F2806x USB Library

USER'S GUIDE

Copyright

Texas Instruments
13905 University Boulevard
Sugar Land, TX 77479
http://www.ti.com/c2000

Revision Information

This is version 2.08.00.00 of this document, last updated on Fri Oct 24 23:39:48 IST 2025.

Table of Contents

Copyright **2**
Revision Information **2**
1 Introduction **5**
2 General Purpose Functions **7**
2.1 C2000 Specific Software 9
2.2 Function Definitions 10
2.3 USB Chapter 9 Definitions 17
2.4 USB Buffer and Ring Buffer APIs 19
3 Device Functions **35**
3.1 USB DriverLib API 37
3.2 USB Device API 37
3.3 USB Device Class Driver APIs 38
3.4 USB Device Class APIs 39
3.5 Generic Bulk Device Class Events 40
3.6 Using the Generic Bulk Device Class 40
3.7 Using the Composite Bulk Device Class 42
3.8 Windows Drivers for Generic Bulk Devices 43
3.9 Bulk Device Class Driver Definitions 46
3.10 CDC Device Class Events 53
3.11 Using the CDC Device Class Driver 54
3.12 Using the Composite CDC Serial Device Class 57
3.13 Windows Drivers for CDC Serial Devices 57
3.14 CDC Device Class Driver Definitions 58
3.15 Defining a Composite Device 67
3.16 Allocating Memory 68
3.17 Example Composite Device 69
3.18 Composite Device Class Driver Definitions 71
3.19 HID Device Class Events 73
3.20 Using the HID Device Class Driver 74
3.21 Using the Composite HID Mouse Device Class 78
3.22 Handling HID Reports 78
3.23 HID Device Class Driver Definitions 80
3.24 HID Mouse Device API Events 98
3.25 Using the HID Mouse Device Class API 99
3.26 HID Mouse Device Class API Definitions 100
3.27 HID Keyboard Device API Events 106
3.28 Using the HID Keyboard Device Class API 106
3.29 HID Keyboard Device Class API Definitions 107
3.30 Building Descriptors 114
3.31 USB Event Handlers 117
3.32 USB FIFO Configuration 122
3.33 Interrupt Vector Selection 122
3.34 Passing Control to the USB Device API 122
3.35 USB Device API Definitions 123
4 Host Functions **131**
4.1 Enumeration 133
4.2 USB Pipes 134
4.3 Control Transactions 134

4.4 Interrupt Handling . . . 134
4.5 Host Controller Driver Definitions . . . 135
4.6 HID Class Driver . . . 152
4.7 Mass Storage Class Driver . . . 154
4.8 Implementing Custom Host Class Drivers . . . 155
4.9 Host Class Driver Definitions . . . 156
4.10 Mouse Device . . . 178
4.11 Keyboard Device . . . 179
4.12 Host Device Interface Definitions . . . 180
4.13 Application Initialization . . . 185
4.14 Application Interface . . . 185
4.15 Application Termination . . . 186
4.16 Example Application Setup . . . 186
4.17 Host HID Mouse Programming Example . . . 187
4.18 Host HID Keyboard Programming Example . . . 188
4.19 Host Mass Storage Programming Example . . . 189
IMPORTANT NOTICE . . . 190

1 Introduction

The Texas Instruments® C2000® USB Library is a set of data types and functions for creating USB device or host applications on C2000 microcontroller-based boards. The contents of the USB library and its associated header files fall into four main groups:

General purpose functions used by both device and host applications. These include functions to parse USB descriptors and set the operating mode of the application.

Device specific functions providing the class-independent features required by all USB device applications such as host connection signaling and responding to standard descriptor requests.

Host specific functions providing class-independent features required by all USB host application such as device detection and enumeration, and endpoint management.

Class specific functions and data types to aid development of applications conforming to several commonly-used USB classes.

The capabilities and organization of the USB library functions are governed by the following design goals:

They are written entirely in C.

They are easy to understand.

They are reasonably efficient in terms of memory and processor usage.

They are as self-contained as possible.

Where possible, computations that can be performed at compile time are done there instead of at run time.

Some consequences of these design goals are:

To ensure ease of use and understandability, the USB functions are not necessarily as efficient as they could be (from a code size and/or execution speed point of view).

The APIs have a means of removing all error checking code. Since the error checking is usually only useful during initial program development, it can be removed to improve code size and speed.

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 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 2.
`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 3.
`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 4.

2 General Purpose Functions

Introduction ..... ??
Porting Summary ..... ??
C2000 Specific Software 9
Function Definitions 10
USB Chapter 9 Definitions 17
USB Buffer and Ring Buffer APIs ..... 19 This chapter describes the set of USB library data types and functions which are of general use in the development of USB applications running on C2000 microcontroller-based boards. These elements are not specific to USB host or device.

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 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. Special type definitions were added to this file in the effort to port the Stellarisware USB stack to the C2000 platform. Special attention should be paid to the types tShort and tLong and their respective accesses.
`f2806x_usbWrapper.h`	The header file containing definitions related to the interrupt service routines belonging to the USB stack. Also contains externs for various USB support functions specific to the f2806x series of devices.
`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.

f2806x_usbWrapper.c	The source code for f2806x specific interrupt wrappers, and support functions related to: VBus Detection, ID Detection, External Power Control, and External Power Fault Detection.
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.

The porting strategy focused on three main differences between the Stellaris and C2000 devices.

Memory Widths

Interrupt Controllers

Support Function Pins

The stack underwent much change because of the different memory widths of the Stellaris and C2000 devices. On a Stellaris device the smallest addressable location is 8 bits, while on a C2000 device the smallest addressable location is 16 bits. In software this means that a char on C2000 is 16 bits while on most other devices a char is 8 bits. Because of this and the way the USB descriptors are passed up and down the stack, all data read from and written to the USB controller is done a byte (8 bits) at a time and is stored in memory a byte (8 bits) per address. This is inefficient because data may be read from the USB FIFOs 32 bits at a time and half of the buffer memory is wasted.

Minor changes were made to the stack such that it would function with the C28x style PIE interrupt controller. Most changes are invisible to the user as much of the functionality of the interrupt controller driver from driverlib was ported to work with the C28x PIE. The one change you might see are the interrupt service routine wrappers in usblib/f2806x_usbWrapper.c.

Because of design constraints, extra pins were not alotted on this device for function such as VBus detection, ID detection, extenal power enable, and power fault detection. All of these functions can easily be implemented in software using GPIOs. As development kits are released for USB, look for software that implements these functions in usblib/f2806x_usbWrapper.c.

2.1 C2000 Specific Software

Data Structures

struct usb16_t

struct usb32_t

Macros

#define readusb16_t(ptr)

#define readusb32_t(ptr)

#define writeusb16_t(ptr, value)

#define writeusb32_t(ptr, value)

2.1.1 Detailed Description

This software was added to the Stellarisware USB protocol stack to add compatability with C2000 devices.

2.1.2 Macro Definition Documentation

2.1.2.1 #define readusb16_t(

ptr)

This define is used to read data from a tShort variable. This is a is a workaround specific to C2000 devices.

Referenced by USBDescGetInterface(), USBDescGetNumAlternateInterfaces(), and USBDevice-Config().

2.1.2.2 #define readusb32_t(

ptr)

This define is used to read data from a tLong variable. This is a is a workaround specific to C2000 devices.

2.1.2.3 #define writeusb16_t(

ptr,

value)

This define is used to write data to a tShort variable. This is a is a workaround specific to C2000 devices.

Referenced by USBDAudioInit(), USBDBulkInit(), USBDCDCInit(), USBDCompositeInit(), USBDHIDInit(), USBHCDSetAddress(), USBHCDSetConfig(), USBHCDSetInterface(), USBHHIDGetReportDescriptor(), USBHHIDSetIdle(), USBHHIDSetProtocol(), USBHHIDSetReport(), and USBHostAudioVolumeSet().

2.1.2.4 #define writeusb32_t(

ptr,

value)

This define is used to write data to a tLong variable. This is a is a workaround specific to C2000 devices.

Referenced by USBHSCSIInquiry(), USBHSCSIModeSense6(), USBHSCSIReadCapacities(), USBHSCSIReadCapacity(), USBHSCSIRequestSense(), and USBHSCSITestUnitReady().

2.2 Function Definitions

Macros

#define USB_DESC_ANY

#define USBERR_DEV_RX_DATA_ERROR

#define USBERR_DEV_RX_FIFO_FULL

#define USBERR_DEV_RX_OVERRUN

#define USBERR_HOST_EP0_ERROR

#define USBERR_HOST_EP0_NAK_TO

#define USBERR_HOST_IN_DATA_ERROR

#define USBERR_HOST_IN_ERROR

#define USBERR_HOST_IN_FIFO_FULL

#define USBERR_HOST_IN_NAK_TO

#define USBERR_HOST_IN_NOT_COMP

#define USBERR_HOST_IN_PID_ERROR

#define USBERR_HOST_IN_STALL

#define USBERR_HOST_OUT_ERROR

#define USBERR_HOST_OUT_NAK_TO

#define USBERR_HOST_OUT_NOT_COMP

#define USBERR_HOST_OUT_STALL

Typedefs

typedef uint32_t(* tUSBCallback)(void *pvCBData, uint32_t ui32Event, uint32_t ui32MsgParam, void *pvMsgData)

Enumerations

enum tUSBMode {
eUSBModeDevice, eUSBModeHost, eUSBModeOTG, eUSBModeNone,
eUSBModeForceHost, eUSBModeForceDevice }

Functions

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 pfnCallback)

2.2.1 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.

2.2.2 Macro Definition Documentation

2.2.2.1 #define USB_DESC_ANY

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.

Referenced by USBDescGet(), USBDescGetInterface(), USBDescGetNum(), USBDeviceConfig(), and USBDeviceConfigAlternate().

2.2.2.2 #define USBERR_DEV_RX_OVERRUN

The device was unable to receive a packet from the host since the receive FIFO is full.

2.2.2.3 #define USBERR_HOST_EP0_NAK_TO

The host received NAK on endpoint 0 for longer than the configured timeout.

2.2.2.4 #define USBERR_HOST_IN_NAK_TO

The host received NAK on an IN endpoint for longer than the specified timeout period (interrupt, bulk and control modes).

2.2.2.5 #define USBERR_HOST_OUT_NAK_TO

The host received NAK on an OUT endpoint for longer than the specified timeout period (bulk, interrupt and control modes).

2.2.3 Typedef Documentation

2.2.3.1 typedef uint32_t(∗ tUSBCallback)(void ∗pvCBData, uint32_t ui32Event, uint32_t ui32MsgParam, void ∗pvMsgData)

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.

2.2.4 Enumeration Type Documentation

2.2.4.1 enum **tUSBMode**

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.

Enumerator

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.

2.2.5 Function Documentation

2.2.5.1 **tDescriptorHeader** * USBDescGet (

tDescriptorHeader * psDesc,

uint32_t ui32Size,

uint32_t ui32Type,

uint32_t ui32Index)

Determines the number of individual descriptors of a particular type within a supplied buffer.

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.

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.

References tDescriptorHeader::bDescriptorType, tDescriptorHeader::bLength, NEXT_USB_DESCRIPTOR, and USB_DESC_ANY.

Referenced by USBDescGetInterface(), and USBDescGetInterfaceEndpoint().

2.2.5.2 **tInterfaceDescriptor** * USBDescGetInterface (

tConfigDescriptor * psConfig,

uint32_t ui32Index,

uint32_t ui32Alt)

Returns a pointer to the n-th interface descriptor in a configuration descriptor that applies to the supplied alternate setting number.

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.

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.

References readusb16_t, USB_DESC_ANY, USBDescGet(), and tConfigDescriptor::wTotalLength.

Referenced by USBHCDDevClass(), USBHCDDevProtocol(), and USBHCDDevSubClass().

2.2.5.3 **tEndpointDescriptor** * USBDescGetInterfaceEndpoint (

tInterfaceDescriptor * psInterface,

uint32_t ui32Index,

uint32_t ui32Size)

Return a pointer to the n-th endpoint descriptor in the supplied interface descriptor.

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.

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.

References tInterfaceDescriptor::bNumEndpoints, and USBDescGet().

2.2.5.4 uint32_t USBDescGetNum (

tDescriptorHeader * psDesc,

uint32_t ui32Size,

uint32_t ui32Type)

Determines the number of individual descriptors of a particular type within a supplied buffer.

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.

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.

References tDescriptorHeader::bDescriptorType, tDescriptorHeader::bLength, NEXT_USB_DESCRIPTOR, and USB_DESC_ANY.

2.2.5.5 uint32_t USBDescGetNumAlternateInterfaces (

tConfigDescriptor * psConfig,

uint8_t ui8InterfaceNumber)

Determines the number of different alternate configurations for a given interface within a configuration descriptor.

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.

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.

References tDescriptorHeader::bDescriptorType, tConfigDescriptor::bLength, NEXT_USB_DESCRIPTOR, readusb16_t, and tConfigDescriptor::wTotalLength.

2.2.5.6 void USBStackModeSet (

uint32_t ui32Index,

tUSBMode iUSBMode,

tUSBModeCallback pfnCallback)

Allows dual mode application to switch between USB device and host modes and provides a method to force the controller into the desired mode.

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.

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.

References eUSBModeDevice, and eUSBModeHost.

2.3 USB Chapter 9 Definitions

Data Structures

struct tConfigDescriptor

struct tDescriptorHeader

struct tDeviceDescriptor

struct tDeviceQualifierDescriptor

struct tEndpointDescriptor

struct tInterfaceDescriptor

struct tString0Descriptor

struct tStringDescriptor

struct tUSBRequest

Macros

#define NEXT_USB_DESCRIPTOR(ptr)

#define USB3Byte(ui32Value)

#define USBLong(ui32Value)

#define USBShort(ui16Value)

2.3.1 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.

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.

The USB 2.0 specification may be downloaded from the USB Implementers Forum (USB-IF) web site at `http://www.usb.org/developers/docs/`.

2.3.2 Macro Definition Documentation

2.3.2.1 #define NEXT_USB_DESCRIPTOR(

ptr)

Traverse to the next USB descriptor in a block.

Parameters *ptr* points to the first byte of a descriptor in a block of USB descriptors.

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*.

Referenced by USBDescGet(), USBDescGetNum(), and USBDescGetNumAlternateInterfaces().

2.3.2.2 #define USB3Byte(

ui32Value)

Write a 24-bit value to a USB descriptor block.

Parameters *ui32Value* is the 24-bit value that to write to the descriptor.

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.

2.3.2.3 #define USBLong(

ui32Value)

Write a 32-bit value to a USB descriptor block.

Parameters *ui32Value* is the 32-bit value that to write to the descriptor.

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.

2.3.2.4 #define USBShort(ui16Value)

Write a 16-bit value to a USB descriptor block.

Parameters *ui16Value* is the 16-bit value to write to the descriptor.

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.

2.4 USB Buffer and Ring Buffer APIs

Data Structures

struct tUSBBuffer

struct tUSBRingBufObject

Macros

#define USB_BUFFER_WORKSPACE_SIZE

Typedefs

typedef uint32_t(* tUSBPacketAvailable)(void *pvHandle)

typedef uint32_t(* tUSBPacketTransfer)(void *pvHandle, uint8_t *pi8Data, uint32_t ui32Length, bool bLast)

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 (const 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)

2.4.1 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.

2.4.2 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 USBBuffer-

Write() 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.

2.4.3 Macro Definition Documentation

2.4.3.1 #define USB_BUFFER_WORKSPACE_SIZE

The number of bytes of workspace that each USB buffer object requires. This workspace memory is provided to the buffer on USBBufferInit() in the *pvWorkspace* field of the *tUSBBuffer* structure.

2.4.4 Typedef Documentation

2.4.4.1 typedef uint32_t(∗ tUSBPacketAvailable)(void ∗pvHandle)

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.

2.4.4.2 typedef uint32_t(∗ tUSBPacketTransfer)(void ∗pvHandle, uint8_t ∗pi8Data, uint32_t ui32Length, bool bLast)

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.

2.4.5 Function Documentation

2.4.5.1 void∗ USBBufferCallbackDataSet (

tUSBBuffer ∗ psBuffer,

void ∗ pvCBData)

Sets the callback pointer supplied to clients of this buffer.

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.

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.

References tUSBBuffer::pvCBData.

2.4.5.2 uint32_t USBBufferDataAvailable (

const **tUSBBuffer** * psBuffer)

Returns the number of bytes of data available in the buffer.

Parameters *psBuffer* is the pointer to the buffer instance which is to be queried.

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 USBBufferRead(). 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.

References tUSBBuffer::pvWorkspace, and USBRingBufUsed().

2.4.5.3 void USBBufferDataRemoved (

const **tUSBBuffer** * psBuffer,

uint32_t ui32Length)

Indicates that a client has read data directly out of the buffer.

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.

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 USBBufferInfoGet() 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.

References tUSBBuffer::pvWorkspace, and USBRingBufAdvanceRead().

2.4.5.4 void USBBufferDataWritten (

const **tUSBBuffer** * psBuffer,

uint32_t ui32Length)

Indicates that a client has written data directly into the buffer and wishes to start transmission.

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.

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.

References tUSBBuffer::pvWorkspace, and USBRingBufAdvanceWrite().

2.4.5.5 uint32_t USBBufferEventCallback (

void * pvCBData,

uint32_t ui32Event,

uint32_t ui32MsgValue,

void * pvMsgData)

Called by the USB buffer to notify the client of asynchronous events.

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.

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.

722.7

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.

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.

References tUSBBuffer::pfnAvailable, tUSBBuffer::pfnCallback, tUSBBuffer::pfnTransfer, tUSBBuffer::pui8Buffer, tUSBBuffer::pvWorkspace, tUSBBuffer::ui32BufferSize, and USBRingBufInit().

2.4.5.9 uint32_t USBBufferRead (

const **tUSBBuffer** * psBuffer,

uint8_t * pui8Data,

uint32_t ui32Length)

Reads a block of data from a USB receive buffer into storage supplied by the caller.

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.

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.

References tUSBBuffer::pvWorkspace, USBRingBufRead(), and USBRingBufUsed().

2.4.5.10 uint32_t USBBufferSpaceAvailable (

const **tUSBBuffer** * psBuffer)

Returns the number of free bytes in the buffer.

Parameters *psBuffer* is the pointer to the buffer instance which is to be queried.

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.

References tUSBBuffer::pvWorkspace, and USBRingBufFree().

2.4.5.11 uint32_t USBBufferWrite (

const **tUSBBuffer** * psBuffer,

const uint8_t * pui8Data,

uint32_t ui32Length)

Writes a block of data to the transmit buffer and queues it for transmission to the USB controller.

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.

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.

References tUSBBuffer::bTransmitBuffer, tUSBBuffer::pvWorkspace, USBRingBufFree(), and USBRingBufWrite().

2.4.5.12 void USBBufferZeroLengthPacketInsert (

const **tUSBBuffer** * psBuffer,

bool bSendZLP)

Enables or disables zero-length packet insertion.

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.

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.

References tUSBBuffer::pvWorkspace.

2.4.5.13 void USBRingBufAdvanceRead (

tUSBRingBufObject * psUSBRingBuf,

uint32_t ui32NumBytes)

Removes bytes from the ring buffer by advancing the read index.

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.

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.

References tUSBRingBufObject::ui32ReadIndex, tUSBRingBufObject::ui32Size, and USBRingBufUsed().

Referenced by USBBufferDataRemoved().

2.4.5.14 void USBRingBufAdvanceWrite (

tUSBRingBufObject * psUSBRingBuf,

uint32_t ui32NumBytes)

Adds bytes to the ring buffer by advancing the write index.

Parameters *psUSBRingBuf* points to the ring buffer to which bytes have been added.

---

ui32NumBytes is the number of bytes added to the buffer.

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.

References tUSBRingBufObject::ui32ReadIndex, tUSBRingBufObject::ui32Size, tUSBRingBufObject::ui32WriteIndex, and USBRingBufFree().

Referenced by USBBufferDataWritten().

2.4.5.15 uint32_t USBRingBufContigFree (

tUSBRingBufObject * psUSBRingBuf)

Returns number of contiguous free bytes available in a ring buffer.

Parameters *psUSBRingBuf* is the ring buffer object to check.

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.

References tUSBRingBufObject::ui32ReadIndex, tUSBRingBufObject::ui32Size, and tUSBRingBufObject::ui32WriteIndex.

2.4.5.16 uint32_t USBRingBufContigUsed (

tUSBRingBufObject * psUSBRingBuf)

Returns number of contiguous bytes of data stored in ring buffer ahead of the current read pointer.

Parameters *psUSBRingBuf* is the ring buffer object to check.

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.

References tUSBRingBufObject::ui32ReadIndex, tUSBRingBufObject::ui32Size, and tUSBRingBufObject::ui32WriteIndex.

2.4.5.17 bool USBRingBufEmpty (

tUSBRingBufObject * psUSBRingBuf)

Determines whether a ring buffer is empty or not.

Parameters *psUSBRingBuf* is the ring buffer object to empty.

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.

References tUSBRingBufObject::ui32ReadIndex, and tUSBRingBufObject::ui32WriteIndex.

2.4.5.18 void USBRingBufFlush (

tUSBRingBufObject * psUSBRingBuf)

Empties the ring buffer.

Parameters *psUSBRingBuf* is the ring buffer object to empty.

Discards all data from the ring buffer.

Returns None.

References tUSBRingBufObject::ui32ReadIndex, and tUSBRingBufObject::ui32WriteIndex.

Referenced by USBBufferFlush().

2.4.5.19 uint32_t USBRingBufFree (

tUSBRingBufObject * psUSBRingBuf)

Returns number of bytes available in a ring buffer.

Parameters *psUSBRingBuf* is the ring buffer object to check.

This function returns the number of bytes available in the ring buffer.

Returns Returns the number of bytes available in the ring buffer.

References tUSBRingBufObject::ui32Size, and USBRingBufUsed().

Referenced by USBBufferSpaceAvailable(), USBBufferWrite(), USBRingBufAdvanceWrite(), USBRingBufWrite(), and USBRingBufWriteOne().

2.4.5.20 bool USBRingBufFull (

tUSBRingBufObject * psUSBRingBuf)

Determines whether a ring buffer is full or not.

Parameters *psUSBRingBuf* is the ring buffer object to empty.

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.

References tUSBRingBufObject::ui32ReadIndex, tUSBRingBufObject::ui32Size, and tUSBRingBufObject::ui32WriteIndex.

2.4.5.21 void USBRingBufInit (

tUSBRingBufObject * psUSBRingBuf,

uint8_t * pui8Buf,

uint32_t ui32Size)

Initializes a ring buffer object.

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.

This function initializes a ring buffer object, preparing it to store data.

Returns None.

References tUSBRingBufObject::pui8Buf, tUSBRingBufObject::ui32ReadIndex, tUSBRingBufObject::ui32Size, and tUSBRingBufObject::ui32WriteIndex.

Referenced by USBBufferInit().

2.4.5.22 void USBRingBufRead (

tUSBRingBufObject * psUSBRingBuf,

uint8_t * pui8Data,

uint32_t ui32Length)

Reads data from a ring buffer.

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.

This function reads a sequence of bytes from a ring buffer.

Returns None.

References USBRingBufReadOne(), and USBRingBufUsed().

Referenced by USBBufferRead().

2.4.5.23 uint8_t USBRingBufReadOne (

tUSBRingBufObject * psUSBRingBuf)

Reads a single byte of data from a ring buffer.

Parameters *psUSBRingBuf* points to the ring buffer to be written to.

This function reads a single byte of data from a ring buffer.

Returns The byte read from the ring buffer.

References tUSBRingBufObject::pui8Buf, tUSBRingBufObject::ui32ReadIndex, tUSBRingBufObject::ui32Size, and USBRingBufUsed().

Referenced by USBRingBufRead().

2.4.5.24 uint32_t USBRingBufSize (

tUSBRingBufObject * psUSBRingBuf)

Returns the size in bytes of a ring buffer.

Parameters *psUSBRingBuf* is the ring buffer object to check.

This function returns the size of the ring buffer.

Returns Returns the size in bytes of the ring buffer.

References tUSBRingBufObject::ui32Size.

2.4.5.25 uint32_t USBRingBufUsed (

tUSBRingBufObject * psUSBRingBuf)

Returns number of bytes stored in ring buffer.

Parameters *psUSBRingBuf* is the ring buffer object to check.

This function returns the number of bytes stored in the ring buffer.

Returns Returns the number of bytes stored in the ring buffer.

References tUSBRingBufObject::ui32ReadIndex, tUSBRingBufObject::ui32Size, and tUSBRingBufObject::ui32WriteIndex.

Referenced by USBBufferDataAvailable(), USBBufferRead(), USBRingBufAdvanceRead(), USBRingBufFree(), USBRingBufRead(), and USBRingBufReadOne().

2.4.5.26 void USBRingBufWrite (

tUSBRingBufObject * psUSBRingBuf,

const uint8_t * pui8Data,

uint32_t ui32Length)

Writes data to a ring buffer.

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.

This function write a sequence of bytes into a ring buffer.

Returns None.

References USBRingBufFree(), and USBRingBufWriteOne().

Referenced by USBBufferWrite().

2.4.5.27 void USBRingBufWriteOne (

tUSBRingBufObject * psUSBRingBuf,

uint8_t ui8Data)

Writes a single byte of data to a ring buffer.

Parameters *psUSBRingBuf* points to the ring buffer to be written to.

---

ui8Data is the byte to be written.

This function writes a single byte of data into a ring buffer.

Returns None.

References tUSBRingBufObject::pui8Buf, tUSBRingBufObject::ui32Size, tUSBRingBufObject::ui32WriteIndex, and USBRingBufFree().

Referenced by USBRingBufWrite().

3 Device Functions

Introduction ..... ??
API choices for USB devices ..... ??
Bulk Device Class Driver ..... ??
Bulk Device Class Driver Definitions 46
CDC Device Class Driver ..... ??
CDC Device Class Driver Definitions 58
Composite Device Class Driver ..... ??
Composite Device Class Driver Definitions 71
HID Device Class Driver ..... ??
HID Device Class Driver Definitions 80
HID Mouse Device Class API ..... ??
HID Mouse Device Class API Definitions 100
HID Keyboard Device Class API ..... ??
HID Keyboard Device Class API Definitions 107
Using the USB Device API ..... ??
Device Function Definitions 123

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.

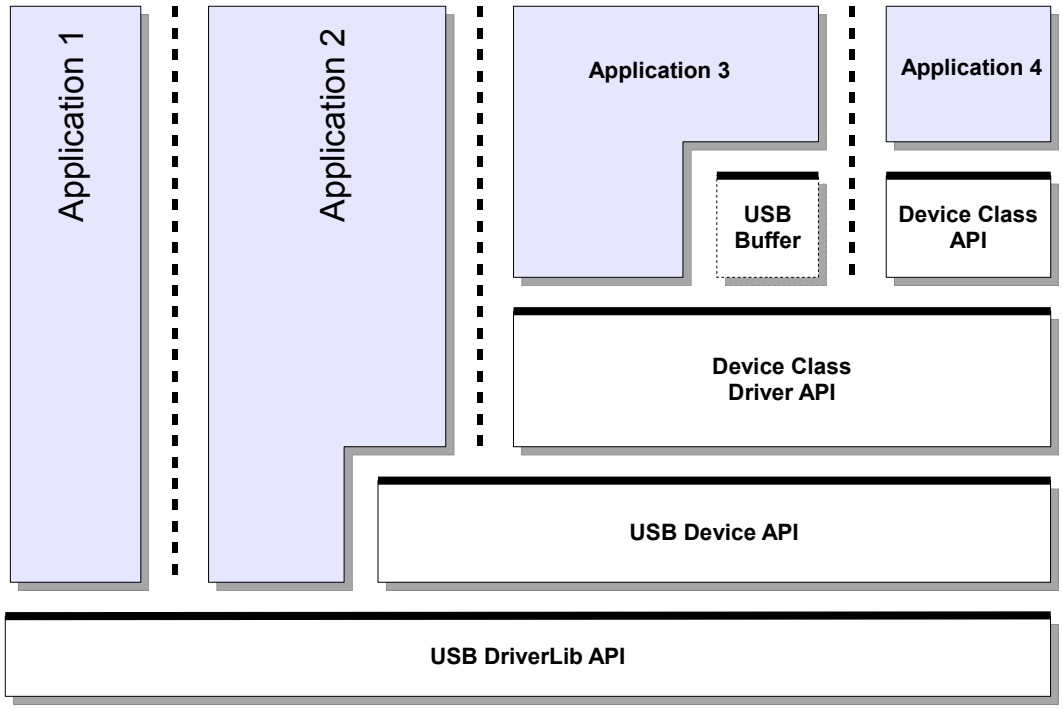
The USB library contains four API layers relevant to the development of USB device applications. Moving down the stack, each API layer offers greater flexibility to an application but this is balanced by the greater effort required to use the lower layers. The available programming interfaces, starting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

3.1 USB DriverLib API

The lowest layer in the USB device stack is the USB driver which can be found within the C2000Ware MWare 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.

3.2 USB Device API

The USB 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`.

3.3 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.

3.4 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.

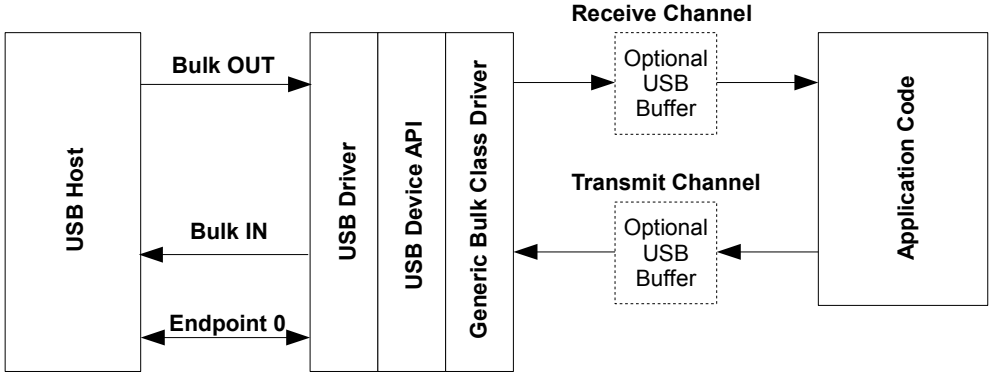
The Human Interface Device (HID) class is one such class. It is used to support a wide variety of devices including keyboards, joysticks, mice and game controllers but the interface is specified in such a way that it could be used for a huge number of vendor-specific devices offering data gathering capability. 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. 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 will have 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 to pass 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.

3.5 Generic Bulk Device Class Events

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

3.5.1 Receive Channel Events

`USB_EVENT_RX_AVAILABLE`

`USB_EVENT_ERROR`

`USB_EVENT_CONNECTED`

`USB_EVENT_DISCONNECTED`

`USB_EVENT_SUSPEND`

`USB_EVENT_RESUME`

3.5.2 Transmit Channel Events

`USB_EVENT_TX_COMPLETE`

3.6 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 "include/usblib.h" include "include/device/usbdevice.h" include "include/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 unsigned char g_pLangDescriptor[] = { 4, USB_DTYPE_STRING, USBShort(USB_LANG_EN_US) };

//*** // // The manufacturer string. // //*** const unsigned char g_pManufacturerString[] = {

2 + (22 * 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, ' ', 0, 'I', 0, 'n', 0, 'c', 0, '.', 0 };

//** // // The product string. // //** const unsigned char g_pProductString[] = { (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 unsigned char g_pSerialNumberString[] = { (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 unsigned char g_pDataInterfaceString[] = { (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 unsigned char g_pConfigString[] = { (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 unsigned char * const g_pStringDescriptors[] = { g_pLangDescriptor, g_pManufacturerString, g_pProductString, g_pSerialNumberString, g_pDataInterfaceString, g_pConfigString };

define NUM_STRING_DESCRIPTORS (sizeof(g_pStringDescriptors) / \ sizeof(unsigned char *))

Define an area of RAM of for the private data for the bulk device class driver. This structure should never be accessed by the application.

//** // // The bulk device private data. // //** tBulkInstance g_sBulkInstance;

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 example 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_pStringDescriptors,

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

// // A pointer to the private memroy allocated for the class driver to use. // g_sBulkInstance };

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 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 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 will be called back once again a few milliseconds later. Although no other events must be handled, `USB_EVENT_CONNECTED` and `USB_EVENT_DISCONNECTED` will 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 will fail.

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 generic bulk device class, there are no events sent to the transmit callback which MUST be handled but applications will 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. pDevice = USBDBulkInit(0, g_sBulkDevice);

Assuming `pDevice` returned is not NULL, your device is now ready to communicate with a USB host.

Once the host connects, your receive event handler will be 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.

3.7 Using the Composite Bulk Device Class

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 will prepare an instance of the bulk device class to be enumerated as part of a composite device. The return value from the USBDBulkCompositeInit() function should be placed in the pvInstance member of the tCompositeEntry structure for the bulk device. The code example below provides an example of how to initialize the tCompositeEntry structure.

// // These should be initialized with valid values for each class. // extern tUSBDCompositeDevice g_sCompDevice; extern tUSBDBulkDevice g_sBulkDevice;

// // 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) unsigned char g_pucDescriptorData[DESCRIPTOR_DATA_SIZE];

tCompositeEntry psCompEntries[2];

// // Set the generic bulk device information. // psCompEntries[0].psDevice = g_sBulkDeviceInfo;

// // Save the instance data for this bulk device. // psCompEntries[0].pvInstance = USBDBulkCompositeInit(0, g_sBulkDevice);

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

...

// // Save the device entries in the composite device. // g_sCompDevice.psDevices = psCompEntries;

...

USBDCompositeInit(0, g_sCompDevice, DESCRIPTOR_DATA_SIZE, g_pucDescriptorData);

All other API calls to the USB bulk device class should use the value returned by USBDBulkCompositeInit() 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_pucDescriptorData array as shown in the example above.

3.8 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 will be 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 C2000Ware USB Windows example application "USB Bulk Example". This application can be found in the "tools" directory of MWare.

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.

3.8.1 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).

; ——————————————————————— ; ; TI Generic Bulk USB device driver installer ; ; This INF file may be used as a template when creating customized applications ; based on the generic bulk device class. 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 = "$WindowsNT$" Class = TIBulkDeviceClass ClassGuid={F5450C06-EB58-420e-8F98-A76C5D4AFB18} Provider = CatalogFile=MyCatFile.cat

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

[Manufacturer]

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

[TIBulkDevice_WinUSB.NTx86]

[TIBulkDevice_WinUSB.NTamd64]

; ================== Installation ==================

[ClassInstall32] AddReg=AddReg_ClassInstall

[AddReg_ClassInstall] HKR,,,

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

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

[WinUSB_ServiceInstall] DisplayName = ServiceType = 1 StartType = 3 ErrorControl = 1 ServiceBinary =

[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","WinUSBCoIn

[CoInstallers_CopyFiles] WinUSBCoInstaller.dll WdfCoInstaller01005.dll

[DestinationDirs] CoInstallers_CopyFiles=11

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

[SourceDisksNames] 1 = 2 =

[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\TIBulkDevice.DeviceDesc="Generic Bulk Device" WinUSB_SvcDesc="WinUSB" DISK_NAME="TI Install Disk" DeviceClassDisplayName=TI Bulk Devices"

3.8.2 Sample libusb-win32 INF File

The following is an example of an INF file that can be used to associate the `usb_dev_bulk` example 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 = 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]

;——————————————————————— ; 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" Service-Type = 1 StartType = 3 ErrorControl = 0 ServiceBinary =

;———————————————————— ; Devices ;————————————————————

[Devices] "Generic Bulk Device"=LIBUSB_DEV, USB\VID_1cbePID_0003

[Devices.NT] "Generic Bulk Device"=LIBUSB_DEV, USB\VID_1cbePID_0003

[Devices.NTAMD64] "Generic Bulk Device"=LIBUSB_DEV, USB\VID_1cbePID_0003

;———————————————————— ; Strings ;————————————————————

[Strings] manufacturer = "Texas Instruments"

3.9 Bulk Device Class Driver Definitions

Data Structures

struct tUSBDBulkDevice

Macros

#define 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)

3.9.1 Detailed Description

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

3.9.2 Macro Definition Documentation

3.9.2.1 #define COMPOSITE_DBULK_SIZE

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.

3.9.3 Function Documentation

3.9.3.1 void * USBDBulkCompositeInit (

uint32_t ui32Index,

tUSBDBulkDevice * psBulkDevice,

tCompositeEntry * psCompEntry)

Initializes bulk device operation for a given USB controller.

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.

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.

References tUSBDBulkDevice::pfnRxCallback, tUSBDBulkDevice::pfnTxCallback, tUSBDBulkDevice::ppui8StringDescriptors, tCompositeEntry::psDevInfo, tCompositeEntry::pvInstance, tUSBDBulkDevice::sPrivateData, tUSBDBulkDevice::ui32NumStringDescriptors, and USBDCDDeviceInfoInit().

Referenced by USBDBulkInit().

3.9.3.2 void * USBDBulkInit (

uint32_t ui32Index,

tUSBDBulkDevice * psBulkDevice)

Initializes bulk device operation for a given USB controller.

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.

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 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, 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.

References tConfigDescriptor::bmAttributes, tConfigDescriptor::bMaxPower, tDeviceDescriptor::idProduct, tDeviceDescriptor::idVendor, tUSBDBulkDevice::sPrivateData, tUSBDBulkDevice::ui16MaxPowermA, tUSBDBulkDevice::ui16PID, tUSBDBulkDevice::ui16VID, tUSBDBulkDevice::ui8PwrAttributes, USBDBulkCompositeInit(), USBDCDInit(), and writeusb16_t.

3.9.3.3 uint32_t USBDBulkPacketRead (

void * pvBulkDevice,

uint8_t * pi8Data,

uint32_t ui32Length,

bool bLast)

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

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.

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.

3.9.3.4 uint32_t USBDBulkPacketWrite (

void * pvBulkDevice,

uint8_t * pi8Data,

uint32_t ui32Length,

bool bLast)

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

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.

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.

3.9.3.5 void USBDBulkPowerStatusSet (

void * pvBulkDevice,

uint8_t ui8Power)

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

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**.

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.

References USBDCDPowerStatusSet().

3.9.3.6 bool USBDBulkRemoteWakeupRequest (

void * pvBulkDevice)

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

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

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.

References USBDCDRemoteWakeupRequest().

3.9.3.7 uint32_t USBDBulkRxPacketAvailable (

void * pvBulkDevice)

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

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

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.

3.9.3.8 void * USBDBulkSetRxCBData (

void * pvBulkDevice,

void * pvCBData)

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

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.

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.

3.9.3.9 void * USBDBulkSetTxCBData (

void * pvBulkDevice,

void * pvCBData)

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

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.

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.

3.9.3.10 void USBDBulkTerm (

void * pvBulkDevice)

Shut down the bulk device.

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

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.

References USBDCDTerm().

3.9.3.11 uint32_t USBDBulkTxPacketAvailable (

void * pvBulkDevice)

Returns the number of free bytes in the transmit buffer.

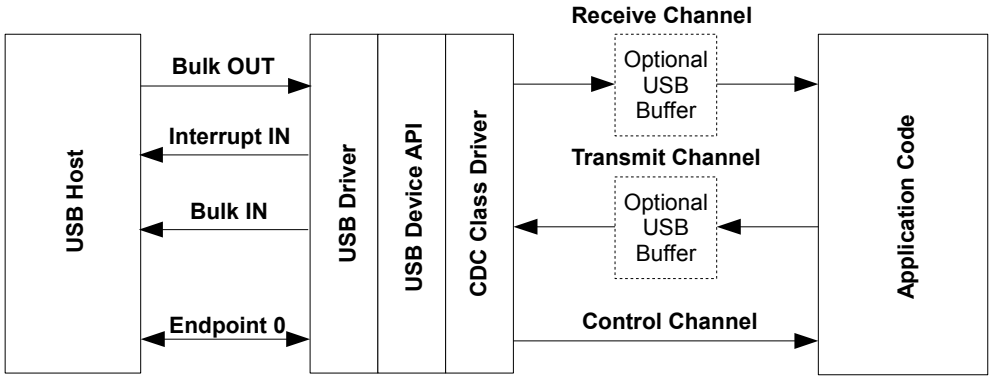
Parameters *pvBulkDevice* is the pointer to the device instance structure as returned by USBDBulkInit().

This function returns the maximum number of bytes that can be passed on a call to USBDBulkPacketWrite 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. 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 will likely be 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.

3.10 CDC Device Class Events

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

3.10.1 Receive Channel Events

USB_EVENT_RX_AVAILABLE

USB_EVENT_DATA_REMAINING

USB_EVENT_ERROR

3.10.2 Transmit Channel Events

USB_EVENT_TX_COMPLETE

3.10.3 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

3.11 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 "include/usblib.h" include "include/device/usbdevice.h" include "include/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 unsigned char g_pLangDescriptor[] = { 4, USB_DTYPE_STRING, USBShort(USB_LANG_EN_US) };

//***
//
// The manufacturer string.
//
//***
const unsigned char g_pManufacturerString[] = {

2 + (22 * 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, ' ', 0, 'I', 0, 'n', 0, 'c', 0, '.', 0 };

//***
//
// The product string.
//
//***
const unsigned char g_pProductString[] = { 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 unsigned char g_pSerialNumberString[] = { 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 unsigned char g_pControlInterfaceString[] = { 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 unsigned char g_pConfigString[] = { 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 unsigned char * const g_pStringDescriptors[] = { g_pLangDescriptor, g_pManufacturerString, g_pProductString, g_pSerialNumberString, g_pControlInterfaceString, g_pConfigString };

define NUM_STRING_DESCRIPTORS (sizeof(g_pStringDescriptors) / \ sizeof(unsigned char *))

Define a tCDCSerInstance structure which the USB CDC serial device class driver uses to store its internal state information. This should never be accessed by the application.

//***
//
// The CDC serial device private data.
//
//***
tCDCSerInstance g_sSerialInstance;

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_pStringDescriptors,

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

// // A pointer to the private structure allocated for the class driver to use. // g_sSerialInstance };

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 will be 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` will 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 will fail.

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 will 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. pDevice = USBDCDCInit(0, g_sCDCDevice);

Assuming `pDevice` returned is not NULL, your device is now ready to communicate with a USB host.

Once the host connects, your control event handler will be 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.

3.12 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 will prepare an instance of the CDC serial device class to be enumerated as part of a composite device. The return value from the USBDCDCCompositeInit() function should be placed in the pvInstance member of the tCompositeEntry structure for the CDC serial device. The code example below provides an example of how to initialize the tCompositeEntry structure.

// // These should be initialized with valid values for each class. // extern tUSBDCompositeDevice g_sCompDevice; extern tUSBDCDCDevice g_sCDCDevice;

// // 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) unsigned char g_pucDescriptorData[DESCRIPTOR_DATA_SIZE];

tCompositeEntry psCompEntries[2];

// // Set the CDC serial device information. // psCompEntries[0].psDevice = g_sCDCSerDeviceInfo;

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

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

...

// // Save the device entries in the composite device. // g_sCompDevice.psDevices = psCompEntries;

...

USBDCompositeInit(0, g_sCompDevice, DESCRIPTOR_DATA_SIZE, g_pucDescriptorData);

All other API calls to the USB CDC serial device class should use the value returned by USBDCDCCompositeInit() 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_pucDescriptorData array as shown in the example above.

3.13 Windows Drivers for CDC Serial Devices

Making your CDC serial) device visible as a virtual COM port on a Windows system is very straightforward 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 USB CDC (serial) driver installation file. ; [Version] Signature="$WindowsNT$" Class=Ports ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318} Provider=LayoutFile=layout.inf DriverVer=08/17/2001,5.1.2600.0

[Manufacturer]

[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. ;

; ; These entries are for the dual serial port composite example usb_dev_cserial. ;

; ; 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. ;

;———————————————————————— ; 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=ServiceType=1 StartType=3 ErrorControl=1 ServiceBinary=

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

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

3.14 CDC Device Class Driver Definitions

Data Structures

struct tLineCoding

struct tUSBDCDCDevice

Macros

#define COMPOSITE_DCDC_SIZE

#define USBD_CDC_EVENT_CLEAR_BREAK

#define USBD_CDC_EVENT_GET_LINE_CODING

#define USBD_CDC_EVENT_SEND_BREAK

#define USBD_CDC_EVENT_SET_CONTROL_LINE_STATE

#define 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)

3.14.1 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 will also need to include `usbcdc.h` which contains general CDC definitions required by both host and device implementations.

3.14.2 Macro Definition Documentation

3.14.2.1 #define COMPOSITE_DCDC_SIZE

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.

3.14.2.2 #define USBD_CDC_EVENT_CLEAR_BREAK

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

3.14.2.3 #define USBD_CDC_EVENT_GET_LINE_CODING

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.

3.14.2.4 #define USBD_CDC_EVENT_SEND_BREAK

The host requests that the device send a BREAK condition on its serial communication channel. The BREAK should remain active until a USBD_CDC_EVENT_CLEAR_BREAK event is received.

3.14.2.5 #define USBD_CDC_EVENT_SET_CONTROL_LINE_STATE

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

3.14.2.6 #define USBD_CDC_EVENT_SET_LINE_CODING

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.

3.14.3 Function Documentation

3.14.3.1 void * USBDCDCCompositeInit (

uint32_t ui32Index,

tUSBDCDCDevice * psCDCDevice,

tCompositeEntry * psCompEntry)

Initializes CDC device operation when used with a composite device.

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.

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.

References tUSBDCDCDevice::pfnControlCallback, tUSBDCDCDevice::pfnRxCallback, tUSBDCDCDevice::pfnTxCallback, tUSBDCDCDevice::ppui8StringDescriptors, tCompositeEntry::psDevInfo, tCompositeEntry::pvInstance, tUSBDCDCDevice::sPrivateData, tUSBDCDCDevice::ui32NumStringDescriptors, and USBDCDDeviceInfoInit().

Referenced by USBDCDCInit().

3.14.3.2 void * USBDCDCInit (

uint32_t ui32Index,

tUSBDCDCDevice * psCDCDevice)

Initializes CDC device operation for a given USB controller.

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.

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.

References tConfigDescriptor::bmAttributes, tConfigDescriptor::bMaxPower, tDeviceDescriptor::idProduct, tDeviceDescriptor::idVendor, tUSBDCDCDevice::sPrivateData, tUSBDCDCDevice::ui16MaxPowermA, tUSBDCDCDevice::ui16PID, tUSBDCDCDevice::ui16VID, tUSBDCDCDevice::ui8PwrAttributes, USBDCDCCompositeInit(), USBDCDInit(), and writeusb16_t.

3.14.3.3 uint32_t USBDCDCPacketRead (

void * pvCDCDevice,

uint8_t * pi8Data,

uint32_t ui32Length,

bool bLast)

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

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.

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 determined without relying upon the client to provide this information. Returns Returns the number of bytes of data read.

3.14.3.4 uint32_t USBDCDCPacketWrite (

void * pvCDCDevice,

uint8_t * pi8Data,

uint32_t ui32Length,

bool bLast)

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

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.

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.

3.14.3.5 void USBDCDCPowerStatusSet (

void * pvCDCDevice,

uint8_t ui8Power)

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

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**.

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.

722.7

file usbcdc.h.

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.

3.14.3.9 void * USBDCDCSetControlCBData (

void * pvCDCDevice,

void * pvCBData)

Sets the client-specific pointer for the control callback.

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.

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.

References tUSBDCDCDevice::pvControlCBData.

3.14.3.10 void * USBDCDCSetRxCBData (

void * pvCDCDevice,

722.7

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

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.

References USBDCDTerm().

3.14.3.13 uint32_t USBDCDCTxPacketAvailable (

void * pvCDCDevice)

Returns the number of free bytes in the transmit buffer.

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

This function returns the maximum number of bytes that can be passed on a call to USBDCDCPacketWrite() 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. 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.

3.15 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

ppStringDescriptors and ulNumStringDescriptors 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 your microcontroller has enough endpoints to satisfy the number of devices included in the composite class.

Example: tCompositeInstance g_CompInstance; unsigned long g_pulCompWorkspace[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,

// // The instance data for the composite class. The first value provided // here must point to a buffer containing NUM_DEVICES unsigned longs. This // is used by the composite device class to hold a lookup table allowing it // to correctly steering callbacks to the relevant interface or endpoint. // The second value is a pointer to the class instance structure which // provides storage for state information required for operation of the // class. // // g_pulCompWorkspace, g_CompInstance };

3.16 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 first is the simple instance data that is provided in the tCompositeInstance structure and is shown in the previous section. This is a static allocation for the composite device instance and is used internally by the composite device class. The second 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 will provide 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. The last type of allocation is any data that is needed by the individual device classes. Each instance of each class will likely need some memory allocated separately to each instance of a device.

3.16.1 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. unsigned char pucDesciptorData[COMPOSITE_DCDC_SIZE*2]; The application must also provide separate

instance data for each instance of the devices that it is including in the composite device. This is true even when including two devices classes of the same type 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 provides the instance data in the array of tCompositeEntry structures passed into the composite class in the psDevices member of the tUSBDCompositeDevice structure. Notice in the example below that the device information is common (g_sCDCSerDeviceInfo) but that each has its own instance data (sCDCDeviceA and sCDCDeviceB).

Example: Two serial instances extern tUSBDCDCDevice g_sCDCDeviceA; extern tUSBDCDCDevice g_sCDCDeviceB;

tCompositeEntry g_psDevices[2]= { { g_sCDCSerDeviceInfo, (void *)g_sCDCDeviceA }, { g_sCDCSerDeviceInfo, (void *)g_sCDCDeviceB } };

3.16.2 Interface Handling

The device class interfaces will be 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.

3.16.3 String Handling

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

3.17 Example Composite Device

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

3.17.1 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:

// // Instance and buffers for Serial Device A. // tCDCSerInstance g_sCDCInstanceA; const tUSBBuffer g_sTxBufferA; const tUSBBuffer g_sRxBufferA;

// // Instance and buffers for Serial Device B. // tCDCSerInstance g_sCDCInstanceB; const tUSBBuffer g_sTxBufferB; const tUSBBuffer g_sRxBufferB;

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

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

Now the application must allocate the device array so that it can be provided to the USB composite device class. The following code shows the two serial devices included above into the array of tCompositeEntry values. tCompositeEntry g_psDevices[2]= { { g_sCDCSerDeviceInfo, (void *)g_sCDCDeviceA }, { g_sCDCSerDeviceInfo, (void *)g_sCDCDeviceB } };

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: // // Allocate the composite instance data and workspace storage. // tCompositeInstance g_CompInstance; unsigned long g_pulCompWorkspace[NUM_DEVICES]; // // Initialize the USB composite device structure. // tUSBDCompositeDevice g_sCompDevice = { // // TI VID. // USB_VID_TI,

// // 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,

NUM_DEVICES, g_psCompDevices,

g_pulCompWorkspace, g_CompInstance };

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: unsigned char 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. // g_sCompDevice.psCompInfo[0].pvInstance = USBDCDCCompositeInit(0, (tUSBDCDCDevice *)g_sCDCDeviceA); g_sCompDevice.psCompInfo[1].pvInstance = USBDCDCCompositeInit(0, (tUSBDCDCDevice *)g_sCDCDeviceB);

// // 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 were included with the 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, ControlHandler(). 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: unsigned long Control-

Handler(void *pvCBData, unsigned long ulEvent, unsigned long ulMsgValue, void *pvMsgData) { tUSBDCDCDevice pCDCDevice;

pCDCDevice = (tUSBDCDCDevice *)pvCBData;

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

3.18 Composite Device Class Driver Definitions

Data Structures

struct tUSBDCompositeDevice

Functions

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

void USBDCompositeTerm (void *pvCompositeInstance)

3.18.1 Detailed Description

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

3.18.2 Function Documentation

3.18.2.1 void * USBDCompositeInit (

uint32_t ui32Index,

tUSBDCompositeDevice * psDevice,

uint32_t ui32Size,

uint8_t * pui8Data)

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

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.

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.

References tUSBDCompositeDevice::ppui8StringDescriptors, tConfigHeader::psSections, tUSBDCompositeDevice::sPrivateData, tUSBDCompositeDevice::ui16MaxPowermA, tUSBDCompositeDevice::ui16PID, tUSBDCompositeDevice::ui16VID, tUSBDCompositeDevice::ui32NumStringDescriptors, tConfigHeader::ui8NumSections, tUSBDCompositeDevice::ui8PwrAttributes, USBDCDDeviceInfoInit(), USBDCDInit(), and writeusb16_t.

3.18.2.2 void USBDCompositeTerm (

void * pvCompositeInstance)

Shuts down the composite device.

Parameters *pvCompositeInstance* is the pointer to the device instance structure as returned by USBDCompositeInit().

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. 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.

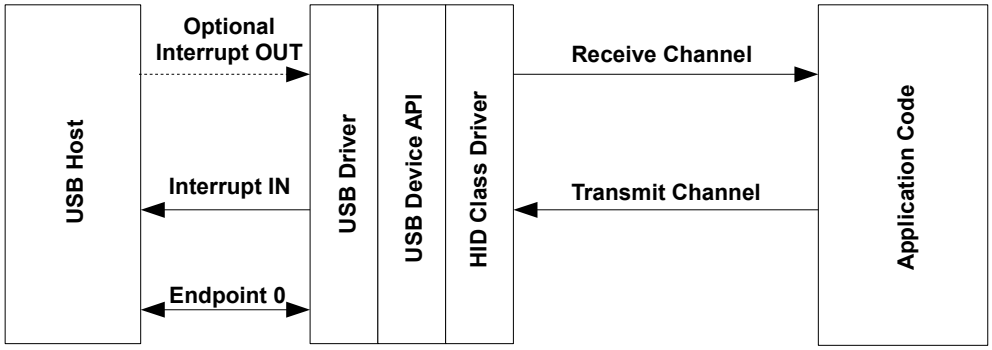
Communication between the HID device and host is via a collection of "report" structures which are defined by the device in HID report descriptors which the host can query. Reports are defined both for communication of device input to the host and for output and feature selection from the host.

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.

3.19 HID Device Class Events

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

3.19.1 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

3.19.2 Transmit Channel Events

USB_EVENT_TX_COMPLETE

3.20 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 "include/usblib.h" include "include/usbhid.h" include "include/device/usbdevice.h" include "include/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 will vary 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 will be 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 unsigned char g_pLangDescriptor[] = { 4, USB_DTYPE_STRING, USBShort(USB_LANG_EN_US) };

//*** // // The manufacturer string. // //*** const unsigned char g_pManufacturerString[] = {

2 + (22 * 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, ' ', 0, 'I', 0, 'n', 0, 'c', 0, '.', 0 };

//*** // // The product string. // //*** const unsigned char

g_pProductString[] = { (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 unsigned char g_pSerialNumberString[] = { (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 unsigned char g_pHIDInterfaceString[] = { (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 unsigned char g_pConfigString[] = { (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 unsigned char * const g_pStringDescriptors[] = { g_pLangDescriptor, g_pManufacturerString, g_pProductString, g_pSerialNumberString, g_pHIDInterfaceString, g_pConfigString };

define NUM_STRING_DESCRIPTORS (sizeof(g_pStringDescriptors) / \ sizeof(unsigned char *))

Define a tHIDInstance structure which the USB HID device class driver uses to store its internal state information. This should never be accessed by the application.

//*** // // The HID device private data. // //*** tHIDInstance g_sHIDInstance;

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 unsigned char g_pucMouseReportDescriptor[]= { 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), Us-

age(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 unsigned char * 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,

// // A pointer to the private instance data allocated for the class driver to // use. // g_sHIDInstance };

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` will 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 will fail.

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() will fail.

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. pDevice = USBDHIDMouseInit(0, g_sHIDMouseDevice);

Assuming `pDevice` returned is not NULL, your device is now ready to communicate with a USB host.

Once the host connects, your control event handler will be 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.

3.21 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 will prepare an instance of the HID mouse device class to be enumerated as part of a composite device. The return value from the USBDHIDMouseCompositeInit() function should be placed in the pvInstance member of the tCompositeEntry structure for the HID mouse device. The code example below provides an example of how to initialize the tCompositeEntry structure.

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

// // 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) unsigned char g_pucDescriptorData[DESCRIPTOR_DATA_SIZE];

tCompositeEntry psCompEntries[2];

// // Set the generic HID device information. // psCompEntries[0].psDevice = g_sHIDDeviceInfo;

// // Save the instance data for this HID mouse device. // psCompEntries[0].pvInstance = USBDHIDMouseCompositeInit(0, g_sHIDMouseDevice);

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

...

// // Save the device entries in the composite device. // g_sCompDevice.psDevices = psCompEntries;

...

USBDCompositeInit(0, g_sCompDevice, DESCRIPTOR_DATA_SIZE, g_pucDescriptorData);

All other API calls to the USB HID mose device class should use the value returned by USBDHIDMouseCompositeInit() 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_pucDescriptorData array as shown in the example above.

3.22 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 (64 bytes), 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 will be 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 will call the application back later to give it another opportunity. Until the packet is read, NAK will be 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 will not access the report buffer after this event is sent and the application may handle the memory as it wishes following this point.

3.23 HID Device Class Driver Definitions

Data Structures

struct tHIDClassDescriptorInfo

struct tHIDDescriptor

struct tHIDKeyboardUsageTable

struct tHIDReportIdle

struct tUSBDHIDDevice

Macros

#define Collection(ui8Value)

#define COMPOSITE_DHID_SIZE

#define EndCollection

#define Feature(ui8Value)

#define Feature2(ui16Value)

#define Input(ui8Value)

#define Input2(ui16Value)

#define LogicalMaximum(i8Value)

#define LogicalMinimum(i8Value)

#define Output(ui8Value)

#define Output2(ui16Value)

#define PhysicalMaximum(i16Value)

#define PhysicalMinimum(i16Value)

#define ReportCount(ui8Value)

#define ReportID(ui8Value)

#define ReportSize(ui8Value)

#define Unit(ui32Value)

#define UnitAccelerationSI

#define UnitAngAccelerationSI

#define UnitCurrent_A

#define UnitDistance_cm

#define UnitDistance_i

#define UnitEnergySI

#define UnitExponent(i8Value)

#define UnitForceSI

#define UnitMass_g

#define UnitMomentumSI

#define UnitRotation_deg

#define UnitRotation_rad

#define UnitTemp_F

#define UnitTemp_K

#define UnitTime_s

#define UnitVelocitySI

#define UnitVoltage

#define Usage(ui8Value)

#define UsageMaximum(ui8Value)

#define UsageMinimum(ui8Value)

#define UsagePage(ui8Value)

#define UsagePageVendor(ui16Value)

#define UsageVendor(ui16Value)

#define USBD_HID_EVENT_GET_PROTOCOL

#define USBD_HID_EVENT_GET_REPORT

#define USBD_HID_EVENT_GET_REPORT_BUFFER

#define USBD_HID_EVENT_IDLE_TIMEOUT

#define USBD_HID_EVENT_REPORT_SENT

#define USBD_HID_EVENT_SET_PROTOCOL

#define 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)

3.23.1 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 will also need to include `usbhid.h` which includes HID-related definitions required by both host and device implementations.

3.23.2 Macro Definition Documentation

3.23.2.1 #define Collection(

ui8Value)

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

Parameters *ui8Value* is the type of Collection.

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.

3.23.2.2 #define COMPOSITE_DHID_SIZE

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.

3.23.2.3 #define EndCollection

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.

3.23.2.4 #define Feature(

ui8Value)

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

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.

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.

3.23.2.5 #define Feature2(

ui16Value)

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

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.

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.

3.23.2.6 #define Input(

ui8Value)

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

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.

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.

3.23.2.7 #define Input2(

ui16Value)

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

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.

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.

3.23.2.8 #define LogicalMaximum(

i8Value)

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

Parameters *i8Value* is the Logical Maximum value.

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.

3.23.2.9 #define LogicalMinimum(

i8Value)

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

Parameters *i8Value* is the Logical Minimum value.

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.

3.23.2.10 #define Output(

ui8Value)

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

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.

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.

3.23.2.11 #define Output2(

ui16Value)

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

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 (bit 8) is not required, the Output macro may be used instead.

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.

3.23.2.12 #define PhysicalMaximum(

i16Value)

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

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

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.

3.23.2.13 #define PhysicalMinimum(

i16Value)

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

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

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.

3.23.2.14 #define ReportCount(

ui8Value)

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

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

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.

3.23.2.15 #define ReportID(

ui8Value)

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

Parameters *ui8Value* is the identifier prefix for the current report.

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.

3.23.2.16 #define ReportSize(

ui8Value)

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

Parameters *ui8Value* is the size, in bits, of items in a report item.

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.

3.23.2.17 #define Unit(

ui32Value)

This is a macro to assist adding Unit entries for uncommon units in HID report descriptors.

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.

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.

3.23.2.18 #define UnitAccelerationSI

This macro inserts a Unit entry for acceleration in cm/s∗∗2 into a report descriptor.

3.23.2.19 #define UnitAngAccelerationSI

This macro inserts a Unit entry for angular acceleration in degrees/s∗∗2 into a report descriptor.

3.23.2.20 #define UnitCurrent_A

This macro inserts a Unit entry for voltage into a a report descriptor.

3.23.2.21 #define UnitDistance_cm

This macro inserts a Unit entry for centimeters into a report descriptor.

3.23.2.22 #define UnitDistance_i

This macro inserts a Unit entry for inches into a report descriptor.

3.23.2.23 #define UnitEnergySI

This macro inserts a Unit entry for energy in (grams ∗ cm∗∗2)/s∗∗2 into a report descriptor.

3.23.2.24 #define UnitExponent(

i8Value)

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

Parameters *i8Value* is the required exponent in the range [-8, 7].

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.

3.23.2.25 #define UnitForceSI

This macro inserts a Unit entry for force in (cm ∗ grams)/s∗∗2 into a report descriptor.

3.23.2.26 #define UnitMass_g

This macro inserts a Unit entry for grams into a report descriptor.

3.23.2.27 #define UnitMomentumSI

This macro inserts a Unit entry for momentum in (grams ∗ cm)/s into a report descriptor.

3.23.2.28 #define UnitRotation_deg

This macro inserts a Unit entry for degrees into a report descriptor.

3.23.2.29 #define UnitRotation_rad

This macro inserts a Unit entry for radians into a report descriptor.

3.23.2.30 #define UnitTemp_F

This macro inserts a Unit entry for temperature in Fahrenheit into a report descriptor.

3.23.2.31 #define UnitTemp_K

This macro inserts a Unit entry for temperature in Kelvin into a report descriptor.

3.23.2.32 #define UnitTime_s

This macro inserts a Unit entry for seconds into a report descriptor.

3.23.2.33 #define UnitVelocitySI

This macro inserts a Unit entry for velocity in cm/s into a report descriptor.

3.23.2.34 #define UnitVoltage

This macro inserts a Unit entry for voltage into a a report descriptor.

3.23.2.35 #define Usage(

ui8Value)

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

Parameters *ui8Value* is the Usage value.

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.

3.23.2.36 #define UsageMaximum(

ui8Value)

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

Parameters *ui8Value* is the Usage Maximum value.

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.

3.23.2.37 #define UsageMinimum(

ui8Value)

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

Parameters *ui8Value* is the Usage Minimum value.

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.

3.23.2.38 #define UsagePage(

ui8Value)

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

Parameters *ui8Value* is the Usage Page value.

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.

3.23.2.39 #define UsagePageVendor(

ui16Value)

This is a macro to assist adding Usage Page entries in HID report descriptors when a vendor-specific value is to be used.

Parameters *ui8Value* is the Usage Page value.

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.

3.23.2.40 #define UsageVendor(

ui16Value)

This is a macro to assist adding vendor-specific Usage entries in HID report descriptors.

Parameters *ui16Value* is the vendor-specific Usage value in the range 0xFF00 to 0xFFFF.

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.

3.23.2.41 #define USBD_HID_EVENT_GET_PROTOCOL

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.

3.23.2.42 #define USBD_HID_EVENT_GET_REPORT

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.

3.23.2.43 #define USBD_HID_EVENT_GET_REPORT_BUFFER

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).

3.23.2.44 #define USBD_HID_EVENT_IDLE_TIMEOUT

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 will contain 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.

3.23.2.45 #define USBD_HID_EVENT_REPORT_SENT

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.

3.23.2.46 #define USBD_HID_EVENT_SET_PROTOCOL

This event is sent in response to a Set_Protocol request from the host. The ui32MsgData value will contain the requested protocol, USB_HID_PROTOCOL_BOOT or USB_HID_PROTOCOL_REPORT.

3.23.2.47 #define USBD_HID_EVENT_SET_REPORT

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 will previously have been returned in response to an earlier USBD_HID_EVENT_GET_REPORT_BUFFER callback. The HID device class driver will 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.

3.23.3 Function Documentation

3.23.3.1 void * USBDHIDCompositeInit (

uint32_t ui32Index,

tUSBDHIDDevice * psHIDDevice,

tCompositeEntry * psCompEntry)

Initializes HID device operation for a given USB controller.

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.

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.

References tEndpointDescriptor::bEndpointAddress, tUSBDHIDDevice::pfnRxCallback, tUSBDHIDDevice::pfnTxCallback, tUSBDHIDDevice::ppui8ClassDescriptors, tUSBDHIDDevice::ppui8StringDescriptors, tCompositeEntry::psDevInfo, tUSBDHIDDevice::psHIDDescriptor, tUSBDHIDDevice::psReportIdle, tCompositeEntry::pvInstance, tUSBDHIDDevice::sPrivateData, tUSBDHIDDevice::ui32NumStringDescriptors, tUSBDHIDDevice::ui8NumInputReports, and USBDCDDeviceInfoInit().

Referenced by USBDHIDInit(), USBDHIDKeyboardCompositeInit(), and USBDHIDMouseCompositeInit().

3.23.3.2 void * USBDHIDInit (

uint32_t ui32Index,

tUSBDHIDDevice * psHIDDevice)

Initializes HID device operation for a given USB controller.

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.

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.

References tDeviceDescriptor::idProduct, tDeviceDescriptor::idVendor, tUSBDHIDDevice::pfnRxCallback, tUSBDHIDDevice::pfnTxCallback, tUSBDHIDDevice::ppui8ClassDescriptors, tUSBDHIDDevice::ppui8StringDescriptors, tUSBDHIDDevice::psHIDDescriptor, tUSBDHIDDevice::psReportIdle, tUSBDHIDDevice::sPrivateData, tUSBDHIDDevice::ui16PID, tUSBDHIDDevice::ui16VID, tUSBDHIDDevice::ui8NumInputReports, USBDCDInit(), USBDHIDCompositeInit(), and writeusb16_t.

Referenced by USBDHIDKeyboardInit(), and USBDHIDMouseInit().

3.23.3.3 uint32_t USBDHIDPacketRead (

void * pvHIDInstance,

uint8_t * pi8Data,

uint32_t ui32Length,

bool bLast)

Reads a packet of data received from the USB host via the interrupt OUT endpoint (if in use).

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.

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.

3.23.3.4 void USBDHIDPowerStatusSet (

void * pvHIDInstance,

uint8_t ui8Power)

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

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**.

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.

References USBDCDPowerStatusSet().

Referenced by USBDHIDKeyboardPowerStatusSet(), and USBDHIDMousePowerStatusSet().

3.23.3.5 bool USBDHIDRemoteWakeupRequest (

void * pvHIDInstance)

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

Parameters *pvHIDInstance* is the pointer to the HID device instance structure.

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.

References USBDCDRemoteWakeupRequest().

Referenced by USBDHIDKeyboardRemoteWakeupRequest(), and USBDHIDMouseRemoteWakeupRequest().

3.23.3.6 uint32_t USBDHIDReportWrite (

void * pvHIDInstance,

uint8_t * pi8Data,

uint32_t ui32Length,

bool bLast)

Transmits a HID device report to the USB host via the HID interrupt IN endpoint.

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.

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.

Referenced by USBDHIDKeyboardKeyStateChange(), and USBDHIDMouseStateChange().

3.23.3.7 uint32_t USBDHIDRxPacketAvailable (

void * pvHIDInstance)

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

Parameters *pvHIDInstance* is the pointer to the device instance structure as returned by USBDHIDInit().

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.

3.23.3.8 void * USBDHIDSetRxCBData (

void * pvHIDInstance,

void * pvCBData)

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

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.

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.

3.23.3.9 void * USBDHIDSetTxCBData (

void * pvHIDInstance,

void * pvCBData)

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

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.

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.

3.23.3.10 void USBDHIDTerm (

void * pvHIDInstance)

Shuts down the HID device.

Parameters *pvHIDInstance* is the pointer to the device instance structure as returned by USBDHIDInit().

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.

References USBDCDTerm().

Referenced by USBDHIDKeyboardTerm(), and USBDHIDMouseTerm().

3.23.3.11 uint32_t USBDHIDTxPacketAvailable (

void * pvHIDInstance)

Returns the number of free bytes in the transmit buffer.

Parameters *pvHIDInstance* is the pointer to the device instance structure as returned by USBDHIDInit().

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.

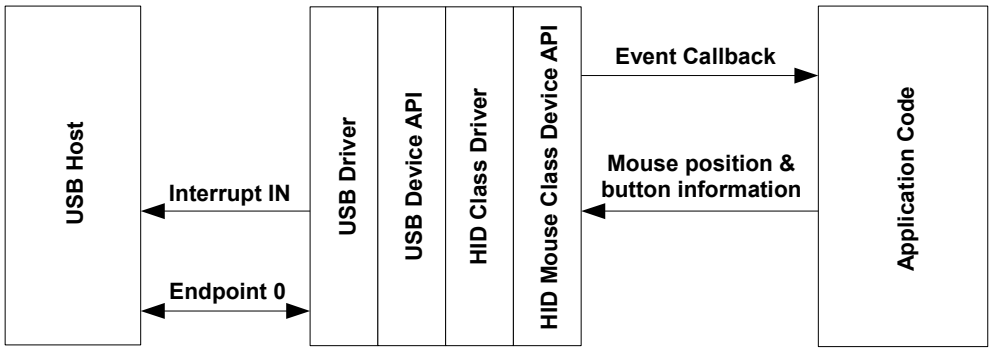
Referenced by USBDHIDKeyboardKeyStateChange(), and USBDHIDMouseStateChange().

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.

3.24 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`

3.25 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 "include/usblib.h" include "include/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 an area of RAM of for the private data for the HID mouse class driver. This structure should never be accessed by the application.

//** // // The HID mouse device private instance data. // //** static tHIDMouseInstance g_sMouseInstance;

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,

// // A pointer to the private instance data allocated for the API to use. // g_sMouseInstance };

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 will result 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. pDevice = USBDHIDMouseInit(0, g_sMouseDevice);

Assuming `pDevice` returned is not NULL, your mouse device is now ready to communicate with a USB host.

Once the host connects, your mouse event handler will be sent `USB_EVENT_CONNECTED` after which calls can be made to USBDHIDMouseStateChange()to inform the host of mouse position

and button state changes.

3.26 HID Mouse Device Class API Definitions

Data Structures

struct tUSBDHIDMouseDevice

Macros

#define MOUSE_ERR_NOT_CONFIGURED

#define MOUSE_ERR_TX_ERROR

#define MOUSE_REPORT_BUTTON_1

#define MOUSE_REPORT_BUTTON_2

#define MOUSE_REPORT_BUTTON_3

#define 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)

3.26.1 Detailed Description

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

722.7

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.

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.

References tUSBDHIDDevice::bUseOutEndpoint, tUSBDHIDMouseDevice::pfnCallback, tUSBDHIDDevice::pfnRxCallback, tUSBDHIDDevice::pfnTxCallback, tUSBDHIDDevice::ppui8ClassDescriptors, tUSBDHIDMouseDevice::ppui8StringDescriptors, tUSBDHIDDevice::ppui8StringDescriptors, tUSBDHIDDevice::psHIDDescriptor, tUSBDHIDDevice::psReportIdle, tUSBDHIDDevice::pvRxCBData, tUSBDHIDDevice::pvTxCBData, tUSBDHIDMouseDevice::sPrivateData, tUSBDHIDMouseDevice::ui16MaxPowermA, tUSBDHIDDevice::ui16MaxPowermA, tUSBDHIDMouseDevice::ui16PID, tUSBDHIDDevice::ui16PID, tUSBDHIDMouseDevice::ui16VID, tUSBDHIDDevice::ui16VID, tUSBDHIDMouseDevice::ui32NumStringDescriptors, tUSBDHIDDevice::ui32NumStringDescriptors, tUSBDHIDDevice::ui8NumInputReports, tUSBDHIDDevice::ui8Protocol, tUSBDHIDMouseDevice::ui8PwrAttributes, tUSBDHIDDevice::ui8PwrAttributes, tUSBDHIDDevice::ui8Subclass, and USBDHIDCompositeInit().

Referenced by USBDHIDMouseInit().

3.26.3.2 void * USBDHIDMouseInit (

uint32_t ui32Index,

tUSBDHIDMouseDevice * psMouseDevice)

Initializes HID mouse device operation for a given USB controller.

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.

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 will return 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 will receive **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 will cause unpredictable (though almost certainly unpleasant) behavior. Returns Returns NULL on failure or the psMouseDevice pointer on success.

References tConfigDescriptor::bmAttributes, tConfigDescriptor::bMaxPower, tUSBDHIDMouseDevice::pfnCallback, tUSBDHIDMouseDevice::ppui8StringDescriptors, tUSBDHIDMouseDevice::sPrivateData, tUSBDHIDMouseDevice::ui16MaxPowermA, tUSBDHIDMouseDevice::ui8PwrAttributes, USBDHIDInit(), and USBDHIDMouseCompositeInit().

3.26.3.3 void USBDHIDMousePowerStatusSet (

void * pvMouseDevice,

uint8_t ui8Power)

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

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**.

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.

References tUSBDHIDMouseDevice::sPrivateData, and USBDHIDPowerStatusSet().

3.26.3.4 bool USBDHIDMouseRemoteWakeupRequest (

void * pvMouseDevice)

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

Parameters *pvMouseDevice* is the pointer to the mouse device instance structure.

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.

References tUSBDHIDMouseDevice::sPrivateData, and USBDHIDRemoteWakeupRequest().

3.26.3.5 void * USBDHIDMouseSetCBData (

void * pvMouseDevice,

void * pvCBData)

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

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.

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.

References tUSBDHIDMouseDevice::pvCBData.

3.26.3.6 uint32_t USBDHIDMouseStateChange (

void * pvMouseDevice,

int8_t i8DeltaX,

int8_t i8DeltaY,

uint8_t ui8Buttons)

Reports a mouse state change, pointer movement or button press, to the USB host.

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**.

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** will be returned and the state change will be 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.

References MOUSE_ERR_NOT_CONFIGURED, MOUSE_ERR_TX_ERROR, MOUSE_SUCCESS, tUSBDHIDMouseDevice::sPrivateData, USBDHIDReportWrite(), and USBDHIDTxPacketAvailable().

3.26.3.7 void USBDHIDMouseTerm (

void * pvMouseDevice)

Shuts down the HID mouse device.

Parameters *pvMouseDevice* is the pointer to the device instance structure.

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.

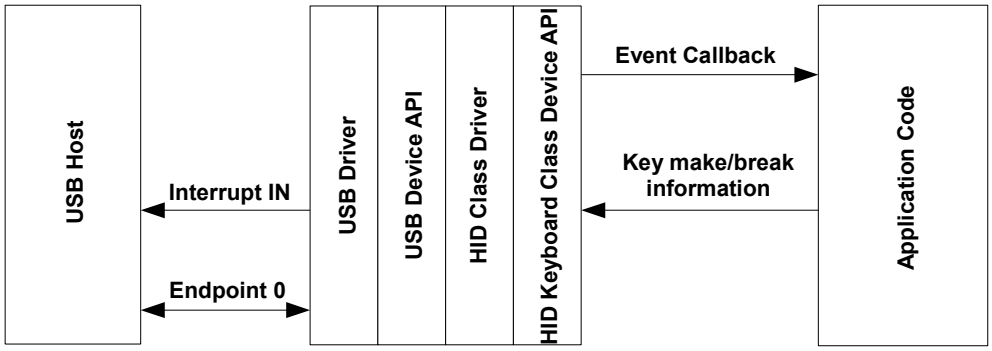
References tUSBDHIDMouseDevice::sPrivateData, and USBDHIDTerm().

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.

3.27 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`

3.28 Using the HID Keyboard Device Class API

To add a USB HID keyboard interface to your application using the HID Keyboard 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 "include/usblib.h" include "include/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 an area of RAM of for the private data for the HID keyboard class driver. This structure should never be accessed by the application.

//** // // The HID keyboard device private instance data. // //** static tHIDKeyboardInstance g_sKeyboardInstance;

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,

// // A pointer to the private instance data allocated for the API to use. // g_sKeyboardInstance };

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. pDevice = USBDHIDKeyboardInit(0, g_sKeyboardDevice);

Assuming `pDevice` returned is not NULL, your keyboard device is now ready to communicate with a USB host.

Once the host connects, your keyboard event handler will be sent `USB_EVENT_CONNECTED` after which calls can be made to USBDHIDKeyboardKeyStateChange()to inform the host of key press and release events.

3.29 HID Keyboard Device Class API Definitions

Data Structures

struct tUSBDHIDKeyboardDevice

Macros

#define KEYB_ERR_NOT_CONFIGURED

#define KEYB_ERR_NOT_FOUND

#define KEYB_ERR_TOO_MANY_KEYS

#define KEYB_ERR_TX_ERROR

#define KEYB_MAX_CHARS_PER_REPORT

#define KEYB_SUCCESS

#define 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)

3.29.1 Detailed Description

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

3.29.2 Macro Definition Documentation

3.29.2.1 #define KEYB_ERR_NOT_CONFIGURED

USBDHIDKeyboardKeyStateChange returns this value if it is called before the USB host has connected and configured the device. Any key usage code passed will be stored and passed to the host once configuration completes.

Referenced by USBDHIDKeyboardKeyStateChange().

3.29.2.2 #define KEYB_ERR_NOT_FOUND

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.

Referenced by USBDHIDKeyboardKeyStateChange().

3.29.2.3 #define KEYB_ERR_TOO_MANY_KEYS

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 will report a roll over error to the host each time it is asked for the keyboard input report.

Referenced by USBDHIDKeyboardKeyStateChange().

3.29.2.4 #define KEYB_ERR_TX_ERROR

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.

Referenced by USBDHIDKeyboardKeyStateChange().

3.29.2.5 #define KEYB_MAX_CHARS_PER_REPORT

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 will result in a rollover error being reported to the host and KEYB_ERR_TOO_MANY_KEYS being returned from USBDHIDKeyboardKeyStateChange.

Referenced by USBDHIDKeyboardCompositeInit(), and USBDHIDKeyboardKeyStateChange().

3.29.2.6 #define USBD_HID_KEYB_EVENT_SET_LEDS

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.

3.29.3 Function Documentation

3.29.3.1 void * USBDHIDKeyboardCompositeInit (

uint32_t ui32Index,

tUSBDHIDKeyboardDevice * psHIDKbDevice,

tCompositeEntry * psCompEntry)

Initializes HID keyboard device operation for a given USB controller.

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.

This call is very similar to USBDKeyboardInit() 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.

References tUSBDHIDDevice::bUseOutEndpoint, KEYB_MAX_CHARS_PER_REPORT, tUSBDHIDKeyboardDevice::pfnCallback, tUSBDHIDDevice::pfnRxCallback, tUSBDHIDDevice::pfnTxCallback, tUSBDHIDDevice::ppui8ClassDescriptors, tUSBDHIDKeyboardDevice::ppui8StringDescriptors, tUSBDHIDDevice::ppui8StringDescriptors, tUSBDHIDDevice::psHIDDescriptor, tUSBDHIDDevice::psReportIdle, tUSBDHIDDevice::pvRxCBData, tUSBDHIDDevice::pvTxCBData, tUSBDHIDKeyboardDevice::sPrivateData, tUSBDHIDKeyboardDevice::ui16MaxPowermA, tUSBDHIDDevice::ui16MaxPowermA, tUSBDHIDKeyboardDevice::ui16PID, tUSBDHIDDevice::ui16PID, tUSBDHIDKeyboardDevice::ui16VID, tUSBDHIDDevice::ui16VID, tUSBDHIDKeyboardDevice::ui32NumStringDescriptors, tUSBDHIDDevice::ui32NumStringDescriptors, tUSBDHIDDevice::ui8NumInputReports, tUSBDHIDDevice::ui8Protocol, tUSBDHIDKeyboardDevice::ui8PwrAttributes, tUSBDHIDDevice::ui8PwrAttributes, tUSBDHIDDevice::ui8Subclass, and USBDHIDCompositeInit().

Referenced by USBDHIDKeyboardInit().

3.29.3.2 void * USBDHIDKeyboardInit (

uint32_t ui32Index,

tUSBDHIDKeyboardDevice * psHIDKbDevice)

Initializes HID keyboard device operation for a given USB controller.

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.

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 will return 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 will receive **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 will cause unpredictable (though almost certainly unpleasant) behavior. Returns Returns NULL on failure or the psHIDKbDevice pointer on success.

References tConfigDescriptor::bmAttributes, tConfigDescriptor::bMaxPower, tUSBDHIDKeyboardDevice::pfnCallback, tUSBDHIDKeyboardDevice::ppui8StringDescriptors, tUSBDHIDKeyboardDevice::sPrivateData, tUSBDHIDKeyboardDevice::ui16MaxPowermA, tUSBDHIDKeyboardDevice::ui8PwrAttributes, USBDHIDInit(), and USBDHIDKeyboardCompositeInit().

3.29.3.3 uint32_t USBDHIDKeyboardKeyStateChange (

void * pvKeyboardDevice,

uint8_t ui8Modifiers,

uint8_t ui8UsageCode,

bool bPress)

Reports a key state change to the USB host.

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.

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 will contain the rollover error code, HID_KEYB_USAGE_ROLLOVER instead of key usage codes and the caller will receive

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.

References KEYB_ERR_NOT_CONFIGURED, KEYB_ERR_NOT_FOUND, KEYB_ERR_TOO_MANY_KEYS, KEYB_ERR_TX_ERROR, KEYB_MAX_CHARS_PER_REPORT, KEYB_SUCCESS, tUSBDHIDKeyboardDevice::sPrivateData, USBDHIDReportWrite(), and USBDHIDTxPacketAvailable().

3.29.3.4 void USBDHIDKeyboardPowerStatusSet (

void * pvKeyboardDevice,

uint8_t ui8Power)

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

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**.

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.

References tUSBDHIDKeyboardDevice::sPrivateData, and USBDHIDPowerStatusSet().

3.29.3.5 bool USBDHIDKeyboardRemoteWakeupRequest (

void * pvKeyboardDevice)

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

Parameters *pvKeyboardDevice* is the pointer to the keyboard device instance structure.

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.

References tUSBDHIDKeyboardDevice::sPrivateData, and USBDHIDRemoteWakeupRequest().

3.29.3.6 void * USBDHIDKeyboardSetCBData (

void * pvKeyboardDevice,

void * pvCBData)

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

Parameters *pvKeyboardDevice* is the pointer to the device instance structure as returned by USBDHIDKeyboardInit().

---

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

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 will not be possible.

Returns Returns the previous callback pointer that was set for this instance.

References tUSBDHIDKeyboardDevice::pvCBData.

3.29.3.7 void USBDHIDKeyboardTerm (

void * pvKeyboardDevice)

Shuts down the HID keyboard device.

Parameters *pvKeyboardDevice* is the pointer to the device instance structure as returned by USBDHIDKeyboardInit().

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.

References tUSBDHIDKeyboardDevice::sPrivateData, and USBDHIDTerm().

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 C2000Ware 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.

3.30 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:

`pDeviceDescriptor`

`ppConfigDescriptors`

`ppStringDescriptors`

`ulNumStringDescriptors`

All descriptors are provided as pointers to arrays of unsigned characters 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).

3.30.1 tDeviceInfo.pDeviceDescriptor

This array must hold the device descriptor that the USB Device API will return 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 unsigned char g_pDeviceDescriptor[] = { 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), // 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.

3.30.2 tDeviceInfo.ppConfigDescriptors

While only a single device descriptor is required, multiple configuration descriptors may be offered so the `ppConfigDescriptors` 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 `pDeviceDescriptor` 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 will calculate 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 unsigned char g_pKeyboardDescriptor[] =
{
    //
    // 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.
//
//*****************************************************************************
unsigned char
```

g_pHIDInterface[] = { // // 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 unsigned char g_pHIDInEndpoint[] = { // // 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 ppConfigDescriptors field of // the tDeviceInfo structure passed to USBDCDInit(). // //*** const tConfigHeader g_sKeyboardConfigHeader = { NUM_KEYBOARD_SECTIONS, g_psKeyboardSections };

3.30.3 tDeviceInfo.ppStringDescriptors and tDeviceInfo.ulNumStringDescriptors

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 `ulNumStringDescriptors` indicates the number of individual string descriptors in the `ppStringDescriptors` array.

The string descriptor array provided to the USB Device API for a USB HID keyboard follows.

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

//*****************************************************************************
//
// The manufacturer string.
//
//*****************************************************************************
const unsigned char g_pManufacturerString[] =
{
    2 + (22 * 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, ' ', 0, 'I', 0, 'n', 0, 'c', 0, '.', 0
};

//*****************************************************************************
//
// The product string.
//
//*****************************************************************************
const unsigned char g_pProductString[] =
{
    (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 unsigned char g_pSerialNumberString[] =
{
    (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 unsigned char *
const g_ppStringDescriptors[] =
{
    g_pLangDescriptor,
    g_pManufacturerString,
    g_pProductString,
    g_pSerialNumberString
};
```

In this example, the `ppStringDescriptors` member of the `tDeviceInfo` structure would be initialized with the value `g_ppStringDescriptors` and the `ulNumStringDescriptors` member would be set to the number of elements in the `g_ppStringDescriptors` array.

3.31 USB Event Handlers

The majority of the work in a USB device application will be 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 USBDCDInit().

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.

3.31.1 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 USBDCDStallEP0() to indicate that the descriptor request is not supported.

If this member of `sCallbacks` is set to NULL, the USB Device API will stall endpoint zero whenever it receives a request for a non-standard descriptor.

3.31.2 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 will follow 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 f2806x Peripheral 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() will be delivered asynchronously via the `pfnDataReceived` callback described below.

If this member of `sCallbacks` is set to NULL, the USB Device API will stall endpoint zero whenever it receives a non-standard request.

3.31.3 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 will stall 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 will note the interface change internally but not report it to the device code.

3.31.4 pfnConfigChange

When the host enumerates a device, it will ultimately select 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 will note the configuration change internally but not report it to the device code.

3.31.5 pfnDataReceived

This callback informs the device code of the arrival of data following an earlier call to USBDCDRequestDataEP0(). On this callback, the received data will have 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 will read 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.

3.31.6 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 will not inform the device code when a block of EP0 data is sent.

3.31.7 pfnResetHandler

The `pfnResetHandler` callback is made by the USB Device API whenever a bus reset is detected. This will typically occur 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 will not inform the device code when a bus reset occurs.

3.31.8 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 will not inform the device code when a bus suspend occurs.

3.31.9 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 will not inform the device code when a bus resume occurs.

3.31.10 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 will not inform the device code when a disconnection event occurs.

3.31.11 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`.

3.31.12 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 will need 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.

3.32 USB FIFO Configuration

The USB controller FIFO must be partitioned appropriately between the various endpoints that an application is using. Although the actual configuration is performed within the USB library, the application must pass a structure to indicate any special sizing or FIFO configuration considerations that need to be taken into account. The `tFIFOConfig` structure contains two arrays, the first defining FIFO configuration parameters for each of the 3 IN endpoints and the second containing the same information for the 3 OUT endpoints. Endpoint zero is handled independently and uses a fixed FIFO configuration.

A pointer to this structure is passed to USBCDCInit() in the `psFIFOConfig` field of the `tDeviceInfo` structure. If the application does not wish to use any special features such as DMA or double buffering, the field can be initialized using global pointer `g_sUSBDefaultFIFOConfig` which configures the FIFO to buffer one full packet for each endpoint in use.

If the default FIFO configuration is not suitable, declare a new `tFIFOConfig` structure and complete all fields before setting `psFIFOConfig` to point to it. For each endpoint, the relevant structure entry allows:

the FIFO size to be defined in terms of a maximum packet size multiplier,

the FIFO buffering mode to be set to either single- or double-buffered, and

and special configuration flags such as DMA mode to be specified.

3.33 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. Note that this should not be done for a device-only application since making use of either of these APIs will cause the host-side USB library code to be included in the final application binary.

3.34 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 will enable 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 will be 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);

3.35 USB Device API Definitions

Data Structures

struct tCompositeEntry

struct tConfigHeader

struct tConfigSection

struct tCustomHandlers

struct tDeviceInfo

Macros

#define USB_MAX_INTERFACES_PER_DEVICE

Functions

void USB0DeviceIntHandler (void)

void USBDCDDeviceInfoInit (uint32_t ui32Index, tDeviceInfo *psDeviceInfo)

void USBDCDInit (uint32_t ui32Index, tDeviceInfo *psDevice, void *pvDCDCBData)

void USBDCDPowerStatusSet (uint32_t ui32Index, uint8_t ui8Power)

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)

3.35.1 Detailed Description

3.35.2 Macro Definition Documentation

3.35.2.1 #define USB_MAX_INTERFACES_PER_DEVICE

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.

3.35.3 Function Documentation

3.35.3.1 void USB0DeviceIntHandler (

)

The USB device interrupt handler.

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.

3.35.3.2 void USBDCDDeviceInfoInit (

void uint32_t ui32Index,

tDeviceInfo * psDeviceInfo)

Initialize an instance of the tDeviceInfo structure.

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. 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.

Referenced by USBDAudioCompositeInit(), USBDBulkCompositeInit(), USBDCDCCompositeInit(), USBDCDInit(), USBDCompositeInit(), USBDDFUCompositeInit(), and USBDHIDCompositeInit().

3.35.3.3 void USBDCDInit (

uint32_t ui32Index,

tDeviceInfo * psDevice,

void * pvDCDCBData)

Initialize the USB library device control driver for a given hardware controller.

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.

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.

References tConfigDescriptor::bmAttributes, eUSBModeDevice, eUSBModeForceDevice, eUSBModeForceHost, eUSBModeHost, eUSBModeNone, eUSBModeOTG, tDeviceInfo::ppsConfigDescriptors, tConfigHeader::psSections, tConfigSection::pui8Data, USBDCDDeviceInfoInit(), and USBLibDMAInit().

Referenced by USBDAudioInit(), USBDBulkInit(), USBDCDCInit(), USBDCompositeInit(), and USBDHIDInit().

3.35.3.4 void USBDCDPowerStatusSet (

uint32_t ui32Index,

uint8_t ui8Power)

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

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**.

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.

Referenced by USBDBulkPowerStatusSet(), USBDCDCPowerStatusSet(), and USBDHIDPowerStatusSet().

3.35.3.5 bool USBDCDRemoteWakeupRequest (uint32_t ui32Index)

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

Parameters *ui32Index* is the index of the USB controller that will request a bus wake up.

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.

Referenced by USBDBulkRemoteWakeupRequest(), USBDCDCRemoteWakeupRequest(), and USBDHIDRemoteWakeupRequest().

3.35.3.6 void USBDCDRequestDataEP0 (uint32_t ui32Index, uint8_t * pui8Data, uint32_t ui32Size)

This function starts the request for data from the host on endpoint zero.

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.

This function handles retrieving data from the host when a custom command has been issued on endpoint zero. If the application needs notification when the data has been received,

`psCallbacks->pfnDataReceived()` in the tDeviceInfo structure must contain valid function 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.

3.35.3.7 void USBDCDSendDataEP0 (

uint32_t ui32Index,

uint8_t * pui8Data,

uint32_t ui32Size)

This function requests transfer of data to the host on endpoint zero.

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.

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 contain 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.

3.35.3.8 void USBDCDSetDefaultConfiguration (

uint32_t ui32Index,

uint32_t ui32DefaultConfig)

This function sets the default configuration for the device.

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 descriptor) 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.

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.

3.35.3.9 void USBDCDStallEP0 (

uint32_t ui32Index)

This function generates a stall condition on endpoint zero.

Parameters *ui32Index* is the index of the USB controller whose endpoint zero is to be stalled.

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.

3.35.3.10 void USBDCDTerm (

uint32_t ui32Index)

Free the USB library device control driver for a given hardware controller.

Parameters *ui32Index* is the index of the USB controller which is to be freed.

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.

Referenced by USBDAudioTerm(), USBDBulkTerm(), USBDCDCTerm(), USBDDFUCompositeTerm(), USBDDFUUpdateBegin(), and USBDHIDTerm().

3.35.3.11 bool USBDeviceConfig (

tDCDInstance * psDevInst,

const **tConfigHeader** * psConfig)

Configure the USB controller appropriately for the device whose configuration descriptor is passed.

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.

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.

USBDCDConfig() is an optional call and applications may chose to make direct calls to SysCtlPeripheralEnable(), SysCtlUSBPLLEnable(), USBDevEndpointConfigSet() and USBFIFOConfigSet() instead of using this function. If this function is used, it must be called prior to USBDCDInit() since this call assumes that the low level hardware configuration has been completed before it is made.

Returns Returns **true** on success or **false** on failure.

References tInterfaceDescriptor::bAlternateSetting, tEndpointDescriptor::bEndpointAddress, tInterfaceDescriptor::bInterfaceNumber, tInterfaceDescriptor::bNumEndpoints, readusb16_t, USB_DESC_ANY, and tEndpointDescriptor::wMaxPacketSize.

3.35.3.12 bool USBDeviceConfigAlternate (

tDCDInstance * psDevInst,

const **tConfigHeader** * psConfig,

uint8_t ui8InterfaceNum,

uint8_t ui8AlternateSetting)

Configure the affected USB endpoints appropriately for one alternate interface setting.

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.

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.

References tInterfaceDescriptor::bAlternateSetting, tInterfaceDescriptor::bInterfaceNumber, tInter-

faceDescriptor::bNumEndpoints, and USB_DESC_ANY.

4 Host Functions

Introduction ..... ??
Host Controller Driver ..... ??
Host Controller Driver Definitions 135
Host Class Driver ..... ??
Host Class Driver Definitions 156
Host Device Interface ..... ??
Host Device Interface Definitions 180
Host Programming Examples ..... ??

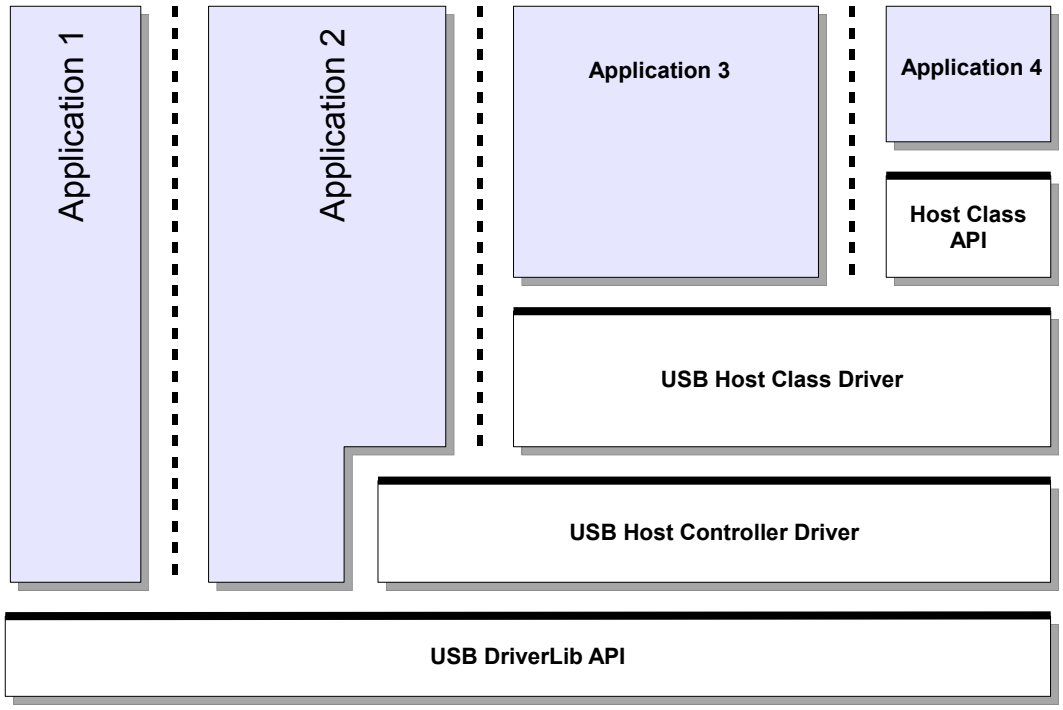
This chapter covers the support provided by the USB library for the USB controller in host mode. In order to simplify the application and the addition of new devices and device classes, the USB library provides a layered interface to the USB host controller. At the top layer of the USB library there 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 will be 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)

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/device`.

`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.

`usbhscsi.c` The source code for a high level SCSI interface which calls the host Mass Storage Class driver.

`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.

`usbhhidkeyboard.c`
The source code for the USB host HID keyboard device.

`usbhhidkeyboard.h`

The header file containing the definitions needed to interact with the USB host HID keyboard device.

`usbhhidmouse.c`
: The source code for the USB host HID mouse device.

`usbhhidmouse.h`
: The header file containing the definitions needed to interact with the USB host HID 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.

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 will support. 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.

4.1 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 USBHCDControlTransfer() 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 will call the open function for that host class driver. If no host class driver is found the host controller driver will ignore the device and there will be 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 will be notified that a new device has been discovered by a callback from the device interface that a device of that given type has been enumerated. When the device is removed the application will also get a callback that the device is no longer present. The events

USB_EVENT_CONNECTED and **USB_EVENT_DISCONNECTED** are the only event notifications that will make it up to the application as a result of enumeration.

4.2 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 can provide status and perform read and write operations. Calling USBHCDPipeStatus() allows a host class driver to check the status of a pipe. However most 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.

4.3 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.

4.4 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.

4.5 Host Controller Driver Definitions

Data Structures

struct tUSBHostClassDriver

Macros

#define DECLARE_EVENT_DRIVER(VarName, pfnOpen, pfnClose, pfnEvent)

Functions

void USB0HostIntHandler (void)

uint32_t USBHCDControlTransfer (uint32_t ui32Index, tUSBRequest *psSetupPacket, tUSBHostDevice *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)

void USBHCDInit (uint32_t ui32Index, void *pvPool, uint32_t ui32PoolSize)

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, tUSBHostDevice *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)

722.7

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.

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.

4.5.3 Function Documentation

4.5.3.1 void USB0HostIntHandler (

)

The USB host mode interrupt handler for controller index 0.

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.

4.5.3.2 uint32_t USBHCDControlTransfer (

void uint32_t ui32Index,

tUSBRequest ∗ psSetupPacket,

tUSBHostDevice ∗ psDevice,

uint8_t ∗ pui8Data,

uint32_t ui32Size,

uint32_t ui32MaxPacketSize)

This function completes a control transaction to a device.

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.

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.

References tUSBHostDevice::bLowSpeed, tUSBRequest::bmRequestType, tUSBHostDevice::ui32Address, tUSBHostDevice::ui8Hub, and tUSBHostDevice::ui8HubPort.

Referenced by USBHCDSetAddress(), USBHCDSetConfig(), USBHCDSetInterface(), USBHHIDGetReportDescriptor(), USBHHIDSetIdle(), USBHHIDSetProtocol(), USBHHIDSetReport(), and USBHostAudioVolumeSet().

4.5.3.3 uint8_t USBHCDDevAddress (

uint32_t ui32Instance)

This function will return the USB address for the requested device instance.

Parameters *ui32Instance* is a unique value indicating which device to query.

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.

4.5.3.4 uint8_t USBHCDDevClass (

uint32_t ui32Instance,

uint32_t ui32Interface)

This function will return the USB class for the requested device instance.

Parameters *ui32Instance* is a unique value indicating which device to query.

ui32Interface is the interface number to query for the USB class.

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.

References tInterfaceDescriptor::bInterfaceClass, and USBDescGetInterface().

4.5.3.5 uint8_t USBHCDDevHubPort (

uint32_t ui32Instance)

This function returns the USB hub port for the requested device instance.

Parameters *ui32Instance* is a unique value indicating which device to query.

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.

4.5.3.6 uint8_t USBHCDDevProtocol (

uint32_t ui32Instance,

uint32_t ui32Interface)

This function returns the USB protocol for the requested device instance.

Parameters *ui32Instance* is a unique value indicating which device to query.

---

ui32Interface is the interface number to query for the USB protocol.

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.

References tInterfaceDescriptor::bInterfaceProtocol, and USBDescGetInterface().

4.5.3.7 uint8_t USBHCDDevSubClass (

uint32_t ui32Instance,

uint32_t ui32Interface)

This function will return the USB subclass for the requested device instance.

Parameters *ui32Instance* is a unique value indicating which device to query.

---

ui32Interface is the interface number to query for the USB subclass.

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.

References tInterfaceDescriptor::bInterfaceSubClass, and USBDescGetInterface().

4.5.3.8 int32_t USBHCDEventDisable (

uint32_t ui32Index,

void * pvEventDriver,

uint32_t ui32Event)

This function is called to disable a specific USB HCD event notification.

Parameters *ui32Index* specifies which USB controller to use.

---

pvEventDriver is the event driver structure that was passed into the USBHCDRegisterDrivers() function as part of the array of tUSBHostClassDriver structures.

---

ui32Event is the event to disable.

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 USBHCDRegisterDrivers() function.

Returns This function returns a non-zero number if the event was successfully disabled and returns zero if the event cannot be disabled.

4.5.3.9 int32_t USBHCDEventEnable (

uint32_t ui32Index,

void * pvEventDriver,

uint32_t ui32Event)

This function is called to enable a specific USB HCD event notification.

Parameters *ui32Index* specifies which USB controller to use.

---

pvEventDriver is the event driver structure that was passed into the USBHCDRegisterDrivers() function as part of the array of tUSBHostClassDriver structures.

---

ui32Event is the event to enable.

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 USBHCDRegisterDrivers() function.

Returns This function returns a non-zero number if the event was successfully enabled and returns zero if the event cannot be enabled.

4.5.3.10 void USBHCDInit (

uint32_t ui32Index,

void * pvPool,

uint32_t ui32PoolSize)

This function is used to initialize the HCD code.

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*.

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.

References eUSBModeDevice, eUSBModeForceDevice, eUSBModeHost, eUSBModeNone, and eUSBModeOTG.

Referenced by USBOTGModeInit().

4.5.3.11 void USBHCDMain (

)

This function is the main routine for the Host Controller Driver.

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.

References tEventInfo::ui32Event, tEventInfo::ui32Instance, USB_EVENT_POWER_FAULT, and USB_EVENT_SOF.

Referenced by USBOTGMain().

4.5.3.12 uint32_t USBHCDPipeAlloc (

uint32_t ui32Index,

uint32_t ui32EndpointType,

tUSBHostDevice * psDevice,

tHCDPipeCallback pfnCallback)

This function is used to allocate a USB HCD pipe.

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.

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.

References USBHCDPipeAllocSize().

4.5.3.13 uint32_t USBHCDPipeAllocSize (

uint32_t ui32Index,

uint32_t ui32EndpointType,

tUSBHostDevice * psDevice,

uint32_t ui32Size,

tHCDPipeCallback pfnCallback)

This function is used to allocate a USB HCD pipe.

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.

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.

References tUSBHostDevice::bLowSpeed, tUSBHostDevice::ui32Address, tUSBHostDevice::ui8Hub, tUSBHostDevice::ui8HubPort, and USBLibDMAChannelAllocate.

Referenced by USBHCDPipeAlloc().

4.5.3.14 uint32_t USBHCDPipeConfig (

uint32_t ui32Pipe,

uint32_t ui32MaxPayload,

uint32_t ui32Interval,

uint32_t ui32TargetEndpoint)

This function is used to configure a USB HCD pipe.

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^(*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^(*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.

References tUSBHostDevice::ui32Flags.

4.5.3.15 void USBHCDPipeDataAck (

uint32_t ui32Pipe)

This function acknowledges data received via an interrupt IN pipe.

Parameters *ui32Pipe* is the USB INT pipe whose last packet is to be acknowledged.

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.

4.5.3.16 void USBHCDPipeFree (

uint32_t ui32Pipe)

This function is used to release a USB pipe.

Parameters *ui32Pipe* is the allocated USB pipe to release.

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.

References USBLibDMAChannelRelease.

4.5.3.17 uint32_t USBHCDPipeRead (

uint32_t ui32Pipe,

uint8_t * pui8Data,

uint32_t ui32Size)

This function is used to read data from a USB HCD pipe.

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*.

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.

References USBLibDMAChannelDisable, USBLibDMAChannelStatus, and USBLibDMATransfer.

4.5.3.18 uint32_t USBHCDPipeReadNonBlocking (

uint32_t ui32Pipe,

uint8_t * pui8Data,

uint32_t ui32Size)

This function is used to read data from a USB HCD pipe.

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*.

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.

Referenced by USBHHIDGetReport().

4.5.3.19 uint32_t USBHCDPipeSchedule (

uint32_t ui32Pipe,

uint8_t * pui8Data,

uint32_t ui32Size)

This function is used to schedule and IN transaction on a USB HCD pipe.

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*.

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.

References USBLibDMATransfer.

Referenced by USBHostAudioPlay(), and USBHostAudioRecord().

4.5.3.20 uint32_t USBHCDPipeStatus (

uint32_t ui32Pipe)

This function is used to return the current status of a USB HCD pipe.

This function will return the current status for a given USB pipe. If there is no status to report this call will simply return **USBHCD_PIPE_NO_CHANGE**.

Parameters *ui32Pipe* is the USB pipe for this status request.

Returns This function returns the current status for the given endpoint. This will be one of the **USBHCD_PIPE_*** values.

4.5.3.21 uint32_t USBHCDPipeWrite (

uint32_t ui32Pipe,

uint8_t * pui8Data,

uint32_t ui32Size)

This function is used to write data to a USB HCD pipe.

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.

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.

References USBLibDMAChannelDisable, and USBLibDMATransfer.

4.5.3.22 uint32_t USBHCDPowerAutomatic (

uint32_t ui32Index)

This function returns if the current power settings will automatically handle enabling and disabling VBUS power.

Parameters *ui32Index* specifies which USB controller to query.

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.

Referenced by USB0OTGModeIntHandler(), USBDualModeInit(), USBOTGMain(), and USBOTGModeInit().

4.5.3.23 uint32_t USBHCDPowerConfigGet (

uint32_t ui32Index)

This function is used to get the power pin and power fault configuration.

Parameters *ui32Index* specifies which USB controller to use.

This function will return 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 returned by this function.

Returns The configuration of the power control pins.

Referenced by USBDualModeInit(), and USBOTGModeInit().

4.5.3.24 void USBHCDPowerConfigInit (

uint32_t ui32Index,

uint32_t ui32PwrConfig)

This function is used to set the power pin and power fault configuration.

Parameters *ui32Index* specifies which USB controller to use.

---

ui32PwrConfig is the power configuration to use for the application.

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.

4.5.3.25 uint32_t USBHCDPowerConfigSet (

uint32_t ui32Index,

uint32_t ui32Config)

This function is used to set the power pin and power fault configuration.

Parameters *ui32Index* specifies which USB controller to use.

---

ui32Config specifies which USB power configuration to use.

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.

Referenced by USBOTGModeInit().

4.5.3.26 void USBHCDRegisterDrivers (

uint32_t ui32Index,

const **tUSBHostClassDriver** *const * ppsHClassDrvs,

uint32_t ui32NumDrivers)

This function is used to initialize the HCD class driver list.

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.

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.

4.5.3.27 void USBHCDReset (

uint32_t ui32Index)

This function generates reset signaling on the USB bus.

Parameters *ui32Index* specifies which USB controller to use.

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.

4.5.3.28 void USBHCDResume (

uint32_t ui32Index)

This function will generate resume signaling on the USB bus.

Parameters *ui32Index* specifies which USB controller to use.

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 USBHCD-Suspend() has been made.

Returns None.

4.5.3.29 void USBHCDSetAddress (

uint32_t ui32DevIndex,

uint32_t ui32DevAddress)

This function is used to send the set address command to a device.

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.

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.

References tUSBRequest::bmRequestType, tUSBRequest::bRequest, USBHCDControlTransfer(), tUSBRequest::wIndex, tUSBRequest::wLength, writeusb16_t, and tUSBRequest::wValue.

4.5.3.30 void USBHCDSetConfig (

uint32_t ui32Index,

uint32_t ui32Device,

uint32_t ui32Configuration)

This function is used to set the current configuration for a device.

Parameters *ui32Index* specifies which USB controller to use.

---

ui32Device is the USB device for this function.

---

ui32Configuration is one of the devices valid configurations.

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.

References tUSBRequest::bmRequestType, tUSBRequest::bRequest, USBHCDControlTransfer(), tUSBRequest::wIndex, tUSBRequest::wLength, writeusb16_t, and tUSBRequest::wValue.

4.5.3.31 void USBHCDSetInterface (

uint32_t ui32Index,

uint32_t ui32Device,

uint32_t ui32Interface,

uint32_t ui32AltSetting)

This function is used to set the current interface and alternate setting for an interface on a device.

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.

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.

References tUSBRequest::bmRequestType, tUSBRequest::bRequest, USBHCDControlTransfer(), tUSBRequest::wIndex, tUSBRequest::wLength, writeusb16_t, and tUSBRequest::wValue.

Referenced by USBHostAudioPlay(), and USBHostAudioRecord().

4.5.3.32 void USBHCDSuspend (

uint32_t ui32Index)

This function will generate suspend signaling on the USB bus.

Parameters *ui32Index* specifies which USB controller to use.

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.

4.5.3.33 void USBHCDTerm (

uint32_t ui32Index)

This function is used to terminate the HCD code.

Parameters *ui32Index* specifies which USB controller to release.

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. 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 will use 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 will be through it's endpoint pipes. The host class driver will parse and allocate any endpoints that it needs by calling the USBHCDPipeAlloc() and USBHCDPipeConfig() functions. These USB pipes will provide the methods to read/write and get callback notification from the USB host controller driver layer.

4.6 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$_pp USBHostClassDrivers[] = g_USBHIDClassDriver$;

// // Register the host class drivers. // USBHCDRegisterDrivers(0, g$_pp USBHostClassDrivers, 1)$;

...

// // Open an instance of a HID mouse class driver. // ulMouseInstance = USBHHIDOpen(USBH$_HID_DEV_MOUSE, USBHMouseCallback, (unsignedlong)g_sUSBHMouse)$;

Once a HID device has been opened the first callback it will received will be 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 will occur. When shutting down or to release a device, the application should call USBHHIDClose() to disable callbacks. This will not actually power down the device but it will stop the driver from calling the application. During normal operation the host class driver will receive 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 USBJHCDPipeSchedule() 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 will send data to the device by using the correct USB OUT pipe.

4.7 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 will perform 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 will load 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

$g_pUSBHostClassDrivers[] = g_USBHostMSCClassDriver;$

// // Register the host class drivers. // USBHCDRegisterDrivers(0, $g_pUSBHostClassDrivers, 1);$

// // Initialize the mass storage class driver on controller 0 with the // MSCCallback() function as the callback for events. // USBHMSCDriveOpen(0, MSCCallback);

The first callback will be 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 will occur. When shutting down, the application should call USBHMSCDriveClose() to disable callbacks. This will not actually power down the mass storage device but it will stop 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 will 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. // USBHMSCBlock-Read(ulMSCDevice, 0, pucBuffer, 1);

// // Write 2 blocks starting at logical block 500. // USBHMSCBlockWrite(ulMSCDevice, 500, pucBuffer, 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 SCSIRead10() and SCSIWrite10() 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.

4.8 Implementing Custom Host Class Drivers

This next section will cover 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 will be called when the host controller driver enumerates a device that matches the “USB_CLASS_SOMECLASS” interface class. The USBGenericClose() function will

be 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 USBGenericClassDriver = USB$_{C}LASS_{S}OMECLASS, USBGenericOpen, USBGenericClose, USBGenericIntHandler$;

The *ulInterfaceClass* member of the tUSBClassDriver structure is the class read from the device's interface descriptor during enumeration. This number will be 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 will be loaded. The *pfnOpen* member of the tUSBClassDriver structure will be 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 will require 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 will be 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 will simply return 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 will be 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 will run 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.

4.9 Host Class Driver Definitions

Macros

#define USBH_AUDIO_EVENT_CLOSE

#define USBH_AUDIO_EVENT_OPEN

#define USBH_EVENT_HID_KB_MOD

#define USBH_EVENT_HID_KB_PRESS

#define USBH_EVENT_HID_KB_REL

#define USBH_EVENT_HID_MS_PRESS

#define USBH_EVENT_HID_MS_REL

#define USBH_EVENT_HID_MS_X

#define USBH_EVENT_HID_MS_Y

Enumerations

enum tHIDSubClassProtocol { eUSBHHIDClassNone, eUSBHHIDClassKeyboard, eUSBHHIDClassMouse, eUSBHHIDClassVendor }

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)

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)

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)

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)

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

4.9.1 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`.

4.9.2 Macro Definition Documentation

4.9.2.1 #define USBH_AUDIO_EVENT_CLOSE

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.

4.9.2.2 #define USBH_AUDIO_EVENT_OPEN

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.

4.9.3 Enumeration Type Documentation

4.9.3.1 enum **tHIDSubClassProtocol**

The following values are used to register callbacks to the USB HOST HID device class layer.

Enumerator

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.

4.9.4 Function Documentation

4.9.4.1 void USBHHIDClose (

tHIDInstance * psHIDInstance)

This function is used to release an instance of a HID device.

Parameters *psHIDInstance* is the instance value for a HID device to release.

This function releases an instance of a HID device that was created by a call to USBHHIDOpen(). 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.

References eUSBHHIDClassNone.

Referenced by USBHKeyboardClose(), and USBHMouseClose().

4.9.4.2 uint32_t USBHHIDGetReport (

tHIDInstance * psHIDInstance,

uint32_t ui32Interface,

uint8_t * pui8Data,

uint32_t ui32Size)

This function is used to retrieve a report from a HID device.

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*.

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.

References USBHCDPipeReadNonBlocking().

4.9.4.3 uint32_t USBHHIDGetReportDescriptor (

tHIDInstance * psHIDInstance,

uint8_t * pui8Buffer,

uint32_t ui32Size)

This function can be used to retrieve the report descriptor for a given device instance.

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*.

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*.

References tUSBRequest::bmRequestType, tUSBRequest::bRequest, USBHCDControlTransfer(), tUSBRequest::wIndex, tUSBRequest::wLength, writeusb16_t, and tUSBRequest::wValue.

Referenced by USBHKeyboardInit(), and USBHMouseInit().

4.9.4.4 tHIDInstance* USBHHIDOpen (

tHIDSubClassProtocol iDeviceType,

tUSBCallback pfnCallback,

void * pvCBData)

This function is used to open an instance of a HID device.

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.

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.

References eUSBHHIDClassNone.

Referenced by USBHKeyboardOpen(), and USBHMouseOpen().

4.9.4.5 uint32_t USBHHIDSetIdle (

tHIDInstance * psHIDInstance,

uint8_t ui8Duration,

uint8_t ui8ReportID)

This function is used to set the idle timeout for a HID device.

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.

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.

References tUSBRequest::bmRequestType, tUSBRequest::bRequest, USBHCDControlTransfer(), tUSBRequest::wIndex, tUSBRequest::wLength, writeusb16_t, and tUSBRequest::wValue.

Referenced by USBHKeyboardInit(), USBHKeyboardPollRateSet(), and USBHMouseInit().

4.9.4.6 uint32_t USBHHIDSetProtocol (

tHIDInstance * psHIDInstance,

uint32_t ui32BootProtocol)

This function is used to set or clear the boot protocol state of a device.

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.

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.

References tUSBRequest::bmRequestType, tUSBRequest::bRequest, USBHCDControlTransfer(), tUSBRequest::wIndex, tUSBRequest::wLength, writeusb16_t, and tUSBRequest::wValue.

Referenced by USBHKeyboardInit(), and USBHMouseInit().

4.9.4.7 uint32_t USBHHIDSetReport (

tHIDInstance * psHIDInstance,

uint32_t ui32Interface,

uint8_t * pui8Data,

uint32_t ui32Size)

This function is used to send a report to a HID device.

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*.

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.

References tUSBRequest::bmRequestType, tUSBRequest::bRequest, USBHCDControlTransfer(), tUSBRequest::wIndex, tUSBRequest::wLength, writeusb16_t, and tUSBRequest::wValue.

Referenced by USBHKeyboardInit(), and USBHKeyboardModifierSet().

4.9.4.8 void USBHHubClose (

tHubInstance * psHubInstance)

This function is used to release a hub device instance.

Parameters *psHubInstance* is the hub device instance that is to be released.

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.

4.9.4.9 void USBHHubEnumerationComplete (

uint8_t ui8Hub,

uint8_t ui8Port)

Informs the hub class driver that a downstream device has been enumerated.

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.

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.

4.9.4.10 void USBHHubEnumerationError (

uint8_t ui8Hub,

uint8_t ui8Port)

Informs the hub class driver that a downstream device failed to enumerate.

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.

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.

4.9.4.11 tHubInstance* USBHHubOpen (

tUSBHHubCallback pfnCallback)

This function is used to enable the host hub class driver before any devices are present.

Parameters *pfnCallback* is the driver call back for host hub events.

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.

4.9.4.12 int32_t USBHMSCBlockRead (

tUSBHMSCInstance * psMSCInstance,

uint32_t ui32LBA,

uint8_t * pui8Data,

uint32_t ui32NumBlocks)

This function performs a block read to an MSC device.

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.

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.

References USBHSCSIRead10().

4.9.4.13 int32_t USBHMSCBlockWrite (

tUSBHMSCInstance ∗ psMSCInstance,

uint32_t ui32LBA,

uint8_t ∗ pui8Data,

uint32_t ui32NumBlocks)

This function performs a block write to an MSC device.

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.

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.

References USBHSCSIWrite10().

4.9.4.14 void USBHMSCDriveClose (

tUSBHMSCInstance * psMSCInstance)

This function should be called to release a drive instance.

Parameters *psMSCInstance* is the device instance that is to be released.

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.

4.9.4.15 tUSBHMSCInstance * USBHMSCDriveOpen (

uint32_t ui32Drive,

tUSBHMSCCallback pfnCallback)

This function should be called before any devices are present to enable the mass storage device class driver.

Parameters *ui32Drive* is the drive number to open.

---

pfnCallback is the driver callback for any mass storage events.

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.

4.9.4.16 int32_t USBHMSCDriveReady (

tUSBHMSCInstance * psMSCInstance)

This function checks if a drive is ready to be accessed.

Parameters *psMSCInstance* is the device instance to use for this read.

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.

References USBHSCSIInquiry(), USBHSCSIReadCapacity(), USBHSCSIRequestSense(), and USBHSCSITestUnitReady().

4.9.4.17 void USBHostAudioClose (

tUSBHostAudioInstance * psAudioInstance)

This function should be called to release an audio device instance.

Parameters *psAudioInstance* is the device instance that is to be released.

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.

4.9.4.18 uint32_t USBHostAudioFormatGet (

tUSBHostAudioInstance * psAudioInstance,

uint32_t ui32SampleRate,

uint32_t ui32Bits,

uint32_t ui32Channels,

uint32_t ui32Flags)

This function is called to determine if an audio format is supported by the connected USB Audio device.

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.

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.

4.9.4.19 uint32_t USBHostAudioFormatSet (

tUSBHostAudioInstance * psAudioInstance,

uint32_t ui32SampleRate,

uint32_t ui32Bits,

uint32_t ui32Channels,

uint32_t ui32Flags)

This function is called to set the current sample rate on an audio interface.

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.

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.

4.9.4.20 tUSBHostAudioInstance * USBHostAudioOpen (

uint32_t ui32Index,

tUSBHostAudioCallback pfnCallback)

This function should be called before any devices are present to enable the host audio class driver.

Parameters *ui32Index* is the audio device to open (currently only 0 is supported).

---

pfnCallback is the driver call back for host audio events.

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.

4.9.4.21 int32_t USBHostAudioPlay (

tUSBHostAudioInstance * psAudioInstance,

void * pvBuffer,

uint32_t ui32Size,

tUSBHostAudioCallback pfnCallback)

This function is called to send an audio buffer to the USB audio device.

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.

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.

References USBHCDPipeSchedule(), and USBHCDSetInterface().

4.9.4.22 int32_t USBHostAudioRecord (

tUSBHostAudioInstance * psAudioInstance,

void * pvBuffer,

uint32_t ui32Size,

tUSBHostAudioCallback pfnCallback)

This function is called to provide an audio buffer to the USB audio device for audio input.

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.

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.

References USBHCDPipeSchedule(), and USBHCDSetInterface().

4.9.4.23 uint32_t USBHostAudioVolumeGet (

tUSBHostAudioInstance * psAudioInstance,

uint32_t ui32Interface,

uint32_t ui32Channel)

This function is used to get the current volume setting for a given audio device.

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.

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.

4.9.4.24 uint32_t USBHostAudioVolumeMaxGet (

tUSBHostAudioInstance * psAudioInstance,

uint32_t ui32Interface,

uint32_t ui32Channel)

This function is used to get the maximum volume setting for a given audio device.

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.

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.

4.9.4.25 uint32_t USBHostAudioVolumeMinGet (

tUSBHostAudioInstance * psAudioInstance,

uint32_t ui32Interface,

uint32_t ui32Channel)

This function is used to get the minimum volume setting for a given audio device.

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.

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.

4.9.4.26 uint32_t USBHostAudioVolumeResGet (

tUSBHostAudioInstance * psAudioInstance,

uint32_t ui32Interface,

uint32_t ui32Channel)

This function is used to get the volume control resolution for a given audio device.

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.

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.

4.9.4.27 void USBHostAudioVolumeSet (

tUSBHostAudioInstance * psAudioInstance,

uint32_t ui32Interface,

uint32_t ui32Channel,

uint32_t ui32Value)

This function is used to set the current volume setting for a given audio device.

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.

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.

References tUSBRequest::bmRequestType, tUSBRequest::bRequest, USBHCDControlTransfer(), tUSBRequest::wIndex, tUSBRequest::wLength, writeusb16_t, and tUSBRequest::wValue.

4.9.4.28 uint32_t USBHSCSIInquiry (

uint32_t ui32InPipe,

uint32_t ui32OutPipe,

uint8_t * pui8Data,

uint32_t * pui32Size)

This will issue the SCSI inquiry command to a device.

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.

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**.

References writeusb32_t.

Referenced by USBHMSCDriveReady().

4.9.4.29 uint32_t USBHSCSIModeSense6 (

uint32_t ui32InPipe,

uint32_t ui32OutPipe,

uint32_t ui32Flags,

uint8_t * pui8Data,

uint32_t * pui32Size)

This will issue the SCSI Mode Sense(6) command to a device.

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.

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$_{M}S_{P}C_{C}URRENT|SCSI_{M}S_{P}C_{A}LL, //!$pui8Data, pui32S$

Returns This function returns the SCSI status from the command. The value will be either **SCSI_CMD_STATUS_PASS** or **SCSI_CMD_STATUS_FAIL**.

References writeusb32_t.

4.9.4.30 uint32_t USBHSCSIRead10 (

uint32_t ui32InPipe,

uint32_t ui32OutPipe,

uint32_t ui32LBA,

uint8_t $*$ pui8Data,

uint32_t $*$ pui32Size,

uint32_t ui32NumBlocks)

This function issues a SCSI Read(10) command to a device.

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.

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**.

Referenced by USBHMSCBlockRead().

4.9.4.31 uint32_t USBHSCSIReadCapacities (

uint32_t ui32InPipe,

uint32_t ui32OutPipe,

uint8_t * pui8Data,

uint32_t * pui32Size)

This will issue the SCSI read capacities command to a device.

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.

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**.

References writeusb32_t.

4.9.4.32 uint32_t USBHSCSIReadCapacity (

uint32_t ui32InPipe,

uint32_t ui32OutPipe,

uint8_t * pui8Data,

uint32_t * pui32Size)

This will issue the SCSI read capacity command to a device.

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.

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**.

References writeusb32_t.

Referenced by USBHMSCDriveReady().

4.9.4.33 uint32_t USBHSCSIRequestSense (

uint32_t ui32InPipe,

uint32_t ui32OutPipe,

uint8_t * pui8Data,

uint32_t * pui32Size)

This function issues a SCSI Request Sense command to a device.

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.

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*.

722.7

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**.

Referenced by USBHMSCBlockWrite().

4.9.5 Variable Documentation

4.9.5.1 const **tUSBHostClassDriver** g_sUSBHIDClassDriver

This constant global structure defines the HID Class Driver that is provided with the USB library.

4.9.5.2 const **tUSBHostClassDriver** g_sUSBHostAudioClassDriver

This constant global structure defines the Audio Class Driver that is provided with the USB library.

4.9.5.3 const **tUSBHostClassDriver** g_sUSBHostMSCClassDriver

This constant global structure defines the Mass Storage Class Driver that is provided with the USB library.

4.9.5.4 const **tUSBHostClassDriver** g_sUSBHubClassDriver

This constant global structure defines the Hub Class Driver that is provided with the USB library. 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 will discuss each briefly and explain how their interfaces can be used by an application.

4.10 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

USBH_EVENT_HID_MS_Y

USBH_EVENT_HID_MS_PRESS

The ulMsgParam parameter will have 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 ulMsgParam parameter will have 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 ulMsgParam parameter will have an 8 bit signed value indicating the delta in the X direction since the last update.

USBH_EVENT_HID_MS_Y

The ulMsgParam parameter will have 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.

4.11 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

USBH_EVENT_HID_KB_MOD

USBH_EVENT_HID_KB_PRESS

The ulMsgParam parameter will have 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 ulMsgParam parameter will have the USB usage identifier for the key that has been released.

USBH_EVENT_HID_KB_MOD

The ulMsgParam parameter will have 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

4.12 Host Device Interface Definitions

Functions

uint32_t USBHKeyboardClose (tUSBHKeyboard *psKbInstance)

uint32_t USBHKeyboardInit (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 tHIDKeyboard-UsageTable *psTable, uint8_t ui8UsageID)

uint32_t USBHMouseClose (tUSBHMouse *psMsInstance)

uint32_t USBHMouseInit (tUSBHMouse *psMsInstance)

tUSBHMouse * USBHMouseOpen (tUSBHIDMouseCallback pfnCallback, uint8_t *pui8Buffer, uint32_t ui32Size)

4.12.1 Detailed Description

The macros and functions defined in this section can be found in header files `host/usbhhidkeyboard.h` and `host/usbhhidmouse.h`.

4.12.2 Function Documentation

4.12.2.1 uint32_t USBHKeyboardClose (

tUSBHKeyboard * psKbInstance)

This function is used close an instance of a keyboard.

Parameters *psKbInstance* is the instance value for this keyboard.

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.

References USBHHIDClose().

4.12.2.2 uint32_t USBHKeyboardInit (

tUSBHKeyboard * psKbInstance)

This function is used to initialize a keyboard interface after a keyboard has been detected.

Parameters *psKbInstance* is the instance value for this keyboard.

This function should be called after receiving a **USB_EVENT_CONNECTED** event in the callback 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.

References USBHHIDGetReportDescriptor(), USBHHIDSetIdle(), USBHHIDSetProtocol(), and USBHHIDSetReport().

4.12.2.3 uint32_t USBHKeyboardModifierSet (

tUSBHKeyboard * psKbInstance,

uint32_t ui32Modifiers)

This function is used to set one of the fixed modifier keys on a keyboard.

Parameters *psKbInstance* is the instance value for this keyboard.

---

ui32Modifiers is a bit mask of the modifiers to set on the keyboard.

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.

References USBHHIDSetReport().

4.12.2.4 tUSBHKeyboard * USBHKeyboardOpen (

tUSBHIDKeyboardCallback pfnCallback,

uint8_t * pui8Buffer,

uint32_t ui32Size)

This function is used open an instance of a keyboard.

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*.

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.

References eUSBHHIDClassKeyboard, and USBHHIDOpen().

4.12.2.5 uint32_t USBHKeyboardPollRateSet (

tUSBHKeyboard * psKbInstance,

uint32_t ui32PollRate)

This function is used to set the automatic poll rate of the keyboard.

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.

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.

References USBHHIDSetIdle().

4.12.2.6 uint32_t USBHKeyboardUsageToChar (

tUSBHKeyboard * psKbInstance,

const **tHIDKeyboardUsageTable** * psTable,

uint8_t ui8UsageID)

This function is used to map a USB usage ID to a printable character.

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.

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.

References tHIDKeyboardUsageTable::pui32CapsLock, tHIDKeyboard-UsageTable::pvCharMapping, and tHIDKeyboardUsageTable::ui8BytesPerChar.

4.12.2.7 uint32_t USBHMouseClose (

tUSBHMouse * psMsInstance)

This function is used close an instance of a mouse.

Parameters *psMsInstance* is the instance value for this mouse.

This function is used to close an instance of the mouse that was opened with a call to USBH-MouseOpen(). The *psMsInstance* value is the value that was returned when the application called USBHMouseOpen().

Returns Returns 0.

References USBHHIDClose().

4.12.2.8 uint32_t USBHMouseInit (

tUSBHMouse * psMsInstance)

This function is used to initialize a mouse interface after a mouse has been detected.

Parameters *psMsInstance* is the instance value for this mouse.

This function should be called after receiving a **USB_EVENT_CONNECTED** event in the callback 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.

References USBHHIDGetReportDescriptor(), USBHHIDSetIdle(), and USBHHIDSetProtocol().

4.12.2.9 tUSBHMouse * USBHMouseOpen (

tUSBHIDMouseCallback pfnCallback,

uint8_t * pui8Buffer,

uint32_t ui32Size)

This function is used open an instance of a mouse.

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*.

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.

References eUSBHHIDClassMouse, and USBHHIDOpen().

The USB library provides examples for three host applications that can access mass storage devices and HID keyboard and mouse devices. These next sections will cover the basics of each of these three applications and how they interact with the USB library.

4.13 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 will not be 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.

4.14 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 will simply return 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 will 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.

4.15 Application Termination

When the application needs to shut down the host controller it will need 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 will leave 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.

4.16 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_pHCDPool 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_ppHostClassDrivers 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 will be 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 will enable 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$_{M}EMORY_{S}IZE$128

//***
// The memory pool to provide to the Host controller driver.
//***
unsigned char g$_{p}HCDPool[HCD_{M}EMORY_{S}IZE]$;

//***
// 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$_{p}pHostClassDrivers[] = g_{U}SBHIDClassDriver, g_{U}SBHostMSCClassDriver;$

//** // // This global holds the number of class drivers in the g$_{p}pHostClassDrivers//list.////***staticconstunsignedlongg_{u}lNumHostClassDrivers = sizeof(g_{p}pHostClassDrivers)/sizeof(tUSBHostClassDriver*);$

...

// // Enable Clocking to the USB controller. // SysCtlPeripheralEnable(SYSCTL$_{P}ERIPH_{U}SB0);$

// // Enable the peripherals used by this example. // SysCtlPeripheralEnable(SYSCTL$_{P}ERIPH_{U}ART0); SysCtlPeripheralEn$

// // Set the USB0EPEN and USB0PFLT pins to be controlled by the USB // controller. // GPIOPinTypeUSBDigital(GPIO$_{P}ORTH_{B}ASE, GPIO_{P}IN_{3}|GPIO_{P}IN_{4});$

// // Register the host class drivers. // USBHCDRegisterDrivers(0, g$_{p}pUSBHostClassDrivers, g_{u}lNumHostClassDrivers);$

...

// // 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$_{p}HCDPool, HCD_{M}EMORY_{S}IZE);$

4.17 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$_{u}lMouseInstance = USBHMouseOpen(MouseCallback, g_{p}ucBuffer, 128);$

...

// // Main loop of application. // while(1) switch(eMouseState) // // This state is entered when they mouse is first detected. // case MOUSE$_{I}NIT : ////Initializedthenewlyconnectedmouse.//USBHMouseInit(g_{u}lMouseInstance);$

// // Proceed to the mouse connected state. // eMouseState = MOUSE$_{C}ONNECTED;$

break; case MOUSE$_{C}ONNECTED : break; case MOUSE_{N}OT_{C}ONNECTED : default : break;$

// // Periodic call the main loop for the Host controller driver. // USBHCDMain(); ...

Once the mouse has been configured the application's mouse callback routine will be 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 unsigned long MouseCallback(void *pvCBData, unsigned long ulEvent, unsigned long ulMsgParam, void *pvMsgData) switch(ulEvent) // // New mouse detected. // case USB$_EVENT_CONNECTED : eMouseState = MOUSE_INIT; break;$

// // Mouse has been unplugged. // case USB$_EVENT_DISCONNECTED : eMouseState = MOUSE_NOT_CONNECTED; break;$

// // New Mouse events detected. // case USBH$_EVENT_HID_MS_PRESS : break;case USBH_EVENT_HID_MS_REL : break;case USBH_EVENT_HID_MS_X : break;case USBH_EVENT_HID_MS_Y : break;return(0);$

4.18 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$_ulKeyboardInstance = USBHKeyboardOpen(KeyboardCallback, g_pucBuffer, 128);$

// // The main loop for the application. // while(1) switch(eKeyboardState) // // This state is entered when they keyboard is first detected. // case KEYBOARD$_INIT$: $////Initializedthenewlyconnectedkeyboard.//USBHKeyboardInit(g_ulKeyboardInstance);$

// // Proceed to the keyboard connected state. // eKeyboardState = KEYBOARD$_CONNECTED;$

break; case KEYBOARD$_UPDATE$: $////Iftheapplicationdetectedachangethatrequiredan//updatetobesenttothekeyboardtoc$

USBHKeyboardModifierSet(g$_ulKeyboardInstance, g_ulModifiers); case KEYBOARD_CONNECTED$: $break;case KEYBOARD_NOT_CONNECTED : default : break;$

// // 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** will let 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

unsigned long KeyboardCallback(void *pvCBData, unsigned long ulEvent, unsigned long ulMsg-Param, void *pvMsgData) unsigned char ucChar;

switch(ulEvent) // // New keyboard detected. // case USB$_EVENT_CONNECTED$: $eKeyboardState = KEYBOARD_INIT; break;$

// // Keyboard has been unplugged. // case USB$_EVENT_DISCONNECTED$: $eKeyboardState = KEYBOARD_NOT_CONNECTED; break;$

// // New Key press detected. // case USBH$_EVENT_HID_KB_PRESS$: $////ulMsgParamholdstheUSBUsageID.//break; caseUSBH_EVENT_HID_KB_MOD$: $////ulMsgParamholdstheUSBModifierbitmask.//break; caseUSBH_EVENT_HID_KB_REL$: $////ulMsgParamholdstheUSBUsageID.//break; return(0);$

4.19 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$_u lMSCInstance = USBHMSCDriveOpen(0, MSCCallback);$

...

// // Wait for the drive to become ready. // while(USBHMSCDriveReady(g$_u lMSCInstance$))$////Systemleveldelaycallshouldbeh$

...

// // Block Read example. // USBHMSCBlockRead(g$_u lMSCInstance, ulLBA, pucData, 1);$

...

// // Block Write example. // USBHMSCBlockWrite(g$_u lMSCInstance, ulLBA, pucData, 1);$

...

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265