LUMINARY MICRO®

Stellaris® USB Library

USER'S GUIDE

PDL-USBL-uvision-UG-2523

Legal Disclaimers and Trademark Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH LUMINARY MICRO PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN LUMINARY MICRO'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, LUMINARY MICRO ASSUMES NO LIABILITY WHATSOEVER, AND LUMINARY MICRO DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF LUMINARY MICRO'S PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. LUMINARY MICRO'S PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE-SUSTAINING APPLICATIONS.

Luminary Micro may make changes to specifications and product descriptions at any time, without notice. Contact your local Luminary Micro sales office or your distributor to obtain the latest specifications and before placing your product order.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Luminary Micro reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Luminary Micro, Inc.
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

Revision Information

This is version 2523 of this document, last updated on April 03, 2008.

Table of Contents

1 Introduction

The Luminary Micro® Stellaris® USB Library is a set of data types and functions for creating USB device, host or On-The-Go (OTG) applications on Stellaris 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.

2 General Purpose Functions

2.1 Introduction

This chapter describes the set of USB library data types and functions which are of general use in the development of USB applications running on Stellaris microcontroller-based boards. These elements are not specific to USB host, device or On-The-Go (OTG) operation.

The functions and types described here fall into three main categories:

- Definitions of USB standard types (as found in Chapter 9 of the USB 2.0 specification) and functions to parse them.
- Functions relating to host/device mode switching for On-The-Go or dual mode applications.
- USB device class header files.

Source Code Overview

Source code and headers for the general-purpose USB functions can be found in the top level of the USB library tree.

`usblib.h`	The main header file for the USB library. All clients of the USB Library should include this header.
`usb-ids.h`	The header file containing labels defining the Luminary Micro USB vendor ID (VID) and product IDs (PIDs) for each of the example devices provided in USB-capable evaluation kits.
`usbmsc.h`	The header file containing definitions specific to the USB Mass Storage Class.
`usbcdc.h`	The header file containing definitions specific to the USB Communication Device Class.
`usbhid.h`	The header file containing definitions specific to the USB Human Interface Device Class.
`usbdesc.c`	The source code for a set of functions allowing parsing of USB descriptors.
`usbmode.c`	The source code for a set of functions related to switching between host and device modes of operation.
`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.

2.2 Function Definitions

Defines

- USB_DESC_ANY

Enumerations

- tUSBMode

Functions

- void USB0DualModeIntHandler (void)
- tDescriptorHeader * USBDescGet (tDescriptorHeader *psDesc, unsigned long ulSize, unsigned long ulType, unsigned long ulIndex)
- tInterfaceDescriptor * USBDescGetAlternateInterface (tConfigDescriptor *psConfig, unsigned char ucInterfaceNumber, unsigned long ulIndex)
- tInterfaceDescriptor * USBDescGetInterface (tConfigDescriptor *psConfig, unsigned long ulIndex, unsigned long ulAlt)
- tEndpointDescriptor * USBDescGetInterfaceEndpoint (tInterfaceDescriptor *psInterface, unsigned long ulIndex, unsigned long ulSize)
- unsigned long USBDescGetNum (tDescriptorHeader *psDesc, unsigned long ulSize, unsigned long ulType)
- unsigned long USBDescGetNumAlternateInterfaces (tConfigDescriptor *psConfig, unsigned char ucInterfaceNumber)
- void USBDualModeInit (unsigned long ulIndex)
- void USBDualModeTerm (unsigned long ulIndex)
- void USBStackModeSet (unsigned long ulIndex, tUSBMode eUSBMode, 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 Define Documentation

2.2.2.1 USB_DESC_ANY

Definition:

```
#define USB_DESC_ANY
```

Description:

The USB_DESC_ANY label is used as a wild card in several of the descriptor parsing APIs to determine whether or not particular search criteria should be ignored.

2.2.3 Enumeration Documentation

2.2.3.1 tUSBMode

Description:

The operating mode required by the USB library client. This type is used by applications which wish to be able to switch between host and device modes by calling the USBStackModeSet() API.

Enumerators:

USB_MODE_DEVICE The application wishes to operate as a USB device.

USB_MODE_HOST The application wishes to operate as a USB host.

USB_MODE_NONE A marker indicating that no USB mode has yet been set by the application.

2.2.4 Function Documentation

2.2.4.1 USB0DualModeIntHandler

Steers USB interrupts from controller to the correct handler in the USB stack.

Prototype:

```
void
USB0DualModeIntHandler(void)
```

Description:

This interrupt handler is used in applications which require to operate in both host and device mode. It steers the USB hardware interrupt to the correct handler in the USB stack depending upon the current operating mode of the application, USB device or host.

For successful dual mode operation, an application must register USB0DualModeIntHandler() in the CPU vector table as the interrupt handler for the USB0 interrupt. This handler is responsible for steering interrupts to the device or host stack depending upon the chosen mode.

Note:

Devices which do not require dual mode capability should register either USB0DeviceIntHandler() or USB0HostIntHandler() instead. Registering USB0DualModeIntHandler() for a single mode application will result in an application binary larger than required since library functions for both USB operating modes will be included even though only one mode is actually required.

Returns:

None.

2.2.4.2 USBDescGet

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

Prototype:

```
tDescriptorHeader *
USBDescGet(tDescriptorHeader *psDesc,
           unsigned long ulSize,
           unsigned long ulType,
           unsigned long ulIndex)
```

Parameters:

psDesc points to the first byte of a block of standard USB descriptors.

ulSize is the number of bytes of descriptor data found at pointer *psDesc*.

ulType 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.

ulIndex is the zero based index of the descriptor whose pointer is to be returned. For example, passing value 1 in *ulIndex* returns the second matching descriptor.

Description:

Return a pointer to the n-th descriptor of a particular type found in the block of *ulSize* bytes starting at *psDesc*.

Returns:

Returns a pointer to the header of the required descriptor if found or NULL otherwise.

2.2.4.3 USBDescGetAlternateInterface

Returns a pointer to the n-th interface descriptor in a config descriptor with the supplied interface number.

Prototype:

```
tInterfaceDescriptor *
USBDescGetAlternateInterface(tConfigDescriptor *psConfig,
                             unsigned char ucInterfaceNumber,
                             unsigned long ulIndex)
```

Parameters:

psConfig points to the first byte of a standard USB configuration descriptor.

ucInterfaceNumber is the interface number of the descriptor that is being queried.

ulIndex is the zero based index of the descriptor to return.

Description:

This function returns a pointer to the n-th interface descriptor in the supplied configuration which has the requested interface number. It may be used by a client to retrieve the descriptors for each alternate setting of a given interface within the configuration passed.

Returns:

Returns a pointer to the n-th interface descriptor with interface number as specified or NULL of this descriptor does not exist.

2.2.4.4 USBDescGetInterface

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

Prototype:

```
tInterfaceDescriptor *
USBDescGetInterface(tConfigDescriptor *psConfig,
                    unsigned long ulIndex,
                    unsigned long ulAlt)
```

Parameters:

psConfig points to the first byte of a standard USB configuration descriptor.

ulIndex is the zero based index of the interface that is to be found. If ulAlt is set to a value other than **USB_DESC_ANY**, this will be equivalent to the interface number being searched for.

ulAlt is the alternate setting number which is to be searched for. If this value is **USB_DESC_ANY**, the alternate setting is ignored and all interface descriptors are considered in the search.

Description:

Return a pointer to the n-th interface descriptor found in the supplied configuration descriptor. If *ulAlt* 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 ulAlt if ulIndex is set to the required interface number and ulAlt 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.

2.2.4.5 USBDescGetInterfaceEndpoint

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

Prototype:

```
tEndpointDescriptor *
USBDescGetInterfaceEndpoint(tInterfaceDescriptor *psInterface,
                            unsigned long ulIndex,
                            unsigned long ulSize)
```

Parameters:

psInterface points to the first byte of a standard USB interface descriptor.

ulIndex is the zero based index of the endpoint that is to be found.

ulSize contains the maximum number of bytes that the function may search beyond *psInterface* while looking for the requested endpoint descriptor.

Description:

Return a pointer to the n-th endpoint descriptor found in the supplied interface descriptor. If the *ulIndex* parameter is invalid (greater than or equal to the bNumEndpoints field of the inter-

face descriptor) or the endpoint cannot be found within *ulSize* 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 ulSize to limit the search range.

Returns:
Returns a pointer to the requested endpoint descriptor if found or NULL otherwise.

2.2.4.6 USBDescGetNum

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

Prototype:
```
unsigned long
USBDescGetNum(tDescriptorHeader *psDesc,
              unsigned long ulSize,
              unsigned long ulType)
```

Parameters:
psDesc points to the first byte of a block of standard USB descriptors.
ulSize is the number of bytes of descriptor data found at pointer *psDesc*.
ulType identifies the type of descriptor that is to be counted. If the value is **USB_DESC_ANY**, the function returns the total number of descriptors regardless of type.

Description:
This function can be used to count the number of descriptors of a particular type within a block of descriptors. The caller can provide a specific type value which the function matches against the second byte of each descriptor or, alternatively, can specify **USB_DESC_ANY** to have the function count all descriptors regardless of their type.

Returns:
Returns the number of descriptors found in the supplied block of data.

2.2.4.7 USBDescGetNumAlternateInterfaces

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

Prototype:
```
unsigned long
USBDescGetNumAlternateInterfaces(tConfigDescriptor *psConfig,
                                 unsigned char ucInterfaceNumber)
```

Parameters:
psConfig points to the first byte of a standard USB configuration descriptor.
ucInterfaceNumber is the interface number for which the number of alternate configurations is to be counted.

Description:
This function can be used to count the number of alternate settings for a specific interface within a configuration.

Returns:
Returns the number of alternate versions of the specified interface or 0 if the interface number supplied cannot be found in the config descriptor.

2.2.4.8 USBDualModeInit

Initializes the USB controller for dual mode operation.

Prototype:
```
void
USBDualModeInit(unsigned long ulIndex)
```

Parameters:
ulIndex specifies the USB controller that is to be initialized for dual mode operation. This parameter must be set to 0.

Description:
This function initializes the USB controller hardware into a state suitable for dual mode operation. Applications may use this function to ensure that the controller is in a neutral state and able to receive appropriate interrupts before host or device mode is chosen using a call to USBStackModeSet().

Returns:
None.

2.2.4.9 USBDualModeTerm

Returns the USB controller to the default mode when in dual mode operation.

Prototype:
```
void
USBDualModeTerm(unsigned long ulIndex)
```

Parameters:
ulIndex specifies the USB controller whose dual mode operation is to be ended. This parameter must be set to 0.

Description:
Applications using both host and device modes may call this function to disable interrupts in preparation for shutdown or a change of operating mode.

Returns:
None.

2.2.4.10 USBStackModeSet

Allows a dual-mode application to switch between USB device and host modes.

Prototype:

```
void
USBStackModeSet(unsigned long ulIndex,
                tUSBMode eUSBMode,
                tUSBModeCallback pfnCallback)
```

Parameters:

ulIndex specifies the USB controller whose mode of operation is to be set. This parameter must be set to 0.

eUSBMode indicates the mode that the application wishes to operate in. Valid values are **USB_MODE_DEVICE** to operate as a USB device and **USB_MODE_HOST** 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 *eUSBMode* is set to either **USB_MODE_DEVICE** or **USB_MODE_HOST**, the callback will be made immediately to allow the application to perform any host or device specific initialization.

Description:

This function allows a USB application, which 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.

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) need not call this function since no interrupt steering is required if the appropriate single mode interrupt handler is installed.

Returns:

None.

2.3 USB Chapter 9 Definitions

Data Structures

- tConfigDescriptor
- tDescriptorHeader
- tDeviceDescriptor
- tDeviceQualifierDescriptor
- tEndpointDescriptor

- tInterfaceDescriptor
- tString0Descriptor
- tStringDescriptor
- tUSBRequest

Defines

- NEXT_USB_DESCRIPTOR(ptr)
- USBLong(ulValue)
- USBShort(usValue)

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. Note that this convention departs from the naming convention applied to all other Luminary Micro data types.

It is important to be aware that all the structures described in this section are byte packed. Appropriate typedef modifiers are included in `usblib.h` to ensure the correct packing for all currently-supported toolchains.

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

2.3.2 Data Structure Documentation

2.3.2.1 tConfigDescriptor

Definition:

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned short wTotalLength;
    unsigned char bNumInterfaces;
    unsigned char bConfigurationValue;
    unsigned char iConfiguration;
    unsigned char bmAttributes;
    unsigned char bMaxPower;
}
tConfigDescriptor
```

Members:

bLength The length of this descriptor in bytes. All configuration descriptors are 9 bytes long.

bDescriptorType The type of the descriptor. For a configuration descriptor, this will be USB_DTYPE_CONFIGURATION (2).

wTotalLength The total length of data returned for this configuration. This includes the combined length of all descriptors (configuration, interface, endpoint and class- or vendor-specific) returned for this configuration.

bNumInterfaces The number of interface supported by this configuration.

bConfigurationValue The value used as an argument to the SetConfiguration standard request to select this configuration.

iConfiguration The index of a string descriptor describing this configuration.

bmAttributes Attributes of this configuration.

bMaxPower The maximum power consumption of the USB device from the bus in this configuration when the device is fully operational. This is expressed in units of 2mA so, for example, 100 represents 200mA.

Description:

This structure describes the USB configuration descriptor as defined in USB 2.0 specification section 9.6.3. This structure also applies to the USB other speed configuration descriptor defined in section 9.6.4.

2.3.2.2 tDescriptorHeader

Definition:

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
}
tDescriptorHeader
```

Members:

bLength The length of this descriptor (including this length byte) expressed in bytes.

bDescriptorType The type identifier of the descriptor whose information follows. For standard descriptors, this field could contain, for example, USB_DTYPE_DEVICE to identify a device descriptor or USB_DTYPE_ENDPOINT to identify an endpoint descriptor.

Description:

This structure describes a generic descriptor header. These fields are to be found at the beginning of all valid USB descriptors.

2.3.2.3 tDeviceDescriptor

Definition:

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned short bcdUSB;
    unsigned char bDeviceClass;
    unsigned char bDeviceSubClass;
    unsigned char bDeviceProtocol;
```

```
    unsigned char bMaxPacketSize0;
    unsigned short idVendor;
    unsigned short idProduct;
    unsigned short bcdDevice;
    unsigned char iManufacturer;
    unsigned char iProduct;
    unsigned char iSerialNumber;
    unsigned char bNumConfigurations;
}
tDeviceDescriptor
```

Members:

bLength The length of this descriptor in bytes. All device descriptors are 18 bytes long.

bDescriptorType The type of the descriptor. For a device descriptor, this will be USB_DTYPE_DEVICE (1).

bcdUSB The USB Specification Release Number in BCD format. For USB 2.0, this will be 0x0200.

bDeviceClass The device class code.

bDeviceSubClass The device subclass code. This value qualifies the value found in the bDeviceClass field.

bDeviceProtocol The device protocol code. This value is qualified by the values of bDeviceClass and bDeviceSubClass.

bMaxPacketSize0 The maximum packet size for endpoint zero. Valid values are 8, 16, 32 and 64.

idVendor The device Vendor ID (VID) as assigned by the USB-IF.

idProduct The device Product ID (PID) as assigned by the manufacturer.

bcdDevice The device release number in BCD format.

iManufacturer The index of a string descriptor describing the manufacturer.

iProduct The index of a string descriptor describing the product.

iSerialNumber The index of a string descriptor describing the device's serial number.

bNumConfigurations The number of possible configurations offered by the device. This field indicates the number of distinct configuration descriptors that the device offers.

Description:

This structure describes the USB device descriptor as defined in USB 2.0 specification section 9.6.1.

2.3.2.4 tDeviceQualifierDescriptor

Definition:

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned short bcdUSB;
    unsigned char bDeviceClass;
    unsigned char bDeviceSubClass;
    unsigned char bDeviceProtocol;
    unsigned char bMaxPacketSize0;
    unsigned char bNumConfigurations;
```

```
    unsigned char bReserved;
}
tDeviceQualifierDescriptor
```

Members:

bLength The length of this descriptor in bytes. All device qualifier descriptors are 10 bytes long.

bDescriptorType The type of the descriptor. For a device descriptor, this will be USB_DTYPE_DEVICE_QUAL (6).

bcdUSB The USB Specification Release Number in BCD format. For USB 2.0, this will be 0x0200.

bDeviceClass The device class code.

bDeviceSubClass The device subclass code. This value qualifies the value found in the bDeviceClass field.

bDeviceProtocol The device protocol code. This value is qualified by the values of bDeviceClass and bDeviceSubClass.

bMaxPacketSize0 The maximum packet size for endpoint zero when operating at a speed other than high speed.

bNumConfigurations The number of other-speed configurations supported.

bReserved Reserved for future use. Must be set to zero.

Description:

This structure describes the USB device qualifier descriptor as defined in the USB 2.0 specification, section 9.6.2.

2.3.2.5 tEndpointDescriptor

Definition:

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned char bEndpointAddress;
    unsigned char bmAttributes;
    unsigned short wMaxPacketSize;
    unsigned char bInterval;
}
tEndpointDescriptor
```

Members:

bLength The length of this descriptor in bytes. All endpoint descriptors are 7 bytes long.

bDescriptorType The type of the descriptor. For an endpoint descriptor, this will be USB_DTYPE_ENDPOINT (5).

bEndpointAddress The address of the endpoint. This field contains the endpoint number ORed with flag USB_EP_DESC_OUT or USB_EP_DESC_IN to indicate the endpoint direction.

bmAttributes The endpoint transfer type, USB_EP_ATTR_CONTROL, USB_EP_ATTR_ISOC, USB_EP_ATTR_BULK or USB_EP_ATTR_INT and, if isochronous, additional flags indicating usage type and synchronization method.

wMaxPacketSize The maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. For high speed isochronous or interrupt endpoints, bits 11 and 12 are used to pass additional information.

bInterval The polling interval for data transfers expressed in frames or microframes depending upon the operating speed.

Description:

This structure describes the USB endpoint descriptor as defined in USB 2.0 specification section 9.6.6.

2.3.2.6 tInterfaceDescriptor

Definition:

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned char bInterfaceNumber;
    unsigned char bAlternateSetting;
    unsigned char bNumEndpoints;
    unsigned char bInterfaceClass;
    unsigned char bInterfaceSubClass;
    unsigned char bInterfaceProtocol;
    unsigned char iInterface;
}
tInterfaceDescriptor
```

Members:

bLength The length of this descriptor in bytes. All interface descriptors are 9 bytes long.

bDescriptorType The type of the descriptor. For an interface descriptor, this will be USB_DTYPE_INTERFACE (4).

bInterfaceNumber The number of this interface. This is a zero based index into the array of concurrent interfaces supported by this configuration.

bAlternateSetting The value used to select this alternate setting for the interface defined in bInterfaceNumber.

bNumEndpoints The number of endpoints used by this interface (excluding endpoint zero).

bInterfaceClass The interface class code as assigned by the USB-IF.

bInterfaceSubClass The interface subclass code as assigned by the USB-IF.

bInterfaceProtocol The interface protocol code as assigned by the USB-IF.

iInterface The index of a string descriptor describing this interface.

Description:

This structure describes the USB interface descriptor as defined in USB 2.0 specification section 9.6.5.

2.3.2.7 tString0Descriptor

Definition:

```
typedef struct
{
```

```
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned short wLANGID[1];
}
tString0Descriptor
```

Members:

bLength The length of this descriptor in bytes. This value will vary depending upon the number of language codes provided in the descriptor.

bDescriptorType The type of the descriptor. For a string descriptor, this will be USB_DTYPE_STRING (3).

wLANGID The language code (LANGID) for the first supported language. Note that this descriptor may support multiple languages, in which case, the number of elements in the wLANGID array will increase and bLength will be updated accordingly.

Description:

This structure describes the USB string descriptor for index 0 as defined in USB 2.0 specification section 9.6.7. Note that the number of language IDs is variable and can be determined by examining bLength. The number of language IDs present in the descriptor is given by ((bLength - 2) / 2).

2.3.2.8 tStringDescriptor

Definition:

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned char bString;
}
tStringDescriptor
```

Members:

bLength The length of this descriptor in bytes. This value will be 2 greater than the number of bytes comprising the UNICODE string that the descriptor contains.

bDescriptorType The type of the descriptor. For a string descriptor, this will be USB_DTYPE_STRING (3).

bString The first byte of the UNICODE string. This string is not NULL terminated. Its length (in bytes) can be computed by subtracting 2 from the value in the bLength field.

Description:

This structure describes the USB string descriptor for all string indexes other than 0 as defined in USB 2.0 specification section 9.6.7.

2.3.2.9 tUSBRequest

Definition:

```
typedef struct
{
    unsigned char bmRequestType;
```

```
    unsigned char bRequest;
    unsigned short wValue;
    unsigned short wIndex;
    unsigned short wLength;
}
tUSBRequest
```

Members:

bmRequestType Determines the type and direction of the request.

bRequest Identifies the specific request being made.

wValue Word-sized field that varies according to the request.

wIndex Word-sized field that varies according to the request; typically used to pass an index or offset.

wLength The number of bytes to transfer if there is a data stage to the request.

Description:

The standard USB request header as defined in section 9.3 of the USB 2.0 specification.

2.3.3 Define Documentation

2.3.3.1 NEXT_USB_DESCRIPTOR

Traverse to the next USB descriptor in a block.

Definition:

```
#define NEXT_USB_DESCRIPTOR(ptr)
```

Parameters:

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

Description:

This macro aids in traversing lists of descriptors by returning a pointer to the next descriptor in the list given a pointer to the current one.

Returns:

Returns a pointer to the next descriptor in the block following *ptr*.

2.3.3.2 USBLong

Write a 4 byte unsigned long value to a USB descriptor block.

Definition:

```
#define USBLong(ulValue)
```

Parameters:

ulValue is the four byte unsigned long that is to be written to the descriptor.

Description:

This helper macro is used in descriptor definitions to write four-byte values. Since the configuration descriptor contains all interface and endpoint descriptors in a contiguous block of

memory, these descriptors are typically defined using an array of bytes rather than as packed structures.

Returns:
Not a function.

2.3.3.3 USBShort

Write a 2 byte unsigned short value to a USB descriptor block.

Definition:
```
#define USBShort(usValue)
```

Parameters:
usValue is the two byte unsigned short that is to be written to the descriptor.

Description:
This helper macro is used in descriptor definitions to write two-byte values. Since the configuration descriptor contains all interface and endpoint descriptors in a contiguous block of memory, these descriptors are typically defined using an array of bytes rather than as packed structures.

Returns:
Not a function.

3 Device Functions

3.1 Introduction

The functions described in this chapter are intended for use by USB devices and provide a framework which handles all the class independent operations required by a USB device. These operations include device, configuration and string descriptor management, handling of all standard USB requests, endpoint management and low level interrupt handling.

Class specific application code interfaces with the USB library by means of exported descriptor structures and a table of callback functions called by the library in response to particular events observed by the USB controller.

Source Code Overview

Source code and headers for the general purpose USB functions can be found in the `device` directory of the USB library tree.

`usbdenum.c`	The source code for the USB device enumeration functions offered by the library.
`usbdevice.h`	The header file containing device mode function prototypes and data types offered by the library.
`usbdhandler.c`	The source code for the USB device interrupt handler.
`usbdhid.h`	The header file containing Human Interface Device definitions specific to devices of this class.

3.2 Creating a USB Device

Creating a device application using the USB library 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.
- Configure endpoints and partition the USB controller FIFO appropriately for your application.
- Call the USB library to connect the device to the bus and manage standard host interaction on your behalf.

The following sections walk through each of these steps offering code examples to illustrate the process. Working examples illustrating use of the library can also be found in the DriverLib release for your USB-capable evaluation kit.

The term “device code” used in the following sections describes all class specific code written above the USB library to implement a particular USB device application.

3.2.1 Building Descriptors

The USB library 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 file `usbdescriptors.c` in each of the USB device example applications.

3.2.1.1 tDeviceInfo.pDeviceDescriptor

This array must hold the device descriptor that the USB library will return to the host in response to a GET_DESCRIPTOR(DEVICE) request. The following example contains the device descriptor provided by the `usb_dev_keyboard` example.

```
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_LUMINARY), // 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.

3.2.1.2 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 each configuration descriptor. 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.

Individual configuration descriptors are, once again, defined as arrays of bytes. The content, however, is somewhat more complex than the device descriptor due to the amount of information passed alongside the 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 library 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 library requires that individual configurations are numbered consecutively starting at 1 for the first configuration.

The following example contains the configuration descriptor provided by the `usb_dev_keyboard` example. 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 library is capable of safely skipping device specific descriptors when parsing these structures.

```
//*****************************************************************************
//
// Keyboard configuration descriptor.
//
// Note that 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 iConfiguration 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.
    1,                          // The number of interfaces in this
                                // configuration.
    1,                          // The unique value for this configuration.
    0,                          // The string identifier that describes this
                                // configuration.
    USB_CONF_ATTR_SELF_PWR,     // Bus Powered, Self Powered, remote wake up.
    250,                        // The maximum power in 2mA increments.

    //
    // 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 constant defined by
```

```
                                // USB-IF.
    USB_HID_SCLASS_BOOT,        // The interface sub-class constant defined
                                // by USB-IF.
    USB_HID_PROTOCOL_KEYB,      // The interface protocol for the sub-class
                                // specified in ucIfaceSubClass.
    0,                          // 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.

    //
    // Endpoint Descriptor
    //
    7,                          // The size of the endpoint descriptor.
    USB_DTYPE_ENDPOINT,         // Descriptor type is an endpoint.
    USB_EP_DESC_IN | 1,         // Endpoint 1 is an IN endpoint.
    USB_EP_ATTR_INT,            // Endpoint is an interrupt endpoint.
    USBShort(8),                // The maximum packet size for this endpoint.
    10,                         // The polling interval for this endpoint.
};

//
// Configuration Descriptors.
//
const unsigned char * const g_ppConfigDescriptors[] =
{
    g_pKeyboardDescriptor
};
```

3.2.1.3 tDeviceInfo.ppStringDescriptors and tDeviceInfo.ulNumStringDescriptors

Descriptive strings referenced by device and configuration descriptors are provided to the USB library 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 library by the `usb_dev_keyboard` example 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[] =
{

    (19 + 1) * 2,
    USB_DTYPE_STRING,
    'L', 0, 'u', 0, 'm', 0, 'i', 0, 'n', 0, 'a', 0, 'r', 0, 'y', 0, ' ', 0,
    'M', 0, 'i', 0, 'c', 0, 'r', 0, 'o', 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.

3.2.2 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 library. These callback functions are made available to the USB library 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

Note that all callbacks 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.

3.2.2.1 pfnGetDescriptor

Standard USB device, configuration and string descriptors are handled by the USB library internally but some device classes also define additional, class specific descriptors. In cases where the host requests one of these non-standard descriptors, this callback is made to give the device code an opportunity to provide its own descriptor to satisfy the request.

If the device can satisfy the request, it must call USBDCDSendDataEP0() to provide the requested descriptor data to the host. If the request cannot be satisfied, the device should call USBDCD-StallEP0() to indicate that the descriptor request is not supported.

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

3.2.2.2 pfnRequestHandler

The USB library 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 Stellaris Peripheral Driver Library USB driver.

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 library 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 library will stall endpoint zero whenever it receives a non-standard request.

3.2.2.3 pfnInterfaceChange

Based on the configuration descriptor published by the device code, several different alternate interface settings may be supported. In cases where the host wishes to change from the default interface configuration and the USB library determines that the requested alternate setting is supported, this callback is made to inform the device code of the change. The parameter passed is the new alternate interface setting number as published in the `bAlternateSetting` field of the interface descriptor.

This callback is only made once the USB library has validated the requested alternate setting. If the requested setting is not available in the published configuration descriptor, the USB library 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 library will note the interface change internally but not report it to the device code.

3.2.2.4 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 library 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 library will note the configuration change internally but not report it to the device code.

3.2.2.5 pfnDataReceived

This callback informs the device code of the arrival of data following an earlier call to USBDCDRequestDataEP0(). On this callback, the received data will have been written into the buffer provided to the USB library 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 library itself.

If this member of sCallbacks is set to NULL, the USB library 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.

3.2.2.6 pfnDataSent

The USBDCDSendDataEP0() function allows device code to send an arbitrarily-sized block of data to the host via endpoint zero. The maximum packet size that can be sent via endpoint zero is, however, 64 bytes so larger blocks of data are sent in multiple packets. This callback function is used by the USB library 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 library will not inform the device code when a block of EP0 data is sent.

3.2.2.7 pfnResetHandler

The pfnResetHandler callback is made by the USB library 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 library will not inform the device code when a bus reset occurs.

3.2.2.8 pfnSuspendHandler

The pfnSuspendHandler callback is made whenever the USB library 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 library will not inform the device code when a bus suspend occurs.

3.2.2.9 pfnResumeHandler

The pfnResumeHandler callback is made whenever the USB library 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 library will not inform the device code when a bus resume occurs.

3.2.2.10 pfnDisconnectHandler

On Stellaris parts supporting USB On-The-Go (OTG) functionality, the pfnDisconnectHandler callback is made whenever the USB library detects that the host or device has disconnected from the bus. On Stellaris parts offering USB Host + Device functionality, the method employed to detect disconnection will be dependent upon the board design and this callback cannot be guaranteed.

If this member of sCallbacks is set to NULL, the USB library will not inform the device code when a disconnection event occurs.

3.2.2.11 pfnEndpointHandler

While the use of endpoint zero is standardized and supported via several of the other callbacks already listed (pfnDataSent, pfnDataReceived, pfnGetDescriptor, pfnRequestHandler, pfnInterfaceChange and pfnConfigChange), the use of other endpoints is entirely dependent upon the device class being implemented. The pfnEndpointHandler callback is, therefore, made to notify the device code of all activity on any endpoint other than endpoint zero and it is the device code's responsibility to determine the correct action to take in response to each callback.

The ulStatus parameter passed to the handler provides information on the actual endpoint for which the callback is being made and allows the handler to determine if the event is due to transmission (if an IN endpoint event occurs) or reception (if an OUT endpoint event occurs) of data.

Having determined the endpoint sourcing the event, the device code can determine the actual event by calling USBEndpointStatus() for the appropriate endpoint then clear the status by calling USBDevEndpointStatusClear().

When incoming data is indicated by the flag USB_DEV_RX_PKT_RDY being set in the endpoint status, data can be received using a call to USBEndpointDataGet() followed by a call to USBDevEndpointDataAck() to acknowledge the reception to the host.

When an event relating to an IN endpoint (data transmitted from the device to the host) is received, the status read from USBEndpointStatus() indicates any errors in transmission. If the value read is 0, this implies that the data was successfully transmitted and acknowledged by the host.

Any device whose configuration descriptor indicates that it uses any endpoint (endpoint zero use is assumed) must populate the pfnEndpointHandler member of tCustomHandlers.

3.2.3 USB Hardware Initialization

The following example shows the procedure required to initialize the USB hardware controller to operate as a USB device. The code here is taken from the usb_dev_mouse example which makes use of a single interrupt endpoint in addition to endpoint zero. The basic steps required are:

- Enable clocking to the controller and the USB PHY.
- Configure the USB endpoints to be used by the device.
- Set the appropriate sizes for each endpoint FIFO.

When configuring the FIFO, the application is responsible for configuring only endpoints other than endpoint zero (which is configured automatically by the USB library). It is important, therefore, to ensure that the first configured endpoint FIFO address is set no lower than MAX_PACKET_SIZE_EP0 (64)since the first 64 bytes of the FIFO RAM is always used by endpoint zero.

The FIFO size for each endpoint must be at least as large as the maximum packet size for that endpoint. Note that when using double buffered FIFOs, the amount of FIFO RAM occupied is double the size specified in the third parameter to USBFIFOConfigSet(). Specifying size USB_FIFO_SZ_64_DB, therefore, uses 128 bytes of the total FIFO space and you must ensure that the next endpoint FIFO configured starts at a FIFO address 128 bytes higher than the previous endpoint FIFO address.

```
//
// Enable Clocking to the USB controller.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_USB0);

//
// Turn on USB Phy clock.
//
SysCtlUSBPLLEnable();

//
// Configure the interrupt endpoint.
//
USBDevEndpointConfig(USB0_BASE,
                     USB_EP_1,
                     64,
                     USB_EP_DEV_IN | USB_EP_MODE_INT);

//
// Configure FIFO.
//
USBFIFOConfigSet(USB0_BASE, USB_EP_1, MAX_PACKET_SIZE_EP0, USB_FIFO_SZ_64,
                 USB_EP_DEV_IN);
```

3.2.4 Passing Control to the USB Library

When all previous setup steps have been completed, control can be passed to the USB library. 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 library our device information and connect the device to
// the bus.
//
USBDCDInit(0, &g_sMouseDeviceInfo);
```

3.3 Device Function Definitions

Data Structures

- tCustomHandlers
- tDeviceInfo

Functions

- void USB0DeviceIntHandler (void)
- void USBDCDInit (unsigned long ulIndex, tDeviceInfo *psDevice)
- void USBDCDRequestDataEP0 (unsigned long ulIndex, unsigned char *pucData, unsigned long ulSize)
- void USBDCDSendDataEP0 (unsigned long ulIndex, unsigned char *pucData, unsigned long ulSize)
- void USBDCDSetDefaultConfiguration (unsigned long ulIndex, unsigned long ulDefaultConfig)
- void USBDCDStallEP0 (unsigned long ulIndex)
- void USBDCDTerm (unsigned long ulIndex)

3.3.1 Data Structure Documentation

3.3.1.1 tCustomHandlers

Definition:

```
typedef struct
{
    tStdRequest pfnGetDescriptor;
    tStdRequest pfnRequestHandler;
    tInfoCallback pfnInterfaceChange;
    tInfoCallback pfnConfigChange;
    tInfoCallback pfnDataReceived;
    tInfoCallback pfnDataSent;
    tUSBIntHandler pfnResetHandler;
    tUSBIntHandler pfnSuspendHandler;
    tUSBIntHandler pfnResumeHandler;
    tUSBIntHandler pfnDisconnectHandler;
    tUSBEPIntHandler pfnEndpointHandler;
}
tCustomHandlers
```

Members:

pfnGetDescriptor This callback is made whenever the USB host requests a non-standard descriptor from the device.

pfnRequestHandler This callback is made whenever the USB host makes a non-standard request.

pfnInterfaceChange This callback is made in response to a SetInterface request from the host.

pfnConfigChange This callback is made in response to a SetConfiguration request from the host.

pfnDataReceived This callback is made when data has been received following to a call to USBDCDRequestDataEP0.

pfnDataSent This calback is made when data has been transmitted following a call to USBDCDSendDataEP0.

pfnResetHandler This callback is made when a USB reset is detected.

pfnSuspendHandler This callback is made when the bus has been inactive long enough to trigger a suspend condition.

pfnResumeHandler This is called when resume signaling is detected.

pfnDisconnectHandler This callback is made when the device is disconnected from the USB bus.

pfnEndpointHandler This callback is made to inform the device of activity on all endpoints other than endpoint zero.

Description:

USB event handler functions used during enumeration and operation of the device stack.

3.3.1.2 tDeviceInfo

Definition:

```
typedef struct
{
    tCustomHandlers sCallbacks;
    const unsigned char *pDeviceDescriptor;
    const unsigned char *const *ppConfigDescriptors;
    const unsigned char *const *ppStringDescriptors;
    unsigned long ulNumStringDescriptors;
}
tDeviceInfo
```

Members:

sCallbacks A pointer to a structure containing pointers to event handler functions provided by the client to support the operation of this device.

pDeviceDescriptor A pointer to the device descriptor for this device.

ppConfigDescriptors A pointer to an array of config descriptor pointers. Each entry in the array corresponds to one configuration that the device may be set to use by the USB host. The number of entries in the array must match the bNumConfigurations value in the device descriptor array, pDeviceDescriptor.

ppStringDescriptors A pointer to the string descriptor array for this device.

ulNumStringDescriptors The number of descriptors provided in the ppStringDescriptors array.

Description:

This structure is passed to the USB library on a call to USBDCDInit and provides the library with information about the device that the application is implementing. It contains functions pointers for the various USB event handlers and pointers to each of the standard device descriptors.

3.3.2 Function Documentation

3.3.2.1 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.

3.3.2.2 USBDCDInit

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

Parameters:
ulIndex 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.

This function must be called by any application which wishes to operate as a USB device. It 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 device information structure passed in *psDevice* must remain unchanged between this call and any matching call to USBDCDTerm() since it is not copied by the USB library.

Returns:
None.

3.3.2.3 USBDCDRequestDataEP0

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

Parameters:
ulIndex is the index of the USB controller from which the data is being requested.
pucData is a pointer to the buffer to fill with data from the USB host.
ulSize 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, `tDeviceInfo.sCallbacks.pfnDataReceived` should 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.

3.3.2.4 USBDCDSendDataEP0

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

Parameters:
ulIndex is the index of the USB controller which is to be used to send the data.
pucData is a pointer to the buffer to send via endpoint zero.
ulSize 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, `tDeviceInfo.sCallbacks.pfnDataSent` should contain a valid function pointer. This callback could be used to free up the buffer passed into this function in the *pucData* parameter. The contents of the *pucData* buffer must remain unchanged until the `pfnDataSent` callback is received.

Returns:
None.

3.3.2.5 USBDCDSetDefaultConfiguration

This function sets the default configuration for the device.

Parameters:
ulIndex is the index of the USB controller whose default configuration is to be set.
ulDefaultConfig 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 config 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 `ppConfigDescriptors` array for the device.

If this function is not called, the USB library will return the first configuration in the `ppConfigDescriptors` array as the default configuration.

Note:
The USB device stack assumes that the configuration IDs (byte 6 of the config descriptor, `bConfigurationValue`) stored within the configuration descriptor array, `ppConfigDescriptors`, 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.

3.3.2.6 USBDCDStallEP0

This function generates a stall condition on endpoint zero.

Parameters:
ulIndex 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.

3.3.2.7 USBDCDTerm

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

Parameters:
ulIndex 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.

3.4 HID Class Device Definitions

Defines

- Collection(usValue)
- EndCollection
- Input(usValue)
- LogicalMaximum(usValue)
- LogicalMinimum(usValue)
- Output(usValue)
- ReportCount(usValue)

- ReportSize(usValue)
- Usage(usValue)
- UsageMaximum(usValue)
- UsageMinimum(usValue)
- UsagePage(usValue)

3.4.1 Detailed Description

The macros defined in this section can be found in header file `device/usbdhid.h` and are specific to the USB Human Interface Device (HID) class. They are intended to simplify the process of creating HID report descriptors. For examples of how these macros can be used, please see the `usb_dev_mouse` and `usb_dev_keyboard` example applications.

3.4.2 Define Documentation

3.4.2.1 Collection

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

Definition:

```
#define Collection(usValue)
```

Description:

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

Parameters:

usValue is the type of Collection.

Returns:

Not a function.

3.4.2.2 EndCollection

Definition:

```
#define EndCollection
```

Description:

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

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

Returns:

Not a function.

3.4.2.3 Input

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

Definition:

```
#define Input(usValue)
```

Description:

This macro takes a value and prepares it to be placed as a 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.

Parameters:

usValue is bit mask to specify the type of a set of input report items.

Returns:

Not a function.

3.4.2.4 LogicalMaximum

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

Definition:

```
#define LogicalMaximum(usValue)
```

Description:

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

Parameters:

usValue is the Logical Maximum value.

Returns:

Not a function.

3.4.2.5 LogicalMinimum

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

Definition:

```
#define LogicalMinimum(usValue)
```

Description:

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

Parameters:

usValue is the Logical Minimum value.

Returns:

Not a function.

3.4.2.6 Output

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

Definition:

```
#define Output(usValue)
```

Description:

This macro takes a value and prepares it to be placed as a 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.

Parameters:

usValue is bit mask to specify the type of a set of output report items.

Returns:

Not a function.

3.4.2.7 ReportCount

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

Definition:

```
#define ReportCount(usValue)
```

Description:

This macro takes a value and prepares it to be placed as a Report Count entry into a HID report structure. This is number of entries of Report Size for a given item.

Parameters:

usValue is the number of items in a report item.

Returns:

Not a function.

3.4.2.8 ReportSize

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

Definition:

```
#define ReportSize(usValue)
```

Description:

This macro takes a value and prepares it to be placed as a Report Size entry into a HID report structure. This is size in bits of the entries of of a report entry. The Report Count specifies how many entries of Report Size are in a given item. These can be individual bits or bit fields.

Parameters:

usValue is the size, in bits, of items in a report item.

Returns:

Not a function.

3.4.2.9 Usage

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

Definition:

```
#define Usage(usValue)
```

Description:

This macro takes a value and prepares it to be placed as a Usage entry into a HID report structure. These are defined by the USB HID specification.

Parameters:

usValue is the Usage value.

Returns:

Not a function.

3.4.2.10 UsageMaximum

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

Definition:

```
#define UsageMaximum(usValue)
```

Description:

This macro takes a value and prepares it to be placed as a Usage Maximum entry into a HID report structure. This is the last or maximum value associated with a usage value.

Parameters:

usValue is the Usage Maximum value.

Returns:

Not a function.

3.4.2.11 UsageMinimum

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

Definition:

```
#define UsageMinimum(usValue)
```

Description:

This macro takes a value and prepares it to be placed as a Usage Minimum entry into a HID report structure. This is the first or minimum value associated with a usage value.

Parameters:

usValue is the Usage Minimum value.

Returns:

Not a function.

3.4.2.12 UsagePage

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

Definition:

```
#define UsagePage(usValue)
```

Description:

This macro takes a value and prepares it to be placed as a Usage Page entry into a HID report structure. These are defined by the USB HID specification.

Parameters:

usValue is the Usage Page value.

Returns:

Not a function.

4 Host Functions

4.1 Introduction

This chapter covers the support provide by the USB library for the USB controller in host mode. The USB library provides a layered interface to the USB host controller. There is an application interface that is provided by host class drivers and a host controller driver interface that interacts with the USB controller. The host class drivers access the lower level host controller driver that handles all accesses to the USB hardware. The host class drivers also provide an interface to the host controller driver so that information can be passed back from the host controller driver. This communication is normally provided by even callbacks or direct APIs that will be discussed in the rest of this chapter. The USB library provides the top layer host class drivers so that the application can run without any knowledge of the lower levels, or it also provides a method to include custom USB library host class drivers.

Source Code Overview

Source code and headers for the general-purpose USB functions can be found in the `host` directory of the USB library tree.

`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.
`usbhmsc.c`	The source code for the USB host Mass Storage Class driver.
`usbhmsc.h`	The header file containing Mass Storage Class definitions specific to hosts supporting this class of device.
`usbhscsi.c`	The source code for a high level SCSI interface which calls the host Mass Storage Class driver.

4.2 Host Application Interface

This section will cover the basics on how the USB library's host controller driver interface initializes, enumerates and communicates with USB devices.

Application Initialization

The USB library host stack initialization is handled in the `USBHCDInit()` function. This function should be called before accessing any other USB library functions. 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 anytime 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.

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 typically 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.

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 in the case of a dual role USB controller or an OTG controller.

4.3 Host Controller Driver

The USB library host controller driver provides an interface to the host controller's hardware register interface. This is the lowest level of the driver interface and it interacts directly with the DriverLib USB APIs. The host controller driver provides all of the functionality necessary to provide enumeration of devices as they are discovered on the USB bus. The host controller driver also provides a method to allow applications to decide what host class drivers will be used with each type of USB device. This allows an application to handle multiple types of devices and only the devices that the application cares about.

Interrupt Handling

All interrupt handling is done by the USB library host controller driver and it provides callbacks to notify the host class drivers and the application of USB events. Most of enumeration code is handled by interrupt handlers but the enumeration does require the USBHCDMain() to progress through the enumeration states.

USB Pipes

The host controller driver uses interfaces called USB pipes as the primary method of communications with USB devices. These USB pipes can be dynamically allocated or statically allocated to devices during enumeration. These USB pipes are only used inside the USB library or other host class drivers. 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. Most access to the USB pipes occurs through USBHCDPipeWrite() and USBHCDPipeRead(). These are used to read or write to endpoints on USB devices on endpoints other than zero. The host controller driver does not use USB pipes for communications over endpoint zero and instead uses the USBHCDControlTransfer() function.

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 some extra control transactions through endpoint zero.

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. The interrupt code for enumeration is contained in the HostEnumHandler(). The enumeration process also requires

periodically calling the USBHCDMain() function. All accesses through endpoint zero are through the USBHCDControlTransfer() interface. 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 it will call the open function for that host class driver. If no host class driver is found the host controller driver will ignores the device and there will be no notification to the application. The host controller driver or the host class driver can provide callbacks to the application for enumeration events. The host class drivers are responsible for configuring the USB pipes based on the type of device that is discovered. There is no interaction needed from the application during enumeration except to provide a callback to notify the application that a device is present or has been removed.

4.4 Host Class Drivers

The host class drivers provide access to applications for the various USB device types. The only host class driver example that is currently provided is the mass storage class driver. The application must provide a list of supported host class drivers by calling the USBHCDRegisterDrivers() function. The next sections will cover the MSC class driver and how to implement a custom host class driver.

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. The mass storage host class driver provided with the USB library provides a driver interface for the library and a top layer interface for an application's file system.

Application Interface

The host class driver for mass storage class 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. The functions that provide the block read/write access are USBHMSCBlockRead() and USBHMSCBlockWrite(). These function will perform block operations at the size specified by the flash drive. Some 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.

Driver Interface

The mass storage class driver also provides an interface to the USB library host controller enumeration code 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 USBMSCOpen() and USBMSCClose() provide the interface for the host controller's enumeration code to call when a mass storage class device is

detected or removed. 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.

Example: Adding Mass Storage Class Driver

```
const tUSBHostClassDriver * const g_ppUSBHostClassDrivers[] =
{
    &g_USBHostMSCClassDriver
};

//
// Register the host class drivers.
//
USBHCDRegisterDrivers(g_ppUSBHostClassDrivers, 1);
```

Application Setup

The application must perform some configuration and be prepared to handle driver callbacks in order to properly interact with the mass storage class driver. The application must first call USBMSCDriveOpen() early in the application to allow the mass storage class driver to notify the application of mass storage class events.

Example: Mass Storage Class Initialization

```
//
// 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 an MSC_EVENT_OPEN 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 USBMSCDriveReady() function should be called to determine when the flash drive is ready for read/write operations. When the device has been removed an MSC_EVENT_CLOSE event will occur. When shutting down, the application should call USBMSCDriveClose() 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 flash drive is ready, the application can use the USBMSCBlockRead() and USBMSCBlockWrite() 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. Since most mass storage class device adhere to the SCSI protocol for these block based calls, the next section covers what SCSI calls are supported.

Example: Block Read/Write Calls

```
//
// Read 1 block starting at logical block 0.
//
USBHMSCBlockRead(ulMSCDevice, 0, pucBuffer, 1);
```

```
//
// Write 2 blocks starting at logical block 500.
//
USBHMSCBlockWrite(ulMSCDevice, 500, pucBuffer, 2);
```

SCSI Functions

The SCSI functions are used to by the mass storage class driver to communicate with the flash drives via USB pipes provided by the host controller driver. The only types of devices that are supported are mass storage class devices that use 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.

Implementing Custom Host Class Driver

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 mass storage class driver, 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. In this example, the host class driver is providing the USBGenericIntHandler() interrupt routine that will be called at interrupt time.

Example: Custom Host Class Driver Interface

```
tUSBClassDriver USBGenericClassDriver =
{
    USB_CLASS_SOMECLASS,
    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 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.

4.5 Host Mass Storage Programming Example

The following programming example demonstrates the initial configuration and some of the basic calls that are made to the mass storage class driver. This includes the memory allocations needed as well as the global class driver definitions.

Example: Basic Configuration as Host

```
//
// The size of the host controller's memory pool in bytes.
//
#define HCD_MEMORY_SIZE         128

//
// The memory pool to provide to the Host controller driver.
//
unsigned char g_pHCDPool[HCD_MEMORY_SIZE];

//
// The instance data for the MSC driver.
//
unsigned long g_ulMSCInstance = 0;

//
// The global that holds all of the host drivers in use in the application.
// In this case, only the MSC class is loaded.
//
const tUSBHostClassDriver * const g_ppUSBHostClassDrivers[] =
{
    &g_USBHostMSCClassDriver
};

    ...

    //
    // Enable Clocking to the USB controller.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_USB0);

    //
    // Enable the peripherals used by this example.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH);

    //
    // Set the USB pins to be controlled by the USB controller.
    //
    GPIOPinTypeUSBDigital(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    GPIOPinTypeUSBDigital(GPIO_PORTH_BASE, GPIO_PIN_3 | GPIO_PIN_4);

    //
    // Turn on USB Phy clock.
    //
    SysCtlUSBPLLEnable();

    //
```

```
// Register the host class drivers.
//
USBHCDRegisterDrivers(g_ppUSBHostClassDrivers, 1);

//
// Open an instance of the mass storage class driver.
//
g_ulMSCInstance = USBHMSCDriveOpen(0, MSCCallback);

//
// Initialize the host controller.
//
USBHCDInit(USB0_BASE, g_pHCDPool, HCD_MEMORY_SIZE);

...

//
// Wait for the drive to become ready.
//
while(USBHMSCDriveReady(g_ulMSCInstance))
{
    //
    // System level delay call should be here to give the device time to
    // become ready.
    //
    SysCtlDelay(SysCtlClockGet() / 100);
}

...

//
// Block Read example.
//
USBHMSCBlockRead(g_ulMSCInstance, ulLBA, pucData, 1);

...

//
// Block Write example.
//
USBHMSCBlockWrite(g_ulMSCInstance, ulLBA, pucData, 1);

...
```

4.6 Definitions

Data Structures

- tUSBHostClassDriver
- tUSBHostDevice

Functions

- void USB0HostIntHandler (void)
- void USBHCDInit (unsigned long ulIndex, void *pData, unsigned long ulSize)
- void USBHCDMain (void)

- unsigned long USBHCDPipeAlloc (unsigned long ulIndex, unsigned long ulEndpointType, unsigned long ulDevAddr, tHCDPipeCallback pCallback)
- unsigned long USBHCDPipeConfig (unsigned long ulPipe, unsigned long ulMaxPayload, unsigned long ulTargetEndpoint)
- void USBHCDPipeFree (unsigned long ulPipe)
- unsigned long USBHCDPipeRead (unsigned long ulPipe, unsigned char *pData, unsigned long ulSize)
- unsigned long USBHCDPipeStatus (unsigned long ulPipe)
- unsigned long USBHCDPipeWrite (unsigned long ulPipe, unsigned char *pData, unsigned long ulSize)
- void USBHCDRegisterDrivers (unsigned long ulIndex, const tUSBHostClassDriver *const *ppHClassDrvrs, unsigned long ulNumDrivers)
- void USBHCDReset (unsigned long ulIndex)
- void USBHCDResume (unsigned long ulIndex)
- void USBHCDSetConfig (unsigned long ulIndex, unsigned long ulDevice, unsigned long ulConfiguration)
- void USBHCDSuspend (unsigned long ulIndex)
- void USBHCDTerm (unsigned long ulIndex)
- long USBHMSCBlockRead (unsigned long ulInstance, unsigned long ulLBA, unsigned char *pucData, unsigned long ulNumBlocks)
- long USBHMSCBlockWrite (unsigned long ulInstance, unsigned long ulLBA, unsigned char *pucData, unsigned long ulNumBlocks)
- void USBHMSCDriveClose (unsigned long ulInstance)
- unsigned long USBHMSCDriveOpen (unsigned long ulDrive, tUSBHMSCCallback pfnCallback)
- long USBHMSCDriveReady (unsigned long ulInstance)

Variables

- tUSBHostClassDriver g_USBHostMSCClassDriver

4.6.1 Data Structure Documentation

4.6.1.1 tUSBHostClassDriver

Definition:

```
typedef struct
{
    unsigned long ulInterfaceClass;
    void * (*pfnOpen)(tUSBHostDevice *pDevice);
    void (*pfnClose)(void *pvInstance);
    void (*pfnIntHandler)(void *pvInstance);
}
tUSBHostClassDriver
```

Members:

ulInterfaceClass The interface class that this device driver supports.

pfnOpen The function called when this class of device has been detected.

pfnClose The function called when the device, originally opened with a call to the pfnOpen function, is disconnected.

pfnIntHandler This is the interrupt handler that will be called when an endpoint associated with this device instance generates an interrupt.

Description:

This is a USB host driver instance, it is parsed to find a “driver” for a class that is enumerated.

4.6.1.2 tUSBHostDevice

Definition:

```
typedef struct
{
    unsigned long ulAddress;
    unsigned long ulInterface;
    tDeviceDescriptor DeviceDescriptor;
    tConfigDescriptor *pConfigDescriptor;
    unsigned long ulConfigDescriptorSize;
}
tUSBHostDevice
```

Members:

ulAddress The current device address for this device.

ulInterface The current interface for this device.

DeviceDescriptor A pointer to the device descriptor for this device.

pConfigDescriptor A pointer to the configuration descriptor for this device.

ulConfigDescriptorSize The size of the buffer allocated to pConfigDescriptor.

Description:

This is the structure that holds all of the information for devices that are enumerated in the system.

4.6.2 Function Documentation

4.6.2.1 USB0HostIntHandler

The USB host mode interrupt handler for controller index 0.

Prototype:

```
void
USB0HostIntHandler(void)
```

Description:

This the main USB interrupt handler entry point. This handler will branch the interrupt off to the appropriate handlers depending on the current status of the USB controller.

Returns:

None.

4.6.2.2 USBHCDInit

This function is used to initialize the HCD code.

Prototype:

```
void
USBHCDInit(unsigned long ulIndex,
           void *pvPool,
           unsigned long ulPoolSize)
```

Parameters:

ulIndex specifies which USB controller to use.

pvPool is a pointer to the data to use as a memory pool for this controller.

ulPoolSize is the size in bytes of the buffer passed in as pvPool.

Description:

This function will perform all the necessary operations to allow the USB host controller to begin enumeration and communication with a device. This function should typically be called once at the start of an application before any other calls are made to the host controller.

Returns:

None.

4.6.2.3 USBHCDMain

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

Prototype:

```
void
USBHCDMain(void)
```

Description:

This function is the main routine for the Host Controller Driver, and must be called periodically by the main application. 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 at interrupt time to prevent this loop from gating USB operations.

Returns:

None.

4.6.2.4 USBHCDPipeAlloc

This function is used to allocate a USB HCD pipe.

Prototype:

```
unsigned long
USBHCDPipeAlloc(unsigned long ulIndex,
                unsigned long ulEndpointType,
                unsigned long ulDevAddr,
                tHCDPipeCallback pCallback)
```

Parameters:

ulIndex specifies which USB controller to use.

ulEndpointType is the type of endpoint that this pipe will be communicating with.

ulDevAddr is the device address to use for this endpoint.

pCallback is the function that will be called when events occur on this USB Pipe.

Description:

Since there are a limited number of USB HCD pipes that can be used in the host controller, this function is used to temporarily or permanently acquire one of the endpoints. It also provides a method to register a callback for status changes on this endpoint. If no callbacks are desired then the *pCallback* function should be set to 0.

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 HCDPipeFree() when releasing the resource.

4.6.2.5 USBHCDPipeConfig

This function is used to configures a USB HCD pipe.

Prototype:

```
unsigned long
USBHCDPipeConfig(unsigned long ulPipe,
                 unsigned long ulMaxPayload,
                 unsigned long ulTargetEndpoint)
```

Description:

This should be called after allocating a USB HCD pipe with a call to USBHCDPipeAlloc(). It is used to set some of the configuration associated with an endpoint like the max payload and target endpoint.

Parameters:

ulPipe is the allocated endpoint to modify.

ulMaxPayload is maxium data that can be handled per transaction.

ulTargetEndpoint is the target endpoint on the device to communicate with.

Returns:

This function returns 0.

4.6.2.6 USBHCDPipeFree

This function is used to release a USB HCD pipe.

Prototype:

```
void
USBHCDPipeFree(unsigned long ulPipe)
```

Parameters:

ulPipe is a unique number returned from HCDPipeAlloc() function.

Description:
This function is used to release a USB HCD Pipe 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.

4.6.2.7 USBHCDPipeRead

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

Prototype:
```
unsigned long
USBHCDPipeRead(unsigned long ulPipe,
               unsigned char *pucData,
               unsigned long ulSize)
```

Parameters:
ulPipe is the USB pipe to put data into.
pucData is a pointer to the data to send.
ulSize is the amount of data to send.

Description:
This function will not block and will only read as much data as requested or as much data is currently available from the USB pipe. The caller should have registered a callback with the USBHCDPipeAlloc() call in order to be informed when the data has been received. The value returned by this function can be less than the *ulSize* requested if the USB pipe has less data available than this request is making.

Returns:
This function returns the number of bytes that were returned in the pData buffer.

4.6.2.8 USBHCDPipeStatus

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

Prototype:
```
unsigned long
USBHCDPipeStatus(unsigned long ulPipe)
```

Description:
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:
ulPipe 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.

4.6.2.9 USBHCDPipeWrite

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

Prototype:

```
unsigned long
USBHCDPipeWrite(unsigned long ulPipe,
                unsigned char *pucData,
                unsigned long ulSize)
```

Parameters:

ulPipe is the USB pipe to put data into.

pucData is a pointer to the data to send.

ulSize is the amount of data to send.

Description:

This function will not block and will only send as much data as will fit into the current USB pipes 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 *ulSize* 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.

4.6.2.10 USBHCDRegisterDrivers

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

Prototype:

```
void
USBHCDRegisterDrivers(unsigned long ulIndex,
                      const tUSBHostClassDriver *const
*ppHClassDrvrs,
                      unsigned long ulNumDrivers)
```

Parameters:

ulIndex specifies which USB controller to use.

ppHClassDrvrs is an array of host class drivers that are supported on this controller.

ulNumDrivers is the number of entries in the *pHostClassDrivers* array.

Description:

This function will set the host classes supported by the host controller specified by the *ulIndex* parameter. This function should be called before enabling the host controller driver with the USBHCDInit() function.

Returns:

None.

4.6.2.11 USBHCDReset

This function generates reset signaling on the USB bus.

Prototype:

```
void
USBHCDReset(unsigned long ulIndex)
```

Parameters:

ulIndex specifies which USB controller to use.

Description:

This function handles sending out reset signaling on the USB bus. After returning from this function, any attached device on the USB bus should have returned to it's reset state.

Returns:

None.

4.6.2.12 USBHCDResume

This function will generate resume signaling on the USB bus.

Prototype:

```
void
USBHCDResume(unsigned long ulIndex)
```

Parameters:

ulIndex specifies which USB controller to use.

Description:

This function is used to generate resume signaling on the USB bus in order to cause with USB devices to leave their suspended state. This call should not be made unless a preceding cal to HCDSuspend() has been made.

Returns:

None.

4.6.2.13 USBHCDSetConfig

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

Prototype:

```
void
USBHCDSetConfig(unsigned long ulIndex,
                unsigned long ulDevice,
                unsigned long ulConfiguration)
```

Parameters:

ulIndex specifies which USB controller to use.

ulDevice is the USB device for this function.

ulConfiguration is one of the devices valid configurations.

Description:
This function is used to set the current device configuration for a USB device address. The *ulConfiguration* 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.

4.6.2.14 USBHCDSuspend

This function will generate suspend signaling on the USB bus.

Prototype:
```
void
USBHCDSuspend(unsigned long ulIndex)
```

Parameters:
ulIndex specifies which USB controller to use.

Description:
This function is used to generate suspend signaling on the USB bus. In order to leave the suspended state, the application should call HCDResume().

Returns:
None.

4.6.2.15 USBHCDTerm

This function is used to terminate the HCD code.

Prototype:
```
void
USBHCDTerm(unsigned long ulIndex)
```

Parameters:
ulIndex specifies which USB controller to use.

Description:
This function will clean up the USB host controller and disable it in preparation for shutdown or a switch to USB device mode. Once this call is made, *USBHCDInit()* may be called to reinitialize the controller and prepare for host mode operation.

Returns:
None.

4.6.2.16 USBHMSCBlockRead

This function performs a block read to an MSC device.

Prototype:

```
long
USBHMSCBlockRead(unsigned long ulInstance,
                 unsigned long ulLBA,
                 unsigned char *pucData,
                 unsigned long ulNumBlocks)
```

Parameters:

ulInstance is the device instance to use for this read.

ulLBA is the logical block address to read on the device.

pucData is a pointer to the returned data buffer.

ulNumBlocks is the number of blocks to read from the device.

Description:

This function will perform a block sized read from the device associated with the *ulInstance* parameter. The *ulLBA* parameter specifies the logical block address to read on the device. This function will only perform *ulNumBlocks* block sized reads. In most cases this is a read of 512 bytes of data. The $*$*pucData* buffer should be at least *ulNumBlocks* $*$ 512 bytes in size.

Returns:

The function returns zero for success and any negative value indicates a failure.

4.6.2.17 USBHMSCBlockWrite

This function performs a block write to an MSC device.

Prototype:

```
long
USBHMSCBlockWrite(unsigned long ulInstance,
                  unsigned long ulLBA,
                  unsigned char *pucData,
                  unsigned long ulNumBlocks)
```

Parameters:

ulInstance is the device instance to use for this write.

ulLBA is the logical block address to write on the device.

pucData is a pointer to the data to write out.

ulNumBlocks is the number of blocks to write to the device.

Description:

This function will perform a block sized write to the device associated with the *ulInstance* parameter. The *ulLBA* parameter specifies the logical block address to write on the device. This function will only perform *ulNumBlocks* block sized writes. In most cases this is a write of 512 bytes of data. The $*$*pucData* buffer should contain at least *ulNumBlocks* $*$ 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.

4.6.2.18 USBHMSCDriveClose

This function should be called to release a drive instance.

Prototype:

```
void
USBHMSCDriveClose(unsigned long ulInstance)
```

Parameters:

ulInstance is the device instance that is to be released.

Description:

This function is called when an MSC drive is to be released in preparation for shutdown or a switch to USB device mode, for example. Following this call, the drive is available for other clients who may open it again using a call to /e USBMSCDriveOpen().

Returns:

None.

4.6.2.19 USBHMSCDriveOpen

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

Prototype:

```
unsigned long
USBHMSCDriveOpen(unsigned long ulDrive,
                 tUSBHMSCCallback pfnCallback)
```

Parameters:

ulDrive is the drive number to open.

pfnCallback is the driver callback for any mass storage events.

Description:

This function is called to open an instance of a mass storage device. It should be called before any devices are connected to allow for proper notification of drive connection and disconnection. The /e ulDrive 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 /e 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.

4.6.2.20 USBHMSCDriveReady

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

Prototype:

```
long
USBHMSCDriveReady(unsigned long ulInstance)
```

Parameters:

ulInstance is the device instance to use for this read.

Description:

This function checks if the current device has become ready to be accessed. It uses the *ulInstance* parameter to determine which device to access and will return zero when the device is ready. Any negative return code indicates that the device was not ready.

Returns:

This function will return zero if the device is ready and it will return a negative value if the device is not ready or if an error occurred.

4.6.3 Variable Documentation

4.6.3.1 g_USBHostMSCClassDriver

Definition:

```
tUSBHostClassDriver g_USBHostMSCClassDriver
```

Description:

The USB MSC bulk only host class driver structure.

Company Information

Founded in 2004, Luminary Micro, Inc. designs, markets, and sells ARM Cortex-M3-based microcontrollers (MCUs). Austin, Texas-based Luminary Micro is the lead partner for the Cortex-M3 processor, delivering the world's first silicon implementation of the Cortex-M3 processor. Luminary Micro's introduction of the Stellaris family of products provides 32-bit performance for the same price as current 8- and 16-bit microcontroller designs. With entry-level pricing at $1.00 for an ARM technology-based MCU, Luminary Micro's Stellaris product line allows for standardization that eliminates future architectural upgrades or software tool changes.

Luminary Micro, Inc.
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com
sales@luminarymicro.com

Support Information

For support on Luminary Micro products, contact:

support@luminarymicro.com
+1-512-279-8800, ext 3