pfnEvent* function is called the void pointer that is passed in as a parameter should
>   be cast to a pointer to a structure of type tEventInfo. This will contain the event that caused the
>   pfnEvent function to be called.

3.3.4   Function Documentation

3.3.4.1   USB0HostIntHandler

The USB host mode interrupt handler for controller index 0.

Prototype:
```
void
USB0HostIntHandler(void)
```

Description:
>   This the main USB interrupt handler entry point. This handler will branch the interrupt off to the
>   appropriate handlers depending on the current status of the USB controller. This function must
>   be placed in the interrupt table in order for the USB Library host stack to function.

Returns:
>   None.

3.3.4.2 USBHCDControlTransfer

This function completes a control transaction to a device.

Prototype:
```
uint32_t
USBHCDControlTransfer(uint32_t ui32Index,
                      tUSBRequest *psSetupPacket,
                      tUSBHostDevice *psDevice,
                      uint8_t *pui8Data,
                      uint32_t ui32Size,
                      uint32_t ui32MaxPacketSize)
```

Parameters:
>**ui32Index** is the controller index to use for this transfer.
>
>**psSetupPacket** is the setup request to be sent.
>
>**psDevice** is the device instance pointer for this request.
>
>**pui8Data** is the data to send for OUT requests or the receive buffer for IN requests.
>
>**ui32Size** is the size of the buffer in *pui8Data*.
>
>**ui32MaxPacketSize** is the maximum packet size for the device for this request.

Description:
>This function handles the state changes necessary to send a control transaction to a device. This function should not be called from within an interrupt callback as it is a blocking function.

Returns:
>The number of bytes of data that were sent or received as a result of this request.

3.3.4.3 USBHCDDevAddress

This function will return the USB address for the requested device instance.

Prototype:
```
uint8_t
USBHCDDevAddress(uint32_t ui32Instance)
```

Parameters:
>**ui32Instance** is a unique value indicating which device to query.

Description:
>This function returns the USB address for the device that is associated with the *ui32Instance* parameter. The caller must use a value for *ui32Instance* have been passed to the application when it receives a **USB_EVENT_CONNECTED** event. The function will return the USB address for the interface number specified by the *ui32Interface* parameter.

Returns:
>The USB address for the requested interface.

3.3.4.4 USBHCDDevClass

This function will return the USB class for the requested device instance.

Prototype:
```
uint8_t
USBHCDDevClass(uint32_t ui32Instance,
               uint32_t ui32Interface)
```

Parameters:
>   ***ui32Instance*** is a unique value indicating which device to query.
>   ***ui32Interface*** is the interface number to query for the USB class.

Description:
>   This function returns the USB class for the device that is associated with the *ui32Instance* parameter. The caller must use a value for *ui32Instance* have been passed to the application when it receives a **USB_EVENT_CONNECTED** event. The function will return the USB class for the interface number specified by the *ui32Interface* parameter. If *ui32Interface* is set to 0xFFFFFFFF then the function will return the USB class for the first interface that is found in the device's USB descriptors.

Returns:
>   The USB class for the requested interface.

3.3.4.5 USBHCDDevHubPort

This function returns the USB hub port for the requested device instance.

Prototype:
```
uint8_t
USBHCDDevHubPort(uint32_t ui32Instance)
```

Parameters:
>   ***ui32Instance*** is a unique value indicating which device to query.

Description:
>   This function returns the USB hub port for the device that is associated with the *ui32Instance* parameter. The caller must use the value for *ui32Instance* was passed to the application when it receives a **USB_EVENT_CONNECTED** event. The function returns the USB hub port for the interface number specified by the *ui32Interface* parameter.

Returns:
>   The USB hub port for the requested interface.

3.3.4.6 USBHCDDevProtocol

This function returns the USB protocol for the requested device instance.

Prototype:
```
uint8_t
USBHCDDevProtocol(uint32_t ui32Instance,
                  uint32_t ui32Interface)
```

Parameters:
>   ***ui32Instance*** is a unique value indicating which device to query.
>
>   ***ui32Interface*** is the interface number to query for the USB protocol.

Description:
>   This function returns the USB protocol for the device that is associated with the *ui32Instance* parameter. The caller must use a value for *ui32Instance* have been passed to the application when it receives a **USB_EVENT_CONNECTED** event. The function will return the USB protocol for the interface number specified by the *ui32Interface* parameter. If *ui32Interface* is set to 0xFFFFFFFF then the function will return the USB protocol for the first interface that is found in the device's USB descriptors.

Returns:
>   The USB protocol for the requested interface.

3.3.4.7   USBHCDDevSubClass

This function will return the USB subclass for the requested device instance.

Prototype:
```
uint8_t
USBHCDDevSubClass(uint32_t ui32Instance,
                  uint32_t ui32Interface)
```

Parameters:
>   ***ui32Instance*** is a unique value indicating which device to query.
>
>   ***ui32Interface*** is the interface number to query for the USB subclass.

Description:
>   This function returns the USB subclass for the device that is associated with the *ui32Instance* parameter. The caller must use a value for *ui32Instance* have been passed to the application when it receives a **USB_EVENT_CONNECTED** event. The function will return the USB subclass for the interface number specified by the *ui32Interface* parameter. If *ui32Interface* is set to 0xFFFFFFFF then the function will return the USB subclass for the first interface that is found in the device's USB descriptors.

Returns:
>   The USB subclass for the requested interface.

3.3.4.8   USBHCDEventDisable

This function is called to disable a specific USB HCD event notification.

Prototype:
```
int32_t
USBHCDEventDisable(uint32_t ui32Index,
                   void *pvEventDriver,
                   uint32_t ui32Event)
```

Parameters:
>   ***ui32Index*** specifies which USB controller to use.

pvEventDriver is the event driver structure that was passed into the USBHCDRegister-Drivers() function as part of the array of tUSBHostClassDriver structures.

ui32Event is the event to disable.

Description:

This function is called to disable event callbacks for a specific USB HCD event. The requested event is passed in the *ui32Event* parameter. Not all events can be enables so the function will return zero if the event provided cannot be enabled. The *pvEventDriver* is a pointer to the event driver structure that the caller passed into the USBHCDRegisterDrivers() function. This structure is typically declared with the DECLARE_EVENT_DRIVER() macro and included as part of the array of pointers to tUSBHostClassDriver structures that is passed to the USBHC-DRegisterDrivers() function.

Returns:

This function returns a non-zero number if the event was successfully disabled and returns zero if the event cannot be disabled.

3.3.4.9   USBHCDEventEnable

This function is called to enable a specific USB HCD event notification.

Prototype:

```
int32_t
USBHCDEventEnable(uint32_t ui32Index,
                  void *pvEventDriver,
                  uint32_t ui32Event)
```

Parameters:

ui32Index specifies which USB controller to use.

pvEventDriver is the event driver structure that was passed into the USBHCDRegister-Drivers() function as part of the array of tUSBHostClassDriver structures.

ui32Event is the event to enable.

Description:

This function is called to enable event callbacks for a specific USB HCD event. The requested event is passed in the *ui32Event* parameter. Not all events can be enables so the function will return zero if the event provided cannot be enabled. The *pvEventDriver* is a pointer to the event driver structure that the caller passed into the USBHCDRegisterDrivers() function. This structure is typically declared with the DECLARE_EVENT_DRIVER() macro and included as part of the array of pointers to tUSBHostClassDriver structures that is passed to the USBHC-DRegisterDrivers() function.

Returns:

This function returns a non-zero number if the event was successfully enabled and returns zero if the event cannot be enabled.

3.3.4.10   USBHCDFeatureSet

This function sets one of the **USBLIB_FEATURE_** requests.

Prototype:
```
bool
USBHCDFeatureSet(uint32_t ui32Index,
                 uint32_t ui32Feature,
                 void *pvFeature)
```

Parameters:
> ***ui32Index*** is the index of the USB controller to access.
>
> ***ui32Feature*** is one of the **USBLIB_FEATURE_** defines.
>
> ***pvFeature*** is a pointer to the data for the **USBLIB_FEATURE** request.

Description:
> This function sends the requested feature request to the USB library. Not all features are supported by all devices so see the documentation for the **USBLIB_FEATURE_** to determine if the feature is supported.

Returns:
> Returns **true** if the feature was set and **false** if the feature is not supported or could not be changed to the requested value.

3.3.4.11 USBHCDInit

This function is used to initialize the HCD code.

Prototype:
```
void
USBHCDInit(uint32_t ui32Index,
           void *pvPool,
           uint32_t ui32PoolSize)
```

Parameters:
> ***ui32Index*** specifies which USB controller to use.
>
> ***pvPool*** is a pointer to the data to use as a memory pool for this controller.
>
> ***ui32PoolSize*** is the size in bytes of the buffer passed in as *pvPool*.

Description:
> This function will perform all the necessary operations to allow the USB host controller to begin enumeration and communication with devices. This function should typically be called once at the start of an application once all of the device and class drivers are ready for normal operation. This call will start up the USB host controller and any connected device will immediately start the enumeration sequence.
>
> The USBStackModeSet() function can be called with eUSBModeHost in order to cause the USB library to force the USB operating mode to a host controller. This allows the application to used the USBVBUS and USBID pins as GPIOs on devices that support forcing OTG to operate as a host only controller. By default the USB library will assume that the USBVBUS and USBID pins are configured as USB pins and not GPIOs.
>
> The memory pool passed to this function must be at least as large as a typical configuration descriptor for devices that are to be supported. This value is application-dependent however it should never be less than 32 bytes and, in most cases, should be at least 64 bytes. If there is not sufficient memory to load a configuration descriptor from a device, the device will not be recognized by the USB library's host controller driver.

Returns:
> None.

3.3.4.12 USBHCDLPMResume

This function generates an LPM request for a device to exit L1 sleep state.

Prototype:
```
void
USBHCDLPMResume(uint32_t ui32Index)
```

Parameters:
> ***ui32Index*** specifies which USB controller to use.

Description:
> This function will start LPM resume signaling on the USB bus. This wakes all devices and is similar to USBHCDResume() call but is triggered by an LPM request.

Returns:
> None.

3.3.4.13 USBHCDLPMSleep

This function generates an LPM request for a device to enter L1 sleep state.

Prototype:
```
uint32_t
USBHCDLPMSleep(tUSBHostDevice *psDevice)
```

Parameters:
> ***psDevice*** is the device to query.

Description:
> This function sends a request to a device to enter the LPM L1 sleep state. The caller must check the return value to see if the request can be attempted at this time. If another LPM transaction is busy on another device this function will return **USBHCD_LPM_PENDING** or **USBHCD_LPM_AVAIL** if the LPM request was scheduled to be sent. The caller should check the USBHCDLPMStatus() function to determine if the request has completed.

Returns:
> This function returns the following values:

> - USBHCD_LPM_AVAIL - The transition to L1 state is scheduled to be sent.
> - USBHCD_LPM_PENDING - There is already an LPM request pending.

3.3.4.14 USBHCDLPMStatus

This function returns the current status of an LPM request.

Prototype:
```
uint32_t
USBHCDLPMStatus(tUSBHostDevice *psDevice)
```

Parameters:
psDevice is the device to query.

Description:
This function returns the current status of LPM requests for a given device. This is called to determine if a previous request completed successfully or if there was an error.

Returns:
This function returns the following values:

- **USBHCD_LPM_AVAIL** - There are no pending LPM requests on this specific device or the last request completed successfully.
- **USBHCD_LPM_ERROR** - The last LPM request for this device did not complete success-fully.
- **USBHCD_LPM_PENDING** - The last LPM request has not completed.

3.3.4.15 USBHCDMain

This function is the main routine for the Host Controller Driver.

Prototype:
```
void
USBHCDMain(void)
```

Description:
This function is the main routine for the host controller driver, and must be called periodically by the main application outside of a callback context. This allows for a simple cooperative system to access the the host controller driver interface without the need for an RTOS. All time critical operations are handled in interrupt context but all blocking operations are run from the this function to allow them to block and wait for completion without holding off other interrupts.

Returns:
None.

3.3.4.16 USBHCDPipeAlloc

This function is used to allocate a USB HCD pipe.

Prototype:
```
uint32_t
USBHCDPipeAlloc(uint32_t ui32Index,
                uint32_t ui32EndpointType,
                tUSBHostDevice *psDevice,
                tHCDPipeCallback pfnCallback)
```

Parameters:
ui32Index specifies which USB controller to use.

ui32EndpointType is the type of endpoint that this pipe will be communicating with.

psDevice is the device instance associated with this endpoint.

pfnCallback is the function that will be called when events occur on this USB Pipe.

Description:

Since there are a limited number of USB HCD pipes that can be used in the host controller, this function is used to temporarily or permanently acquire one of the endpoints. It also provides a method to register a callback for status changes on this endpoint. If no callbacks are desired then the *pfnCallback* function should be set to 0. The callback should be used when using the USBHCDPipeSchedule() function so that the caller is notified when the action is complete.

Returns:

This function returns a value indicating which pipe was reserved. If the value is 0 then there were no pipes currently available. This value should be passed to any USBHCDPipe APIs to indicate which pipe is being accessed.

3.3.4.17   USBHCDPipeAllocSize

This function is used to allocate a USB HCD pipe.

Prototype:
```
uint32_t
USBHCDPipeAllocSize(uint32_t ui32Index,
                    uint32_t ui32EndpointType,
                    tUSBHostDevice *psDevice,
                    uint32_t ui32Size,
                    tHCDPipeCallback pfnCallback)
```

Parameters:

ui32Index specifies which USB controller to use.

ui32EndpointType is the type of endpoint that this pipe will be communicating with.

psDevice is the device instance associated with this endpoint.

ui32Size is the size of the FIFO in bytes.

pfnCallback is the function that will be called when events occur on this USB Pipe.

Description:

Since there are a limited number of USB HCD pipes that can be used in the host controller, this function is used to temporarily or permanently acquire one of the endpoints. Unlike the USBHCDPipeAlloc() function this function allows the caller to specify the size of the FIFO allocated to this endpoint in the *ui32Size* parameter. This function also provides a method to register a callback for status changes on this endpoint. If no callbacks are desired then the *pfnCallback* function should be set to 0. The callback should be used when using the USBHCDPipeSchedule() function so that the caller is notified when the action is complete.

Returns:

This function returns a value indicating which pipe was reserved. If the value is 0 then there were no pipes currently available. This value should be passed to any USBHCDPipe APIs to indicate which pipe is being accessed.

3.3.4.18 USBHCDPipeConfig

This function is used to configure a USB HCD pipe.

Prototype:
```
uint32_t
USBHCDPipeConfig(uint32_t ui32Pipe,
                 uint32_t ui32MaxPayload,
                 uint32_t ui32Interval,
                 uint32_t ui32TargetEndpoint)
```

Description:
This should be called after allocating a USB pipe with a call to USBHCDPipeAlloc(). It is used to set the configuration associated with an endpoint like the max payload and target endpoint. The *ui32MaxPayload* parameter is typically read directly from the devices endpoint descriptor and is expressed in bytes.

Setting the *ui32Interval* parameter depends on the type of endpoint being configured. For endpoints that do not need to use the *ui32Interval* parameter *ui32Interval* should be set to 0. For Bulk *ui32Interval* is a value from 2-16 and will set the NAK timeout value as $2^\wedge$(*ui32Interval-1*) frames. For interrupt endpoints *ui32Interval* is a value from 1-255 and is the count in frames between polling the endpoint. For isochronous endpoints *ui32Interval* ranges from 1-16 and is the polling interval in frames represented as $2^\wedge$(*ui32Interval-1*) frames.

Parameters:
ui32Pipe is the allocated endpoint to modify.
ui32MaxPayload is maximum data that can be handled per transaction.
ui32Interval is the polling interval for data transfers expressed in frames.
ui32TargetEndpoint is the target endpoint on the device to communicate with.

Returns:
If the call was successful, this function returns zero any other value indicates an error.

3.3.4.19 USBHCDPipeDataAck

This function acknowledges data received via an interrupt IN pipe.

Prototype:
```
void
USBHCDPipeDataAck(uint32_t ui32Pipe)
```

Parameters:
ui32Pipe is the USB INT pipe whose last packet is to be acknowledged.

Description:
This function is used to acknowledge reception of data on an interrupt IN pipe. A transfer on an interrupt IN endpoint is scheduled via a call to USBHCDPipeSchedule() and the application is notified when data is received using a **USB_EVENT_RX_AVAILABLE** event. In the handler for this event, the application must call USBHCDPipeDataAck() to have the USB controller ACK the data from the device and complete the transaction.

Returns:
None.

3.3.4.20 USBHCDPipeFree

This function is used to release a USB pipe.

Prototype:
```
void
USBHCDPipeFree(uint32_t ui32Pipe)
```

Parameters:
>**ui32Pipe** is the allocated USB pipe to release.

Description:
>This function is used to release a USB pipe that was allocated by a call to USBHCDPipeAlloc()
>for use by some other device endpoint in the system. Freeing an unallocated or invalid pipe will
>not generate an error and will instead simply return.

Returns:
>None.

3.3.4.21 USBHCDPipeRead

This function is used to read data from a USB HCD pipe.

Prototype:
```
uint32_t
USBHCDPipeRead(uint32_t ui32Pipe,
               uint8_t *pui8Data,
               uint32_t ui32Size)
```

Parameters:
>**ui32Pipe** is the USB pipe to read data from.
>**pui8Data** is a pointer to store the data that is received.
>**ui32Size** is the size in bytes of the buffer pointed to by *pui8Data*.

Description:
>This function will block and will only return when it has read as much data as requested from
>the USB pipe. The caller must register a callback with the USBHCDPipeAlloc() call in order to
>be informed when the data has been received. If the caller provides a non-zero pointer in the
>*pui8Data* parameter then the data is copied into the buffer before the callback occurs. If the
>caller provides a zero in *pui8Data* parameter then the caller is responsible for reading the data
>out of the FIFO when the **USB_EVENT_RX_AVAILABLE** callback event occurs. The value
>returned by this function can be less than the *ui32Size* requested if the USB pipe has less data
>available than was requested.

Returns:
>This function returns the number of bytes that were returned in the *pui8Data* buffer.

3.3.4.22 USBHCDPipeReadNonBlocking

This function is used to read data from a USB HCD pipe.

Prototype:
```
uint32_t
USBHCDPipeReadNonBlocking(uint32_t ui32Pipe,
                          uint8_t *pui8Data,
                          uint32_t ui32Size)
```

Parameters:
ui32Pipe is the USB pipe to read data from.

pui8Data is a pointer to store the data that is received.

ui32Size is the size in bytes of the buffer pointed to by *pui8Data*.

Description:
This function will not block and will only read as much data as requested or as much data is currently available from the USB pipe. The caller should have registered a callback with the USBHCDPipeAlloc() call in order to be informed when the data has been received. The value returned by this function can be less than the *ui32Size* requested if the USB pipe has less data available than was requested.

Returns:
This function returns the number of bytes that were returned in the *pui8Data* buffer.

3.3.4.23 USBHCDPipeSchedule

This function is used to schedule and IN transaction on a USB HCD pipe.

Prototype:
```
uint32_t
USBHCDPipeSchedule(uint32_t ui32Pipe,
                   uint8_t *pui8Data,
                   uint32_t ui32Size)
```

Parameters:
ui32Pipe is the USB pipe to read data from.

pui8Data is a pointer to store the data that is received.

ui32Size is the size in bytes of the buffer pointed to by *pui8Data*.

Description:
This function will not block depending on the type of pipe passed in will schedule either a send of data to the device or a read of data from the device. In either case the amount of data will be limited to what will fit in the FIFO for a given endpoint.

Returns:
This function returns the number of bytes that were sent in the case of a transfer of data or it will return 0 for a request on a USB IN pipe.

3.3.4.24 USBHCDPipeTransferSizeGet

This function returns the current byte count of a USB HCD pipe.

Prototype:
```
uint32_t
USBHCDPipeTransferSizeGet(uint32_t ui32Pipe)
```

Parameters:
   ui32Pipe is the allocated endpoint to modify.

Description:
   This call returns the current or last byte count for a transfer using the pipe specified by the *ui32Pipe* parameter. This is typically used to determine the actual byte count received when a **USB_EVENT_RX_AVAILABLE** occurs.

Returns:
   If the call was successful, this function returns the number of bytes transfered by the USB pipe.

3.3.4.25 USBHCDPipeWrite

This function is used to write data to a USB HCD pipe.

Prototype:
```
uint32_t
USBHCDPipeWrite(uint32_t ui32Pipe,
                uint8_t *pui8Data,
                uint32_t ui32Size)
```

Parameters:
   ui32Pipe is the USB pipe to put data into.
   pui8Data is a pointer to the data to send.
   ui32Size is the amount of data to send.

Description:
   This function will block until it has sent as much data as was requested using the USB pipe's FIFO. The caller should have registered a callback with the USBHCDPipeAlloc() call in order to be informed when the data has been transmitted. The value returned by this function can be less than the *ui32Size* requested if the USB pipe has less space available than this request is making.

Returns:
   This function returns the number of bytes that were scheduled to be sent on the given USB pipe.

3.3.4.26 USBHCDPowerAutomatic

This function returns if the current power settings will automatically handle enabling and disabling VBUS power.

Prototype:
```
uint32_t
USBHCDPowerAutomatic(uint32_t ui32Index)
```

Parameters:
   ui32Index specifies which USB controller to query.

Description:
> This function returns if the current power control pin configuration will automatically apply power or whether it will be left to the application to turn on power when it is notified.

Returns:
> A non-zero value indicates that power is automatically applied and a value of zero indicates that the application must manually apply power.

3.3.4.27 USBHCDPowerConfigGet

This function is used to get the power pin and power fault configuration.

Prototype:
```
uint32_t
USBHCDPowerConfigGet(uint32_t ui32Index)
```

Parameters:
> ***ui32Index*** specifies which USB controller to use.

Description:
> This function will return the current power control pin configuration as set by the USBHCD-PowerConfigInit() function or the defaults if not yet set. See the USBHCDPowerConfigInit() documentation for the meaning of the bits that are returned by this function.

Returns:
> The configuration of the power control pins.

3.3.4.28 USBHCDPowerConfigInit

This function is used to set the power pin and power fault configuration.

Prototype:
```
void
USBHCDPowerConfigInit(uint32_t ui32Index,
                      uint32_t ui32PwrConfig)
```

Parameters:
> ***ui32Index*** specifies which USB controller to use.
> ***ui32PwrConfig*** is the power configuration to use for the application.

Description:
> This function must be called before HCDInit() is called so that the power pin configuration can be set before power is enabled. The *ui32PwrConfig* flags specify the power fault level sensitivity, the power fault action, and the power enable pin level and source.
>
> One of the following can be selected as the power fault level sensitivity:
>
> - **USBHCD_FAULT_LOW** - An external power fault is indicated by the pin being driven low.
> - **USBHCD_FAULT_HIGH** - An external power fault is indicated by the pin being driven high.
>
> One of the following can be selected as the power fault action:

- **USBHCD_FAULT_VBUS_NONE** - No automatic action when power fault detected.
- **USBHCD_FAULT_VBUS_TRI** - Automatically Tri-state the USBnEPEN pin on a power fault.
- **USBHCD_FAULT_VBUS_DIS** - Automatically drive the USBnEPEN pin to it's inactive state on a power fault.

One of the following can be selected as the power enable level and source:

- **USBHCD_VBUS_MANUAL** - Power control is completely managed by the application, the USB library will provide a power callback to request power state changes.
- **USBHCD_VBUS_AUTO_LOW** - USBEPEN is driven low by the USB controller automatically if USBOTGSessionRequest() has enabled a session.
- **USBHCD_VBUS_AUTO_HIGH** - USBEPEN is driven high by the USB controller automatically if USBOTGSessionRequest() has enabled a session.

If **USBHCD_VBUS_MANUAL** is used then the application must provide an event driver to receive the **USB_EVENT_POWER_ENABLE** and **USB_EVENT_POWER_DISABLE** events and enable and disable power to VBUS when requested by the USB library. The application should respond to a power control callback by enabling or disabling VBUS as soon as possible and before returning from the callback function.

Note:
The following values should no longer be used with the USB library: **USB_HOST_PWRFLT_LOW**, **USB_HOST_PWRFLT_HIGH**, **USB_HOST_PWRFLT_EP_NONE**, **USB_HOST_PWRFLT_EP_TRI**, **USB_HOST_PWRFLT_EP_LOW**, **USB_HOST_PWRFLT_EP_HIGH**, **USB_HOST_PWREN_LOW**, **USB_HOST_PWREN_HIGH**, **USB_HOST_PWREN_VBLOW**, and **USB_HOST_PWREN_VBHIGH**.

Returns:
None.

3.3.4.29 USBHCDPowerConfigSet

This function is used to set the power pin and power fault configuration.

Prototype:
```
uint32_t
USBHCDPowerConfigSet(uint32_t ui32Index,
                     uint32_t ui32Config)
```

Parameters:
ui32Index specifies which USB controller to use.
ui32Config specifies which USB power configuration to use.

Description:
This function will set the current power control pin configuration as set by the USBHCDPowerConfigInit() function or the defaults if not yet set. See the USBHCDPowerConfigInit() documentation for the meaning of the bits that are set by this function.

Returns:
Returns zero to indicate the power setting is now active.

3.3.4.30 USBHCDRegisterDrivers

This function is used to initialize the HCD class driver list.

Prototype:
```
void
USBHCDRegisterDrivers(uint32_t ui32Index,
                          const tUSBHostClassDriver *const
*ppsHClassDrvs,
                          uint32_t ui32NumDrivers)
```

Parameters:
>   ***ui32Index*** specifies which USB controller to use.
>   ***ppsHClassDrvs*** is an array of host class drivers that are supported on this controller.
>   ***ui32NumDrivers*** is the number of entries in the *pHostClassDrivers* array.

Description:
>   This function will set the host classes supported by the host controller specified by the *ui32Index* parameter. This function should be called before enabling the host controller driver with the USBHCDInit() function.

Returns:
>   None.

3.3.4.31 USBHCDReset

This function generates reset signaling on the USB bus.

Prototype:
```
void
USBHCDReset(uint32_t ui32Index)
```

Parameters:
>   ***ui32Index*** specifies which USB controller to use.

Description:
>   This function handles sending out reset signaling on the USB bus. After returning from this function, any attached device on the USB bus should have returned to it's reset state.

Returns:
>   None.

3.3.4.32 USBHCDResume

This function will generate resume signaling on the USB bus.

Prototype:
```
void
USBHCDResume(uint32_t ui32Index)
```

Parameters:
> ***ui32Index*** specifies which USB controller to use.

Description:
> This function is used to generate resume signaling on the USB bus in order to cause USB
> devices to leave their suspended state. This call should not be made unless a preceding call
> to USBHCDSuspend() has been made.

Returns:
> None.

3.3.4.33 USBHCDSetAddress

This function is used to send the set address command to a device.

Prototype:
```
void
USBHCDSetAddress(uint32_t ui32DevIndex,
                 uint32_t ui32DevAddress)
```

Parameters:
> ***ui32DevIndex*** is the index of the device whose address is to be set. This value must be 0 to
> indicate that the device is connected directly to the host controller. Higher values indicate
> devices connected via a hub.
>
> ***ui32DevAddress*** is the new device address to use for a device.

Description:
> The USBHCDSetAddress() function is used to set the USB device address, once a device has
> been discovered on the bus. This call is typically issued following a USB reset triggered by
> a call the USBHCDReset(). The address passed into this function via the *ui32DevAddress*
> parameter is used for all further communications with the device after this function returns.

Returns:
> None.

3.3.4.34 USBHCDSetConfig

This function is used to set the current configuration for a device.

Prototype:
```
void
USBHCDSetConfig(uint32_t ui32Index,
                uint32_t ui32Device,
                uint32_t ui32Configuration)
```

Parameters:
> ***ui32Index*** specifies which USB controller to use.
>
> ***ui32Device*** is the USB device for this function.
>
> ***ui32Configuration*** is one of the devices valid configurations.

Description:

This function is used to set the current device configuration for a USB device. The *ui32Configuration* value must be one of the configuration indexes that was returned in the configuration descriptor from the device, or a value of 0. If 0 is passed in, the device will return to it's addressed state and no longer be in a configured state. If the value is non-zero then the device will change to the requested configuration.

Returns:

None.

3.3.4.35 USBHCDSetInterface

This function is used to set the current interface and alternate setting for an interface on a device.

Prototype:

```
void
USBHCDSetInterface(uint32_t ui32Index,
                   uint32_t ui32Device,
                   uint32_t ui32Interface,
                   uint32_t ui32AltSetting)
```

Parameters:

ui32Index specifies which USB controller to use.

ui32Device is the USB device for this function.

ui32Interface is one of the valid interface numbers for a device.

ui32AltSetting is one of the valid alternate interfaces for the *ui32Interface* number.

Description:

This function is used to change the alternate setting for one of the valid interfaces on a USB device. The *ui32Device* specifies the device instance that was returned when the device was connected. This call will set the USB device's interface based on the *ui32Interface* and *ui32AltSetting*.

Example: Set the USB device interface 2 to alternate setting 1.

```
USBHCDSetInterface(0, ui32Device, 2, 1);
```

Returns:

None.

3.3.4.36 USBHCDStringDescriptorGet

This function issues a request for a string descriptor from a device.

Prototype:

```
uint32_t
USBHCDStringDescriptorGet(tUSBHostDevice *psDevice,
                          uint8_t *pui8Buffer,
                          uint32_t ui32Size,
                          uint32_t ui32LangID,
                          uint32_t ui32StringIndex)
```

Parameters:

psDevice is the device for this request.

pui8Buffer is the pointer to the buffer to store the requested string descriptor.

ui32Size is the size of the buffer passed in the buffer that will be used for this request.

ui32LangID is the ID of the language for the requested string.

ui32StringIndex is the index for the request.

Description:

This function will request a string descriptor from the device of the type specified in the *ui32DescriptorType* parameter. The *pui8Descriptor* pointer is the location where the request results will be stored. The *ui32Size* should be passed in to indicate the size of the *pui8Descriptor* buffer. The *ui32DevAddress* parameter is used to specify the device address to communicate with on the USB bus. This value should be specified as 0 for any non-configured device on the USB bus and be changed to the address set by a call to USBHCDSetAddress().

Returns:

The number of bytes returned in the *pui8Buffer* due to the request. This value can be zero if the device did not respond.

3.3.4.37 USBHCDSuspend

This function will generate suspend signaling on the USB bus.

Prototype:
```
void
USBHCDSuspend(uint32_t ui32Index)
```

Parameters:

ui32Index specifies which USB controller to use.

Description:

This function is used to generate suspend signaling on the USB bus. In order to leave the suspended state, the application should call USBHCDResume().

Returns:

None.

3.3.4.38 USBHCDTerm

This function is used to terminate the HCD code.

Prototype:
```
void
USBHCDTerm(uint32_t ui32Index)
```

Parameters:

ui32Index specifies which USB controller to release.

Description:

This function will clean up the USB host controller and disable it in preparation for shutdown or a switch to USB device mode. Once this call is made, *USBHCDInit()* may be called to reinitialize the controller and prepare for host mode operation.

Returns:
      None.

3.4   Host Class Driver

The host class drivers provide access to devices that use a common USB class interface. The USB library currently supports the following two USB class drivers: Mass Storage Class(MSC) and Human Interface Device(HID). In order to use these class drivers, the application must provide a list of the host class drivers that it uses by calling the USBHCDRegisterDrivers() function. The *g_USBHIDClassDriver* structure defines the interface for the Host HID class driver and the *g_USBHostMSCClassDriver* structure defines the interface for the Host MSC class driver.

The host class driver provides interfaces at its bottom layer to the USB host controller driver and device specific interfaces at it's top layer. The lower layer interface to the USB host controller interface is the same for all USB host class drivers while the device interface layer on top is common to all USB host device interface of a given class. Thus the top layer of the of the MSC class driver does not need to match the top layer of the HID class driver, however the lower layer must be the same for both. Aside from enumeration, all communication with the host class driver is through its endpoint pipes. The host class driver parses and allocate any endpoints that it needs by calling the USBHCDPipeAlloc() and USBHCDPipeConfig() functions. These USB pipes provide the methods to read/write and get callback notification from the USB host controller driver layer.

3.4.1   USB Events Driver

The USB host library includes a method to receive non-device class specific events in the application by using a USB event driver. This driver can be included in applications by declaring an instance of the USB event driver using the DECLARE_EVENT_DRIVER() macro and then adding the variable that is declared to the list of drivers supported by the application. This event driver allows applications to notify users that an unsupported device has been inserted or to provide notification that a power fault has occurred and power may have been shut off, depending on the settings provided to the USBHCDPowerConfigInit() function. Depending on configuration the following events can occur:

- **USB_EVENT_CONNECTED** indicates that a support device has been connected.
- **USB_EVENT_UNKNOWN_CONNECTED** indicates that an unsupported device has been connected.
- **USB_EVENT_DISCONNECTED** indicates that an unsupported device has been disconnected.
- **USB_EVENT_POWER_FAULT** indicates that a power fault has occurred.
- **USB_EVENT_POWER_ENABLE** indicates that power should be enabled by the application since it has requested to manually control the power.
- **USB_EVENT_POWER_DISABLE** indicates that power should be enabled by the application since it has requested to manually control the power.
- **USB_EVENT_SOF** indicates that a SOF event has occurred(default disabled).

The USB host library provides the ability to enable or disable any of these events by calling the USBHCDEventEnable() or USBHCDEventDisable() functions. All events except the **USB_EVENT_SOF**

are enabled by default when the USB host library is initialized. The **USB_EVENT_SOF** event is left disabled by default to avoid the excess overhead because this event occurs once per millisecond.

Because the USB events driver reuses the interrupt handler callback that is used for a normal host controller drivers, the application is required to cast the void pointer that is passed in to the function to a pointer to a tEventInfo structure. The following code example shows a basic implementation of a USB library event driver callback function.

Example: A USB Event Driver Callback Function

```
//
// Declare the driver.
//
DECLARE_EVENT_DRIVER(g_sEventDriver, 0, 0, USBHCDEvents)

void
USBHCDEvents(void *pvData)
{
    tEventInfo *psEventInfo;

    //
    // Cast this pointer to its actual type.
    //
    psEventInfo = (tEventInfo *)pvData;

    switch(psEventInfo->ui32Event)
    {
        //
        // Unknown device connected.
        //
        case USB_EVENT_CONNECTED:
        {
            ...

            break;
        }

        //
        // Unknown device disconnected.
        //
        case USB_EVENT_DISCONNECTED:
        {
            ...

            break;
        }

        //
        // Power Fault detected.
        //
        case USB_EVENT_POWER_FAULT:
        {
            ...

            break;
        }

        default:
        {
            break;
        }
    }
}
```

3.4.2   Hub Class Driver

The USB Hub class driver provides support for a USB hub device that allows the USB controller to communicate with multiple USB devices. The maximum number of devices that are supported is controlled by the **MAX_USB_DEVICES** definition in usblib.h. The value defined by **MAX_USB_DEVICES** defaults to 5, meaning that the USB library supports one hub and four other devices. Cascaded USB hubs are not supported because the USB library only supports a single instance of a USB hub. The application-level interface to the USB hub class consists of only an initialization function and requires no additional application-level changes to handle any of the supported USB classes. When USB hub support is enabled, non-hub devices can still be directly connected to the USB controller with no special handling by the application. The next section covers the application interfaces to the USB hub class and the memory requirements when enabling the USB hub support.

Device Interface

The application layer of the USB hub class driver provides functions that an application uses to configure or disable the USB hub class driver. To initialize the USB hub class driver, the application must call USBHHubOpen() and provide it with a memory pool suitable to hold the configuration descriptors of the maximum number of attached devices and a hub instance structure. The memory pool allocation is very similar to how an application provides memory to the USBHCDInit() function with the exception that that the amount of memory should be multiplied by the number of devices supported. This memory pool size should be the expected maximum configuration descriptor size multiplied by **MAX_USB_DEVICES**. The application also provides a tHubInstance structure that holds private instance data that should not be accessed by the application. In order to release an instance of a hub class driver, the application must call USBHHubClose(). This call to USBHHubClose() is only made if the application is shutting down the USB interface or when in OTG mode, switching roles from host to device.

Example: USB Hub with Keyboard and MSC support.

```
//*****************************************************************************
//
// The size of the host controller's memory pool in bytes.
//
//*****************************************************************************
#define HCD_MEMORY_SIZE         128

//*****************************************************************************
//
// The memory pool to provide to the Host controller driver.
//
//*****************************************************************************
uint8_t g_pui8HCDPool[HCD_MEMORY_SIZE];

//*****************************************************************************
//
// The size of the host controller's memory pool in bytes.
//
//*****************************************************************************
#define HUB_POOL_SIZE           (HCD_MEMORY_SIZE * MAX_USB_DEVICES)

//*****************************************************************************
//
// The memory pool to provide to the hub driver.  This pool is used to hold the
// configuration descriptors of the devices attached to the hub.  It must be
```

```
// sized to be at least
// (MAX_USB_DEVICES * (largest expected configuration descriptor)) bytes.
//
//*****************************************************************************
uint8_t g_pui8HubPool[HUB_POOL_SIZE];

//*****************************************************************************
//
// The instance data for the hub, which is internal data and should not be
// accessed by the application.
//
//*****************************************************************************
tHubInstance g_sHubInstance;

//*****************************************************************************
//
// The global that holds all of the host drivers in use in the application.
// In this case, the Mass Storage, HID, and Hub class drivers are present.
//
//*****************************************************************************
static tUSBHostClassDriver const * const g_ppsHostClassDrivers[] =
{
    &g_sUSBHostMSCClassDriver,
    &g_sUSBHIDClassDriver,
    &g_sUSBHubClassDriver,
    &g_sUSBEventDriver
};

//*****************************************************************************
//
// The global that holds the number of class drivers in the
// g_ppsHostClassDrivers list.
//
//*****************************************************************************
static const uint32_t g_ui32NumHostClassDrivers =
    sizeof(g_ppsHostClassDrivers) / sizeof(tUSBHostClassDriver *);

//
// Initialize the USB stack mode to host.
//
USBStackModeSet(0, USB_MODE_HOST, 0);

//
// Register the host class drivers.
//
USBHCDRegisterDrivers(0, g_ppsHostClassDrivers, g_ui32NumHostClassDrivers);

//
// Open the Keyboard and Mass storage interfaces.
//
KeyboardOpen();
MSCOpen();

//
// Open a hub instance and provide it with the memory required to hold
// configuration descriptors for each attached device and the private hub
// instance data.
//
USBHHubOpen(HubCallback);

//
// Initialize the power configuration by configuring the power enable signal
// to be active high and not enabling the power fault.
//
USBHCDPowerConfigInit(0, USBHCD_VBUS_AUTO_HIGH | USBHCD_VBUS_FILTER);
```

```
//
// Initialize the USB controller for OTG operation with a 2ms polling
// rate.
//
HCDInit(0, g_pui8HCDPool, HCD_MEMORY_SIZE);

while(1)
{
    HCDMain();
}
```

3.4.3 HID Class Driver

The HID class driver provides access to any type of HID class by leaving the details of the HID device to the layer above the HID class driver. The top layer of the HID class driver provides common functions to open or close an instance of a HID device, read a device's report descriptor so that it can be parsed by the HID device code, and get and set reports on a HID device. The lower level interface that is connected to the host controller driver is specified in the *g_USBHIDClassDriver* structure. This structure is used to register the HID class driver with the host class driver so that it is called when a HID device is connected and enumerated. The functions in the *g_USBHIDClassDriver* structure should never be called directly by and application or a host class driver as they are reserved for access by the host controller driver.

In the following example the generic HID class driver is registered with the USB host controller driver and then a call is made to open an instance of a mouse class device. Typically the call to USBHHIDOpen() is made from within a device class interface while the USBHCDRegisterDrivers() call is made from the main application. For instance the USBHHIDOpen() for the mouse device provided with the USB library is made in the USBHMouseOpen() function which is part of the USB mouse interface.

Device Interface

At the top layer of the HID class driver, the driver has a device class interface for used by various HID devices. In order for the HID class driver to recognize a device, the device class is responsible for calling the USBHHIDOpen(). This call specifies the type of device and a callback for this device type so that any events related to this device type can be passed back to the device class driver. The defined classes are in the type defined values in the tHIDSubClassProtocol type and are passed into the USBHHIDOpen() call via the eDeviceType parameter. In order to release an instance of a HID class driver, the HID device class or application must call the USBHHIDClose() to allow a new or different type of device to be connected. In the examples provided in the USB library the report descriptors are retrieved but are not used as the examples rely on the "boot" mode of the USB keyboard and mouse to fix the format of the report descriptors. This is accomplished by using the USBHHIDSetReport() interface to force the device into its boot protocol mode. As this could be limiting or not available in other types of applications or devices, the USBHHIDGetReportDescriptor() provides the ability of a generic HID device to query the device for its report descriptor(s). The last two remaining HID interfaces, USBHHIDSetReport() and USBHHIDGetReport(), provide access to the HID reports.

Example: Adding HID Class Driver

```
const tUSBHostClassDriver * const g_ppsUSBHostClassDrivers[] =
{
    &g_USBHIDClassDriver
```

```
    };

    //
    // Register the host class drivers.
    //
    USBHCDRegisterDrivers(0, g_ppsUSBHostClassDrivers, 1);

    ...

    //
    // Open an instance of a HID mouse class driver.
    //
    psMouseInstance = USBHHIDOpen(USBH_HID_DEV_MOUSE,
                                  USBHMouseCallback,
                                  (void *)&g_sUSBHMouse);
```

Once a HID device has been opened the first callback it receives is a **USB_EVENT_CONNECTED** event, indicating that a HID device of the type passed into the USBHHIDOpen() has been connected and the USB library host controller driver has completed enumeration of the device. When the HID device has been removed a **USB_EVENT_DISCONNECTED** event occurs. When shutting down or to release a device, the application should call USBHHIDClose() to disable callbacks. This does not actually power down the device but it stops the driver from calling the application. During normal operation the host class driver receives **USB_EVENT_SCHEDULER** and **USB_EVENT_RX_AVAILABLE** events. The **USB_EVENT_SCHEDULER** indicates that the HID class driver should schedule a new request if it is ready to do so. This done by calling USBJHCD-PipeSchedule() to request that a new IN request is made on the given Interrupt IN pipe. When the **USB_EVENT_RX_AVAILABLE** occurs this indicates that new data is available due to completion of the previous request for data on the Interrupt IN pipe. The **USB_EVENT_RX_AVAILABLE** is passed on the device class interface to allow it to request the data via a call to USBHHIDGetReport(). It is up to the device class driver to interpret the data in the report structure that is returned. In some cases, like the keyboard example, the device class may also need to call the host class driver to issue a set report to send data to the device. This is done by calling the USBHHIDSetReport() interface of the host class driver. This sends data to the device by using the correct USB OUT pipe.

3.4.4   Mass Storage Class Driver

The mass storage host class driver provides access to devices that support the mass storage class protocol. The most common of these devices are USB flash drives. This host class driver provides a simple block based interface to the devices that can be matched up with an application's file system. A USB host class driver for mass storage devices is included with the USB library. It provides a simple block based interface that can be used with an application's file system as it provides direct block interface to mass storage devices based on logical block address.

The mass storage host class driver provides an application API for access to USB flash drives. The API provided is meant to match with file systems that need block based read/write access to flash drives. The USBHMSCBlockRead() and USBHMSCBlockWrite() functions provide the block read and block write device access. These function performs block operations at the size specified by the flash drive. Since some flash drives require some setup time after enumeration before they are ready for drive access, the mass storage class driver provides the USBHMSCDriveReady() function to check if the drive is ready for normal operation.

The mass storage host class driver also provides an interface to the USB library host controller driver to complete enumeration of mass storage class devices. The mass storage class driver information is held in the global structure *g_USBMSCClassDriver*. This structure should only be

referenced by the application and the function pointers in this structure should never called directly by anything other than the host controller driver. The USBHMSCOpen() and USBHMSCClose() provide the interface for the host controller's enumeration code to call when a mass storage class device is detected or removed. It is up to the mass storage host class driver to provide a callback to the file system or application for notification of the drive being removed or added. To make the the mass storage class driver visible to the host controller driver it must be added in the list of drivers provided in the USBHCDRegisterDrivers() function call. The class enumeration constant is set to **USB_CLASS_MASS_STORAGE** so any devices enumerating with value loads this class driver.

Device Interface

This next section covers how an application or file system interacts with the host mass storage class driver provided with the USB library. The application or file system must register the mass storage class driver with a call to USBHCDRegisterDrivers() with the *g_USBHostMSCClassDriver* as a member of the array passed in to the call. Once the host mass storage class driver has been registered, the application must call USBHMSCDriveOpen() to allow the application or file system to be called when a new mass storage device is connected or disconnected or any other mass storage class event occurs.

Example: Adding Mass Storage Class Driver

```
const tUSBHostClassDriver * const g_ppsUSBHostClassDrivers[] =
{
    &g_sUSBHostMSCClassDriver
};

//
// Register the host class drivers.
//
USBHCDRegisterDrivers(0, g_ppsUSBHostClassDrivers, 1);

//
// Initialize the mass storage class driver on controller 0 with the
// MSCCallback() function as the callback for events.
//
USBHMSCDriveOpen(0, MSCCallback);
```

The first callback is a **USB_EVENT_CONNECTED** event, indicating that a mass storage class flash drive was inserted and the USB library host stack has completed enumeration of the device. This does not indicate that the flash drive is ready for read/write operations but that is has been detected. The USBHMSCDriveReady() function should be called to determine when the flash drive is ready for read/write operations. When the device has been removed an **USB_EVENT_DISCONNECTED** event occurs. When shutting down, the application should call USBHMSCDriveClose() to disable callbacks. This does not actually power down the mass storage device but it stops the driver from calling the application.

Once the USBHMSCDriveReady() call indicates that the flash drive is ready, the application can use the USBHMSCBlockRead() and USBHMSCBlockWrite() functions to access the device. These are block based functions that use the logical block address to indicate which block to access. It is important to note that the size passed in to these functions is in blocks and not bytes and that the most common block size is 512 bytes. These calls always read or write a full block so space must be allocated appropriately. The following example shows calls for both reading and writing blocks from the mass storage class device.

Example: Block Read/Write Calls

```
//
// Read 1 block starting at logical block 0.
//
USBHMSCBlockRead(ui32MSCDevice, 0, pui8Buffer, 1);

//
// Write 2 blocks starting at logical block 500.
//
USBHMSCBlockWrite(ui32MSCDevice, 500, pui8Buffer, 2);
```

SCSI Functions

Since most mass storage class device adhere to the SCSI protocol for block based calls, the USB library provides SCSI functions for the mass storage class driver to communicate with flash drives. The commands and data pass over the USB pipes provided by the host controller driver. The only types of mass storage class devices that are supported are devices that use the SCSI protocol. Since flash drives only support a limited subset of the SCSI protocol, only the SCSI functions needed by mass storage class to mount and access flash drives are implemented. The USBHSCSIRead10() and USBHSCSIWrite10() functions are the two functions used for reading and writing to the mass storage class devices. The remaining SCSI functions are used to get information about the mass storage devices like the size of the blocks on the device and the number of blocks present. Others are used for error handling or testing if the device is ready for a new command.

3.4.5 Audio Class Driver

The USB audio host class driver provides access to devices that support the USB audio class protocol. This driver provides access to both audio in and audio out interfaces. The application opens an instance of the audio device by calling USBHostAudioOpen() and providing a callback function to receive events notifications when an audio device has been enumerated and is ready for normal operation(USBH_AUDIO_EVENT_OPEN), or when an active audio device has been disconnected(USBH_AUDIO_EVENT_CLOSE). The application should not access any other APIs that use the interface returned from the USBHostAudioOpen() function until an USBH_AUDIO_EVENT_OPEN event is received and not after a USBH_AUDIO_EVENT_CLOSE event is received. When the application no longer needs the audio interface it can call USBHostAudioClose() to stop the audio device and no longer be notified of changes to the audio device. Audio output is handled by providing buffers to the host audio driver by calling USBHostAudioPlay() and including a callback function for the buffer. The buffers are returned to the application by the callback to the application provided in the USBHostAudioPlay(). This allows the application to gain control while the audio is being scheduled for output. The audio input is handled by providing buffers to the host audio driver by calling USBHostAudioRecord() and passing in a buffer callback as well. Buffers are then scheduled to be filled by the USB controller and returned to the application by the callback function that the application provided in the USBHostAudioRecord() call. The next section provides more detail and examples for each application level API.

Application Interface

The USB host audio application interface provides a basic method for controlling audio output, input and some volume control. Since the USB host audio provides only a small amount of buffering, it is up to the application to provide adequate buffering based on it's other functions to keep the audio stream from starving for data.

The USB host audio driver requires some initial configuration by the application that is outside of the USB audio driver's control. The first of these is to enable the uDMA controller and configure a DMA control table that includes the USB DMA channels. The application must also register the USB host audio driver by calling USBHCDRegisterDrivers() with the *g_USBHostAudioClassDriver* structure pointer in the list of supported drivers. Finally the application must create an instance of the USB host audio device by calling the USBHostAudioOpen() function and provide it with a callback for basic USB audio events, saving the value returned for use with other APIs.

Example: Initial USB Audio Setup

```
//
// The instance data for the USB host audio driver.
//
uint32_t g_psAudioInstance = 0;

//
// The control table used by the uDMA controller.  This table must be aligned
// to a 1024 byte boundary.  When using USB with uDMA if it is only used for
// USB then only first 6 channels are needed.
//
// Note: If other DMA channels are used then the table must be large enough
// to hold all channels in use.
//
tDMAControlTable g_psDMAControlTable[6];

//
// The global that holds all of the host drivers in use in the application.
// In this case, only the host audio class is loaded.
//
static tUSBHostClassDriver const * const g_ppsHostClassDrivers[] =
{
    &g_sUSBHostAudioClassDriver,
    &g_sUSBEventDriver
};

    ...

    //
    // Enable the uDMA controller and set up the control table base.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UDMA);
    uDMAEnable();
    uDMAControlBaseSet(g_psDMAControlTable);

    //
    // Register the host class drivers.
    //
    USBHCDRegisterDrivers(0, g_ppsHostClassDrivers, g_ui32NumHostClassDrivers);

    //
    // Open an instance of the mass storage class driver.
    //
    g_psAudioInstance = USBHostAudioOpen(0, AudioCallback);

    ...
```

Audio output is handled by setting the format of the audio stream and then by calling the USBHostAudioPlay() function to provide new buffers to the audio device. The callback function that is provided with this call returns the buffers when the driver is no longer using them. In order for audio output to start, the application must first set the audio format with a successful call to USBHostAudioFormatSet(). If the format was not supported by the audio device then this function returns a non-zero value and USBHostAudioPlay() should not be called until a valid format is selected. Once

a valid format is set the application should provide audio data to the host audio driver by calling USBHostAudioPlay() and then always waiting for the callback to indicate that the buffer has been released. Calling the USBHostAudioPlay() function before the previous buffer has been released can cause the previous transfer to be interrupted or canceled. Since the USB host audio driver provides limited buffering it is up to the application to have data ready for output. The application can safely call USBHostAudioPlay() function directly from the callback function to provide a new buffer to the USB audio device.

Example: Audio Output

```
void AudioOutCallback(void *pvBuffer, uint32_t ui32Param, uint32_t ui32Event)
{
    //
    // Check if this was a buffer free event and provide a new buffer to the
    // host audio driver.
    //
    if(ui32Event == USB_EVENT_TX_COMPLETE)
    {
        USBHostAudioPlay(psAudioInstance, pNewBuffer, ui32Size,
                        AudioOutCallback));
    }

}

void AudioPlay(void)
{
    //
    // Wait for USBH_AUDIO_EVENT_OPEN event.
    //

    ...

    //
    // Set the audio format to 48KHz 16 bit stereo output.
    //
    USBHostAudioFormatSet(psAudioInstance, 48000, 16, 2,
                        USBH_AUDIO_FORMAT_OUT);

    ...

    //
    // Start the output of the first buffer and let the callback start the
    // remaining buffers.
    //
    USBHostAudioPlay(psAudioInstance, pBuffer, ui32Size, AudioOutCallback));

    //
    // Handle filling returned buffers.
    //

    ...
}
```

Audio input is handled by setting the format of the audio stream and then by calling the USBHostAudioRecord() function to provide a new buffer to be filled by the host audio driver. The callback function that is provided with this call returns the buffer when the audio driver has new data available. In order for audio input to start, the application must first set the audio input format with a successful call to USBHostAudioFormatSet(). If the format was not supported by the audio device then this function returns a non-zero value. USBHostAudioRecord() should not be called until a valid format is selected. Once a valid format is set the application should provide an audio buffer to the host audio driver by calling USBHostAudioRecord() and wait for the callback to indicate that the buffer has been filled. Calling the USBHostAudioRecord() function before the previous buffer has been

filled can cause the previous input transfer to be interrupted or lost. Since the USB host audio driver provides limited buffering it is up to the application to handle the input buffers and provide new buffers. The application can safely call USBHostAudioRecord() function directly from the callback function to provide a new buffer to the USB audio device, however the same buffer should not be passed back until it has been processed or the host audio driver may overwrite the data.

Example: Audio Input

```
void AudioInCallback(tUSBHostAudioInstance *psAudioInstance, uint32_t ui32Event,
                     uint32_t ui32Param, void *pvMsgData)
{
    //
    // Check if this was a buffer full event and provide a new buffer to the
    // host audio driver.
    //
    if(ui32Event == USB_EVENT_RX_AVAILABLE)
    {
        USBHostAudioRecord(psAudioInstance, pNewBuffer, ui32Size,
                           AudioInCallback));
    }

}

void AudioRecord(void)
{
    //
    // Wait for USBH_AUDIO_EVENT_OPEN event.
    //

    ...

    //
    // Set the audio format to 48KHz 16 bit stereo output.
    //
    USBHostAudioFormatSet(psAudioInstance, 48000, 16, 2,
                          USBH_AUDIO_FORMAT_IN);

    ...

    //
    // Start the input of the first buffer and let the callback start the
    // remaining buffers.
    //
    USBHostAudioRecord(psAudioInstance, pBuffer, ui32Size, AudioInCallback));

    //
    // Handle filling returned buffers.
    //

    ...
}
```

3.4.6   Implementing Custom Host Class Drivers

This next section covers how to implement a custom host class driver and how the host controller driver finds the driver. All host class drivers must provide their own driver interface that is visible to the host controller driver. As with the host class drivers that are included with the USB library, this means exposing a driver interface of the type tUSBClassDriver. In the example below the USBGenericOpen() function is called when the host controller driver enumerates a device that matches the "USB_CLASS_SOMECLASS" interface class. The USBGenericClose() function is

called when the device of this class is removed. The following example shows a definition of a custom host class driver.

Example: Custom Host Class Driver Interface

```
tUSBClassDriver sUSBGenericClassDriver =
{
    USB_CLASS_SOMECLASS,
    USBGenericOpen,
    USBGenericClose,
    USBGenericIntHandler
};
```

The *uIInterfaceClass* member of the tUSBClassDriver structure is the class read from the device's interface descriptor during enumeration. This number is used to as the primary search value for a host class driver. If a device is connected that matches this structure member then that host class driver is loaded. The *pfnOpen* member of the tUSBClassDriver structure is called when a device with a matching interface class is detected. This function should do whatever is necessary to handle device detection and initial configuration of the device, this includes allocating any USB pipes that the device may need for communications. This requires parsing the endpoint descriptors for a device's endpoints and then allocating the USB pipes based on the types and number of endpoints discover. The host class drivers provided with the USB library demonstrate how to parse and allocate USB pipes. This call is not at made interrupt level so it can be interrupted by other USB events. Anything that must be done immediately before any other communications with the device should be done in the *pfnOpen* function. The *pfnOpen* member should should return a handle that is passed to the remaining functions *pfnClose* and *pfnIntHandler*. This handle should enable the host class driver to differentiate between different instances of the same type of device. The value returned can be any value as the USB library simply returns it unmodified to the other host class driver functions. The *pfnClose* structure member is called when the device that was created with *pfnOpen* call is removed from the system. All driver clean up should be done in the *pfnClose* call as no more calls are made to the host class driver. If the host class driver needs to respond to USB interrupts, an optional *pfnIntHandler* function pointer is provided. This function runs at interrupt time and called for any interrupt that occurs due to this device or for generic USB events. This function is not required and should only be implemented if it is necessary. It is completely up to the custom USB host class driver to determine it's own upper layer interface to applications or to other device interface layers. With the addition of hub support, the application interface layer should take into account multiple instances of a device class if multiple instances of devices are supported.

3.5   **Host Class Driver Definitions**

Defines

- USBH_AUDIO_EVENT_CLOSE
- USBH_AUDIO_EVENT_OPEN
- USBH_EVENT_HID_KB_MOD
- USBH_EVENT_HID_KB_PRESS
- USBH_EVENT_HID_KB_REL
- USBH_EVENT_HID_MS_PRESS
- USBH_EVENT_HID_MS_REL
- USBH_EVENT_HID_MS_X
- USBH_EVENT_HID_MS_Y

Enumerations

- tHIDSubClassProtocol

Functions

- void USBHHIDClose (tHIDInstance *psHIDInstance)
- uint32_t USBHHIDGetReport (tHIDInstance *psHIDInstance, uint32_t ui32Interface, uint8_t *pui8Data, uint32_t ui32Size)
- uint32_t USBHHIDGetReportDescriptor (tHIDInstance *psHIDInstance, uint8_t *pui8Buffer, uint32_t ui32Size)
- uint32_t USBHHIDLPMSleep (tHIDInstance *psHIDInstance)
- uint32_t USBHHIDLPMStatus (tHIDInstance *psHIDInstance)
- tHIDInstance * USBHHIDOpen (tHIDSubClassProtocol iDeviceType, tUSBCallback pfnCallback, void *pvCBData)
- uint32_t USBHHIDSetIdle (tHIDInstance *psHIDInstance, uint8_t ui8Duration, uint8_t ui8ReportID)
- uint32_t USBHHIDSetProtocol (tHIDInstance *psHIDInstance, uint32_t ui32BootProtocol)
- uint32_t USBHHIDSetReport (tHIDInstance *psHIDInstance, uint32_t ui32Interface, uint8_t *pui8Data, uint32_t ui32Size)
- void USBHHubClose (tHubInstance *psHubInstance)
- void USBHHubEnumerationComplete (uint8_t ui8Hub, uint8_t ui8Port)
- void USBHHubEnumerationError (uint8_t ui8Hub, uint8_t ui8Port)
- uint32_t USBHHubLPMSleep (tHubInstance *psHubInstance)
- uint32_t USBHHubLPMStatus (tHubInstance *psHubInstance)
- tHubInstance * USBHHubOpen (tUSBHHubCallback pfnCallback)
- int32_t USBHMSCBlockRead (tUSBHMSCInstance *psMSCInstance, uint32_t ui32LBA, uint8_t *pui8Data, uint32_t ui32NumBlocks)
- int32_t USBHMSCBlockWrite (tUSBHMSCInstance *psMSCInstance, uint32_t ui32LBA, uint8_t *pui8Data, uint32_t ui32NumBlocks)
- void USBHMSCDriveClose (tUSBHMSCInstance *psMSCInstance)
- tUSBHMSCInstance * USBHMSCDriveOpen (uint32_t ui32Drive, tUSBHMSCCallback pfnCallback)
- int32_t USBHMSCDriveReady (tUSBHMSCInstance *psMSCInstance)
- uint32_t USBHMSCLPMSleep (tUSBHMSCInstance *psMSCInstance)
- uint32_t USBHMSCLPMStatus (tUSBHMSCInstance *psMSCInstance)
- void USBHostAudioClose (tUSBHostAudioInstance *psAudioInstance)
- uint32_t USBHostAudioFormatGet (tUSBHostAudioInstance *psAudioInstance, uint32_t ui32SampleRate, uint32_t ui32Bits, uint32_t ui32Channels, uint32_t ui32Flags)
- uint32_t USBHostAudioFormatSet (tUSBHostAudioInstance *psAudioInstance, uint32_t ui32SampleRate, uint32_t ui32Bits, uint32_t ui32Channels, uint32_t ui32Flags)
- uint32_t USBHostAudioLPMSleep (tUSBHostAudioInstance *psAudioInstance)
- uint32_t USBHostAudioLPMStatus (tUSBHostAudioInstance *psAudioInstance)
- tUSBHostAudioInstance * USBHostAudioOpen (uint32_t ui32Index, tUSBHostAudioCallback pfnCallback)
- int32_t USBHostAudioPlay (tUSBHostAudioInstance *psAudioInstance, void *pvBuffer, uint32_t ui32Size, tUSBHostAudioCallback pfnCallback)

- int32_t USBHostAudioRecord (tUSBHostAudioInstance ∗psAudioInstance, void ∗pvBuffer, uint32_t ui32Size, tUSBHostAudioCallback pfnCallback)
- uint32_t USBHostAudioVolumeGet (tUSBHostAudioInstance ∗psAudioInstance, uint32_t ui32Interface, uint32_t ui32Channel)
- uint32_t USBHostAudioVolumeMaxGet (tUSBHostAudioInstance ∗psAudioInstance, uint32_t ui32Interface, uint32_t ui32Channel)
- uint32_t USBHostAudioVolumeMinGet (tUSBHostAudioInstance ∗psAudioInstance, uint32_t ui32Interface, uint32_t ui32Channel)
- uint32_t USBHostAudioVolumeResGet (tUSBHostAudioInstance ∗psAudioInstance, uint32_t ui32Interface, uint32_t ui32Channel)
- void USBHostAudioVolumeSet (tUSBHostAudioInstance ∗psAudioInstance, uint32_t ui32Interface, uint32_t ui32Channel, uint32_t ui32Value)
- uint32_t USBHSCSIInquiry (uint32_t ui32InPipe, uint32_t ui32OutPipe, uint8_t ∗pui8Data, uint32_t ∗pui32Size)
- uint32_t USBHSCSIModeSense6 (uint32_t ui32InPipe, uint32_t ui32OutPipe, uint32_t ui32Flags, uint8_t ∗pui8Data, uint32_t ∗pui32Size)
- uint32_t USBHSCSIRead10 (uint32_t ui32InPipe, uint32_t ui32OutPipe, uint32_t ui32LBA, uint8_t ∗pui8Data, uint32_t ∗pui32Size, uint32_t ui32NumBlocks)
- uint32_t USBHSCSIReadCapacities (uint32_t ui32InPipe, uint32_t ui32OutPipe, uint8_t ∗pui8Data, uint32_t ∗pui32Size)
- uint32_t USBHSCSIReadCapacity (uint32_t ui32InPipe, uint32_t ui32OutPipe, uint8_t ∗pui8Data, uint32_t ∗pui32Size)
- uint32_t USBHSCSIRequestSense (uint32_t ui32InPipe, uint32_t ui32OutPipe, uint8_t ∗pui8Data, uint32_t ∗pui32Size)
- uint32_t USBHSCSITestUnitReady (uint32_t ui32InPipe, uint32_t ui32OutPipe)
- uint32_t USBHSCSIWrite10 (uint32_t ui32InPipe, uint32_t ui32OutPipe, uint32_t ui32LBA, uint8_t ∗pui8Data, uint32_t ∗pui32Size, uint32_t ui32NumBlocks)

Variables

- const tUSBHostClassDriver g_sUSBHIDClassDriver
- const tUSBHostClassDriver g_sUSBHostAudioClassDriver
- const tUSBHostClassDriver g_sUSBHostMSCClassDriver
- const tUSBHostClassDriver g_sUSBHubClassDriver

3.5.1 Detailed Description

The macros and functions defined in this section can be found in header files `host/usbhhid.h`, `host/usbhmsc.h` and `host/usbhscsi.h`.

3.5.2 Define Documentation

3.5.2.1 USBH_AUDIO_EVENT_CLOSE

Definition:
```
#define USBH_AUDIO_EVENT_CLOSE
```

Description:

This USB host audio event indicates that the previously connected device has been disconnected. The *pvBuffer* and *ui32Param* values are not used in this event.

3.5.2.2 USBH_AUDIO_EVENT_OPEN

Definition:

```
#define USBH_AUDIO_EVENT_OPEN
```

Description:

This USB host audio event indicates that the device is connected and ready to send or receive buffers. The *pvBuffer* and *ui32Param* values are not used in this event.

3.5.2.3 USBH_EVENT_HID_KB_MOD

Definition:

```
#define USBH_EVENT_HID_KB_MOD
```

Description:

The HID keyboard detected one of the keyboard modifiers being pressed.

3.5.2.4 USBH_EVENT_HID_KB_PRESS

Definition:

```
#define USBH_EVENT_HID_KB_PRESS
```

Description:

The HID keyboard detected a key being pressed.

3.5.2.5 USBH_EVENT_HID_KB_REL

Definition:

```
#define USBH_EVENT_HID_KB_REL
```

Description:

The HID keyboard detected a key being released.

3.5.2.6 USBH_EVENT_HID_MS_PRESS

Definition:

```
#define USBH_EVENT_HID_MS_PRESS
```

Description:

A button was pressed on a HID mouse.

3.5.2.7 USBH_EVENT_HID_MS_REL

Definition:
```
#define USBH_EVENT_HID_MS_REL
```

Description:
A button was released on a HID mouse.

3.5.2.8 USBH_EVENT_HID_MS_X

Definition:
```
#define USBH_EVENT_HID_MS_X
```

Description:
The HID mouse detected movement in the X direction.

3.5.2.9 USBH_EVENT_HID_MS_Y

Definition:
```
#define USBH_EVENT_HID_MS_Y
```

Description:
The HID mouse detected movement in the Y direction.

3.5.3 Enumeration Documentation

3.5.3.1 tHIDSubClassProtocol

Description:
The following values are used to register callbacks to the USB HOST HID device class layer.

Enumerators:
eUSBHHIDClassNone No device should be used. This value should not be used by applications.
eUSBHHIDClassKeyboard This is a keyboard device.
eUSBHHIDClassMouse This is a mouse device.
eUSBHHIDClassVendor This is a vendor specific device.

3.5.4 Function Documentation

3.5.4.1 USBHHIDClose

This function is used to release an instance of a HID device.

Prototype:
```
void
USBHHIDClose(tHIDInstance *psHIDInstance)
```

Parameters:
>   ***psHIDInstance*** is the instance value for a HID device to release.

Description:
>   This function releases an instance of a HID device that was created by a call to USBHHI-DOpen(). This call is required to allow other HID devices to be enumerated after another HID device has been disconnected. The *psHIDInstance* parameter should hold the value that was returned from the previous call to USBHHIDOpen().

Returns:
>   None.

3.5.4.2   USBHHIDGetReport

This function is used to retrieve a report from a HID device.

Prototype:
```
uint32_t
USBHHIDGetReport(tHIDInstance *psHIDInstance,
                 uint32_t ui32Interface,
                 uint8_t *pui8Data,
                 uint32_t ui32Size)
```

Parameters:
>   ***psHIDInstance*** is the value that was returned from the call to USBHHIDOpen().
>
>   ***ui32Interface*** is the interface to retrieve the report from.
>
>   ***pui8Data*** is the memory buffer to use to store the report.
>
>   ***ui32Size*** is the size in bytes of the buffer pointed to by *pui8Buffer*.

Description:
>   This function is used to retrieve a report from a USB pipe. It is usually called when the USB HID layer has detected a new data available in a USB pipe. The USB HID host device code will receive a **USB_EVENT_RX_AVAILABLE** event when data is available, allowing the callback function to retrieve the data.

Returns:
>   Returns the number of bytes read from report.

3.5.4.3   USBHHIDGetReportDescriptor

This function can be used to retrieve the report descriptor for a given device instance.

Prototype:
```
uint32_t
USBHHIDGetReportDescriptor(tHIDInstance *psHIDInstance,
                           uint8_t *pui8Buffer,
                           uint32_t ui32Size)
```

Parameters:
>   ***psHIDInstance*** is the value that was returned from the call to USBHHIDOpen().

pui8Buffer is the memory buffer to use to store the report descriptor.

ui32Size is the size in bytes of the buffer pointed to by *pui8Buffer*.

Description:

This function is used to return a report descriptor from a HID device instance so that it can determine how to interpret reports that are returned from the device indicated by the *psHIDInstance* parameter. This call is blocking and will return the number of bytes read into the *pui8Buffer*.

Returns:

Returns the number of bytes read into the *pui8Buffer*.

3.5.4.4 USBHHIDLPMSleep

This function forwards an LPM request for a device to enter L1 sleep state.

Prototype:

```
uint32_t
USBHHIDLPMSleep(tHIDInstance *psHIDInstance)
```

Parameters:

psHIDInstance is the HID instance that was returned from the call to USBHHIDOpen().

Description:

This function forwards a request from a HID device class to the host controller to request that a device enter the LPM L1 sleep state. The caller must check the return value to see if the request can be attempted at this time. If another LPM transaction is busy on this or another device, then this function returns **USBHCD_LPM_PENDING**. If the LPM request was scheduled to be sent the function returns **USBHCD_LPM_AVAIL**. The caller should check the USBHCDLPMStatus() function to determine if the request completed successfully or if there was an error.

Returns:

This function returns the following values:

- USBHCD_LPM_AVAIL - The transition to L1 state is scheduled to be sent.
- USBHCD_LPM_PENDING - There is already an LPM request pending.

3.5.4.5 USBHHIDLPMStatus

This function returns the current status of an LPM request.

Prototype:

```
uint32_t
USBHHIDLPMStatus(tHIDInstance *psHIDInstance)
```

Parameters:

psHIDInstance is the HID instance that was returned from the call to USBHHIDOpen().

Description:

This function returns the current status of LPM requests for a given device. This is called to determine if a previous request completed successfully or if there was an error.

Returns:
This function returns the following values:

- **USBHCD_LPM_AVAIL** - There are no pending LPM requests on this specific device or the last request completed successfully.
- **USBHCD_LPM_ERROR** - The last LPM request for this device did not complete successfully.
- **USBHCD_LPM_PENDING** - The last LPM request has not completed.

3.5.4.6 USBHHIDOpen

This function is used to open an instance of a HID device.

Prototype:
```
tHIDInstance *
USBHHIDOpen(tHIDSubClassProtocol iDeviceType,
            tUSBCallback pfnCallback,
            void *pvCBData)
```

Parameters:
iDeviceType is the type of device that should be loaded for this instance of the HID device.
pfnCallback is the function that will be called whenever changes are detected for this device.
pvCBData is the data that will be returned in when the *pfnCallback* function is called.

Description:
This function creates an instance of an specific type of HID device. The *iDeviceType* parameter is one subclass/protocol values of the types specified in enumerated types tHIDSubClassProtocol. Only devices that enumerate with this type will be called back via the *pfnCallback* function. The *pfnCallback* parameter is the callback function for any events that occur for this device type. The *pfnCallback* function must point to a valid function of type *tUSBCallback* for this call to complete successfully. To release this device instance the caller of USBHHIDOpen() should call USBHHIDClose() and pass in the value returned from the USBHHIDOpen() call.

Returns:
This function returns and instance value that should be used with any other APIs that require an instance value. If a value of 0 is returned then the device instance could not be created.

3.5.4.7 USBHHIDSetIdle

This function is used to set the idle timeout for a HID device.

Prototype:
```
uint32_t
USBHHIDSetIdle(tHIDInstance *psHIDInstance,
               uint8_t ui8Duration,
               uint8_t ui8ReportID)
```

Parameters:
psHIDInstance is the value that was returned from the call to USBHHIDOpen().
ui8Duration is the duration of the timeout in milliseconds.

ui8ReportID is the report identifier to set the timeout on.

Description:
This function will send the Set Idle command to a HID device to set the idle timeout for a given report. The length of the timeout is specified by the *ui8Duration* parameter and the report the timeout for is in the *ui8ReportID* value.

Returns:
Always returns 0.

3.5.4.8 USBHHIDSetProtocol

This function is used to set or clear the boot protocol state of a device.

Prototype:
```
uint32_t
USBHHIDSetProtocol(tHIDInstance *psHIDInstance,
                   uint32_t ui32BootProtocol)
```

Parameters:
psHIDInstance is the value that was returned from the call to USBHHIDOpen().
ui32BootProtocol is either zero or non-zero to indicate which protocol to use for the device.

Description:
A USB host device can use this function to set the protocol for a connected HID device. This is commonly used to set keyboards and mice into their simplified boot protocol modes to fix the report structure to a know state.

Returns:
This function returns 0.

3.5.4.9 USBHHIDSetReport

This function is used to send a report to a HID device.

Prototype:
```
uint32_t
USBHHIDSetReport(tHIDInstance *psHIDInstance,
                 uint32_t ui32Interface,
                 uint8_t *pui8Data,
                 uint32_t ui32Size)
```

Parameters:
psHIDInstance is the value that was returned from the call to USBHHIDOpen().
ui32Interface is the interface to send the report to.
pui8Data is the memory buffer to use to store the report.
ui32Size is the size in bytes of the buffer pointed to by *pui8Buffer*.

Description:

This function is used to send a report to a USB HID device. It can be only be called from outside the callback context as this function will not return from the call until the data has been sent successfully.

Returns:

Returns the number of bytes sent to the device.

3.5.4.10 USBHHubClose

This function is used to release a hub device instance.

Prototype:
```
void
USBHHubClose(tHubInstance *psHubInstance)
```

Parameters:

psHubInstance is the hub device instance that is to be released.

Description:

This function is called when an instance of the hub device must be released. This function is typically made in preparation for shutdown or a switch to function as a USB device when in OTG mode. Following this call, the hub device is no longer available, but it can be opened again using a call to USBHHubOpen(). After calling USBHHubClose(), the host hub driver no longer provides any callbacks or accepts calls to other hub driver APIs.

Returns:

None.

3.5.4.11 USBHHubEnumerationComplete

Informs the hub class driver that a downstream device has been enumerated.

Prototype:
```
void
USBHHubEnumerationComplete(uint8_t ui8Hub,
                           uint8_t ui8Port)
```

Parameters:

ui8Hub is the address of the hub to which the downstream device is attached.

ui8Port is the port on the hub to which the downstream device is attached.

Description:

This function is called by the host controller driver to inform the hub class driver that a downstream device has been enumerated successfully. The hub driver then moves on and continues enumeration of any other newly connected devices.

Returns:

None.

3.5.4.12 USBHHubEnumerationError

Informs the hub class driver that a downstream device failed to enumerate.

Prototype:
```
void
USBHHubEnumerationError(uint8_t ui8Hub,
                        uint8_t ui8Port)
```

Parameters:
>   ***ui8Hub*** is the address of the hub to which the downstream device is attached.
>   ***ui8Port*** is the port on the hub to which the downstream device is attached.

Description:
>   This function is called by the host controller driver to inform the hub class driver that an attempt
>   to enumerate a downstream device has failed. The hub driver then cleans up and continues
>   enumeration of any other newly connected devices.

Returns:
>   None.

3.5.4.13 USBHHubLPMSleep

This function forwards an LPM request for a device to enter L1 sleep state.

Prototype:
```
uint32_t
USBHHubLPMSleep(tHubInstance *psHubInstance)
```

Parameters:
>   ***psHubInstance*** is the hub device instance that was returned from the call to USBH-
>   HubOpen().

Description:
>   This function forwards a request from an application to the hub device class to request that a
>   device enter the LPM L1 sleep state. The caller must check the return value to see if the request
>   can be attempted at this time. If another LPM transaction is busy on this or another device, then
>   this function returns **USBHCD_LPM_PENDING**. If the LPM request was scheduled to be sent
>   the function returns **USBHCD_LPM_AVAIL**. The caller should check the USBHCDLPMStatus()
>   function to determine if the request completed successfully or if there was an error.

Returns:
>   This function returns the following values:
>
>   - **USBHCD_LPM_AVAIL** - The transition to L1 state is scheduled to be sent.
>   - **USBHCD_LPM_PENDING** - There is already an LPM request pending.

3.5.4.14 USBHHubLPMStatus

This function returns the current status of an LPM request.

Prototype:
```
uint32_t
USBHHubLPMStatus(tHubInstance *psHubInstance)
```

Parameters:
> ***psHubInstance*** is the hub device instance that was returned from the call to USBH-HubOpen().

Description:
> This function returns the current status of LPM requests for a given device. This is called to determine if a previous request completed successfully or if there was an error.

Returns:
> This function returns the following values:
>
> - **USBHCD_LPM_AVAIL** - There are no pending LPM requests on this specific device or the last request completed successfully.
> - **USBHCD_LPM_ERROR** - The last LPM request for this device did not complete successfully.
> - **USBHCD_LPM_PENDING** - The last LPM request has not completed.

3.5.4.15 USBHHubOpen

This function is used to enable the host hub class driver before any devices are present.

Prototype:
```
tHubInstance *
USBHHubOpen(tUSBHHubCallback pfnCallback)
```

Parameters:
> ***pfnCallback*** is the driver call back for host hub events.

Description:
> This function is called to open an instance of a host hub device and provides a valid callback function for host hub events in the *pfnCallback* parameter. This function must be called before the USB host code can successfully enumerate a hub device or any devices attached to the hub. The *pui8HubPool* is memory provided to the hub class to manage the devices that are connected to the hub. The *ui32PoolSize* is the number of bytes and should be at least 32 bytes per device including the hub device itself. A simple formula for providing memory to the hub class is **MAX_USB_DEVICES** $*$ 32 bytes of data to allow for proper enumeration of connected devices. The value for **MAX_USB_DEVICES** is defined in the usblib.h file and controls the number of devices supported by the USB library. The *ui32NumHubs* parameter defaults to one and only one buffer of size tHubInstance is required to be passed in the *psHubInstance* parameter.

Note:
> Changing the value of **MAX_USB_DEVICES** requires a rebuild of the USB library to have an effect on the library.

Returns:
> This function returns the driver instance to use for the other host hub functions. If there is no instance available at the time of this call, this function returns zero.

3.5.4.16 USBHMSCBlockRead

This function performs a block read to an MSC device.

Prototype:
```
int32_t
USBHMSCBlockRead(tUSBHMSCInstance *psMSCInstance,
                 uint32_t ui32LBA,
                 uint8_t *pui8Data,
                 uint32_t ui32NumBlocks)
```

Parameters:
>*psMSCInstance* is the device instance to use for this read.
>*ui32LBA* is the logical block address to read on the device.
>*pui8Data* is a pointer to the returned data buffer.
>*ui32NumBlocks* is the number of blocks to read from the device.

Description:
>This function will perform a block sized read from the device associated with the *psMSCInstance* parameter. The *ui32LBA* parameter specifies the logical block address to read on the device. This function will only perform *ui32NumBlocks* block sized reads. In most cases this is a read of 512 bytes of data. The *∗pui8Data* buffer should be at least *ui32NumBlocks ∗ 512* bytes in size.

Returns:
>The function returns zero for success and any negative value indicates a failure.

3.5.4.17 USBHMSCBlockWrite

This function performs a block write to an MSC device.

Prototype:
```
int32_t
USBHMSCBlockWrite(tUSBHMSCInstance *psMSCInstance,
                  uint32_t ui32LBA,
                  uint8_t *pui8Data,
                  uint32_t ui32NumBlocks)
```

Parameters:
>*psMSCInstance* is the device instance to use for this write.
>*ui32LBA* is the logical block address to write on the device.
>*pui8Data* is a pointer to the data to write out.
>*ui32NumBlocks* is the number of blocks to write to the device.

Description:
>This function will perform a block sized write to the device associated with the *psMSCInstance* parameter. The *ui32LBA* parameter specifies the logical block address to write on the device. This function will only perform *ui32NumBlocks* block sized writes. In most cases this is a write of 512 bytes of data. The *∗pui8Data* buffer should contain at least *ui32NumBlocks ∗ 512* bytes in size to prevent unwanted data being written to the device.

Returns:
>The function returns zero for success and any negative value indicates a failure.

3.5.4.18 USBHMSCDriveClose

This function should be called to release a drive instance.

Prototype:
```
void
USBHMSCDriveClose(tUSBHMSCInstance *psMSCInstance)
```

Parameters:
> ***psMSCInstance*** is the device instance that is to be released.

Description:
> This function is called when an MSC drive is to be released in preparation for shutdown or a switch to USB device mode, for example. Following this call, the drive is available for other clients who may open it again using a call to USBHMSCDriveOpen().

Returns:
> None.

3.5.4.19 USBHMSCDriveOpen

This function should be called before any devices are present to enable the mass storage device class driver.

Prototype:
```
tUSBHMSCInstance *
USBHMSCDriveOpen(uint32_t ui32Drive,
                 tUSBHMSCCallback pfnCallback)
```

Parameters:
> ***ui32Drive*** is the drive number to open.
> ***pfnCallback*** is the driver callback for any mass storage events.

Description:
> This function is called to open an instance of a mass storage device. It should be called before any devices are connected to allow for proper notification of drive connection and disconnection. The *ui32Drive* parameter is a zero based index of the drives present in the system. There are a constant number of drives, and this number should only be greater than 0 if there is a USB hub present in the system. The application should also provide the *pfnCallback* to be notified of mass storage related events like device enumeration and device removal.

Returns:
> This function will return the driver instance to use for the other mass storage functions. If there is no driver available at the time of this call, this function will return zero.

3.5.4.20 USBHMSCDriveReady

This function checks if a drive is ready to be accessed.

Prototype:
```
int32_t
USBHMSCDriveReady(tUSBHMSCInstance *psMSCInstance)
```

Parameters:
> ***psMSCInstance*** is the device instance to use for this read.

Description:
> This function checks if the current device is ready to be accessed. It uses the *psMSCInstance* parameter to determine which device to check and returns zero when the device is ready. Any non-zero return code indicates that the device was not ready.

Returns:
> This function returns zero if the device is ready and it returns a other value if the device is not ready or if an error occurred.

3.5.4.21 USBHMSCLPMSleep

This function forwards an LPM request for a device to enter L1 sleep state.

Prototype:
```
uint32_t
USBHMSCLPMSleep(tUSBHMSCInstance *psMSCInstance)
```

Parameters:
> ***psMSCInstance*** is the host mass storage class instance that was returned from the call to USBHMSCDriveOpen().

Description:
> This function forwards a request from an application to the mass storage device class to request that a device enter the LPM L1 sleep state. The caller must check the return value to see if the request can be attempted at this time. If another LPM transaction is busy on this or another device, then this function returns **USBHCD_LPM_PENDING**. If the LPM request was scheduled to be sent the function returns **USBHCD_LPM_AVAIL**. The caller should check the USBHCDLPMStatus() function to determine if the request completed successfully or if there was an error.

Returns:
> This function returns the following values:
>
> - USBHCD_LPM_AVAIL - The transition to L1 state is scheduled to be sent.
> - USBHCD_LPM_PENDING - There is already an LPM request pending.

3.5.4.22 USBHMSCLPMStatus

This function returns the current status of an LPM request.

Prototype:
```
uint32_t
USBHMSCLPMStatus(tUSBHMSCInstance *psMSCInstance)
```

Parameters:
> ***psMSCInstance*** is the host mass storage class instance that was returned from the call to USBHMSCDriveOpen().

Description:
> This function returns the current status of LPM requests for a given device. This is called to determine if a previous request completed successfully or if there was an error.

Returns:
> This function returns the following values:
>
> - **USBHCD_LPM_AVAIL** - There are no pending LPM requests on this specific device or the last request completed successfully.
> - **USBHCD_LPM_ERROR** - The last LPM request for this device did not complete successfully.
> - **USBHCD_LPM_PENDING** - The last LPM request has not completed.

3.5.4.23 USBHostAudioClose

This function should be called to release an audio device instance.

Prototype:
```
void
USBHostAudioClose(tUSBHostAudioInstance *psAudioInstance)
```

Parameters:
> ***psAudioInstance*** is the device instance that is to be released.

Description:
> This function is called when a host audio device needs to be released. This could be in preparation for shutdown or a switch to USB device mode, for example. Following this call, the audio device is available and can be opened again using a call to USBHostAudioOpen(). After calling this function, the host audio driver will no longer provide any callbacks or accept calls to other audio driver APIs.

Returns:
> None.

3.5.4.24 USBHostAudioFormatGet

This function is called to determine if an audio format is supported by the connected USB Audio device.

Prototype:
```
uint32_t
USBHostAudioFormatGet(tUSBHostAudioInstance *psAudioInstance,
                      uint32_t ui32SampleRate,
                      uint32_t ui32Bits,
                      uint32_t ui32Channels,
                      uint32_t ui32Flags)
```

Parameters:
> ***psAudioInstance*** is the device instance for this call.
> ***ui32SampleRate*** is the sample rate of the audio stream.
> ***ui32Bits*** is the number of bits per sample in the audio stream.

ui32Channels is the number of channels in the audio stream.

ui32Flags is a set of flags to determine what type of interface to retrieve.

Description:

This function is called when an application needs to determine which audio formats are supported by a USB audio device that has been connected. The *psAudioInstance* value that is used with this call is the value that was returned from the USBHostAudioOpen() function. This call checks the USB audio device to determine if it can support the values provided in the *ui32SampleRate*, *ui32Bits*, and *ui32Channels* values. The *ui32Flags* currently only supports either the **USBH_AUDIO_FORMAT_IN** or **USBH_AUDIO_FORMAT_OUT** values that indicates if a request is for an audio input and an audio output. If the format is supported this function returns zero, and this function returns a non-zero value if the format is not supported. This function does not set the current output or input format.

Returns:

A value of zero indicates the supplied format is supported and a non-zero value indicates that the format is not supported.

3.5.4.25 USBHostAudioFormatSet

This function is called to set the current sample rate on an audio interface.

Prototype:

```
uint32_t
USBHostAudioFormatSet(tUSBHostAudioInstance *psAudioInstance,
                      uint32_t ui32SampleRate,
                      uint32_t ui32Bits,
                      uint32_t ui32Channels,
                      uint32_t ui32Flags)
```

Parameters:

psAudioInstance specifies the device instance for this call.

ui32SampleRate is the sample rate in Hz.

ui32Bits is the number of bits per sample.

ui32Channels is then number of audio channels.

ui32Flags is a set of flags that determine the access type.

Description:

This function is called when to set the current audio output or input format for a USB audio device. The *psAudioInstance* value that is used with this call is the value that was returned from the USBHostAudioOpen() function. The application can use this call to insure that the audio format is supported and set the format at the same time. If the application is just checking for supported rates, then it should call the USBHostAudioFormatGet().

Note:

This function must be called before attempting to send or receive audio with the USBHostAudioPlay() or USBHostAudioRecord() functions.

Returns:

A non-zero value indicates the supplied format is not supported and a zero value indicates that the format was supported and has been configured.

3.5.4.26 USBHostAudioLPMSleep

This function forwards an LPM request for a device to enter L1 sleep state.

Prototype:
```
uint32_t
USBHostAudioLPMSleep(tUSBHostAudioInstance *psAudioInstance)
```

Parameters:
    psAudioInstance is the audio device instance that was returned from the call to USBHostAudioOpen().

Description:
    This function forwards a request from an application to the audio device class to request that a device enter the LPM L1 sleep state. The caller must check the return value to see if the request can be attempted at this time. If another LPM transaction is busy on this or another device, then this function returns **USBHCD_LPM_PENDING**. If the LPM request was scheduled to be sent the function returns **USBHCD_LPM_AVAIL**. The caller should check the USBHCDLPMStatus() function to determine if the request completed successfully or if there was an error.

Returns:
    This function returns the following values:

- **USBHCD_LPM_AVAIL** - The transition to L1 state is scheduled to be sent.
- **USBHCD_LPM_PENDING** - There is already an LPM request pending.

3.5.4.27 USBHostAudioLPMStatus

This function returns the current status of an LPM request.

Prototype:
```
uint32_t
USBHostAudioLPMStatus(tUSBHostAudioInstance *psAudioInstance)
```

Parameters:
    psAudioInstance is the audio device instance that was returned from the call to USBHostAudioOpen().

Description:
    This function returns the current status of LPM requests for a given device. This is called to determine if a previous request completed successfully or if there was an error.

Returns:
    This function returns the following values:

- **USBHCD_LPM_AVAIL** - There are no pending LPM requests on this specific device or the last request completed successfully.
- **USBHCD_LPM_ERROR** - The last LPM request for this device did not complete successfully.
- **USBHCD_LPM_PENDING** - The last LPM request has not completed.

3.5.4.28 USBHostAudioOpen

This function should be called before any devices are present to enable the host audio class driver.

Prototype:
```
tUSBHostAudioInstance *
USBHostAudioOpen(uint32_t ui32Index,
                 tUSBHostAudioCallback pfnCallback)
```

Parameters:
    ui32Index is the audio device to open (currently only 0 is supported).
    pfnCallback is the driver call back for host audio events.

Description:
    This function is called to open an instance of a host audio device and should provide a valid callback function for host audio events in the *pfnCallback* parameter. This function must be called before the USB host code can successfully enumerate an audio device.

Returns:
    This function returns the driver instance to use for the other host audio functions. If there is no instance available at the time of this call, this function returns zero.

3.5.4.29 USBHostAudioPlay

This function is called to send an audio buffer to the USB audio device.

Prototype:
```
int32_t
USBHostAudioPlay(tUSBHostAudioInstance *psAudioInstance,
                 void *pvBuffer,
                 uint32_t ui32Size,
                 tUSBHostAudioCallback pfnCallback)
```

Parameters:
    psAudioInstance specifies the device instance for this call.
    pvBuffer is the audio buffer to send.
    ui32Size is the size of the buffer in bytes.
    pfnCallback is a pointer to a callback function that is called when the buffer can be used again.

Description:
    This function is called when an application needs to schedule a new buffer for output to the USB audio device. Since this call schedules the transfer and returns immediately, the application should provide a *pfnCallback* function to be notified when the buffer can be used again by the application. The *pfnCallback* function provided is called with the *pvBuffer* parameter set to the *pvBuffer* provided by this call, the *ui32Param* can be ignored and the *ui32Event* parameter is **USB_EVENT_TX_COMPLETE**.

Returns:
    This function returns the number of bytes that were scheduled to be sent. If this function returns zero then there was no USB audio device present or the request could not be satisfied at this time.

3.5.4.30 USBHostAudioRecord

This function is called to provide an audio buffer to the USB audio device for audio input.

Prototype:
```
int32_t
USBHostAudioRecord(tUSBHostAudioInstance *psAudioInstance,
                   void *pvBuffer,
                   uint32_t ui32Size,
                   tUSBHostAudioCallback pfnCallback)
```

Parameters:
> **psAudioInstance** specifies the device instance for this call.
> **pvBuffer** is the audio buffer to send.
> **ui32Size** is the size of the buffer in bytes.
> **pfnCallback** is a pointer to a callback function that is called when the buffer has been filled.

Description:
> This function is called when an application needs to schedule a new buffer for input from the USB audio device. Since this call schedules the transfer and returns immediately, the application should provide a *pfnCallback* function to be notified when the buffer has been filled with audio data. When the *pfnCallback* function is called, the *pvBuffer* parameter is set to *pvBuffer* provided in this call, the *ui32Param* is the number of valid bytes in the pvBuffer and the *ui32Event* is set to **USB_EVENT_RX_AVAILABLE**.

Returns:
> This function returns the number of bytes that were scheduled to be sent. If this function returns zero then there was no USB audio device present or the device does not support audio input.

3.5.4.31 USBHostAudioVolumeGet

This function is used to get the current volume setting for a given audio device.

Prototype:
```
uint32_t
USBHostAudioVolumeGet(tUSBHostAudioInstance *psAudioInstance,
                      uint32_t ui32Interface,
                      uint32_t ui32Channel)
```

Parameters:
> **psAudioInstance** is an instance of the USB audio device.
> **ui32Interface** is the interface number to use to query the current volume setting.
> **ui32Channel** is the 0 based channel number to query.

Description:
> The function is used to retrieve the current volume setting for an audio device on the channel specified by *ui32Channel*. The *ui32Interface* is ignored for now and should be set to 0 to access the default audio control interface. The *ui32Channel* value starts with 0 which is the master audio volume control interface. The remaining *ui32Channel* values provide access to various other audio channels, with 1 and 2 being left and right audio channels.

Note:
On devices that do not support volume control interfaces, this call returns 0, indicating a 0db setting.

Returns:
Returns the current volume setting for the requested interface.

3.5.4.32 USBHostAudioVolumeMaxGet

This function is used to get the maximum volume setting for a given audio device.

Prototype:
```
uint32_t
USBHostAudioVolumeMaxGet(tUSBHostAudioInstance *psAudioInstance,
                         uint32_t ui32Interface,
                         uint32_t ui32Channel)
```

Parameters:
psAudioInstance is an instance of the USB audio device.
ui32Interface is the interface number to use to query the maximum volume control value.
ui32Channel is the 0 based channel number to query.

Description:
The function is used to retrieve the maximum volume setting for an audio device on the channel specified by *ui32Channel*. The *ui32Interface* is ignored for now and should be set to 0 to access the default audio control interface. The *ui32Channel* value starts with 0 which is the master audio volume control interface. The remaining *ui32Channel* values provide access to various other audio channels, with 1 and 2 being left and right audio channels.

Note:
On devices that do not support volume control interfaces, this call returns 0, indicating a 0db setting.

Returns:
Returns the maximum volume setting for the requested interface.

3.5.4.33 USBHostAudioVolumeMinGet

This function is used to get the minimum volume setting for a given audio device.

Prototype:
```
uint32_t
USBHostAudioVolumeMinGet(tUSBHostAudioInstance *psAudioInstance,
                         uint32_t ui32Interface,
                         uint32_t ui32Channel)
```

Parameters:
psAudioInstance is an instance of the USB audio device.
ui32Interface is the interface number to use to query the minimum volume control value.
ui32Channel is the 0 based channel number to query.

Description:
The function is used to retrieve the minimum volume setting for an audio device on the channel specified by *ui32Channel*. The *ui32Interface* is ignored for now and should be set to 0 to access the default audio control interface. The *ui32Channel* value starts with 0 which is the master audio volume control interface. The remaining *ui32Channel* values provide access to various other audio channels, with 1 and 2 being left and right audio channels.

Note:
On devices that do not support volume control interfaces, this call returns 0, indicating a 0db setting.

Returns:
Returns the minimum volume setting for the requested interface.

3.5.4.34 USBHostAudioVolumeResGet

This function is used to get the volume control resolution for a given audio device.

Prototype:
```
uint32_t
USBHostAudioVolumeResGet(tUSBHostAudioInstance *psAudioInstance,
                         uint32_t ui32Interface,
                         uint32_t ui32Channel)
```

Parameters:
psAudioInstance is an instance of the USB audio device.
ui32Interface is the interface number to use to query the resolution for the volume control.
ui32Channel is the 0 based channel number to query.

Description:
The function is used to retrieve the volume control resolution for an audio device on the channel specified by *ui32Channel*. The *ui32Interface* is ignored for now and should be set to 0 to access the default audio control interface. The *ui32Channel* value starts with 0 which is the master audio volume control interface. The remaining *ui32Channel* values provide access to various other audio channels, with 1 and 2 being left and right audio channels.

Note:
On devices that do not support volume control interfaces, this call returns 0, indicating a 0db setting.

Returns:
Returns the volume control resolution for the requested interface.

3.5.4.35 USBHostAudioVolumeSet

This function is used to set the current volume setting for a given audio device.

Prototype:
```
void
USBHostAudioVolumeSet(tUSBHostAudioInstance *psAudioInstance,
```

```
                        uint32_t ui32Interface,
                        uint32_t ui32Channel,
                        uint32_t ui32Value)
```

Parameters:
> ***psAudioInstance*** is an instance of the USB audio device.
> ***ui32Interface*** is the interface number to use to set the current volume setting.
> ***ui32Channel*** is the 0 based channel number to query.
> ***ui32Value*** is the value to write to the USB audio device.

Description:
> The function is used to set the current volume setting for an audio device on the channel specified by *ui32Channel*. The *ui32Interface* is ignored for now and should be set to 0 to access the default audio control interface. The *ui32Channel* value starts with 0 which is the master audio volume control interface. The remaining *ui32Channel* values provide access to various other audio channels, with 1 and 2 being left and right audio channels.

Note:
> On devices that do not support volume control interfaces, this call returns 0, indicating a 0db setting.

Returns:
> None.

3.5.4.36 USBHSCSIInquiry

This will issue the SCSI inquiry command to a device.

Prototype:
```
uint32_t
USBHSCSIInquiry(uint32_t ui32InPipe,
                uint32_t ui32OutPipe,
                uint8_t *pui8Data,
                uint32_t *pui32Size)
```

Parameters:
> ***ui32InPipe*** is the USB IN pipe to use for this command.
> ***ui32OutPipe*** is the USB OUT pipe to use for this command.
> ***pui8Data*** is the data buffer to return the results into.
> ***pui32Size*** is the size of buffer that was passed in on entry and the number of bytes returned.

Description:
> This function should be used to issue a SCSI Inquiry command to a mass storage device. To allow for multiple devices, the *ui32InPipe* and *ui32OutPipe* parameters indicate which USB pipes to use for this call.

Note:
> The *pui8Data* buffer pointer should have at least **SCSI_INQUIRY_DATA_SZ** bytes of data or this function will overflow the buffer.

Returns:
> This function returns the SCSI status from the command. The value will be either **SCSI_CMD_STATUS_PASS** or **SCSI_CMD_STATUS_FAIL**.

3.5.4.37 USBHSCSIModeSense6

This will issue the SCSI Mode Sense(6) command to a device.

Prototype:
```
uint32_t
USBHSCSIModeSense6(uint32_t ui32InPipe,
                   uint32_t ui32OutPipe,
                   uint32_t ui32Flags,
                   uint8_t *pui8Data,
                   uint32_t *pui32Size)
```

Parameters:
> ***ui32InPipe*** is the USB IN pipe to use for this command.
> ***ui32OutPipe*** is the USB OUT pipe to use for this command.
> ***ui32Flags*** is a combination of flags defining the exact query that is to be made.
> ***pui8Data*** is the data buffer to return the results into.
> ***pui32Size*** is the size of the buffer on entry and number of bytes read on exit.

Description:
> This function should be used to issue a SCSI Mode Sense(6) command to a mass storage device. To allow for multiple devices, the *ui32InPipe* and *ui32OutPipe* parameters indicate which USB pipes to use for this call. The call will return at most the number of bytes in the *pui32Size* parameter, however it can return less and change the *pui32Size* parameter to the number of valid bytes in the ∗*pui32Size* buffer.
>
> The *ui32Flags* parameter is a combination of the following three sets of definitions:
>
> One of the following values must be specified:
>
> - **SCSI_MS_PC_CURRENT** request for current settings.
> - **SCSI_MS_PC_CHANGEABLE** request for changeable settings.
> - **SCSI_MS_PC_DEFAULT** request for default settings.
> - **SCSI_MS_PC_SAVED** request for the saved values.
>
> One of these following values must also be specified to determine the page code for the request:
>
> - **SCSI_MS_PC_VENDOR** is the vendor specific page code.
> - **SCSI_MS_PC_DISCO** is the disconnect/reconnect page code.
> - **SCSI_MS_PC_CONTROL** is the control page code.
> - **SCSI_MS_PC_LUN** is the protocol specific LUN page code.
> - **SCSI_MS_PC_PORT** is the protocol specific port page code.
> - **SCSI_MS_PC_POWER** is the power condition page code.
> - **SCSI_MS_PC_INFORM** is the informational exceptions page code.
> - **SCSI_MS_PC_ALL** will request all pages codes supported by the device.
>
> The last value is optional and supports the following global flag:
>
> - **SCSI_MS_DBD** disables returning block descriptors.
>
> Example: Request for all current settings.

```
SCSIModeSense6(ui32InPipe,  ui32OutPipe,
               SCSI_MS_PC_CURRENT | SCSI_MS_PC_ALL,
               pui8Data,  pui32Size);
```

Returns:
This function returns the SCSI status from the command.   The value will be either
SCSI_CMD_STATUS_PASS or **SCSI_CMD_STATUS_FAIL**.

3.5.4.38   USBHSCSIRead10

This function issues a SCSI Read(10) command to a device.

Prototype:
```
uint32_t
USBHSCSIRead10(uint32_t ui32InPipe,
               uint32_t ui32OutPipe,
               uint32_t ui32LBA,
               uint8_t *pui8Data,
               uint32_t *pui32Size,
               uint32_t ui32NumBlocks)
```

Parameters:
ui32InPipe is the USB IN pipe to use for this command.

ui32OutPipe is the USB OUT pipe to use for this command.

ui32LBA is the logical block address to read.

pui8Data is the data buffer to return the data.

pui32Size is the size of the buffer on entry and number of bytes read on exit.

ui32NumBlocks is the number of contiguous blocks to read from the device.

Description:
This function is used to issue a SCSI Read(10) command to a device. The *ui32LBA* parameter
specifies the logical block address to read from the device. The data from this block will be
returned in the buffer pointed to by *pui8Data*. The parameter *pui32Size* should indicate enough
space to hold a full block size, or only the first *pui32Size* bytes of the LBA are returned.

Returns:
This function returns the results of the SCSI Read(10) command. The value will be either
SCSI_CMD_STATUS_PASS or **SCSI_CMD_STATUS_FAIL**.

3.5.4.39   USBHSCSIReadCapacities

This will issue the SCSI read capacities command to a device.

Prototype:
```
uint32_t
USBHSCSIReadCapacities(uint32_t ui32InPipe,
                       uint32_t ui32OutPipe,
                       uint8_t *pui8Data,
                       uint32_t *pui32Size)
```

Parameters:

ui32InPipe is the USB IN pipe to use for this command.

ui32OutPipe is the USB OUT pipe to use for this command.

pui8Data is the data buffer to return the results into.

pui32Size is the size of buffer that was passed in on entry and the number of bytes returned.

Description:

This function should be used to issue a SCSI Read Capacities command to a mass storage device that is connected. To allow for multiple devices, the *ui32InPipe* and *ui32OutPipe* parameters indicate which USB pipes to use for this call.

Returns:

This function returns the SCSI status from the command. The value will be either **SCSI_CMD_STATUS_PASS** or **SCSI_CMD_STATUS_FAIL**.

3.5.4.40 USBHSCSIReadCapacity

This will issue the SCSI read capacity command to a device.

Prototype:

```
uint32_t
USBHSCSIReadCapacity(uint32_t ui32InPipe,
                     uint32_t ui32OutPipe,
                     uint8_t *pui8Data,
                     uint32_t *pui32Size)
```

Parameters:

ui32InPipe is the USB IN pipe to use for this command.

ui32OutPipe is the USB OUT pipe to use for this command.

pui8Data is the data buffer to return the results into.

pui32Size is the size of buffer that was passed in on entry and the number of bytes returned.

Description:

This function should be used to issue a SCSI Read Capacity command to a mass storage device that is connected. To allow for multiple devices, the *ui32InPipe* and *ui32OutPipe* parameters indicate which USB pipes to use for this call.

Note:

The *pui8Data* buffer pointer should have at least **SCSI_READ_CAPACITY_SZ** bytes of data or this function will overflow the buffer.

Returns:

This function returns the SCSI status from the command. The value will be either **SCSI_CMD_STATUS_PASS** or **SCSI_CMD_STATUS_FAIL**.

3.5.4.41 USBHSCSIRequestSense

This function issues a SCSI Request Sense command to a device.

Prototype:
```
uint32_t
USBHSCSIRequestSense(uint32_t ui32InPipe,
                     uint32_t ui32OutPipe,
                     uint8_t *pui8Data,
                     uint32_t *pui32Size)
```

Parameters:
>   ***ui32InPipe*** is the USB IN pipe to use for this command.
>
>   ***ui32OutPipe*** is the USB OUT pipe to use for this command.
>
>   ***pui8Data*** is the data buffer to return the results into.
>
>   ***pui32Size*** is the size of the buffer on entry and number of bytes read on exit.

Description:
>   This function is used to issue a SCSI Request Sense command to a device. It will return the data in the buffer pointed to by *pui8Data*. The parameter *pui32Size* should have the allocation size in bytes of the buffer pointed to by *pui8Data*.

Returns:
>   This function returns the results of the SCSI Request Sense command. The value will be either **SCSI_CMD_STATUS_PASS** or **SCSI_CMD_STATUS_FAIL**.

3.5.4.42 USBHSCSITestUnitReady

This function issues a SCSI Test Unit Ready command to a device.

Prototype:
```
uint32_t
USBHSCSITestUnitReady(uint32_t ui32InPipe,
                      uint32_t ui32OutPipe)
```

Parameters:
>   ***ui32InPipe*** is the USB IN pipe to use for this command.
>
>   ***ui32OutPipe*** is the USB OUT pipe to use for this command.

Description:
>   This function is used to issue a SCSI Test Unit Ready command to a device. This call will simply return the results of issuing this command.

Returns:
>   This function returns the results of the SCSI Test Unit Ready command. The value will be either **SCSI_CMD_STATUS_PASS** or **SCSI_CMD_STATUS_FAIL**.

3.5.4.43 USBHSCSIWrite10

This function issues a SCSI Write(10) command to a device.

Prototype:
```
uint32_t
USBHSCSIWrite10(uint32_t ui32InPipe,
```

```
                              uint32_t ui32OutPipe,
                              uint32_t ui32LBA,
                              uint8_t *pui8Data,
                              uint32_t *pui32Size,
                              uint32_t ui32NumBlocks)
```

Description:
> This function is used to issue a SCSI Write(10) command to a device. The *ui32LBA* parameter specifies the logical block address on the device. The data to write to this block should be in the buffer pointed to by *pui8Data* parameter. The parameter *pui32Size* should indicate the amount of data to write to the specified LBA.

Parameters:
> *ui32InPipe* is the USB IN pipe to use for this command.
>
> *ui32OutPipe* is the USB OUT pipe to use for this command.
>
> *ui32LBA* is the logical block address to read.
>
> *pui8Data* is the data buffer to write out.
>
> *pui32Size* is the size of the buffer.
>
> *ui32NumBlocks* is the number of contiguous blocks to write to the device.

Returns:
> This function returns the results of the SCSI Write(10) command. The value will be either **SCSI_CMD_STATUS_PASS** or **SCSI_CMD_STATUS_FAIL**.

3.5.5 Variable Documentation

3.5.5.1 g_sUSBHIDClassDriver

Definition:
> const tUSBHostClassDriver g_sUSBHIDClassDriver

Description:
> This constant global structure defines the HID Class Driver that is provided with the USB library.

3.5.5.2 g_sUSBHostAudioClassDriver

Definition:
> const tUSBHostClassDriver g_sUSBHostAudioClassDriver

Description:
> This constant global structure defines the Audio Class Driver that is provided with the USB library.

3.5.5.3 g_sUSBHostMSCClassDriver

Definition:
> const tUSBHostClassDriver g_sUSBHostMSCClassDriver

Description:

This constant global structure defines the Mass Storage Class Driver that is provided with the USB library.

3.5.5.4 g_sUSBHubClassDriver

Definition:

```
const tUSBHostClassDriver g_sUSBHubClassDriver
```

Description:

This constant global structure defines the Hub Class Driver that is provided with the USB library.

3.6 Host Device Interface

The USB library provides a set of example host device interfaces for a HID mouse, a HID keyboard and a mass storage device. The next few sections discusses each briefly and explain how their interfaces can be used by an application.

3.6.1 Mouse Device

The HID mouse device interface is controlled mainly through a callback function that is provided as part of the call to open the mouse device interface. In order to open an instance of the mouse device the application calls USBHMouseOpen() and passes in a callback function as well as some buffer data for use by the mouse device. The buffer provided is used internally by the mouse device and should not be used by the application. Once the device has been opened, the application should wait for a **USB_EVENT_CONNECTED** event to indicate that a mouse has been successfully detected and enumerated. At this point the application should call the USBHMouseInit() function to initialize the actual device that is connected. After this, the application can expect to start receiving the following events via the callback that was provided in the USBHMouseOpen() call: **USBH_EVENT_HID_MS_PRESS**, **USBH_EVENT_HID_MS_REL**, **USBH_EVENT_HID_MS_X**, or **USBH_EVENT_HID_MS_Y**.

USBH_EVENT_HID_MS_PRESS

The ui32MsgParam parameter has one of the following values **HID_MOUSE_BUTTON_1**, **HID_MOUSE_BUTTON_2**, **HID_MOUSE_BUTTON_3** indicating which buttons have changed to the pressed state.

USBH_EVENT_HID_MS_REL

The ui32MsgParam parameter has one of the following values **HID_MOUSE_BUTTON_1**, **HID_MOUSE_BUTTON_2**, **HID_MOUSE_BUTTON_3** indicating which buttons have changed to the released state.

USBH_EVENT_HID_MS_X

The ui32MsgParam parameter has an 8 bit signed value indicating the delta in the X direction since the last update.

USBH_EVENT_HID_MS_Y

The ui32MsgParam parameter has an 8 bit signed value indicating the delta in the Y direction since the last update.

When the application is done using the mouse device it can call USBHMouseClose() to release the instance of the mouse device and free up the buffer that it passed to the mouse device.

3.6.2   Keyboard Device

Like the mouse, the HID keyboard device interface is controlled mainly through a callback function that is provided as part of the call to open the keyboard device interface. In order to open an instance of the keyboard device the application calls USBHKeyboardOpen() and passes in a callback function as well as some buffer data for use by the keyboard device. The buffer provided is used internally by the keyboard device and should not be used by the application. Once the device has been opened, the application should wait for a **USB_EVENT_CONNECTED** event to indicate that a keyboard has been successfully detected and enumerated. At this point the application should call the USBHKeyboardInit() function to initialize the actual keyboard device that is connected. After this, the application can expect to receive the following events via the callback that was provided in the USBHKeyboardOpen() call: **USBH_EVENT_HID_KB_PRESS**, **USBH_EVENT_HID_KB_REL**, or **USBH_EVENT_HID_KB_MOD**.

USBH_EVENT_HID_KB_PRESS

The ui32MsgParam parameter has the USB usage identifier for the key that has been pressed. It is up to the application to map this usage identifier to an actual printable character using the USBHKeyboardUsageToChar() function, or it can simple respond to the key press without echoing the key to any output device. It should be noted that "special" keys like the Caps Lock key require notifying the actual keyboard device that the host application has detected that the key has been pressed.

USBH_EVENT_HID_KB_REL

The ui32MsgParam parameter has the USB usage identifier for the key that has been released.

USBH_EVENT_HID_KB_MOD

The ui32MsgParam parameter has the current state of all of the modifier keys on the connected keyboard. This value is a bit mapped representation of the modifier keys that can have any of the following bits set:

- **HID_KEYB_LEFT_CTRL**
- **HID_KEYB_LEFT_SHIFT**
- **HID_KEYB_LEFT_ALT**
- **HID_KEYB_LEFT_GUI**
- **HID_KEYB_RIGHT_CTRL**
- **HID_KEYB_RIGHT_SHIFT**
- **HID_KEYB_RIGHT_ALT**
- **HID_KEYB_RIGHT_GUI**

3.7   Host Device Interface Definitions

Functions

- uint32_t USBHKeyboardClose (tUSBHKeyboard ∗psKbInstance)
- uint32_t USBHKeyboardInit (tUSBHKeyboard ∗psKbInstance)
- uint32_t USBHKeyboardLPMSleep (tUSBHKeyboard ∗psKbInstance)
- uint32_t USBHKeyboardLPMStatus (tUSBHKeyboard ∗psKbInstance)
- uint32_t USBHKeyboardModifierSet (tUSBHKeyboard ∗psKbInstance, uint32_t ui32Modifiers)
- tUSBHKeyboard ∗ USBHKeyboardOpen (tUSBHIDKeyboardCallback pfnCallback, uint8_t ∗pui8Buffer, uint32_t ui32Size)
- uint32_t USBHKeyboardPollRateSet (tUSBHKeyboard ∗psKbInstance, uint32_t ui32PollRate)
- uint32_t USBHKeyboardUsageToChar (tUSBHKeyboard ∗psKbInstance, const tHIDKeyboardUsageTable ∗psTable, uint8_t ui8UsageID)
- uint32_t USBHMouseClose (tUSBHMouse ∗psMsInstance)
- uint32_t USBHMouseInit (tUSBHMouse ∗psMsInstance)
- uint32_t USBHMouseLPMSleep (tUSBHMouse ∗psMsInstance)
- uint32_t USBHMouseLPMStatus (tUSBHMouse ∗psMsInstance)
- tUSBHMouse ∗ USBHMouseOpen (tUSBHIDMouseCallback pfnCallback, uint8_t ∗pui8Buffer, uint32_t ui32Size)

3.7.1   Detailed Description

The macros and functions defined in this section can be found in header files `host/usbhhidkeyboard.h` and `host/usbhhidmouse.h`.

3.7.2   Function Documentation

3.7.2.1   USBHKeyboardClose

This function is used close an instance of a keyboard.

Prototype:
```
uint32_t
USBHKeyboardClose(tUSBHKeyboard *psKbInstance)
```

Parameters:
psKbInstance is the instance value for this keyboard.

Description:
This function is used to close an instance of the keyboard that was opened with a call to USBHKeyboardOpen(). The *psKbInstance* value is the value that was returned when the application called USBHKeyboardOpen().

Returns:
This function returns 0 to indicate success any non-zero value indicates an error condition.

3.7.2.2   USBHKeyboardInit

This function is used to initialize a keyboard interface after a keyboard has been detected.

Prototype:
```
uint32_t
USBHKeyboardInit(tUSBHKeyboard *psKbInstance)
```

Parameters:
> ***psKbInstance*** is the instance value for this keyboard.

Description:
> This function should be called after receiving a **USB_EVENT_CONNECTED** event in the call-back function provided by USBHKeyboardOpen(), however this function should only be called outside the callback function. This will initialize the keyboard interface and determine the keyboard's layout and how it reports keys to the USB host controller. The *psKbInstance* value is the value that was returned when the application called USBHKeyboardOpen(). This function only needs to be called once per connection event but it should be called every time a **USB_EVENT_CONNECTED** event occurs.

Returns:
> This function returns 0 to indicate success any non-zero value indicates an error condition.

3.7.2.3   USBHKeyboardLPMSleep

This function forwards an LPM request for a device to enter L1 sleep state.

Prototype:
```
uint32_t
USBHKeyboardLPMSleep(tUSBHKeyboard *psKbInstance)
```

Parameters:
> ***psKbInstance*** is the HID keyboard instance that was returned from the call to USBHKeyboardOpen().

Description:
> This function forwards a request from an application to the HID device class to request that a device enter the LPM L1 sleep state. The caller must check the return value to see if the request can be attempted at this time. If another LPM transaction is busy on this or another device, then this function returns **USBHCD_LPM_PENDING**. If the LPM request was scheduled to be sent the function returns **USBHCD_LPM_AVAIL**. The caller should check the USBHCDLPMStatus() function to determine if the request completed successfully or if there was an error.

Returns:
> This function returns the following values:
>
> - **USBHCD_LPM_AVAIL** - The transition to L1 state is scheduled to be sent.
> - **USBHCD_LPM_PENDING** - There is already an LPM request pending.

3.7.2.4   USBHKeyboardLPMStatus

This function returns the current status of an LPM request.

Prototype:
```
uint32_t
USBHKeyboardLPMStatus(tUSBHKeyboard *psKbInstance)
```

Parameters:
> ***psKbInstance*** is the HID keyboard instance that was returned from the call to USBHKeyboardOpen().

Description:
> This function returns the current status of LPM requests for a given device. This is called to determine if a previous request completed successfully or if there was an error.

Returns:
> This function returns the following values:

> - **USBHCD_LPM_AVAIL** - There are no pending LPM requests on this specific device or the last request completed successfully.
> - **USBHCD_LPM_ERROR** - The last LPM request for this device did not complete successfully.
> - **USBHCD_LPM_PENDING** - The last LPM request has not completed.

3.7.2.5   USBHKeyboardModifierSet

This function is used to set one of the fixed modifier keys on a keyboard.

Prototype:
```
uint32_t
USBHKeyboardModifierSet(tUSBHKeyboard *psKbInstance,
                        uint32_t ui32Modifiers)
```

Parameters:
> ***psKbInstance*** is the instance value for this keyboard.
> ***ui32Modifiers*** is a bit mask of the modifiers to set on the keyboard.

Description:
> This function is used to set the modifier key states on a keyboard. The *ui32Modifiers* value is a bitmask of the following set of values:

> - **HID_KEYB_NUM_LOCK**
> - **HID_KEYB_CAPS_LOCK**
> - **HID_KEYB_SCROLL_LOCK**
> - **HID_KEYB_COMPOSE**
> - **HID_KEYB_KANA**

> Not all of these will be supported on all keyboards however setting values on a keyboard that does not have them should have no effect. The *psKbInstance* value is the value that was returned when the application called USBHKeyboardOpen(). If the value **HID_KEYB_CAPS_LOCK** is used it will modify the values returned from the USBHKeyboardUsageToChar() function.

Returns:
   This function returns 0 to indicate success any non-zero value indicates an error condition.

3.7.2.6   USBHKeyboardOpen

This function is used open an instance of a keyboard.

Prototype:
```
tUSBHKeyboard *
USBHKeyboardOpen(tUSBHIDKeyboardCallback pfnCallback,
                 uint8_t *pui8Buffer,
                 uint32_t ui32Size)
```

Parameters:
   pfnCallback is the callback function to call when new events occur with the keyboard returned.
   pui8Buffer is the memory used by the keyboard to interact with the USB keyboard.
   ui32Size is the size of the buffer provided by *pui8Buffer*.

Description:
   This function is used to open an instance of the keyboard. The value returned from this function should be used as the instance identifier for all other USBHKeyboard calls. The *pui8Buffer* memory buffer is used to access the keyboard. The buffer size required is at least enough to hold a normal report descriptor for the device. If there is not enough space only a partial report descriptor will be read out.

Returns:
   Returns the instance identifier for the keyboard that is attached. If there is no keyboard present this will return 0.

3.7.2.7   USBHKeyboardPollRateSet

This function is used to set the automatic poll rate of the keyboard.

Prototype:
```
uint32_t
USBHKeyboardPollRateSet(tUSBHKeyboard *psKbInstance,
                        uint32_t ui32PollRate)
```

Parameters:
   psKbInstance is the instance value for this keyboard.
   ui32PollRate is the rate in ms to cause the keyboard to update the host regardless of no change in key state.

Description:
   This function will allow an application to tell the keyboard how often it should send updates to the USB host controller regardless of any changes in keyboard state. The *psKbInstance* value is the value that was returned when the application called USBHKeyboardOpen(). The *ui32PollRate* is the new value in ms for the update rate on the keyboard. This value is initially set to 0 which indicates that the keyboard should only to update when the keyboard state changes. Any value other than 0 can be used to force the keyboard to generate auto-repeat sequences for the application.

Returns:
This function returns 0 to indicate success any non-zero value indicates an error condition.

3.7.2.8   USBHKeyboardUsageToChar

This function is used to map a USB usage ID to a printable character.

Prototype:
```
uint32_t
USBHKeyboardUsageToChar(tUSBHKeyboard *psKbInstance,
                        const tHIDKeyboardUsageTable *psTable,
                        uint8_t ui8UsageID)
```

Parameters:
psKbInstance is the instance value for this keyboard.
psTable is the table to use to map the usage ID to characters.
ui8UsageID is the USB usage ID to map to a character.

Description:
This function is used to map a USB usage ID to a character. The provided *psTable* is used to perform the mapping and is described by the tHIDKeyboardUsageTable type defined structure. See the documentation on the tHIDKeyboardUsageTable structure for more details on the internals of this structure. This function uses the current state of the shift keys and the Caps Lock key to modify the data returned by this function. The psTable structure has values indicating which keys are modified by Caps and alternate values for shifted cases. The number of bytes returned from Lock this function depends on the *psTable* structure passed in as it holds the number of bytes per character in the table.

Returns:
Returns the character value for the given usage id.

3.7.2.9   USBHMouseClose

This function is used close an instance of a mouse.

Prototype:
```
uint32_t
USBHMouseClose(tUSBHMouse *psMsInstance)
```

Parameters:
psMsInstance is the instance value for this mouse.

Description:
This function is used to close an instance of the mouse that was opened with a call to USBHMouseOpen(). The *psMsInstance* value is the value that was returned when the application called USBHMouseOpen().

Returns:
Returns 0.

3.7.2.10 USBHMouseInit

This function is used to initialize a mouse interface after a mouse has been detected.

Prototype:
```
uint32_t
USBHMouseInit(tUSBHMouse *psMsInstance)
```

Parameters:
psMsInstance is the instance value for this mouse.

Description:
This function should be called after receiving a **USB_EVENT_CONNECTED** event in the call-back function provided by USBHMouseOpen(), however it should only be called outside of the callback function. This will initialize the mouse interface and determine how it reports events to the USB host controller. The *psMsInstance* value is the value that was returned when the application called USBHMouseOpen(). This function only needs to be called once per connection event but it should be called every time a **USB_EVENT_CONNECTED** event occurs.

Returns:
Non-zero values should be assumed to indicate an error condition.

3.7.2.11 USBHMouseLPMSleep

This function forwards an LPM request for a device to enter L1 sleep state.

Prototype:
```
uint32_t
USBHMouseLPMSleep(tUSBHMouse *psMsInstance)
```

Parameters:
psMsInstance is the HID keyboard instance that was returned from the call to USBH-MouseOpen().

Description:
This function forwards a request from an application to the HID device class to request that a device enter the LPM L1 sleep state. The caller must check the return value to see if the request can be attempted at this time. If another LPM transaction is busy on this or another device, then this function returns **USBHCD_LPM_PENDING**. If the LPM request was scheduled to be sent the function returns **USBHCD_LPM_AVAIL**. The caller should check the USBHCDLPMStatus() function to determine if the request completed successfully or if there was an error.

Returns:
This function returns the following values:

- **USBHCD_LPM_AVAIL** - The transition to L1 state is scheduled to be sent.
- **USBHCD_LPM_PENDING** - There is already an LPM request pending.

3.7.2.12 USBHMouseLPMStatus

This function returns the current status of an LPM request.

Prototype:
```
uint32_t
USBHMouseLPMStatus(tUSBHMouse *psMsInstance)
```

Parameters:
> ***psMsInstance*** is the HID keyboard instance that was returned from the call to USBH-MouseOpen().

Description:
> This function returns the current status of LPM requests for a given device. This is called to determine if a previous request completed successfully or if there was an error.

Returns:
> This function returns the following values:

> - **USBHCD_LPM_AVAIL** - There are no pending LPM requests on this specific device or the last request completed successfully.
> - **USBHCD_LPM_ERROR** - The last LPM request for this device did not complete successfully.
> - **USBHCD_LPM_PENDING** - The last LPM request has not completed.

3.7.2.13 USBHMouseOpen

This function is used open an instance of a mouse.

Prototype:
```
tUSBHMouse *
USBHMouseOpen(tUSBHIDMouseCallback pfnCallback,
              uint8_t *pui8Buffer,
              uint32_t ui32Size)
```

Parameters:
> ***pfnCallback*** is the callback function to call when new events occur with the mouse returned.
> ***pui8Buffer*** is the memory used by the driver to interact with the USB mouse.
> ***ui32Size*** is the size of the buffer provided by *pui8Buffer*.

Description:
> This function is used to open an instance of the mouse. The value returned from this function should be used as the instance identifier for all other USBHMouse calls. The *pui8Buffer* memory buffer is used to access the mouse. The buffer size required is at least enough to hold a normal report descriptor for the device.

Returns:
> Returns the instance identifier for the mouse that is attached. If there is no mouse present this will return 0.

3.8 Host Programming Examples

The USB library provides examples for three host applications that can access mass storage devices and HID keyboard and mouse devices. These next sections cover the basics of each of these three applications and how they interact with the USB library.

3.8.1   Application Initialization

The USB library host stack initialization is handled in the USBHCDInit() function. This function should be called after registering class drivers using USBHCDRegisterDrivers() and, optionally, configuring power pins using USBHCDPowerConfigInit(). Both of these functions are described later.

The USBHCDInit() function takes three parameters, the first of which specifies which USB controller to initialize. This value is a zero based index of the host controller to initialize. The next two parameters specify a memory pool for use by the host controller driver. The size of this buffer should be at least large enough to hold a typical configuration descriptor for devices that are going to be supported. This value is system dependent so it is left to the application to set the size, however it should never be less than 32 bytes and in most cases should be at least 64 bytes. If there is not enough memory to load a configuration descriptor from a device, the device is not recognized by USB library's host controller driver. The USB library also provides a method to shut down an instance of the host controller driver by calling the USBHCDTerm() function. The USBHCDTerm() function should be called any time the application wants to shut down the USB host controller in order to disable it, or possibly switch modes in the case of a dual role controller.

The USB library assumes that the power pin configuration has an active high signal for controlling the external power. If this is not the case or if the application wants control over the power fault logic provided by the library, then the application should call the USBHCDPowerConfigInit() function before calling USBHCDInit() in order to properly configure the power control pins. The polarity of the power pin, the polarity of the the power fault pin and any actions taken in response to a power fault are all controlled by passing a combination of sets of values in the *ulPwrConfig* parameter. See the documentation for the USBHCDPowerConfigInit() function for more details on this function.

3.8.2   Application Interface

The USB library host stack requires some portion of the code to not run in the interrupt handler so it provides the USBHCDMain() function that must be called periodically in the main application. This can be as a result of a timer tick or just once per main loop in a simple application. It should not be called in an interrupt handler. Calling the function too often is harmless as it simply returns if the USB host stack has nothing to do. Calling USBHCDMain() too infrequently can cause enumeration to take longer than normal. It is up to the application to prioritize the importance of USB communications by calling USBHCDMain() at a rate that is reasonable to the application.

All support devices must have a host class driver loaded in order to communicate with each type of device that is supported. The details of interacting with these host class drivers is explained in the host class driver sections that follow in this document.

3.8.3   Application Termination

When the application needs to shut down the host controller it needs to shutdown all host class drivers and then shut down the host controller itself. This gives the host class drivers a chance to close cleanly by calling each host class driver's close function. Then the USBHCDTerm() function should be called to shut down the host controller. This sequence leaves the USB controller and the USB library stack in a state so that it is ready to be re-initialized or in order to switch USB mode from host to device.

3.8.4   Example Application Setup

The following example shows the basic setup code needed for any application that is using the USB library in host mode. The *g_pui8HCDPool* array which is passed in to the USBHCDInit() is used as heap memory for by the USB library and thus the memory should not be used by the application. In this example, the *g_ppsHostClassDrivers* array holds both HID and MSC class drivers making it possible for both types of devices to be supported. However if the application only needs to include the classes that it needs to support in order to save code and memory space. The pin and peripheral configuration is left to the application as the USB pins may not always be on the same physical pins for every part supported by the USB library. The macros provided in the pin_map.h file included with DriverLib can be used to indicate which pin and peripheral to use for a given part. See the DriverLib documentation on pin mapping for more details on how it provides mapping of peripherals to pins on devices. The USBHCDRegisterDrivers() call passes in the static array of supported USB host class drivers that are supported by the application. As shown in the example, the application should always call the USB device interfaces open routines before calling USBHCDInit() since this call enables the USB host controller and start enumerating any connected device. If the device interface has not been called it may miss the connection notification and could miss some state information that occurred before the device interface was ready.

Example: Basic Configuration as Host

```
//*****************************************************************************
//
// The size of the host controller's memory pool in bytes.
//
//*****************************************************************************
#define HCD_MEMORY_SIZE         128

//*****************************************************************************
//
// The memory pool to provide to the Host controller driver.
//
//*****************************************************************************
uint8_t g_pui8HCDPool[HCD_MEMORY_SIZE];

//*****************************************************************************
//
// The global that holds all of the host drivers in use in the application.
// In this case, only the Keyboard class is loaded.
//
//*****************************************************************************
static tUSBHostClassDriver const * const g_ppsHostClassDrivers[] =
{
    &g_sUSBHIDClassDriver,
    &g_sUSBHostMSCClassDriver
};

//*****************************************************************************
//
// This global holds the number of class drivers in the g_ppsHostClassDrivers
// list.
//
//*****************************************************************************
static const uint32_t g_ui32NumHostClassDrivers =
    sizeof(g_ppsHostClassDrivers) / sizeof(tUSBHostClassDriver *);

...

//
// Enable the GPIO peripherals used by the USB pins.
//
```

```
MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOG);
MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOL);

//
// Configure the USB EPEN as a USB pin.
//
MAP_GPIOPinConfigure(GPIO_PG4_USB0EPEN);
MAP_GPIOPinTypeUSBDigital(GPIO_PORTG_BASE, GPIO_PIN_4);

//
// Enable PB0 and PB1 as VBUS and ID pins.
//
MAP_GPIOPinTypeUSBAnalog(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);

//
// Enable the D+/D- as USB pins.
//
MAP_GPIOPinTypeUSBAnalog(GPIO_PORTL_BASE, GPIO_PIN_6 | GPIO_PIN_7);

//
// Initialize the USB stack for host mode.
//
USBStackModeSet(0, eUSBModeHost, 0);

//
// Register the host class drivers.
//
USBHCDRegisterDrivers(0, g_ppsUSBHostClassDrivers, g_ui32NumHostClassDrivers);

//
// Initialize the power configuration. This sets the power enable signal
// to be active high and does not enable the power fault.
//
USBHCDPowerConfigInit(0, USBHCD_VBUS_AUTO_HIGH | USBHCD_VBUS_FILTER);

...

//
// Call any open routines on the device class interfaces here so that they
// are ready to receive callbacks if the device is already inserted on
// power on.
//
// Eg: USBHMSCDriveOpen(0, MSCCallback);
//

...

//
// Initialize the host controller.
//
USBHCDInit(0, g_pui8HCDPool, HCD_MEMORY_SIZE);
```

3.8.5   Host HID Mouse Programming Example

The USB library HID mouse example provides support for HID mouse devices that support the USB HID mouse BIOS protocol. Since most mice support the BIOS protocol nearly any mouse should be able to be connected and be supported. The initial call to USBHMouseOpen() prepares the mouse device application interface to receive notifications from any USB mouse device that is connected. Since the mouse interface needs some basic configuration after being connected the application needs to wait for the mouse to be connected and then call the USBHMouseInit() function to finish

off the mouse configuration.

Example: Mouse Configuration

```
//
// Open an instance of the mouse driver.  The mouse does not need
// to be present at this time, this just saves a place for it and allows
// the applications to be notified when a mouse is present.
//
g_psMouseInstance = USBHMouseOpen(MouseCallback, g_pui8Buffer, 128);

...

//
// Main loop of application.
//
while(1)
{
    switch(iMouseState)
    {
        //
        // This state is entered when they mouse is first detected.
        //
        case MOUSE_INIT:
        {
            //
            // Initialized the newly connected mouse.
            //
            USBHMouseInit(g_psMouseInstance);

            //
            // Proceed to the mouse connected state.
            //
            iMouseState = MOUSE_CONNECTED;

            break;
        }
        case MOUSE_CONNECTED:
        {
            break;
        }
        case MOUSE_NOT_CONNECTED:
        default:
        {
            break;
        }
    }

    //
    // Periodic call the main loop for the Host controller driver.
    //
    USBHCDMain();
}
...
```

Once the mouse has been configured the application's mouse callback routine is notified any time there is a state change with the mouse. This includes the switching to the MOUSE_INIT state when a **USB_EVENT_CONNECTED** event occurs in order to trigger initialization of the mouse device. The **USB_EVENT_DISCONNECTED** simply switches the state of the application to let it know that the mouse is no longer present. The remaining events are mouse state changes that can be used by the application to move a cursor or make a selection based on a mouse click.

Example: Mouse Callback Routine

```
uint32_t
MouseCallback(tUSBHMouse *psMsInstance, uint32_t ui32Event,
              uint32_t ui32MsgParam, void *pvMsgData)
{
    switch(ui32Event)
    {
        //
        // New mouse detected.
        //
        case USB_EVENT_CONNECTED:
        {
            iMouseState = MOUSE_INIT;
            break;
        }

        //
        // Mouse has been unplugged.
        //
        case USB_EVENT_DISCONNECTED:
        {
            iMouseState = MOUSE_NOT_CONNECTED;
            break;
        }

        //
        // New Mouse events detected.
        //
        case USBH_EVENT_HID_MS_PRESS:
        {
            break;
        }
        case USBH_EVENT_HID_MS_REL:
        {
            break;
        }
        case USBH_EVENT_HID_MS_X:
        {
            break;
        }
        case USBH_EVENT_HID_MS_Y:
        {
            break;
        }
    }
    return(0);
}
```

3.8.6   Host HID Keyboard Programming Example

The USB library HID keyboard example provides support for HID keyboard devices that support the USB HID keyboard BIOS protocol. Since most keyboards support the BIOS protocol most keyboards should be able to be connected and be supported. The initial call to USBHKeyboardOpen() prepares the keyboard device application interface to receive notifications from any USB keyboard device that is connected. The keyboard interface needs some basic configuration and needs to set the current state of LEDs on the keyboard, the application must wait for the keyboard to be connected and then call the USBHKeyboardInit() function.

Example: Keyboard Configuration

. . .

```
//
// Open an instance of the keyboard driver.  The keyboard does not need
// to be present at this time, this just save a place for it and allows
// the applications to be notified when a keyboard is present.
//
g_psKeyboardInstance = USBHKeyboardOpen(KeyboardCallback, g_pui8Buffer, 128);


//
// The main loop for the application.
//
while(1)
{
    switch(iKeyboardState)
    {
        //
        // This state is entered when they keyboard is first detected.
        //
        case KEYBOARD_INIT:
        {
            //
            // Initialized the newly connected keyboard.
            //
            USBHKeyboardInit(g_psKeyboardInstance);

            //
            // Proceed to the keyboard connected state.
            //
            iKeyboardState = KEYBOARD_CONNECTED;

            break;
        }
        case KEYBOARD_UPDATE:
        {
            //
            // If the application detected a change that required an
            // update to be sent to the keyboard to change the modifier
            // state then call it and return to the connected state.
            //
            iKeyboardState = KEYBOARD_CONNECTED;

            USBHKeyboardModifierSet(g_psKeyboardInstance, g_ui32Modifiers);
        }
        case KEYBOARD_CONNECTED:
        {
            break;
        }
        case KEYBOARD_NOT_CONNECTED:
        default:
        {
            break;
        }
    }

    //
    // Periodic call the main loop for the Host controller driver.
    //
    USBHCDMain();
}
```

Much like the mouse, the keyboard handles the reception of events entirely in the callback handler. This function should receive and store the keyboard events and handle them in the main program loop when the device is in the connected state. The **USB_EVENT_CONNECTED** lets the main loop know that it is time to call the USBHKeyboardInit() routine to configure the keyboard. The **USB_EVENT_DISCONNECTED** event simply informs the application that the keyboard is not

longer present and not to expect any more callbacks until another **USB_EVENT_CONNECTED** occurs. The remaining events all indicate that a key has been pressed or released. Normal key presses/releases generate **USBH_EVENT_HID_KB_PRESS** or **USBH_EVENT_HID_KB_REL** events while hitting keys like the shift, ctrl, alt and gui keys generate **USBH_EVENT_HID_KB_MOD** events.

Example: Keyboard Callback

```
uint32_t
KeyboardCallback(tUSBHKeyboard *psKbInstance, uint32_t ui32Event,
                 uint32_t ui32MsgParam, void *pvMsgData)
{
    uint8_t ui8Char;

    switch(ui32Event)
    {
        //
        // New keyboard detected.
        //
        case USB_EVENT_CONNECTED:
        {
            iKeyboardState = KEYBOARD_INIT;
            break;
        }

        //
        // Keyboard has been unplugged.
        //
        case USB_EVENT_DISCONNECTED:
        {
            iKeyboardState = KEYBOARD_NOT_CONNECTED;
            break;
        }

        //
        // New Key press detected.
        //
        case USBH_EVENT_HID_KB_PRESS:
        {
            //
            // ui32MsgParam holds the USB Usage ID.
            //
            break;
        }
        case USBH_EVENT_HID_KB_MOD:
        {
            //
            // ui32MsgParam holds the USB Modifier bit mask.
            //
            break;
        }
        case USBH_EVENT_HID_KB_REL:
        {
            //
            // ui32MsgParam holds the USB Usage ID.
            //
            break;
        }
    }
    return(0);
}
```

3.8.7   Host Mass Storage Programming Example

The following programming example demonstrates some of the basic interfaces that are available from the USB mass storage class application interface. See the "Basic Configuration as Host" example above for the initial configuration. The application should call USBHMSCDriveOpen() in order for the application to be ready for a new mass storage device. The application should also wait for the mass storage device to be ready to receive commands by calling USBHMSCDriveReady() and waiting for the value returned to go to 0 before attempting to read or write the device. Typically the reading and writing of the device is left to a file system layer as is the case in the example application, however the calls to directly read or write a block are shown in the example below.

Example: Mass Storage Coding Example

```
//
// Open an instance of the mass storage class driver.
//
g_psMSCInstance = USBHMSCDriveOpen(0, MSCCallback);

...

//
// Wait for the drive to become ready.
//
while(USBHMSCDriveReady(g_psMSCInstance))
{
    //
    // System level delay call should be here to give the device time to
    // become ready.
    //
    SysCtlDelay(g_ui32ClockRate / 100);
}

...

//
// Block Read example.
//
USBHMSCBlockRead(g_psMSCInstance, ui32LBA, pui8Data, 1);

...

//
// Block Write example.
//
USBHMSCBlockWrite(g_psMSCInstance, ui32LBA, pui8Data, 1);

...
```

4   OTG Functions

4.1   Introduction

This chapter covers the support provided by the USB library for the USB OTG controller. The USB library provides a simplified interface that allows the application to configure how often OTG sessions are requested and a callback method to notify the application when any mode switch has occurred. Since the OTG portion of the USB library must handle both device and host mode operation it requires some setup for both device and host modes. This puts some requirements on the order of initialization of the stacks in order to insure proper operation.

/note The USB library currently only supports OTG session request protocol(SRP) and does not support host negotiation protocol(HNP).

Source Code Overview

Source code and headers for the host specific USB functions can be found in the root directory of the USB library tree, typically `DriverLib/usblib`.

`usblib.h`	The header file containing OTG mode and other function prototypes and data types offered by the USB library.
`usbmode.c`	The source code for the USB OTG functions are in this file.

4.2   OTG Controller Driver

The USB library OTG functions provide a higher level interface to the USB controller's OTG hardware register interface. The hardware register interface is the lowest level of the interface and it interacts directly with the DriverLib USB APIs. The OTG functions provide the ability to detect which end of the USB cable was connected (A side or B side) as well as the presence of a device or host being connected. The OTG portion of the USB library requires initialization of the USB library in a certain order to safely bring up the the controller in OTG mode. This is because the stack requires both host and device mode operation. Once the USB library stack is initialized for OTG mode operation, the application will manage OTG notifications via a mode callback function that is called any time the USB mode changes.

4.2.1   Initialization

The application using the USB library in OTG mode must perform not only the initialization for the OTG functionality but also configure the host and device mode library functions as well. For host mode this includes registering the supported host classes, configuring the power pins, and calling any application level initialization functions that are required. For device mode, only the application level initialization is required as the OTG initialization will complete all other initialization required by the USB device library.

To properly configure host mode, the application must first configure and enable the host power pins for USBEPEN and USBPFLT. These are configured using in the library the USBHCDPower-ConfigInit(). The actual physical pins must also be enabled and configured by the application using SysCtlPeripheralEnable() followed by GPIOPinTypeUSBDigital(). These pins are located on different physical pins on devices and thus must be configured by the application. The remaining USB pins are handled by the library and do not require any configuration.

Once the pins are configured, the USB library must be put into OTG mode in order to properly handle incoming requests. The very first call after pin configuration should be to call USBStack-ModeSet() to register a callback function for OTG events as well as put the library into OTG mode. Next any calls needed to initialize the application level interface for device and host mode need to be called. For device this would be functions like USBDHIDMouseInit() for a HID mouse or it could be USBDCDInit() for a non-library provided device class. For host mode the the drivers must be registered with a call to USBHCDRegisterDrivers() and if a USB library provided class is used then it's application initialization interface must be called as well. For example in the case of supporting a USB mouse, the USBHMouseOpen() should be called to initialize the USB libraries HID mouse functionality. The last function that should be called by the application to initialize the USB OTG library is to call USBOTGModeInit() in order to finalize the initialization and enable the controller to start functioning in OTG mode. The USBOTGModeInit() also sets the polling rate for determining the mode of the controller as well as provide the memory that is required by the USB library host stack. Once this call is complete the application should be ready to receive any callbacks from the function provided to the USBStackModeSet() and any other device or host specific calls.

4.2.2   Mode Detection

The detection of which end of the cable that the USB controller is on and the current mode is done automatically in the library. The callback function passed in to USBStackModeSet() provides information on mode changes to allow the application to respond. Initially the USB library will start in idle state and remain in that state until a new mode is detected. The rate at which the mode is checked is passed into the USBOTGModeInit() function or can be set by the USBOTGPollRate() function once the OTG library code has been initialized. If USBOTGPollRate() is called with a polling rate of zero then polling for mode will stop until the rate is changed to a non-zero value. In a full OTG system, the polling rate is not required, but it will set how often the USB library issues session request on B side of the USB cable and how often it will look for devices if on the A side of the cable. Regardless of the mode that is detected the application should periodically call the USBOTGMain() function to allow the polling to function as well as call the USB library's main function once host mode is determined. The USBOTGMain() takes the number of elapsed milliseconds since the last call to USBOTGMain() in order to provide the USB OTG library functions with some basic timing information.

When the USB library detects that it is on the B side of the cable and has been powered, then the library will call back with the mode set to **eUSBModeDevice**. If the USB controller is connected to a non-OTG host then the library will simply detect device mode and start operating as a USB

device. All other device handling will work exactly as it does for normal device operation. When the USB cable is disconnected the USB library will return to the idle state and inform the application by calling the mode callback function with the mode set to **eUSBModeNone**.

The USB library detects that is is on the A side of the cable via a session request from the B side or detecting that the A side is connected to a cable due to the polling check. The USB library will then power the USB bus, call the mode callback function with the mode set to **eUSBModeHost**, and then begin enumerating the device. If a non-OTG device was connected the USB library will simply enumerate the device and function exactly as if the USB controller was in normal host mode.

Once the USB controller is unplugged from the host or detects that a device has been removed in host mode, the session will end and polling will resume. In both cases the USB library will call the callback function with the mode set to **eUSBModeNone** to inform the application that the USB library has returned to and idle state and is neither a host or device. Once the USB controller mode has been determined, no other USB library OTG API functions need to be called.

4.2.3 Interrupt Handling

All interrupt handling is done by the USB library USB0OTGModeIntHandler() function which is the interrupt handler for all USB interrupts in OTG mode. This function must be registered as the interrupt handler for the USB controller. To simplify handling both modes of operation, the OTG interrupt handler will take care of routing interrupts to the correct handler in USB library based on the current mode of operation. The application should be ready to receive any calls from other parts of the USB library based on the current mode of operation that was returned by the mode callback function. Since most callbacks are done in interrupt context they should be treated like interrupt handlers and should defer any real processing of events to occur outside the interrupt context. The callbacks are used to notify the applications of events that occur due to OTG negotiations or during normal operation.

4.3 OTG Controller Definitions

Functions

- void USB0OTGModeIntHandler (void)
- bool USBOTGFeatureSet (uint32_t ui32Index, uint32_t ui32Feature, void *pvFeature)
- void USBOTGMain (uint32_t ui32MsTicks)
- void USBOTGModeInit (uint32_t ui32Index, uint32_t ui32PollingRate, void *pvPool, uint32_t ui32PoolSize)
- void USBOTGModeTerm (uint32_t ui32Index)
- void USBOTGPollRate (uint32_t ui32Index, uint32_t ui32PollRate)

4.3.1 Detailed Description

The macros and functions defined in this section can be found in header file `usblib.h`.

4.3.2   Function Documentation

4.3.2.1   USB0OTGModeIntHandler

Handles OTG mode changes and also steers other USB interrupts from the controller to the correct handler in the USB stack.

Prototype:
```
void
USB0OTGModeIntHandler(void)
```

Description:
> This interrupt handler is used in applications which require to operate in both host and device mode using OTG. When in host or device mode, it steers the USB hardware interrupt to the correct handler in the USB stack depending upon the current operating mode. It also handles other OTG related interrupt events.
>
> For successful OTG mode operation, an application must register USB0OTGModeIntHandler() in the CPU vector table as the interrupt handler for the USB0 interrupt.

Note:
> This interrupt handler should only be used on controllers that support OTG functionality.

Returns:
> None.

4.3.2.2   USBOTGFeatureSet

This function sets one of the **USBLIB_FEATURE_** requests.

Prototype:
```
bool
USBOTGFeatureSet(uint32_t ui32Index,
                 uint32_t ui32Feature,
                 void *pvFeature)
```

Parameters:
> ***ui32Index*** is the index of the USB controller to access.
>
> ***ui32Feature*** is one of the **USBLIB_FEATURE_** defines.
>
> ***pvFeature*** is a pointer to the data for the **USBLIB_FEATURE** request.

Description:
> This function sends the requested feature request to the USB library. Not all features are supported by all devices so see the documentation for the **USBLIB_FEATURE_** to determine if the feature is supported.

Returns:
> Returns **true** if the feature was set and **false** if the feature is not supported or could not be changed to the requested value.

4.3.2.3   USBOTGMain

This function is the main routine for the OTG Controller Driver.

Prototype:
```
void
USBOTGMain(uint32_t ui32MsTicks)
```

Parameters:
> ***ui32MsTicks*** is the number of milliseconds that have passed since the last time this function was called.

Description:
> This function is the main routine for the USB controller when using the library in OTG mode. This routine must be called periodically by the main application outside of a callback context. The *ui32MsTicks* value is used for basic timing needed by the USB library when operating in OTG mode. This allows for a simple cooperative system to access the the OTG controller driver interface without the need for an RTOS. All time critical operations are handled in interrupt context but all longer operations are run from the this function to allow them to block and wait for completion without holding off other interrupts.

Returns:
> None.

4.3.2.4   USBOTGModeInit

Initializes the USB controller for OTG mode operation.

Prototype:
```
void
USBOTGModeInit(uint32_t ui32Index,
               uint32_t ui32PollingRate,
               void *pvPool,
               uint32_t ui32PoolSize)
```

Parameters:
> ***ui32Index*** specifies the USB controller that is to be initialized for OTG mode operation.
> ***ui32PollingRate*** is the rate in milliseconds to poll the controller for changes in mode.
> ***pvPool*** is a pointer to the data to use as a memory pool for this controller.
> ***ui32PoolSize*** is the size in bytes of the buffer passed in as *pvPool*.

Description:
> This function initializes the USB controller hardware into a state suitable for OTG mode operation. Applications must use this function to ensure that the controller is in a neutral state and able to receive appropriate interrupts before host or device mode is chosen by OTG negotiation. The *ui32PollingRate* parameter is used to set the rate at which the USB library will poll the controller to determine the mode. This has the most effect on how quickly the USB library will detect changes when going to host mode. The parameters *pvPool* and *ui32PoolSize* are passed on to the USB host library functions to provide memory for the USB library when it is acting as a host. Any device and host initialization should have been called before calling this function to prevent the USB library from attempting to run in device or host mode before the USB library is fully configured.

Returns:
    None.

4.3.2.5   USBOTGModeTerm

Returns the USB controller to and inactive state when in OTG mode operation.

Prototype:
```
void
USBOTGModeTerm(uint32_t ui32Index)
```

Parameters:
    ui32Index specifies the USB controller to end OTG mode operations.

Description:
    Applications using OTG mode may call this function to disable interrupts in preparation for shutdown or a change of operating mode.

Returns:
    None.

4.3.2.6   USBOTGPollRate

This call sets the USB OTG controllers poll rate when checking for the mode of the controller.

Prototype:
```
void
USBOTGPollRate(uint32_t ui32Index,
               uint32_t ui32PollRate)
```

Parameters:
    ui32Index specifies which USB controller to set the polling rate.
    ui32PollRate is the rate in milliseconds to poll for changes in the controller mode.

Description:
    This function is called to set the USB OTG libraries polling rate when checking the status of the cable. The *ui32PollRate* value used sets the rate in milliseconds that the USB OTG library will poll the cable to see if the controller should enter host mode. This value has no effect on device detection rate as the controller will detect being connected to a host controller automatically. The *ui32PollRate* can be set to 0 to disable polling. The USB OTG library can still function with the polling rate set to zero, however it will fail to detect host mode properly when no device is present at the end of the USB OTG B side of the cable.

Note:
    This function should only be called on devices that support OTG functionality.

Returns:
    None.

4.4 OTG Programming Examples

The USB library provides an example OTG application that can act as a mouse device or can support being a host to a mouse device. This application uses the USB HID mouse device support as well as the USB HID mouse host support that is provided with the USB library. To accomplish this the application must initialize the host stack to support a mouse device and initialize the device stack to operate as a mouse device. The application should always call USBStackModeSet() in order to set the mode of the stack to OTG and in order to receive callbacks for any mode changes.

Example: USBStackModeSet

```
USBStackModeSet(0, eUSBModeOTG, ModeCallback);
```

4.4.1 OTG Host Mode Initialization

The USB OTG library requires the application to initialize the host stack almost identically as is done for normal host mode. The only difference for the USB OTG configuration is that USBHC-DInit() is not called by the application and is handled by the call to USBOTGModeInit(). As with normal host mode, the application first registers the supported device drivers with a call to USBHC-DRegisterDrivers(). If the top level driver for a device requires any initialization it should be called as well. Then the application should configure the USB power pins EPEN and FAULT to allow the OTG stack to be able to power the device and detect fault conditions if required. At this point the host mode drivers and stack are ready for normal operation. Unlike normal host mode the application must periodically call USBOTGMain() instead of USBHCDMain() to allow non-interrupt based library code to run.

Example: Host Mode Configuration

```
//
// Register the host class drivers.
//
USBHCDRegisterDrivers(0, g_ppHostClassDrivers, g_ulNumHostClassDrivers);

//
// Configure the power pins for host mode.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);
GPIOPinTypeUSBDigital(GPIO_PORTH_BASE, GPIO_PIN_3 | GPIO_PIN_4);

//
// Initialize the power configuration. This sets the power enable signal
// to be active high and does not enable the power fault.
//
USBHCDPowerConfigInit(0, USBHCD_VBUS_AUTO_HIGH);

//
// This is the top level call to initialize a mouse driver.
//
USBHMouseOpen(MouseCallback, g_pucBuffer, MOUSE_MEMORY_SIZE);
```

4.4.2 OTG Device Mode Initialization

The USB OTG library requires the application to initialize the device stack identically to how it is done for normal device mode. This includes calling USBDCDInit() for custom classes or a specific

USB library provided device initialization function like USBDHIDMouseInit() in the example below. Once this is done the device mode stack is ready for normal operation in OTG mode.

Example: Device Mode Configuration

```
//
// This is the call to initialize the mouse device stack.
//
USBDHIDMouseInit(0, (tUSBDHIDMouseDevice *)&g_sMouseDevice);
```

4.4.3 OTG Mode Initialization

The only remaining initialization required after configuring the device and host stacks is to enable OTG mode with a call to USBOTGModeInit(). This function takes a parameter to specify the polling intervall for the OTG detection and also two other parameters that are the same as the ones passed to USBHCDInit() as it will pass the memory buffer provided in these two parameters directly to USBHCDInit(). Once this function returns the OTG stack will be operational and ready for new devices.

Example: Mouse Configuration

```
//
// Pass in the host stack buffer to the OTG stack.
//
USBOTGModeInit(0, g_pHCDPool, HCD_MEMORY_SIZE);
```

4.4.4 Application Interface

Once all the host, device and OTG initialization is complete, the USB library is ready to start normal operation. For device mode this simply means that the next time the device is connected to a host the application will receive a callback via the function provided with the USBStackModeSet() API. This will inform the application that a device has been detected. Because the USB bus is not be powered in idle mode, detection of host mode requires polling or actual session request signaling from a device in order to detect that host mode should be enabled. In the case where the USB controller is connected to the A side of the cable(default Host mode) this will check the cable and inform the application via the mode callback if a device is present. In the case where the USB controller is connected to the B side of the cable and an OTG device is on the A side, this call will request that the A side power the USB bus and enumerate the device. If nothing is present then the application will receive a callback indicating **eUSBModeNone** as the current mode was not able to be detected. The application will be informed of a device being disconnected or being disconnected as a device by receiving a callback that the USB library has returned to **eUSBModeNone** and possibly also from the USB library device or host library functions as well.

5 General Purpose Functions

5.1 Introduction

This chapter describes the set of USB library data types and functions which are of general use in the development of USB applications using this library These elements are not specific to USB host, device or On-The-Go (OTG) operation.

The functions and types described here fall into three main categories:

- Definitions of USB standard types (as found in Chapter 9 of the USB 2.0 specification) and functions to parse them.
- Functions relating to host/device mode switching for On-The-Go or dual mode applications.
- USB device class header files.

Source Code Overview

Source code and headers for the general-purpose USB functions can be found in the top level directory of the USB library tree, typically `DriverLib/usblib`.

`usblib.h`	The main header file for the USB library containing all data types and definitions which are common across the library as a whole. Prototypes for general-purpose API functions are also included. All clients of the USB Library should include this header.
`usb-ids.h`	The header file containing labels defining the Texas Instruments USB vendor ID (VID) and product IDs (PIDs) for each of the example devices provided in USB-capable evaluation kits.
`usbmsc.h`	The header file containing definitions specific to the USB Mass Storage Class.
`usbcdc.h`	The header file containing definitions specific to the USB Communication Device Class.
`usbhid.h`	The header file containing definitions specific to the USB Human Interface Device Class.
`usbdesc.c`	The source code for a set of functions allowing parsing of USB descriptors.
`usbbuffer.c`	The source code for a set of functions use to support buffering of USB endpoint data streams in some applications.

usbringbuf.c   The source code for a set of functions implementing simple ring buffers.

usbmode.c   The source code for a set of functions related to switching between host and
                device modes of operation.

usbtick.c   The source code for internal USB library tick handler functions. This file does
                not include any functions accessible by applications.

usblibpriv.h   The private header file used to share variables and definitions between the
                various components of the USB Library. Client applications must not include
                this header.

5.2   Function Definitions

Defines

- USB_DESC_ANY
- USBERR_DEV_RX_DATA_ERROR
- USBERR_DEV_RX_FIFO_FULL
- USBERR_DEV_RX_OVERRUN
- USBERR_HOST_EP0_ERROR
- USBERR_HOST_EP0_NAK_TO
- USBERR_HOST_IN_DATA_ERROR
- USBERR_HOST_IN_ERROR
- USBERR_HOST_IN_FIFO_FULL
- USBERR_HOST_IN_NAK_TO
- USBERR_HOST_IN_NOT_COMP
- USBERR_HOST_IN_PID_ERROR
- USBERR_HOST_IN_STALL
- USBERR_HOST_OUT_ERROR
- USBERR_HOST_OUT_NAK_TO
- USBERR_HOST_OUT_NOT_COMP
- USBERR_HOST_OUT_STALL

Typedefs

- tUSBCallback

Enumerations

- tUSBMode

Functions

- void [USB0DualModeIntHandler](void)
- [tDescriptorHeader](*) ∗ [USBDescGet](*) ([tDescriptorHeader](*) ∗psDesc, uint32_t ui32Size, uint32_t ui32Type, uint32_t ui32Index)
- [tInterfaceDescriptor](*) ∗ [USBDescGetInterface](*) ([tConfigDescriptor](*) ∗psConfig, uint32_t ui32Index, uint32_t ui32Alt)
- [tEndpointDescriptor](*) ∗ [USBDescGetInterfaceEndpoint](*) ([tInterfaceDescriptor](*) ∗psInterface, uint32_t ui32Index, uint32_t ui32Size)
- uint32_t [USBDescGetNum](*) ([tDescriptorHeader](*) ∗psDesc, uint32_t ui32Size, uint32_t ui32Type)
- uint32_t [USBDescGetNumAlternateInterfaces](*) ([tConfigDescriptor](*) ∗psConfig, uint8_t ui8InterfaceNumber)
- void [USBStackModeSet](*) (uint32_t ui32Index, [tUSBMode](*) iUSBMode, tUSBModeCallback pfn-Callback)

5.2.1 Detailed Description

This group of functions relates to standard USB descriptor parsing and host/device mode control. Source for these functions can be found in files `usbenum.c` and `usbmode.c`. Header file `usblib.h` contains prototypes for these functions along with all data type definitions which are not device or host specific.

5.2.2 Define Documentation

5.2.2.1 USB_DESC_ANY

Definition:
        #define USB_DESC_ANY

Description:
        The USB_DESC_ANY label is used as a wild card in several of the descriptor parsing APIs to determine whether or not particular search criteria should be ignored.

5.2.2.2 USBERR_DEV_RX_DATA_ERROR

Definition:
        #define USBERR_DEV_RX_DATA_ERROR

Description:
        The device detected a CRC error in received data.

5.2.2.3 USBERR_DEV_RX_FIFO_FULL

Definition:
        #define USBERR_DEV_RX_FIFO_FULL

Description:
The device receive FIFO is full.

5.2.2.4 USBERR_DEV_RX_OVERRUN

Definition:
```
#define USBERR_DEV_RX_OVERRUN
```

Description:
The device was unable to receive a packet from the host since the receive FIFO is full.

5.2.2.5 USBERR_HOST_EP0_ERROR

Definition:
```
#define USBERR_HOST_EP0_ERROR
```

Description:
The host failed to communicate with a device via an endpoint zero.

5.2.2.6 USBERR_HOST_EP0_NAK_TO

Definition:
```
#define USBERR_HOST_EP0_NAK_TO
```

Description:
The host received NAK on endpoint 0 for longer than the configured timeout.

5.2.2.7 USBERR_HOST_IN_DATA_ERROR

Definition:
```
#define USBERR_HOST_IN_DATA_ERROR
```

Description:
The host detected a CRC or bit-stuffing error (isochronous mode).

5.2.2.8 USBERR_HOST_IN_ERROR

Definition:
```
#define USBERR_HOST_IN_ERROR
```

Description:
The host failed to communicate with a device via an IN endpoint.

5.2.2.9   USBERR_HOST_IN_FIFO_FULL

Definition:
```
#define USBERR_HOST_IN_FIFO_FULL
```

Description:
The host receive FIFO is full.

5.2.2.10 USBERR_HOST_IN_NAK_TO

Definition:
```
#define USBERR_HOST_IN_NAK_TO
```

Description:
The host received NAK on an IN endpoint for longer than the specified timeout period (interrupt, bulk and control modes).

5.2.2.11 USBERR_HOST_IN_NOT_COMP

Definition:
```
#define USBERR_HOST_IN_NOT_COMP
```

Description:
The host did not receive a response from a device.

5.2.2.12 USBERR_HOST_IN_PID_ERROR

Definition:
```
#define USBERR_HOST_IN_PID_ERROR
```

Description:
The host received an invalid PID in a transaction.

5.2.2.13 USBERR_HOST_IN_STALL

Definition:
```
#define USBERR_HOST_IN_STALL
```

Description:
The host received a stall on an IN endpoint.

5.2.2.14 USBERR_HOST_OUT_ERROR

Definition:
```
#define USBERR_HOST_OUT_ERROR
```

Description:
The host failed to communicate with a device via an OUT endpoint.

5.2.2.15 USBERR_HOST_OUT_NAK_TO

Definition:
```
#define USBERR_HOST_OUT_NAK_TO
```

Description:
The host received NAK on an OUT endpoint for longer than the specified timeout period (bulk, interrupt and control modes).

5.2.2.16 USBERR_HOST_OUT_NOT_COMP

Definition:
```
#define USBERR_HOST_OUT_NOT_COMP
```

Description:
The host did not receive a response from a device (isochronous mode).

5.2.2.17 USBERR_HOST_OUT_STALL

Definition:
```
#define USBERR_HOST_OUT_STALL
```

Description:
The host received a stall on an OUT endpoint.

5.2.3   Typedef Documentation

5.2.3.1 tUSBCallback

Definition:
```
typedef uint32_t(* tUSBCallback)(void *pvCBData,
                                 uint32_t ui32Event,
                                 uint32_t ui32MsgParam,
                                 void *pvMsgData)
```

Description:
USB callback function.

Parameters:
pvCBData is the callback pointer associated with the instance generating the callback. This is a value provided by the client during initialization of the instance making the callback.
ui32Event is the identifier of the asynchronous event which is being notified to the client.
ui32MsgParam is an event-specific parameter.
pvMsgData is an event-specific data pointer.

A function pointer provided to the USB layer by the application which will be called to notify it of all asynchronous events relating to data transmission or reception. This callback is used by device class drivers and host pipe functions.

Returns:
>   Returns an event-dependent value.

5.2.4   Enumeration Documentation

5.2.4.1   tUSBMode

Description:
>   The operating mode required by the USB library client. This type is used by applications which wish to be able to switch between host and device modes by calling the USBStackModeSet() API.

Enumerators:
>   ***eUSBModeDevice***   Operate in USB device mode with active monitoring of VBUS and the ID pin must be pulled to a logic high value.
>
>   ***eUSBModeHost***   Operate in USB host mode with active monitoring of VBUS and the ID pin must be pulled to a logic low value.
>
>   ***eUSBModeOTG***   Operate as an On-The-Go device which requires both VBUS and ID to be connected directly to the USB controller from the USB connector.
>
>   ***eUSBModeNone***   A marker indicating that no USB mode has yet been set by the application.
>
>   ***eUSBModeForceHost***   Force host mode so that the VBUS and ID pins are not used or monitored by the USB controller.
>
>   ***eUSBModeForceDevice***   Forcing device mode so that the VBUS and ID pins are not used or monitored by the USB controller.

5.2.5   Function Documentation

5.2.5.1   USB0DualModeIntHandler

Steers USB interrupts from controller to the correct handler in the USB stack.

Prototype:
```
void
USB0DualModeIntHandler(void)
```

Description:
>   This interrupt handler is used in applications which require to operate in both host and device mode. It steers the USB hardware interrupt to the correct handler in the USB stack depending upon the current operating mode of the application, USB device or host.
>
>   For successful dual mode operation, an application must register USB0DualModeIntHandler() in the CPU vector table as the interrupt handler for the USB0 interrupt. This handler is responsible for steering interrupts to the device or host stack depending upon the chosen mode.

Note:
>   Devices which do not require dual mode capability should register either USB0DeviceIntHandler() or USB0HostIntHandler() instead. Registering USB0DualModeIntHandler() for a single mode application will result in an application binary larger than required since library functions for both USB operating modes will be included even though only one mode is actually required.

Returns:
>   None.

5.2.5.2   USBDescGet

Determines the number of individual descriptors of a particular type within a supplied buffer.

Prototype:
```
tDescriptorHeader *
USBDescGet(tDescriptorHeader *psDesc,
           uint32_t ui32Size,
           uint32_t ui32Type,
           uint32_t ui32Index)
```

Parameters:
>   ***psDesc*** points to the first byte of a block of standard USB descriptors.
>
>   ***ui32Size*** is the number of bytes of descriptor data found at pointer *psDesc*.
>
>   ***ui32Type*** identifies the type of descriptor that is to be found. If the value is **USB_DESC_ANY**, the function returns a pointer to the n-th descriptor regardless of type.
>
>   ***ui32Index*** is the zero based index of the descriptor whose pointer is to be returned. For example, passing value 1 in *ui32Index* returns the second matching descriptor.

Description:
>   Return a pointer to the n-th descriptor of a particular type found in the block of *ui32Size* bytes starting at *psDesc*.

Returns:
>   Returns a pointer to the header of the required descriptor if found or NULL otherwise.

5.2.5.3   USBDescGetInterface

Returns a pointer to the n-th interface descriptor in a configuration descriptor that applies to the supplied alternate setting number.

Prototype:
```
tInterfaceDescriptor *
USBDescGetInterface(tConfigDescriptor *psConfig,
                    uint32_t ui32Index,
                    uint32_t ui32Alt)
```

Parameters:
>   ***psConfig*** points to the first byte of a standard USB configuration descriptor.
>
>   ***ui32Index*** is the zero based index of the interface that is to be found. If *ui32Alt* is set to a value other than **USB_DESC_ANY**, this will be equivalent to the interface number being searched for.
>
>   ***ui32Alt*** is the alternate setting number which is to be searched for. If this value is **USB_DESC_ANY**, the alternate setting is ignored and all interface descriptors are considered in the search.

Description:

Return a pointer to the n-th interface descriptor found in the supplied configuration descriptor. If *ui32Alt* is not **USB_DESC_ANY**, only interface descriptors which are part of the supplied alternate setting are considered in the search otherwise all interface descriptors are considered.

Note that, although alternate settings can be applied on an interface-by- interface basis, the number of interfaces offered is fixed for a given config descriptor. Hence, this function will correctly find the unique interface descriptor for that interface's alternate setting number *ui32Alt* if *ui32Index* is set to the required interface number and *ui32Alt* is set to a valid alternate setting number for that interface.

Returns:

Returns a pointer to the required interface descriptor if found or NULL otherwise.

5.2.5.4 USBDescGetInterfaceEndpoint

Return a pointer to the n-th endpoint descriptor in the supplied interface descriptor.

Prototype:

```
tEndpointDescriptor *
USBDescGetInterfaceEndpoint(tInterfaceDescriptor *psInterface,
                            uint32_t ui32Index,
                            uint32_t ui32Size)
```

Parameters:

psInterface points to the first byte of a standard USB interface descriptor.

ui32Index is the zero based index of the endpoint that is to be found.

ui32Size contains the maximum number of bytes that the function may search beyond *psInterface* while looking for the requested endpoint descriptor.

Description:

Return a pointer to the n-th endpoint descriptor found in the supplied interface descriptor. If the *ui32Index* parameter is invalid (greater than or equal to the bNumEndpoints field of the interface descriptor) or the endpoint cannot be found within *ui32Size* bytes of the interface descriptor pointer, the function will return NULL.

Note that, although the USB 2.0 specification states that endpoint descriptors must follow the interface descriptor that they relate to, it also states that device specific descriptors should follow any standard descriptor that they relate to. As a result, we cannot assume that each interface descriptor will be followed by nothing but an ordered list of its own endpoints and, hence, the function needs to be provided *ui32Size* to limit the search range.

Returns:

Returns a pointer to the requested endpoint descriptor if found or NULL otherwise.

5.2.5.5 USBDescGetNum

Determines the number of individual descriptors of a particular type within a supplied buffer.

Prototype:

```
uint32_t
USBDescGetNum(tDescriptorHeader *psDesc,
```

```
                                 uint32_t ui32Size,
                                 uint32_t ui32Type)
```

Parameters:
   psDesc points to the first byte of a block of standard USB descriptors.

   ui32Size is the number of bytes of descriptor data found at pointer *psDesc*.

   ui32Type identifies the type of descriptor that is to be counted. If the value is
      USB_DESC_ANY, the function returns the total number of descriptors regardless of type.

Description:
   This function can be used to count the number of descriptors of a particular type within a block
   of descriptors. The caller can provide a specific type value which the function matches against
   the second byte of each descriptor or, alternatively, can specify **USB_DESC_ANY** to have the
   function count all descriptors regardless of their type.

Returns:
   Returns the number of descriptors found in the supplied block of data.

5.2.5.6   USBDescGetNumAlternateInterfaces

Determines the number of different alternate configurations for a given interface within a configuration descriptor.

Prototype:
```
uint32_t
USBDescGetNumAlternateInterfaces(tConfigDescriptor *psConfig,
                                 uint8_t ui8InterfaceNumber)
```

Parameters:
   psConfig points to the first byte of a standard USB configuration descriptor.

   ui8InterfaceNumber is the interface number for which the number of alternate configurations
      is to be counted.

Description:
   This function can be used to count the number of alternate settings for a specific interface
   within a configuration.

Returns:
   Returns the number of alternate versions of the specified interface or 0 if the interface number
   supplied cannot be found in the config descriptor.

5.2.5.7   USBStackModeSet

Allows dual mode application to switch between USB device and host modes and provides a
method to force the controller into the desired mode.

Prototype:
```
void
USBStackModeSet(uint32_t ui32Index,
                tUSBMode iUSBMode,
                tUSBModeCallback pfnCallback)
```

Parameters:

   ui32Index specifies the USB controller whose mode of operation is to be set. This parameter must be set to 0.

   iUSBMode indicates the mode that the application wishes to operate in. Valid values are **eUSBModeDevice** to operate as a USB device and **eUSBModeHost** to operate as a USB host.

   pfnCallback is a pointer to a function which the USB library will call each time the mode is changed to indicate the new operating mode. In cases where *iUSBMode* is set to either **eUSBModeDevice** or **eUSBModeHost**, the callback will be made immediately to allow the application to perform any host or device specific initialization.

Description:

   This function allows a USB application that can operate in host or device mode to indicate to the USB stack the mode that it wishes to use. The caller is responsible for cleaning up the interface and removing itself from the bus prior to making this call and reconfiguring afterwards. The *pfnCallback* function can be a NULL(0) value to indicate that no notification is required.

   For successful dual mode mode operation, an application must register USB0DualModeIntHandler() as the interrupt handler for the USB0 interrupt. This handler is responsible for steering interrupts to the device or host stack depending upon the chosen mode. Devices which do not require dual mode capability should register either *USB0DeviceIntHandler()* or *USB0HostIntHandler()* instead. Registering *USB0DualModeIntHandler()* for a single mode application will result in an application binary larger than required since library functions for both USB operating modes will be included even though only one mode is required.

   Single mode applications (those offering exclusively USB device or USB host functionality) are only required to call this function if they need to force the mode of the controller to Host or Device mode. This is usually in the event that the application needs to reused the USBVBUS and/or USBID pins as GPIOs.

Returns:

   None.

5.3   USB Events

Data Structures

- tEventInfo

Defines

- USB_EVENT_COMP_CONFIG
- USB_EVENT_COMP_EP_CHANGE
- USB_EVENT_COMP_IFACE_CHANGE
- USB_EVENT_COMP_STR_CHANGE
- USB_EVENT_CONFIG_CHANGE
- USB_EVENT_CONNECTED
- USB_EVENT_DATA_REMAINING

- USB_EVENT_DISCONNECTED
- USB_EVENT_ERROR
- USB_EVENT_LPM_ERROR
- USB_EVENT_LPM_RESUME
- USB_EVENT_LPM_SLEEP
- USB_EVENT_POWER_DISABLE
- USB_EVENT_POWER_ENABLE
- USB_EVENT_POWER_FAULT
- USB_EVENT_REQUEST_BUFFER
- USB_EVENT_RESUME
- USB_EVENT_RX_AVAILABLE
- USB_EVENT_SCHEDULER
- USB_EVENT_SOF
- USB_EVENT_STALL
- USB_EVENT_SUSPEND
- USB_EVENT_TX_COMPLETE
- USB_EVENT_UNKNOWN_CONNECTED

5.3.1 Detailed Description

defined events.

5.3.2 Data Structure Documentation

5.3.2.1 tEventInfo

Definition:
```
typedef struct
{
    uint32_t ui32Event;
    uint32_t ui32Instance;
}
tEventInfo
```

Members:
ui32Event One of the USB_EVENT_ values.
ui32Instance The caller supplied instance value that is passed to event handlers.

Description:
This structure is used to return generic event based information to an application. The following events are currently supported: USB_EVENT_CONNECTED, USB_EVENT_DISCONNECTED, USB_EVENT_POWER_FAULT, USB_EVENT_POWER_FAULT, USB_EVENT_POWER_ENABLE, USB_EVENT_POWER_DISABLE and USB_EVENT_SOF.

5.3.3 Define Documentation

5.3.3.1 USB_EVENT_COMP_CONFIG

Definition:
```
#define USB_EVENT_COMP_CONFIG
```

Description:
This define is used with a device class's pfnDeviceHandler handler function to indicate that the USB library has changed the configuration descriptor. This allows the class to make final adjustments to the configuration descriptor. This event is typically due to the class being included in a composite device.

The *pvInstance* is a pointer to an instance of the device being accessed.

The *ui32Request* is USB_EVENT_COMP_CONFIG.

The *pvRequestData* is a pointer to the beginning of the configuration descriptor for the device instance.

5.3.3.2 USB_EVENT_COMP_EP_CHANGE

Definition:
```
#define USB_EVENT_COMP_EP_CHANGE
```

Description:
This define is used with a device class's pfnDeviceHandler handler function to indicate that the USB library has changed the endpoint number. This event is typically due to the class being included in a composite device.

The *pvInstance* is a pointer to an instance of the device being accessed.

The *ui32Request* is USB_EVENT_COMP_EP_CHANGE.

The *pvRequestData* is a pointer to a two byte array where the first value is the old endpoint number and the second is the new endpoint number. The endpoint numbers should be exactly as USB specification defines them and bit 7 set indicates an IN endpoint and bit 7 clear indicates an OUT endpoint.

5.3.3.3 USB_EVENT_COMP_IFACE_CHANGE

Definition:
```
#define USB_EVENT_COMP_IFACE_CHANGE
```

Description:
This define is used with a device class's pfnDeviceHandler handler function to indicate that the USB library has changed the interface number. This event is typically due to the class being included in a composite device.

The *pvInstance* is a pointer to an instance of the device being accessed.

The *ui32Request* is USB_EVENT_COMP_IFACE_CHANGE.

The *pvRequestData* is a pointer to a two byte array where the first value is the old interface number and the second is the new interface number.

5.3.3.4   USB_EVENT_COMP_STR_CHANGE

Definition:

```
#define USB_EVENT_COMP_STR_CHANGE
```

Description:

This define is used with a device class's pfnDeviceHandler handler function to indicate that the USB library has changed the string index number for a string. This event is typically due to the class being included in a composite device.

The *pvInstance* is a pointer to an instance of the device being accessed.

The *ui32Request* is USB_EVENT_COMP_STR_CHANGE.

The *pvRequestData* is a pointer to a two byte array where the first value is the old string index and the second is the new string index.

5.3.3.5   USB_EVENT_CONFIG_CHANGE

Definition:

```
#define USB_EVENT_CONFIG_CHANGE
```

Description:

This event occurs when a device has been issued a configuration change.

5.3.3.6   USB_EVENT_CONNECTED

Definition:

```
#define USB_EVENT_CONNECTED
```

Description:

The device is now attached to a USB host and ready to begin sending and receiving data (used by device classes only).

5.3.3.7   USB_EVENT_DATA_REMAINING

Definition:

```
#define USB_EVENT_DATA_REMAINING
```

Description:

This event is sent by a lower layer to inquire about the amount of unprocessed data buffered in the layers above. It is used in cases where a low level driver needs to ensure that all preceding data has been processed prior to performing some action or making some notification. Clients receiving this event should return the number of bytes of data that are unprocessed or 0 if no outstanding data remains.

5.3.3.8 USB_EVENT_DISCONNECTED

Definition:

```
#define USB_EVENT_DISCONNECTED
```

Description:

The device has been disconnected from the USB host (used by device classes only).

Note:

In device mode, the USB_EVENT_DISCONNECTED will not be reported if the MCU's PB1/USB0VBUS pin is connected to a fixed +5 Volts rather than directly to the VBUS pin on the USB connector.

5.3.3.9 USB_EVENT_ERROR

Definition:

```
#define USB_EVENT_ERROR
```

Description:

An error has been reported on the channel or pipe. The *ui32MsgValue* parameter indicates the source(s) of the error and is the logical OR combination of "USBERR_" flags defined below.

5.3.3.10 USB_EVENT_LPM_ERROR

Definition:

```
#define USB_EVENT_LPM_ERROR
```

Description:

This event occurs when a device has responded with a NYET to an LPM request because LPM responses were disabled.

5.3.3.11 USB_EVENT_LPM_RESUME

Definition:

```
#define USB_EVENT_LPM_RESUME
```

Description:

This event occurs when a device is resumed from LPM sleep mode.

5.3.3.12 USB_EVENT_LPM_SLEEP

Definition:

```
#define USB_EVENT_LPM_SLEEP
```

Description:

This event occurs when a device enters LPM sleep mode.

5.3.3.13 USB_EVENT_POWER_DISABLE

Definition:
```
#define USB_EVENT_POWER_DISABLE
```

Description:
The controller needs power removed, This is only generated on OTG parts if automatic power control is disabled.

5.3.3.14 USB_EVENT_POWER_ENABLE

Definition:
```
#define USB_EVENT_POWER_ENABLE
```

Description:
The controller has detected a A-Side cable and needs power applied This is only generated on OTG parts if automatic power control is disabled.

5.3.3.15 USB_EVENT_POWER_FAULT

Definition:
```
#define USB_EVENT_POWER_FAULT
```

Description:
The host detected a power fault condition.

5.3.3.16 USB_EVENT_REQUEST_BUFFER

Definition:
```
#define USB_EVENT_REQUEST_BUFFER
```

Description:
This event is sent by a lower layer supporting DMA to request a buffer in which the next received packet may be stored. The *ui32MsgValue* parameter indicates the maximum size of packet that can be received in this channel and *pvMsgData* points to storage which should be written with the returned buffer pointer. The return value from the callback should be the size of the buffer allocated (which may be less than the maximum size passed in *ui32MsgValue* if the client knows that fewer bytes are expected to be received) or 0 if no buffer is being returned.

5.3.3.17 USB_EVENT_RESUME

Definition:
```
#define USB_EVENT_RESUME
```

Description:
The bus has left suspend state.

5.3.3.18 USB_EVENT_RX_AVAILABLE

Definition:
```
#define USB_EVENT_RX_AVAILABLE
```

Description:
Data has been received and is in the buffer provided or is ready to be read from the FIFO. If the *pvMsgData* value is 0 then the *ui32MsgParam* value contains the amount of data in bytes ready to be read from the device. If the *pvMsgData* value is not 0 then *pvMsgData* is a pointer to the data that was read and *ui32MsgParam* is the number of valid bytes in the array pointed to by *pvMsgData*.

5.3.3.19 USB_EVENT_SCHEDULER

Definition:
```
#define USB_EVENT_SCHEDULER
```

Description:
A scheduler event has occurred.

5.3.3.20 USB_EVENT_SOF

Definition:
```
#define USB_EVENT_SOF
```

Description:
A start of frame event has occurred. This event is disabled by default and must be enabled via a call from the application to USBHCDEventEnable().

5.3.3.21 USB_EVENT_STALL

Definition:
```
#define USB_EVENT_STALL
```

Description:
A device or host has detected a stall condition.

5.3.3.22 USB_EVENT_SUSPEND

Definition:
```
#define USB_EVENT_SUSPEND
```

Description:
The bus has entered suspend state.

5.3.3.23 USB_EVENT_TX_COMPLETE

Definition:

```
#define USB_EVENT_TX_COMPLETE
```

Description:

Data has been sent and acknowledged. If this event is received via the USB buffer callback, the *ui32MsgValue* parameter indicates the number of bytes from the transmit buffer that have been successfully transmitted and acknowledged.

5.3.3.24 USB_EVENT_UNKNOWN_CONNECTED

Definition:

```
#define USB_EVENT_UNKNOWN_CONNECTED
```

Description:

An unknown device is now attached to a USB host. This value is only valid for the generic event handler and not other device handlers. It is useful for applications that want to know when an unknown device is connected and what the class is of the unknown device.

The *ui32Instance* is the device instance for the unknown device.

5.4   USB Chapter 9 Definitions

Data Structures

- tConfigDescriptor
- tDescriptorHeader
- tDeviceDescriptor
- tDeviceQualifierDescriptor
- tEndpointDescriptor
- tInterfaceDescriptor
- tString0Descriptor
- tStringDescriptor
- tUSBRequest

Defines

- NEXT_USB_DESCRIPTOR(ptr)
- USB3Byte(ui32Value)
- USBLong(ui32Value)
- USBShort(ui16Value)

5.4.1 Detailed Description

This section describes the various data structures and labels relating to standard USB descriptors and requests as defined in chapter 9 of the USB 2.0 specification. These definitions can be found in `usblib.h`.

For ease of use alongside the USB specification, members of the structures defined here are named to according to the equivalent field in the USB documentation. Note that this convention departs from the naming convention applied to all other Texas Instruments data types.

It is important to be aware that all the structures described in this section are byte packed. Appropriate typedef modifiers are included in `usblib.h` to ensure the correct packing for all currently-supported toolchains.

The USB 2.0 specification may be downloaded from the USB Implementers Forum (USB-IF) web site at `http://www.usb.org/developers/docs/`.

5.4.2 Data Structure Documentation

5.4.2.1 tConfigDescriptor

Definition:
```
typedef struct
{
    uint8_t bLength;
    uint8_t bDescriptorType;
    uint16_t wTotalLength;
    uint8_t bNumInterfaces;
    uint8_t bConfigurationValue;
    uint8_t iConfiguration;
    uint8_t bmAttributes;
    uint8_t bMaxPower;
}
tConfigDescriptor
```

Members:
> **bLength** The length of this descriptor in bytes. All configuration descriptors are 9 bytes long.
>
> **bDescriptorType** The type of the descriptor. For a configuration descriptor, this will be USB_DTYPE_CONFIGURATION (2).
>
> **wTotalLength** The total length of data returned for this configuration. This includes the combined length of all descriptors (configuration, interface, endpoint and class- or vendor-specific) returned for this configuration.
>
> **bNumInterfaces** The number of interface supported by this configuration.
>
> **bConfigurationValue** The value used as an argument to the SetConfiguration standard request to select this configuration.
>
> **iConfiguration** The index of a string descriptor describing this configuration.
>
> **bmAttributes** Attributes of this configuration.
>
> **bMaxPower** The maximum power consumption of the USB device from the bus in this configuration when the device is fully operational. This is expressed in units of 2mA so, for example, 100 represents 200mA.

Description:
> This structure describes the USB configuration descriptor as defined in USB 2.0 specification section 9.6.3. This structure also applies to the USB other speed configuration descriptor defined in section 9.6.4.

5.4.2.2   tDescriptorHeader

Definition:
```
typedef struct
{
    uint8_t bLength;
    uint8_t bDescriptorType;
}
tDescriptorHeader
```

Members:
> ***bLength*** The length of this descriptor (including this length byte) expressed in bytes.
>
> ***bDescriptorType*** The type identifier of the descriptor whose information follows. For standard descriptors, this field could contain, for example, USB_DTYPE_DEVICE to identify a device descriptor or USB_DTYPE_ENDPOINT to identify an endpoint descriptor.

Description:
> This structure describes a generic descriptor header. These fields are to be found at the beginning of all valid USB descriptors.

5.4.2.3   tDeviceDescriptor

Definition:
```
typedef struct
{
    uint8_t bLength;
    uint8_t bDescriptorType;
    uint16_t bcdUSB;
    uint8_t bDeviceClass;
    uint8_t bDeviceSubClass;
    uint8_t bDeviceProtocol;
    uint8_t bMaxPacketSize0;
    uint16_t idVendor;
    uint16_t idProduct;
    uint16_t bcdDevice;
    uint8_t iManufacturer;
    uint8_t iProduct;
    uint8_t iSerialNumber;
    uint8_t bNumConfigurations;
}
tDeviceDescriptor
```

Members:
> ***bLength*** The length of this descriptor in bytes. All device descriptors are 18 bytes long.
>
> ***bDescriptorType*** The type of the descriptor. For a device descriptor, this will be USB_DTYPE_DEVICE (1).

bcdUSB The USB Specification Release Number in BCD format. For USB 2.0, this will be 0x0200.

bDeviceClass The device class code.

bDeviceSubClass The device subclass code. This value qualifies the value found in the bDeviceClass field.

bDeviceProtocol The device protocol code. This value is qualified by the values of bDeviceClass and bDeviceSubClass.

bMaxPacketSize0 The maximum packet size for endpoint zero. Valid values are 8, 16, 32 and 64.

idVendor The device Vendor ID (VID) as assigned by the USB-IF.

idProduct The device Product ID (PID) as assigned by the manufacturer.

bcdDevice The device release number in BCD format.

iManufacturer The index of a string descriptor describing the manufacturer.

iProduct The index of a string descriptor describing the product.

iSerialNumber The index of a string descriptor describing the device's serial number.

bNumConfigurations The number of possible configurations offered by the device. This field indicates the number of distinct configuration descriptors that the device offers.

Description:

This structure describes the USB device descriptor as defined in USB 2.0 specification section 9.6.1.

5.4.2.4 tDeviceQualifierDescriptor

Definition:

```
typedef struct
{
    uint8_t bLength;
    uint8_t bDescriptorType;
    uint16_t bcdUSB;
    uint8_t bDeviceClass;
    uint8_t bDeviceSubClass;
    uint8_t bDeviceProtocol;
    uint8_t bMaxPacketSize0;
    uint8_t bNumConfigurations;
    uint8_t bReserved;
}
tDeviceQualifierDescriptor
```

Members:

bLength The length of this descriptor in bytes. All device qualifier descriptors are 10 bytes long.

bDescriptorType The type of the descriptor. For a device descriptor, this will be USB_DTYPE_DEVICE_QUAL (6).

bcdUSB The USB Specification Release Number in BCD format. For USB 2.0, this will be 0x0200.

bDeviceClass The device class code.

bDeviceSubClass The device subclass code. This value qualifies the value found in the bDeviceClass field.

bDeviceProtocol The device protocol code. This value is qualified by the values of bDevice-Class and bDeviceSubClass.

bMaxPacketSize0 The maximum packet size for endpoint zero when operating at a speed other than high speed.

bNumConfigurations The number of other-speed configurations supported.

bReserved Reserved for future use. Must be set to zero.

Description:

This structure describes the USB device qualifier descriptor as defined in the USB 2.0 specification, section 9.6.2.

5.4.2.5 tEndpointDescriptor

Definition:

```
typedef struct
{
    uint8_t bLength;
    uint8_t bDescriptorType;
    uint8_t bEndpointAddress;
    uint8_t bmAttributes;
    uint16_t wMaxPacketSize;
    uint8_t bInterval;
}
tEndpointDescriptor
```

Members:

bLength The length of this descriptor in bytes. All endpoint descriptors are 7 bytes long.

bDescriptorType The type of the descriptor. For an endpoint descriptor, this will be USB_DTYPE_ENDPOINT (5).

bEndpointAddress The address of the endpoint. This field contains the endpoint number ORed with flag USB_EP_DESC_OUT or USB_EP_DESC_IN to indicate the endpoint direction.

bmAttributes The endpoint transfer type, USB_EP_ATTR_CONTROL, USB_EP_ATTR_ISOC, USB_EP_ATTR_BULK or USB_EP_ATTR_INT and, if isochronous, additional flags indicating usage type and synchronization method.

wMaxPacketSize The maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. For high speed isochronous or interrupt endpoints, bits 11 and 12 are used to pass additional information.

bInterval The polling interval for data transfers expressed in frames or micro frames depending upon the operating speed.

Description:

This structure describes the USB endpoint descriptor as defined in USB 2.0 specification section 9.6.6.

5.4.2.6 tInterfaceDescriptor

Definition:

```
typedef struct
{
```

```
        uint8_t bLength;
        uint8_t bDescriptorType;
        uint8_t bInterfaceNumber;
        uint8_t bAlternateSetting;
        uint8_t bNumEndpoints;
        uint8_t bInterfaceClass;
        uint8_t bInterfaceSubClass;
        uint8_t bInterfaceProtocol;
        uint8_t iInterface;
    }
    tInterfaceDescriptor
```

Members:
    bLength The length of this descriptor in bytes. All interface descriptors are 9 bytes long.

    bDescriptorType The type of the descriptor. For an interface descriptor, this will be USB_DTYPE_INTERFACE (4).

    bInterfaceNumber The number of this interface. This is a zero based index into the array of concurrent interfaces supported by this configuration.

    bAlternateSetting The value used to select this alternate setting for the interface defined in bInterfaceNumber.

    bNumEndpoints The number of endpoints used by this interface (excluding endpoint zero).

    bInterfaceClass The interface class code as assigned by the USB-IF.

    bInterfaceSubClass The interface subclass code as assigned by the USB-IF.

    bInterfaceProtocol The interface protocol code as assigned by the USB-IF.

    iInterface The index of a string descriptor describing this interface.

Description:
    This structure describes the USB interface descriptor as defined in USB 2.0 specification section 9.6.5.

5.4.2.7 tString0Descriptor

Definition:
```
    typedef struct
    {
        uint8_t bLength;
        uint8_t bDescriptorType;
        uint16_t wLANGID[1];
    }
    tString0Descriptor
```

Members:
    bLength The length of this descriptor in bytes. This value will vary depending upon the number of language codes provided in the descriptor.

    bDescriptorType The type of the descriptor. For a string descriptor, this will be USB_DTYPE_STRING (3).

    wLANGID The language code (LANGID) for the first supported language. Note that this descriptor may support multiple languages, in which case, the number of elements in the wLANGID array will increase and bLength will be updated accordingly.

Description:
This structure describes the USB string descriptor for index 0 as defined in USB 2.0 specification section 9.6.7. Note that the number of language IDs is variable and can be determined by examining bLength. The number of language IDs present in the descriptor is given by ((bLength - 2) / 2).

5.4.2.8   tStringDescriptor

Definition:
```
typedef struct
{
    uint8_t bLength;
    uint8_t bDescriptorType;
    uint8_t bString;
}
tStringDescriptor
```

Members:
> **bLength** The length of this descriptor in bytes. This value will be 2 greater than the number of bytes comprising the UNICODE string that the descriptor contains.
>
> **bDescriptorType** The type of the descriptor.   For a string descriptor, this will be USB_DTYPE_STRING (3).
>
> **bString** The first byte of the UNICODE string. This string is not NULL terminated. Its length (in bytes) can be computed by subtracting 2 from the value in the bLength field.

Description:
This structure describes the USB string descriptor for all string indexes other than 0 as defined in USB 2.0 specification section 9.6.7.

5.4.2.9   tUSBRequest

Definition:
```
typedef struct
{
    uint8_t bmRequestType;
    uint8_t bRequest;
    uint16_t wValue;
    uint16_t wIndex;
    uint16_t wLength;
}
tUSBRequest
```

Members:
> **bmRequestType** Determines the type and direction of the request.
>
> **bRequest** Identifies the specific request being made.
>
> **wValue** Word-sized field that varies according to the request.
>
> **wIndex** Word-sized field that varies according to the request; typically used to pass an index or offset.
>
> **wLength** The number of bytes to transfer if there is a data stage to the request.

Description:
The standard USB request header as defined in section 9.3 of the USB 2.0 specification.

5.4.3   Define Documentation

5.4.3.1   NEXT_USB_DESCRIPTOR

Traverse to the next USB descriptor in a block.

Definition:
```
#define NEXT_USB_DESCRIPTOR(ptr)
```

Parameters:
ptr points to the first byte of a descriptor in a block of USB descriptors.

Description:
This macro aids in traversing lists of descriptors by returning a pointer to the next descriptor in the list given a pointer to the current one.

Returns:
Returns a pointer to the next descriptor in the block following *ptr*.

5.4.3.2   USB3Byte

Write a 24-bit value to a USB descriptor block.

Definition:
```
#define USB3Byte(ui32Value)
```

Parameters:
ui32Value is the 24-bit value that to write to the descriptor.

Description:
This helper macro is used in descriptor definitions to write three-byte values. Since the configuration descriptor contains all interface and endpoint descriptors in a contiguous block of memory, these descriptors are typically defined using an array of bytes rather than as packed structures.

Returns:
Not a function.

5.4.3.3   USBLong

Write a 32-bit value to a USB descriptor block.

Definition:
```
#define USBLong(ui32Value)
```

Parameters:
ui32Value is the 32-bit value that to write to the descriptor.

Description:
This helper macro is used in descriptor definitions to write four-byte values. Since the configuration descriptor contains all interface and endpoint descriptors in a contiguous block of memory, these descriptors are typically defined using an array of bytes rather than as packed structures.

Returns:
Not a function.

5.4.3.4 USBShort

Write a 16-bit value to a USB descriptor block.

Definition:
```
#define USBShort(ui16Value)
```

Parameters:
ui16Value is the 16-bit value to write to the descriptor.

Description:
This helper macro is used in descriptor definitions to write two-byte values. Since the configuration descriptor contains all interface and endpoint descriptors in a contiguous block of memory, these descriptors are typically defined using an array of bytes rather than as packed structures.

Returns:
Not a function.

5.5 USB Buffer and Ring Buffer APIs

Data Structures

- tUSBBuffer
- tUSBRingBufObject

Defines

- USBLIB_FEATURE_CPUCLK
- USBLIB_FEATURE_LPM
- USBLIB_FEATURE_POWER
- USBLIB_FEATURE_USBPLL
- USBLIB_FEATURE_USBULPI

Typedefs

- tUSBPacketAvailable
- tUSBPacketTransfer

Functions

- void ∗ USBBufferCallbackDataSet (tUSBBuffer ∗psBuffer, void ∗pvCBData)
- uint32_t USBBufferDataAvailable (const tUSBBuffer ∗psBuffer)
- void USBBufferDataRemoved (const tUSBBuffer ∗psBuffer, uint32_t ui32Length)
- void USBBufferDataWritten (const tUSBBuffer ∗psBuffer, uint32_t ui32Length)
- uint32_t USBBufferEventCallback (void ∗pvCBData, uint32_t ui32Event, uint32_t ui32MsgValue, void ∗pvMsgData)
- void USBBufferFlush (const tUSBBuffer ∗psBuffer)
- void USBBufferInfoGet (const tUSBBuffer ∗psBuffer, tUSBRingBufObject ∗psRingBuf)
- const tUSBBuffer ∗ USBBufferInit (tUSBBuffer ∗psBuffer)
- uint32_t USBBufferRead (const tUSBBuffer ∗psBuffer, uint8_t ∗pui8Data, uint32_t ui32Length)
- uint32_t USBBufferSpaceAvailable (const tUSBBuffer ∗psBuffer)
- uint32_t USBBufferWrite (const tUSBBuffer ∗psBuffer, const uint8_t ∗pui8Data, uint32_t ui32Length)
- void USBBufferZeroLengthPacketInsert (const tUSBBuffer ∗psBuffer, bool bSendZLP)
- void USBRingBufAdvanceRead (tUSBRingBufObject ∗psUSBRingBuf, uint32_t ui32NumBytes)
- void USBRingBufAdvanceWrite (tUSBRingBufObject ∗psUSBRingBuf, uint32_t ui32NumBytes)
- uint32_t USBRingBufContigFree (tUSBRingBufObject ∗psUSBRingBuf)
- uint32_t USBRingBufContigUsed (tUSBRingBufObject ∗psUSBRingBuf)
- bool USBRingBufEmpty (tUSBRingBufObject ∗psUSBRingBuf)
- void USBRingBufFlush (tUSBRingBufObject ∗psUSBRingBuf)
- uint32_t USBRingBufFree (tUSBRingBufObject ∗psUSBRingBuf)
- bool USBRingBufFull (tUSBRingBufObject ∗psUSBRingBuf)
- void USBRingBufInit (tUSBRingBufObject ∗psUSBRingBuf, uint8_t ∗pui8Buf, uint32_t ui32Size)
- void USBRingBufRead (tUSBRingBufObject ∗psUSBRingBuf, uint8_t ∗pui8Data, uint32_t ui32Length)
- uint8_t USBRingBufReadOne (tUSBRingBufObject ∗psUSBRingBuf)
- uint32_t USBRingBufSize (tUSBRingBufObject ∗psUSBRingBuf)
- uint32_t USBRingBufUsed (tUSBRingBufObject ∗psUSBRingBuf)
- void USBRingBufWrite (tUSBRingBufObject ∗psUSBRingBuf, const uint8_t ∗pui8Data, uint32_t ui32Length)
- void USBRingBufWriteOne (tUSBRingBufObject ∗psUSBRingBuf, uint8_t ui8Data)

5.5.1 Detailed Description

At the lowest level, USB communication is packet-based with the size of each packet dependent upon the configuration of the USB endpoint. In addition, when a packet is in transit, no more data may be sent on that endpoint until the transmission completes so state machines are required to ensure that data is only sent when it is safe to do so.

This model is suitable for some applications but in other cases a simple read/write model allowing arbitrarily sized blocks of data to be received or transmitted at times suitable to the application is more appropriate. The USB buffer API allows an application to chose this type of operation when used in conjunction with particular host- or device-class drivers.

A USB buffer provides a unidirectional buffer for a single endpoint and may be configured for operation as either a receive buffer (accepting data from the USB controller and passing it to an application) or a transmit buffer (accepting data from the application and passing it to the USB controller for transmission). In each case, the buffer handles all packetization or depacketization of data and allows the application to read or write arbitrarily-sized blocks of data (subject to the space limitations in the buffer, of course) at times suitable to it.

Each USB buffer makes use of a ring buffer object to store the buffered data. The ring buffer object is not USB-specific and does not interact directly with any USB drivers but the API is made available since the functionality may be useful to an application in areas outside USB communication, for example to buffer data from a UART or other peripheral. If attempting to buffer a USB data stream, however, the USB buffer API should be used since it handles the USB driver-side interaction on behalf of the application. An application must not mix calls to the two APIs for the same object - if using a USB buffer, only APIs of the form USBBufferXxx() should be used to access that buffer and, similarly, if using a plain ring buffer, only USBRingBufXxx() calls must be used.

Source for the USB buffer and ring buffer functions can be found in files `usbbuffer.c` and `usbringbuf.c`. Header file `usblib.h` contains prototypes and data type definitions for these functions.

5.5.2 Using USB Buffers

The USB buffer object is designed to allow insertion between a USB device class driver and the device application or between the USB host controller driver and a host class driver in an application- and class-independent way. Driver data transfer APIs all use a common prototype as do event callbacks so the USB buffer is inserted into the data path using driver function and instance pointers provided in static structures during application initialization. This method has the advantage that the USB buffer is not directly dependent upon any specific functions in the USB library and, as a result, using it does not pull extraneous code into the final application image.

During operation, events from the layer below the buffer are inspected in the buffer's event handler. If they are unrecognized or have no effect on the flow of data, they are passed to the higher layer unaltered. If they relate to data flow, however, the buffer intercepts them and performs the necessary actions to transmit or receive data before passing appropriate events to the layer above.

To insert a buffer for use on a transmit or receive channel or pipe, a `tUSBBuffer` structure must be initialized as follows.

 `bTransmitBuffer` This field must be set to true if the buffer is passing data from the application code to the USB controller or false if passing data from the USB controller to the application.

 `pfnCallback` This field should point to the event handler callback function in the application code. Notifications of asynchronous events relating to the buffer will be made by calls to this function.

 `pvCBData` The callback data pointer written to this field will be passed as the first parameter on all future calls to the application event handler (set in pfnCallback). Typically an application will set this pointer to some value allowing it easy access to data associated with the channel, for example a pointer to an internal instance data structure. The actual content is application specific and the USB buffer merely stores the value and passes it back to the caller when required.

`pfnTransfer`   This field informs the USB buffer of the function to call whenever data is to be transferred between the buffer and the lower layer. This is used to transmit a packet of data if this is a transmit buffer (bTransmitBuffer set to true) or to receive a packet of data if this is a receive buffer (bTransmitBuffer set to false). Taking the example of a buffer used to transmit data to the USB generic bulk device class driver, this would be set to point to USBDBulkPacketWrite(). A receive buffer used with the same driver would have this field set to point to USBDBulkPacketRead().

`pfnAvailable`   For a transmit buffer, this function pointer must be set to point to the lower layer function that can be called to determine whether the relevant USB endpoint or pipe is ready to accept a new packet for transmission. For a receive buffer, this field points to the function that should be called to determine the size of buffer required to read a newly-received packet. Using the same example, a transmit buffer above the USB generic bulk device class driver would have this field set to point to USBDBulkTxPacketAvailable() and a receive buffer above the same driver would set the field to point to USBDBulkRxPacketAvailable().

`pvHandle`   This field must be set to the handle which should be passed as the first parameter to the functions provided in pfnTransfer and pfnAvailable. This will typically be a pointer to the instance structure for the lower layer object in use. In the case of the USB generic bulk device class, this would be the tUSBDBulkDevice pointer originally passed to (and returned from) USBDBulkInit().

`pcBuffer`   This field must be initialized to point to the block of RAM that will be used to buffer data on this channel. The buffer will be managed as a ring buffer. If the application wishes to access the buffer directly rather than via the USBBufferRead() and USBBufferWrite() APIs (thus avoiding a copy operation), it is vital to ensure that ring wrap conditions are correctly handled in the application code.

`ulBufferSize`   This field provides the size of the buffer pointed to by pcBuffer in bytes.

`pvWorkspace`   The USB buffer requires a block of RAM in which it can store state variables. This field points to application-supplied RAM that can be used as workspace by the buffer object. This RAM must not be accessed by the application and must remain accessible to the USB buffer for as long as the buffer exists (between calls to USBBufferInit() and USBBufferTerm()). The label USB_BUFFER_WORKSPACE_SIZE defines the number of bytes of workspace required.

Once a transmit buffer is initialized, the application can write data to it using function USBBufferWrite() whenever space is available and the USB buffer driver will handle packet transmission to the lower layer. Similarly USBBufferRead() can be called to read received data from a receive buffer at any time. In both cases, the USB buffer uses the same event protocol that the lower layers use to indicate to the application when more data can be transferred or when data has been sent. When data from the USB controller is added to a receive buffer, USB_EVENT_RX_AVAILABLE is passed to the application and when data is removed from a transmit buffer after having been sent to the lower layer, USB_EVENT_TX_COMPLETE is sent.

Applications `usb_dev_bulk` and `usb_dev_serial` provide examples of how to use USB buffers in a device application.

5.5.3   Data Structure Documentation

5.5.3.1   tUSBBuffer

Definition:
```
typedef struct
{
    bool bTransmitBuffer;
    tUSBCallback pfnCallback;
    void *pvCBData;
    tUSBPacketTransfer pfnTransfer;
    tUSBPacketAvailable pfnAvailable;
    void *pvHandle;
    uint8_t *pui8Buffer;
    uint32_t ui32BufferSize;
    tUSBBufferVars sPrivateData;
}
tUSBBuffer
```

Members:

bTransmitBuffer This field sets the mode of the buffer. If true, the buffer operates as a transmit buffer and supports calls to USBBufferWrite by the client. If false, the buffer operates as a receive buffer and supports calls to USBBufferRead.

pfnCallback A pointer to the callback function which will be called to notify the application of all asynchronous events related to the operation of the buffer.

pvCBData A pointer that the buffer will pass back to the client in the first parameter of all callbacks related to this instance.

pfnTransfer The function which should be called to transmit a packet of data in transmit mode or receive a packet in receive mode.

pfnAvailable The function which should be called to determine if the endpoint is ready to accept a new packet for transmission in transmit mode or to determine the size of the buffer required to read a packet in receive mode.

pvHandle The handle to pass to the low level function pointers provided in the pfnTransfer and pfnAvailable members. For USB device use, this is the psDevice parameter required by the relevant device class driver APIs. For USB host use, this is the pipe identifier returned by USBHCDPipeAlloc.

pui8Buffer A pointer to memory to be used as the ring buffer for this instance.

ui32BufferSize The size, in bytes, of the buffer pointed to by pi8Buffer.

sPrivateData The private data for the USB buffer that is allocated by the application.

Description:

The structure used by the application to initialize a buffer object that will provide buffered access to either a transmit or receive channel.

5.5.3.2 tUSBRingBufObject

Definition:
```
typedef struct
{
    uint32_t ui32Size;
    uint32_t ui32WriteIndex;
    uint32_t ui32ReadIndex;
    uint8_t *pui8Buf;
}
tUSBRingBufObject
```

Members:
> ***ui32Size*** The ring buffer size.
> ***ui32WriteIndex*** The ring buffer write index.
> ***ui32ReadIndex*** The ring buffer read index.
> ***pui8Buf*** The ring buffer.

Description:
> The structure used for encapsulating all the items associated with a ring buffer.

5.5.4 Define Documentation

5.5.4.1 USBLIB_FEATURE_CPUCLK

Definition:
> ```
> #define USBLIB_FEATURE_CPUCLK
> ```

Description:
> This feature setting allows an application to inform the USB library of the current processor speed that is can use for internal timing when the frame counter is not yet running. The *pvFeature* is a pointer to a 32-bit value that holds the processor frequency in Hz.

5.5.4.2 USBLIB_FEATURE_LPM

Definition:
> ```
> #define USBLIB_FEATURE_LPM
> ```

Description:
> This feature setting enables or disables LPM support in the USB library in either host or device mode depending on if the USBHCDFeatureSet() or USBDCDFeatureSet() is called. If no action is taken the default behavior of USB library is to not support LPM transactions. The *pvFeature* value is a pointer to a 32-bit value containing the a logical OR of the following values:
>
> - **USBLIB_FEATURE_LPM_EN** is used to enable LPM support in host or device mode.
> - **USBLIB_FEATURE_LPM_DIS(default)** is used to disable LPM support in host or device mode.
> - **USBLIB_FEATURE_LPM_RMT_WAKE** is used to enable remote wake from an LPM suspended state.

5.5.4.3 USBLIB_FEATURE_POWER

Definition:
```
#define USBLIB_FEATURE_POWER
```

Description:
This feature setting enables or disables various power settings in the USB library in either host or device mode depending on if the USBHCDFeatureSet() or USBDCDFeatureSet() is called. The *pvFeature* value is a pointer to a 32-bit value containing the a logical OR of the following values:

- **USBLIB_FEATURE_POWER_BUS** - USB device mode is bus powered(default).
- **USBLIB_FEATURE_POWER_SELF** - USB device mode is self powered.
- **USBLIB_FEATURE_REMOTE_WAKE** - Enable USB remote wake feature.

5.5.4.4 USBLIB_FEATURE_USBPLL

Definition:
```
#define USBLIB_FEATURE_USBPLL
```

Description:
This feature setting allows an application to inform the USB library of the current USB PLL rate in cases where the USB library needs this information for internal configuration. If this feature is not set, then default rate for the USB PLL is 480MHz. The *pvFeature* is a pointer to an 32-bit value that holds the USB PLL speed in Hz. If the application needs to use an external USB clock the PLL value should be set to zero. This is used when connecting to an external USB phy which is providing the 60-MHz clock.

5.5.4.5 USBLIB_FEATURE_USBULPI

Definition:
```
#define USBLIB_FEATURE_USBULPI
```

Description:
This feature setting allows an application to disable or configure and enable the ULPI features in the USB library. If this feature is not set, the default behavior is to not support ULPI operation. The *pvFeature* is a pointer to an 32-bit value that holds the USB ULPI configuration. The following are the valid settings for this feature:

- USBLIB_FEATURE_ULPI_NONE disables all ULPI support.
- USBLIB_FEATURE_ULPI_HS enable ULPI with high speed support.
- USBLIB_FEATURE_ULPI_FS enable ULPI with full speed support.

5.5.5 Typedef Documentation

5.5.5.1 tUSBPacketAvailable

Definition:
```
typedef uint32_t(* tUSBPacketAvailable)(void *pvHandle)
```

Description:
> A function pointer type which describes either a class driver transmit or receive packet available function (both have the same prototype) to the USB buffer object.

Parameters:
> ***pvHandle*** is the handle of the device.

Returns:
> None.

5.5.5.2 tUSBPacketTransfer

Definition:
```
typedef uint32_t(* tUSBPacketTransfer)(void *pvHandle,
                                        uint8_t *pi8Data,
                                        uint32_t ui32Length,
                                        bool bLast)
```

Description:
> A function pointer type which describes either a class driver packet read or packet write function (both have the same prototype) to the USB buffer object.

5.5.6 Function Documentation

5.5.6.1 USBBufferCallbackDataSet

Sets the callback pointer supplied to clients of this buffer.

Prototype:
```
void *
USBBufferCallbackDataSet(tUSBBuffer *psBuffer,
                         void *pvCBData)
```

Parameters:
> ***psBuffer*** is the pointer to the buffer instance whose callback data is to be changed.
> ***pvCBData*** is the pointer the client wishes to receive on all future callbacks from this buffer.

Description:
> This function sets the callback pointer which this buffer will supply to clients as the *pvCBData* parameter in all future calls to the event callback.

Note:
> If this function is to be used, the application must ensure that the tUSBBuffer structure used to describe this buffer is held in RAM rather than flash. The *pvCBData* value passed is written directly into this structure.

Returns:
> Returns the previous callback pointer set for the buffer.

5.5.6.2 USBBufferDataAvailable

Returns the number of bytes of data available in the buffer.

Prototype:
```
uint32_t
USBBufferDataAvailable(const tUSBBuffer *psBuffer)
```

Parameters:
>   ***psBuffer*** is the pointer to the buffer instance which is to be queried.

Description:
>   This function may be used to determine the number of bytes of data in a buffer. For a receive buffer, this indicates the number of bytes that the client can read from the buffer using USB-BufferRead(). For a transmit buffer, this indicates the amount of data that remains to be sent to the USB controller.

Returns:
>   Returns the number of bytes of data in the buffer.

5.5.6.3 USBBufferDataRemoved

Indicates that a client has read data directly out of the buffer.

Prototype:
```
void
USBBufferDataRemoved(const tUSBBuffer *psBuffer,
                     uint32_t ui32Length)
```

Parameters:
>   ***psBuffer*** is the pointer to the buffer instance from which data has been read.
>   ***ui32Length*** is the number of bytes of data that the client has read.

Description:
>   This function updates the USB buffer read pointer to remove data that the client has read directly rather than via a call to USBBufferRead(). The function is provided to aid a client wishing to minimize data copying. To read directly from the buffer, a client must call USBBufferInfo-Get() to retrieve the current buffer inpsBufVarsdices. With this information, the data following the current read index can be read. Once the client has processed much data as it needs, USBBufferDataRemoved() must be called to advance the read pointer past the data that has been read and free up that section of the buffer. The client must take care to correctly handle the wrap point if accessing the buffer directly.

Returns:
>   None.

5.5.6.4 USBBufferDataWritten

Indicates that a client has written data directly into the buffer and wishes to start transmission.

Prototype:
```
void
USBBufferDataWritten(const tUSBBuffer *psBuffer,
                     uint32_t ui32Length)
```

Parameters:
>**psBuffer** is the pointer to the buffer instance into which data has been written.
>**ui32Length** is the number of bytes of data that the client has written.

Description:
>This function updates the USB buffer write pointer and starts transmission of the data in the buffer assuming the lower layer is ready to receive a new packet. The function is provided to aid a client wishing to write data directly into the USB buffer rather than using the USBBufferWrite() function. This may be necessary to control when the USB buffer starts transmission of a large block of data, for example.
>
>A transmit buffer will immediately send a new packet on any call to USBBufferWrite() if the underlying layer indicates that a transmission can be started. In some cases this is not desirable and a client may wish to write more data to the buffer in advance of starting transmission to the lower layer. In such cases, USBBufferInfoGet() may be called to retrieve the current ring buffer indices and the buffer accessed directly. Once the client has written all data it wishes to send (taking care to handle the ring buffer wrap), it should call this function to indicate that transmission may begin.

Returns:
>None.

5.5.6.5   USBBufferEventCallback

Called by the USB buffer to notify the client of asynchronous events.

Prototype:
```
uint32_t
USBBufferEventCallback(void *pvCBData,
                       uint32_t ui32Event,
                       uint32_t ui32MsgValue,
                       void *pvMsgData)
```

Parameters:
>**pvCBData** is the client-supplied callback pointer associated with this buffer instance.
>**ui32Event** is the identifier of the event being sent. This will be a general event identifier of the form **USBD_EVENT_xxxx** or a device class-dependent event of the form **USBD_CDC_EVENT_xxx** or **USBD_HID_EVENT_xxx**.
>**ui32MsgValue** is an event-specific parameter value.
>**pvMsgData** is an event-specific data pointer.

Description:
>This function is the USB buffer event handler that applications should register with the USB device class driver as the callback for the channel which is to be buffered using this buffer.

Note:
>This function will never be called by an application. It is the handler that allows the USB buffer to be inserted above the device class driver or host pipe driver and below the application to offer buffering support.

Returns:
   The return value is dependent upon the event being processed.

5.5.6.6   USBBufferFlush

Flushes a USB buffer, discarding any data that it contains.

Prototype:
```
void
USBBufferFlush(const tUSBBuffer *psBuffer)
```

Parameters:
   psBuffer is the pointer to the buffer instance which is to be flushed.

Description:
   This function discards all data currently in the supplied buffer without processing (transmitting it via the USB controller or passing it to the client depending upon the buffer mode).

Returns:
   None.

5.5.6.7   USBBufferInfoGet

Returns the current ring buffer indices for this USB buffer.

Prototype:
```
void
USBBufferInfoGet(const tUSBBuffer *psBuffer,
                 tUSBRingBufObject *psRingBuf)
```

Parameters:
   psBuffer is the pointer to the buffer instance whose information is being queried.

   psRingBuf is a pointer to storage that will be written with the current ring buffer control structure for this USB buffer.

Description:
   This function is provided to aid a client wishing to write data directly into the USB buffer rather than using the USBBufferWrite() function. This may be necessary to control when the USB-Buffer starts transmission of a large block of data, for example.

   A transmit buffer will immediately send a new packet on any call to USBBufferWrite() if the underlying layer indicates that a transmission can be started. In some cases this is not desirable and a client may wish to wishes to write more data to the buffer in advance of starting transmission to the lower layer. In such cases, this function may be called to retrieve the current ring buffer indices and the buffer accessed directly. Once the client has written all data it wishes to send, it should call function USBBufferDataWritten() to indicate that transmission may begin.

Returns:
   None.

5.5.6.8   USBBufferInit

Initializes a USB buffer object to be used with a given USB controller and device or host class driver.

Prototype:
```
const tUSBBuffer *
USBBufferInit(tUSBBuffer *psBuffer)
```

Parameters:
>   ***psBuffer*** points to a structure containing information on the buffer memory to be used and the underlying device or host class driver whose data is to be buffered. This structure must remain accessible for as long as the buffer is in use.

Description:
>   This function is used to initialize a USB buffer object and insert it into the function and callback interfaces between an underlying driver and the application. The caller supplies information on both the RAM to be used to buffer data, the type of buffer to be created (transmit or receive) and the functions to be called in the lower layer to transfer data to or from the USB controller.

Returns:
>   Returns the original buffer structure pointer if successful or NULL if an error is detected.

5.5.6.9   USBBufferRead

Reads a block of data from a USB receive buffer into storage supplied by the caller.

Prototype:
```
uint32_t
USBBufferRead(const tUSBBuffer *psBuffer,
              uint8_t *pui8Data,
              uint32_t ui32Length)
```

Parameters:
>   ***psBuffer*** is the pointer to the buffer instance from which data is to be read.
>
>   ***pui8Data*** points to a buffer into which the received data will be written.
>
>   ***ui32Length*** is the size of the buffer pointed to by pui8Data.

Description:
>   This function reads up to *ui32Length* bytes of data received from the USB host into the supplied application buffer. If the receive buffer contains fewer than *ui32Length* bytes of data, the data that is present will be copied and the return code will indicate the actual number of bytes copied to *pui8Data*.

Returns:
>   Returns the number of bytes of data read.

5.5.6.10   USBBufferSpaceAvailable

Returns the number of free bytes in the buffer.

Prototype:
```
uint32_t
USBBufferSpaceAvailable(const tUSBBuffer *psBuffer)
```

Parameters:
> **psBuffer** is the pointer to the buffer instance which is to be queried.

Description:
> This function returns the number of free bytes in the buffer. For a transmit buffer, this indicates the maximum number of bytes that can be passed on a call to USBBufferWrite() and accepted for transmission. For a receive buffer, it indicates the number of bytes that can be read from the USB controller before the buffer will be full.

Returns:
> Returns the number of free bytes in the buffer.

5.5.6.11 USBBufferWrite

Writes a block of data to the transmit buffer and queues it for transmission to the USB controller.

Prototype:
```
uint32_t
USBBufferWrite(const tUSBBuffer *psBuffer,
               const uint8_t *pui8Data,
               uint32_t ui32Length)
```

Parameters:
> **psBuffer** points to the pointer instance into which data is to be written.
> **pui8Data** points to the first byte of data which is to be written.
> **ui32Length** is the number of bytes of data to write to the buffer.

Description:
> This function copies the supplied data into the transmit buffer. The transmit buffer data will be packetized according to the constraints imposed by the lower layer in use and sent to the USB controller as soon as possible. Once a packet is transmitted and acknowledged, a **USB_EVENT_TX_COMPLETE** event will be sent to the application callback indicating the number of bytes that have been sent from the buffer.

> Attempts to send more data than there is space for in the transmit buffer will result in fewer bytes than expected being written. The value returned by the function indicates the actual number of bytes copied to the buffer.

Returns:
> Returns the number of bytes actually written.

5.5.6.12 USBBufferZeroLengthPacketInsert

Enables or disables zero-length packet insertion.

Prototype:
```
void
USBBufferZeroLengthPacketInsert(const tUSBBuffer *psBuffer,
                                bool bSendZLP)
```

Parameters:
> ***psBuffer*** is the pointer to the buffer instance whose information is being queried.
>
> ***bSendZLP*** is **true** to send zero-length packets or **false** to prevent them from being sent.

Description:
> This function allows the use of zero-length packets to be controlled by an application. In cases where the USB buffer has sent a full (64 byte) packet and then discovers that the transmit buffer is empty, the default behavior is to do nothing. Some protocols, however, require that a zero- length packet be inserted to signal the end of the data. When using such a protocol, this function should be called with *bSendZLP* set to **true** to enable the desired behavior.

Returns:
> None.

5.5.6.13 USBRingBufAdvanceRead

Removes bytes from the ring buffer by advancing the read index.

Prototype:
```
void
USBRingBufAdvanceRead(tUSBRingBufObject *psUSBRingBuf,
                      uint32_t ui32NumBytes)
```

Parameters:
> ***psUSBRingBuf*** points to the ring buffer from which bytes are to be removed.
>
> ***ui32NumBytes*** is the number of bytes to be removed from the buffer.

Description:
> This function advances the ring buffer read index by a given number of bytes, removing that number of bytes of data from the buffer. If *ui32NumBytes* is larger than the number of bytes currently in the buffer, the buffer is emptied.

Returns:
> None.

5.5.6.14 USBRingBufAdvanceWrite

Adds bytes to the ring buffer by advancing the write index.

Prototype:
```
void
USBRingBufAdvanceWrite(tUSBRingBufObject *psUSBRingBuf,
                       uint32_t ui32NumBytes)
```

Parameters:
> ***psUSBRingBuf*** points to the ring buffer to which bytes have been added.

ui32NumBytes is the number of bytes added to the buffer.

Description:
> This function should be used by clients who wish to add data to the buffer directly rather than via calls to USBRingBufWrite() or USBRingBufWriteOne(). It advances the write index by a given number of bytes.

Note:
> It is considered an error if the *ui32NumBytes* parameter is larger than the amount of free space in the buffer and a debug build of this function will fail (ASSERT) if this condition is detected. In a release build, the buffer read pointer will be advanced if too much data is written but this will, of course, result in some of the oldest data in the buffer being discarded and also, depending upon how data is being read from the buffer, may result in a race condition which could corrupt the read pointer.

Returns:
> None.

5.5.6.15 USBRingBufContigFree

Returns number of contiguous free bytes available in a ring buffer.

Prototype:
```
uint32_t
USBRingBufContigFree(tUSBRingBufObject *psUSBRingBuf)
```

Parameters:
> ***psUSBRingBuf*** is the ring buffer object to check.

Description:
> This function returns the number of contiguous free bytes ahead of the current write pointer in the ring buffer.

Returns:
> Returns the number of contiguous bytes available in the ring buffer.

5.5.6.16 USBRingBufContigUsed

Returns number of contiguous bytes of data stored in ring buffer ahead of the current read pointer.

Prototype:
```
uint32_t
USBRingBufContigUsed(tUSBRingBufObject *psUSBRingBuf)
```

Parameters:
> ***psUSBRingBuf*** is the ring buffer object to check.

Description:
> This function returns the number of contiguous bytes of data available in the ring buffer ahead of the current read pointer. This represents the largest block of data which does not straddle the buffer wrap.

Returns:
Returns the number of contiguous bytes available.

5.5.6.17 USBRingBufEmpty

Determines whether a ring buffer is empty or not.

Prototype:
```
bool
USBRingBufEmpty(tUSBRingBufObject *psUSBRingBuf)
```

Parameters:
psUSBRingBuf is the ring buffer object to empty.

Description:
This function is used to determine whether or not a given ring buffer is empty. The structure is specifically to ensure that we do not see warnings from the compiler related to the order of volatile accesses being undefined.

Returns:
Returns **true** if the buffer is empty or **false** otherwise.

5.5.6.18 USBRingBufFlush

Empties the ring buffer.

Prototype:
```
void
USBRingBufFlush(tUSBRingBufObject *psUSBRingBuf)
```

Parameters:
psUSBRingBuf is the ring buffer object to empty.

Description:
Discards all data from the ring buffer.

Returns:
None.

5.5.6.19 USBRingBufFree

Returns number of bytes available in a ring buffer.

Prototype:
```
uint32_t
USBRingBufFree(tUSBRingBufObject *psUSBRingBuf)
```

Parameters:
psUSBRingBuf is the ring buffer object to check.

Description:

This function returns the number of bytes available in the ring buffer.

Returns:

Returns the number of bytes available in the ring buffer.

5.5.6.20 USBRingBufFull

Determines whether a ring buffer is full or not.

Prototype:
```
bool
USBRingBufFull(tUSBRingBufObject *psUSBRingBuf)
```

Parameters:

psUSBRingBuf is the ring buffer object to empty.

Description:

This function is used to determine whether or not a given ring buffer is full. The structure is specifically to ensure that we do not see warnings from the compiler related to the order of volatile accesses being undefined.

Returns:

Returns **true** if the buffer is full or **false** otherwise.

5.5.6.21 USBRingBufInit

Initializes a ring buffer object.

Prototype:
```
void
USBRingBufInit(tUSBRingBufObject *psUSBRingBuf,
               uint8_t *pui8Buf,
               uint32_t ui32Size)
```

Parameters:

psUSBRingBuf points to the ring buffer to be initialized.

pui8Buf points to the data buffer to be used for the ring buffer.

ui32Size is the size of the buffer in bytes.

Description:

This function initializes a ring buffer object, preparing it to store data.

Returns:

None.

5.5.6.22 USBRingBufRead

Reads data from a ring buffer.

Prototype:
```
void
USBRingBufRead(tUSBRingBufObject *psUSBRingBuf,
               uint8_t *pui8Data,
               uint32_t ui32Length)
```

Parameters:
> ***psUSBRingBuf*** points to the ring buffer to be read from.
> ***pui8Data*** points to where the data should be stored.
> ***ui32Length*** is the number of bytes to be read.

Description:
> This function reads a sequence of bytes from a ring buffer.

Returns:
> None.

5.5.6.23 USBRingBufReadOne

Reads a single byte of data from a ring buffer.

Prototype:
```
uint8_t
USBRingBufReadOne(tUSBRingBufObject *psUSBRingBuf)
```

Parameters:
> ***psUSBRingBuf*** points to the ring buffer to be written to.

Description:
> This function reads a single byte of data from a ring buffer.

Returns:
> The byte read from the ring buffer.

5.5.6.24 USBRingBufSize

Returns the size in bytes of a ring buffer.

Prototype:
```
uint32_t
USBRingBufSize(tUSBRingBufObject *psUSBRingBuf)
```

Parameters:
> ***psUSBRingBuf*** is the ring buffer object to check.

Description:
> This function returns the size of the ring buffer.

Returns:

Returns the size in bytes of the ring buffer.

5.5.6.25 USBRingBufUsed

Returns number of bytes stored in ring buffer.

Prototype:
```
uint32_t
USBRingBufUsed(tUSBRingBufObject *psUSBRingBuf)
```

Parameters:

psUSBRingBuf is the ring buffer object to check.

Description:

This function returns the number of bytes stored in the ring buffer.

Returns:

Returns the number of bytes stored in the ring buffer.

5.5.6.26 USBRingBufWrite

Writes data to a ring buffer.

Prototype:
```
void
USBRingBufWrite(tUSBRingBufObject *psUSBRingBuf,
                const uint8_t *pui8Data,
                uint32_t ui32Length)
```

Parameters:

psUSBRingBuf points to the ring buffer to be written to.

pui8Data points to the data to be written.

ui32Length is the number of bytes to be written.

Description:

This function write a sequence of bytes into a ring buffer.

Returns:

None.

5.5.6.27 USBRingBufWriteOne

Writes a single byte of data to a ring buffer.

Prototype:
```
void
USBRingBufWriteOne(tUSBRingBufObject *psUSBRingBuf,
                   uint8_t ui8Data)
```

Parameters:
> ***psUSBRingBuf*** points to the ring buffer to be written to.
> ***ui8Data*** is the byte to be written.

Description:
> This function writes a single byte of data into a ring buffer.

Returns:
> None.

5.6   Internal USB DMA functions

Defines

- USBLibDMAArbSizeSet(psUSBDMAInst, ui32Channel, ui32ArbSize)
- USBLibDMAChannelAllocate(psUSBDMAInst, ui8Endpoint, ui32MaxPacketSize, ui32Config)
- USBLibDMAChannelDisable(psUSBDMAInst, ui32Channel)
- USBLibDMAChannelEnable(psUSBDMAInst, ui32Channel)
- USBLibDMAChannelIntDisable(psUSBDMAInst, ui32Channel)
- USBLibDMAChannelIntEnable(psUSBDMAInst, ui32Channel)
- USBLibDMAChannelRelease(psUSBDMAInst, ui8Endpoint)
- USBLibDMAChannelStatus(psUSBDMAInst, ui32Channel)
- USBLibDMAIntHandler(psUSBDMAInst, ui32Status)
- USBLibDMAIntStatus(psUSBDMAInst)
- USBLibDMAIntStatusClear(psUSBDMAInst, ui32Status)
- USBLibDMAStatus(psUSBDMAInst)
- USBLibDMATransfer(psUSBDMAInst, ui32Channel, pvBuffer, ui32Size)
- USBLibDMAUnitSizeSet(psUSBDMAInst, ui32Channel, ui32BitSize)

Functions

- void ∗ USBLibDMAAddrGet (tUSBDMAInstance ∗psUSBDMAInst, uint32_t ui32Channel)
- tUSBDMAInstance ∗ USBLibDMAInit (uint32_t ui32Index)
- uint32_t USBLibDMASizeGet (tUSBDMAInstance ∗psUSBDMAInst, uint32_t ui32Channel)

5.6.1   Define Documentation

5.6.1.1   USBLibDMAArbSizeSet

This function is used to set the arbitration size for a DMA channel.

Definition:
```
#define USBLibDMAArbSizeSet(psUSBDMAInst,
                            ui32Channel,
                            ui32ArbSize)
```

Parameters:
> ***psUSBDMAInst*** is the DMA instance data for a USB controller.
>
> ***ui32Channel*** is the DMA channel number to modify.
>
> ***ui32ArbSize*** is the transfer arbitration size in bytes.

Description:
> This function configures the individual transfer size of the DMA channel provided in the *ui32Channel* parameter. The *ui32Channel* must already be allocated to an endpoint by calling the USBLibDMAChannelAllocate() function.

Returns:
> None.

5.6.1.2   USBLibDMAChannelAllocate

This function is used to assign a DMA channel to an endpoint.

Definition:
```
#define USBLibDMAChannelAllocate(psUSBDMAInst,
                                 ui8Endpoint,
                                 ui32MaxPacketSize,
                                 ui32Config)
```

Parameters:
> ***psUSBDMAInst*** is the DMA instance data for a USB controller.
>
> ***ui8Endpoint*** is the endpoint number to assign a DMA channel.
>
> ***ui32MaxPacketSize*** is the maximum packet size for the endpoint assigned that is being assigned to the DMA channel.
>
> ***ui32Config*** are the basic configuration options for the DMA channel.

Description:
> This function assigns a DMA channel to a given endpoint. The *ui8Endpoint* parameter is the zero based endpoint number that is assigned a DMA channel. The *ui32Config* parameter contains any configuration options for the DMA channel. The current options include the following:
>
> - **USB_DMA_EP_TX** - this request is for a transmit DMA channel.
> - **USB_DMA_EP_RX** - this request is for a receive DMA channel.

Note:
> The maximum number of available DMA channels to endpoints varies between devices.

Returns:
> Zero or the DMA channel assigned to the endpoint.

5.6.1.3   USBLibDMAChannelDisable

This function disables DMA for a given DMA channel.

Definition:
```
#define USBLibDMAChannelDisable(psUSBDMAInst,
                                ui32Channel)
```

Parameters:
    psUSBDMAInst is the DMA structure pointer for this instance.
    ui32Channel is the DMA channel to disable.

Description:
    This function disables DMA on the channel number passed in the *ui32Channel* parameter.

Returns:
    None.

5.6.1.4 USBLibDMAChannelEnable

This function enables DMA for a given channel.

Definition:
```
#define USBLibDMAChannelEnable(psUSBDMAInst,
                               ui32Channel)
```

Parameters:
    psUSBDMAInst is the DMA structure pointer for this instance.
    ui32Channel is the DMA channel to enable.

Description:
    This function enables DMA on the channel number passed in the *ui32Channel* parameter.

Returns:
    None.

5.6.1.5 USBLibDMAChannelIntDisable

This function disables DMA interrupt for a given DMA channel.

Definition:
```
#define USBLibDMAChannelIntDisable(psUSBDMAInst,
                                   ui32Channel)
```

Parameters:
    psUSBDMAInst is the DMA structure pointer for this instance.
    ui32Channel is the DMA channel interrupt to disable.

Description:
    This function disables the DMA interrupt on the channel number passed in the *ui32Channel* parameter.

Returns:
    None.

5.6.1.6   USBLibDMAChannelIntEnable

This function enables the DMA interrupt for a given channel.

Definition:
```
#define USBLibDMAChannelIntEnable(psUSBDMAInst,
                                  ui32Channel)
```

Parameters:
psUSBDMAInst is the DMA structure pointer for this instance.
ui32Channel is the DMA channel interrupt to enable.

Description:
This function enables DMA interrupt on the channel number passed in the *ui32Channel* parameter.

Returns:
None.

5.6.1.7   USBLibDMAChannelRelease

This function is used to free a DMA channel that was assigned to an endpoint.

Definition:
```
#define USBLibDMAChannelRelease(psUSBDMAInst,
                                ui8Endpoint)
```

Parameters:
psUSBDMAInst is the DMA instance data for a USB controller.
ui8Endpoint is the DMA channel number to free up.

Description:
This function frees up a DMA channel that was allocated to an endpoint by the USBLibD-MAChannelAllocate() function.

Returns:
None.

5.6.1.8   USBLibDMAChannelStatus

This function returns the current DMA status for a given DMA channel.

Definition:
```
#define USBLibDMAChannelStatus(psUSBDMAInst,
                               ui32Channel)
```

Parameters:
psUSBDMAInst is the DMA structure pointer for this instance.
ui32Channel is the DMA channel number used to retrieve the DMA status.

Description:
This function returns the current status of a DMA transfer on a given DMA channel. The DMA channel is specified by the *ui32Channel* parameter.

Returns:
This function returns one of the **USBLIBSTATUS_DMA_***∗* values.

5.6.1.9 USBLibDMAIntHandler

This function is called by the USB interrupt handler.

Definition:
```
#define USBLibDMAIntHandler(psUSBDMAInst,
                            ui32Status)
```

Parameters:
psUSBDMAInst is the DMA structure pointer for this instance.
ui32Status is the DMA interrupt status.

Description:
This function is called by the USB interrupt handler to allow the DMA interface to handle interrupts outside of the context of the normal USB interrupt handler. The *ui32Status* is the current DMA interrupt status at the time of the USB interrupt. Since some DMA controller interrupts are cleared automatically when read, this value must be retrieved by calling the USBLibDMAIntStatus() function and passed into this function.

Returns:
None.

5.6.1.10 USBLibDMAIntStatus

This function returns the current DMA interrupt status.

Definition:
```
#define USBLibDMAIntStatus(psUSBDMAInst)
```

Parameters:
psUSBDMAInst is the DMA structure pointer for this instance.

Description:
This function returns the interrupt status for all DMA channels. The value returned is a per channel interrupt mapping with the DMA channels mapped into bits 0-31 by channel number with channel 1 starting at bit 0.

Note:
This function does not return an endpoint interrupt status, but the interrupt status for the DMA interface used with the USB controller.

Returns:
This function returns the pending DMA interrupts.

5.6.1.11 USBLibDMAIntStatusClear

This function clears the requested DMA interrupt status.

Definition:
```
#define USBLibDMAIntStatusClear(psUSBDMAInst,
                                ui32Status)
```

Parameters:
> ***psUSBDMAInst*** is the DMA structure pointer for this instance.
> ***ui32Status*** contains the interrupts to clear.

Description:
> This function clears the current DMA interrupt status for the controller specified by the *ui32Instance* parameter. The *ui32Status* value has the same format as the value returned from the USBLibDMAIntStatus() function which is a per channel interrupt mapping. The DMA channels are mapped into bits 0-31 by channel number with channel 1 starting at bit 0.

Returns:
> None.

5.6.1.12 USBLibDMAStatus

This function is used to return any global status information for USB DMA.

Definition:
```
#define USBLibDMAStatus(psUSBDMAInst)
```

Parameters:
> ***psUSBDMAInst*** is a generic instance pointer that can be used to distinguish between different hardware instances.

Description:
> This function performs returns the global status for the USB DMA interface.

Returns:
> Always returns 0.

5.6.1.13 USBLibDMATransfer

This function is configures a USB transfer on a given DMA channel.

Definition:
```
#define USBLibDMATransfer(psUSBDMAInst,
                          ui32Channel,
                          pvBuffer,
                          ui32Size)
```

Parameters:
> ***psUSBDMAInst*** is the DMA structure pointer for this instance.
> ***ui32Channel*** is the DMA channel to use.

pvBuffer is a pointer to the buffer to use for the transfer.

ui32Size is the size of the data to be transferred in bytes.

Description:
This function is called to configure a transfer using the USB controller depending on the parameters. The *ui32Channel* parameter holds the channel number to use for this transfer which must have already been allocated with a call to the USBLibDMAChannelAllocate() function. The transaction is configured to transfer *ui32Size* bytes to/from the buffer held in the *pvBuffer* pointer.

Returns:
This function returns the number of bytes scheduled to be transferred.

5.6.1.14 USBLibDMAUnitSizeSet

This function is used to set the individual transfer size of a DMA channel.

Definition:
```
#define USBLibDMAUnitSizeSet(psUSBDMAInst,
                             ui32Channel,
                             ui32BitSize)
```

Parameters:
psUSBDMAInst is the DMA instance data for a USB controller.

ui32Channel is the DMA channel number to modify.

ui32BitSize is the individual transfer size in bits(8, 16 or 32).

Description:
This function configures the individual transfer size of the DMA channel provided in the *ui32Channel* parameter. The *ui32Channel* must already be allocated to an endpoint by calling the USBLibDMAChannelAllocate() function. The *ui32BitSize* parameter should be on of the following values: 8, 16 or 32.

Returns:
None.

5.6.2 Function Documentation

5.6.2.1 USBLibDMAAddrGet

This function is used to return the current DMA pointer for a given DMA channel.

Prototype:
```
void *
USBLibDMAAddrGet(tUSBDMAInstance *psUSBDMAInst,
                 uint32_t ui32Channel)
```

Parameters:
psUSBDMAInst is a generic instance pointer that can be used to distinguish between different hardware instances.

ui32Channel is the DMA channel number for this request.

Description:

This function returns the address that is in use by the DMA channel passed in via the *ui32Channel* parameter. This is not the real-time pointer, but the starting address of the DMA transfer for this DMA channel.

Returns:

The current DMA address for the given DMA channel.

5.6.2.2   USBLibDMAInit

This function is used to initialize the DMA interface for a USB instance.

Prototype:
```
tUSBDMAInstance *
USBLibDMAInit(uint32_t ui32Index)
```

Parameters:

ui32Index is the index of the USB controller for this instance.

Description:

This function performs any initialization and configuration of the DMA portions of the USB controller. This function returns a pointer that is used with the remaining USBLibDMA APIs or the function returns zero if the requested controller cannot support DMA. If this function is called when already initialized it will not reinitialize the DMA controller and will instead return the previously initialized DMA instance.

Returns:

A pointer to use with USBLibDMA APIs.

5.6.2.3   USBLibDMASizeGet

This function is used to return the current DMA transfer size for a given DMA channel.

Prototype:
```
uint32_t
USBLibDMASizeGet(tUSBDMAInstance *psUSBDMAInst,
                 uint32_t ui32Channel)
```

Parameters:

psUSBDMAInst is a generic instance pointer that can be used to distinguish between different hardware instances.

ui32Channel is the DMA channel number for this request.

Description:

This function returns the DMA transfer size that is in use by the DMA channel passed in via the *ui32Channel* parameter.

Returns:

The current DMA transfer size for the given DMA channel.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.