



MPLAB® Harmony Help - USB Libraries

MPLAB Harmony Integrated Software Framework v1.10

USB Libraries Help

This section provides descriptions of the USB libraries that are available in MPLAB Harmony.

USB Device Library

This section provides information on the USB Device libraries that are available in MPLAB Harmony.

USB Device Library - Getting Started

This section provides information for getting started with the USB Device Library.

Introduction

Provides an introduction to the MPLAB Harmony USB Device Library

Description

The MPLAB Harmony USB Device Library (referred to as the USB Device Library) provides embedded application developers with a framework to design and develop a wide variety of USB Devices. A choice of Full Speed only or Full Speed and Hi-Speed USB operations are available, depending on the selected PIC32 microcontroller. The USB Device Library facilitates development of standard USB devices through function drivers that implement standard USB Device class specification. Vendor USB devices can be implemented via USB Device Layer Endpoint functions. The USB Device Library is modular, thus allowing application developers to readily design composite USB devices. The USB Device Library is a part of the MPLAB Harmony installation and is accompanied by demonstration applications that highlight library usage. These demonstrations can also be modified or updated to build custom applications. The USB Device Library also features the following:

- Support for different USB device classes (CDC, Audio, HID, MSD, and Vendor)
- Supports multiple instance of the same class in a composite device
- Supports multiple configurations at different speeds
- Supports Full-Speed and High-Speed operation
- Supports multiple USB peripherals (allows multiple device stacks)
- Modular and Layered architecture
- Supports deferred control transfer responses
- Completely non-blocking
- Supports both polled and interrupt operation
- Works readily in an RTOS application

This document serves as a getting started guide and provides information on the following:

- USB Device Library architecture
- USB Device Library - application interaction
- Creating your own USB device



Note: It is assumed that the reader is familiar with the USB 2.0 specification (available at www.usbif.org). While the document, for the sake completeness, does cover certain aspects of the USB 2.0 protocol, it is recommended that the reader refer to the specification for a complete description of USB operation.

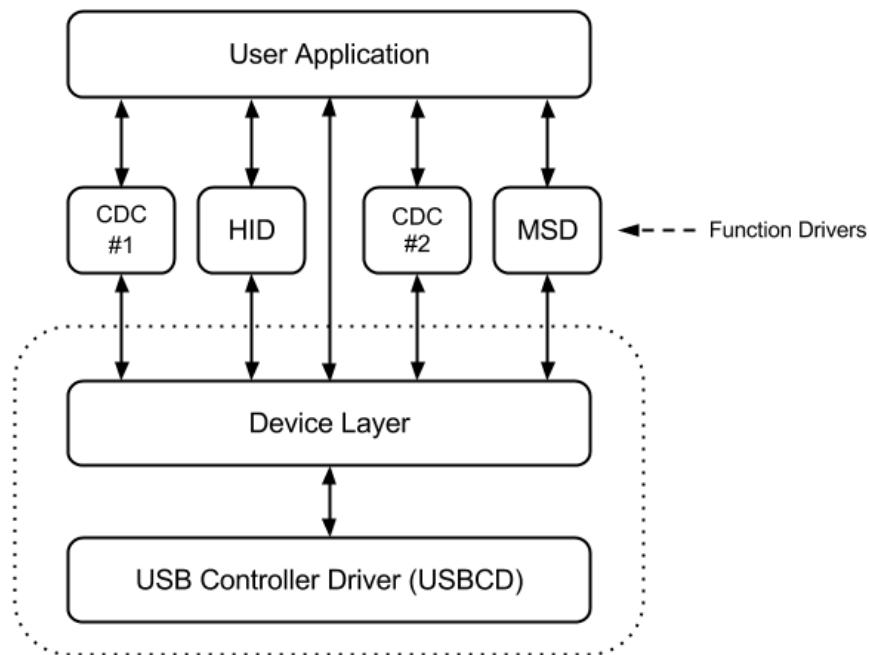
USB Device Library Architecture

Describes the USB Device Library Architecture.

Description

The USB Device Library features a modular and layered architecture, as illustrated in the following figure.

USB Device Library Architecture



As seen in the figure, the USB Device Library consists of the following three major components.

USB Controller Driver (USBCD)

The USBCD manages the state of the USB peripheral and provides the Device Layer with structured data access methods to the USB. It also provides the Device layer with USB events. The USBCD is a MPLAB Harmony driver and uses the MPLAB Harmony framework components (the USB peripheral Library, the Interrupt System Service) for its operation. It supports only one client per instance of the USB Peripheral. This client would typically be the Device Layer. In case of multiple USB peripherals, the USBCD can manage multiple USB peripherals, each being accessed by one client. The driver is accessed exclusively by the Device Layer in the USB Device Layer Architecture. It is initialized by the Device Layer (when the Device Layer is initialized) and in case of polling operation, its Tasks routine is called by the Device Layer. The USBCD provides functions to:

- Enable, disable and stall endpoints
- Schedule USB transfers
- Attach or detach the device
- Control resume signalling

The USB Controller Driver can be configured for Polled or Interrupt mode. Configuring the driver for Interrupt mode configures the Device Stack for Device mode operation. Configuring the driver for Polled mode configures the USB Device Stack for Polled mode operation. An USB device can be designed using either the polled or interrupt mode. However, it is recommended that the USB Controller Driver (and therefore the USB Device Stack) be configured for Interrupt mode. This reduces the impact of other application components on the operation of the Device Stack. Configuring for Interrupt mode also ensure timely stack event generation and stack response to bus events on the USB.

Device Layer

The Device Layer responds to the enumeration requests issued by the USB Host. It has exclusive access to an instance of the USBCD and the control endpoint (Endpoint 0). When the Host issues a class specific control transfer request, the Device Layer will analyze the setup packet of the control transfer and will route the control transfer to the appropriate function driver. The Device Layer must be initialized with the following data:

- Master Descriptor Table - This is a table of all the configuration descriptors and string descriptors.
- Function Driver Registration Table - This table contains information about the function drivers in the application
- USBCD initialization information - This specifies the USB peripheral interrupt, the USB Peripheral instance and Sleep mode operation options

The Device Layer initializes all function drivers that are registered with it when it receives a Set Configuration (for a supported configuration) from the Host. It deinitializes the function drivers when a USB reset event occurs. It initializes the USBCD, opens it and registers a event handler to receive USB events. The Device Layer can also be opened by the application (the application becomes a client to the Device Layer). The application can then receive bus and device events and respond to control transfer requests. The Device Layer provides events to the application such as device configured or device reset. Some of these events are notification-only events, while other events require the application to take action.

Function Drivers

The Function Drivers implements various USB device classes as per the class specification. The USB Device Library architecture can support multiple instances of a function driver. An example would be a USB CDC device that emulates two serial ports. Function drivers provide an abstracted and an easy to use interface to the application. The application must register an event handler with the function driver to receive function driver events and must respond to some of these events with control transfer read/write functions. Function drivers access the bus

through the Device Layer.

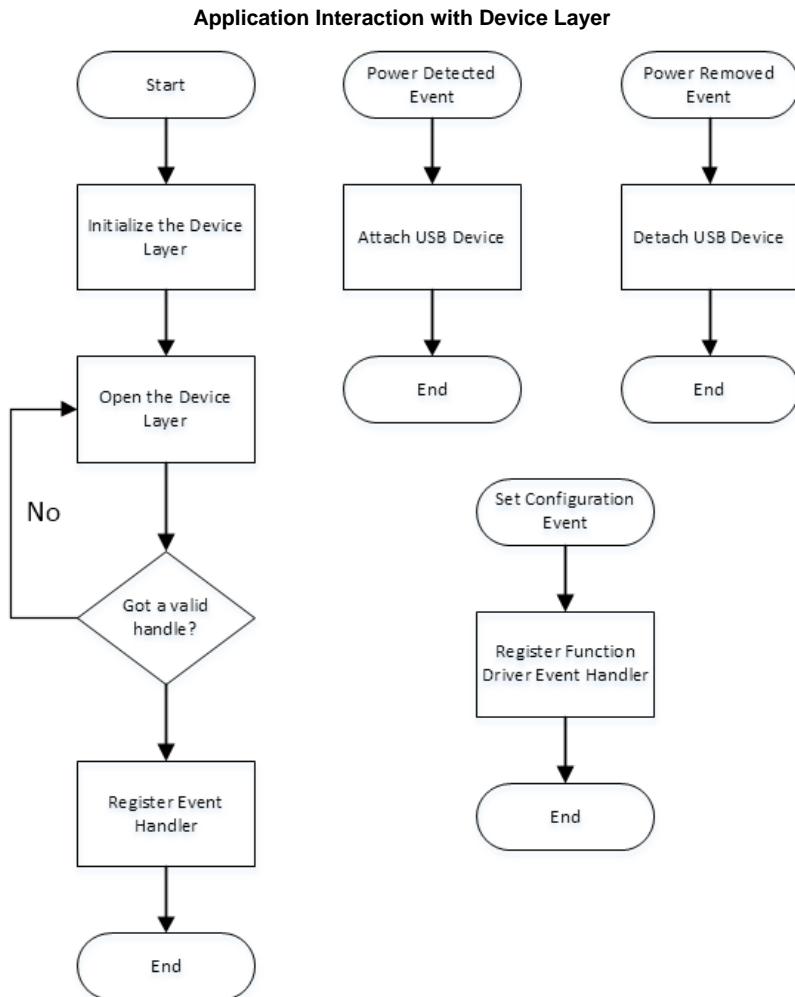
USB Device Library - Application Interaction

Describes how the application must interact with the USB Device Stack.

Description

 **Note:** Additional information on program, data, and stack memory requirements, and a list of USB application configurations is available in the USB Demonstrations section.

The following figure highlights the steps that the application must follow to use the USB Device Library.

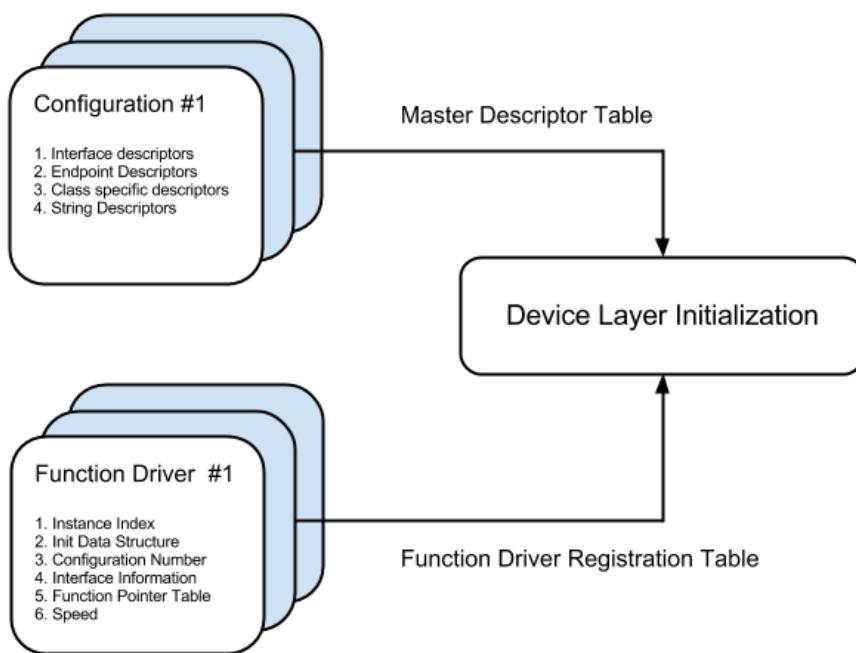


The application must first initialize the Device Layer. As a part of the Device Layer initialization process, the Device Layer initialization structure must be defined which in turn requires the following data structures to be designed

- The master descriptor table
- The function driver registration table

The following figure shows a pictorial representation of the data that forms the Device Layer initialization structure. Additional information on Device Layer initialization is available in the Device Layer Help File.

Device Layer Initialization



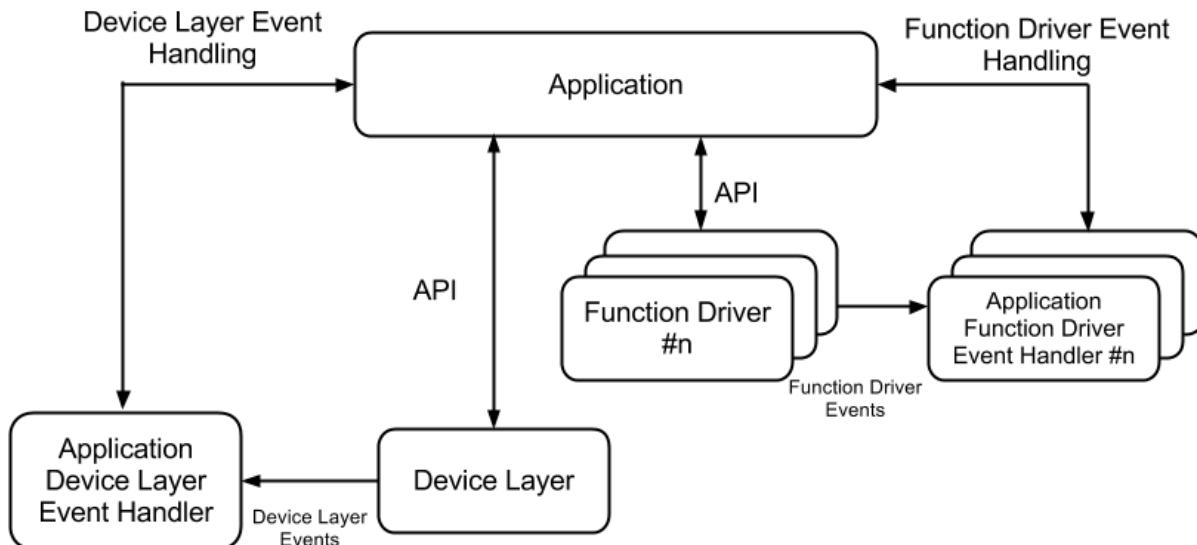
After successful initialization of the Device layer, the application can open the Device layer and register a Device layer event handler. The Device layer event handler receives device level events such as device configured, device deconfigured, device reset and device suspended. The device configured event and deconfigured event are important. The application can use the device deconfigured event to reinitialize its internal state machine. When the application receives a device configured event, it must register event handlers for each function driver that is relevant to the configuration that was set. The function driver event handler registration must be done in the device configured event context because the Device layer acknowledges the set configuration request from the host when it exits the device configured event handler context. The application at this point should be ready to respond to function driver events.

Note: Not registering the function driver event handler in the Device layer configured event could cause the device to not respond to the host requests and therefore, be non-compliant.

Once configured, the device is now ready to serve its intended function on the USB. The application interacts with the Device layer and function drivers through API function and event handlers. The application must be aware of function driver events which require application response. For example, the USB_DEVICE_CDC_EVENT_SET_LINE_CODING event from the USB CDC Function Driver requires the application to respond with a USB_DEVICE_ControlRead function. This function provides the buffer to receive the line coding parameters that the Host sends in the data stage of the Set Line Coding control transfer.

The following figure shows the application interaction with Device layer and function driver after the device has been configured.

Application - Device Layer Interaction after device configuration



In the previous figure, the application should have registered the Device layer event handler before attaching the device on the bus. It should have

registered the function driver event handler before exiting the device configured - Device layer event. The application will then receive function driver instance specific events via the function driver event handlers.

Deferring Control Transfer Responses

Class-specific control transfer related function driver events require the application to complete the data stage and/or the status of the control transfer. The application does this by using the Device Layer Control Transfer API to complete the Control Read/Write transfers. The application may typically be able to complete required data processing, and to continue (or end) the control transfer within the function driver event handler context. However, there could be cases where the required control transfer data processing may require hardware access or extended computation. Performing extended processing or waiting for external hardware within the function driver event handler context is not recommended as the USB 2.0 Specification places restrictions on the control transfer response time.

In cases where the application is not ready to respond to control transfer requests within the function driver event handler context, the USB Device Library provides the option of deferring the response to the control transfer event. The application can respond to the control transfer request after exiting the handler function. The application must still observe the USB 2.0 Specification control transfer timing requirements while responding to the control transfer. Deferring the response in such a manner provides the application with flexibility to analyze the control transfer without degrading the performance of the device on the USB.

Creating Your Own USB Device

Describes how to create a USB device with the MPLAB Harmony USB Device Library.

Description

The first step in creating a USB device is identifying whether the desired device function fits into any of the standard USB device class functions. Using standard USB classes may be advantageous as major operating systems feature Host driver support for standard USB devices. However, the application may not want to tolerate the overhead associated with standard USB device class protocols, in which case, a Vendor USB device can be implemented. A Vendor USB device can be implemented by using the USB Device Layer Endpoint functions; however, these devices will require custom USB host drivers for their operation. Having identified the device class to be used, the following approaches are available for developing a USB device by using the USB Device Library.

Use the Available Library Demonstration Applications

The USB Device Library release package contains a set of demonstration applications that are representative of common USB devices. These can be modified easily to include application specific initialization and application logic. The application logic must be non-blocking and could be implemented as a state machine. Note that the function names and file names referred to in the following section are those used in the USB Device Library demonstration applications.

- The application specific initialization can be called in the APP_Initialize function (in the `app.c` file). The APP_Initialize function is called from the SYS_Initialize function, which in turn is called when the device comes out of Power-on Reset (POR).
- The application logic is implemented as a state machine in the APP_Tasks function (in the `app.c` file). The application logic can interact with the function driver and the Device layer by using available API calls.
- The application logic can track device events by processing the events in the application USB device event handler function (APP_USBDeviceEventHandler function in `app.c`).

Build a USB Device Library Application Using the MPLAB Harmony Configurator (MHC)

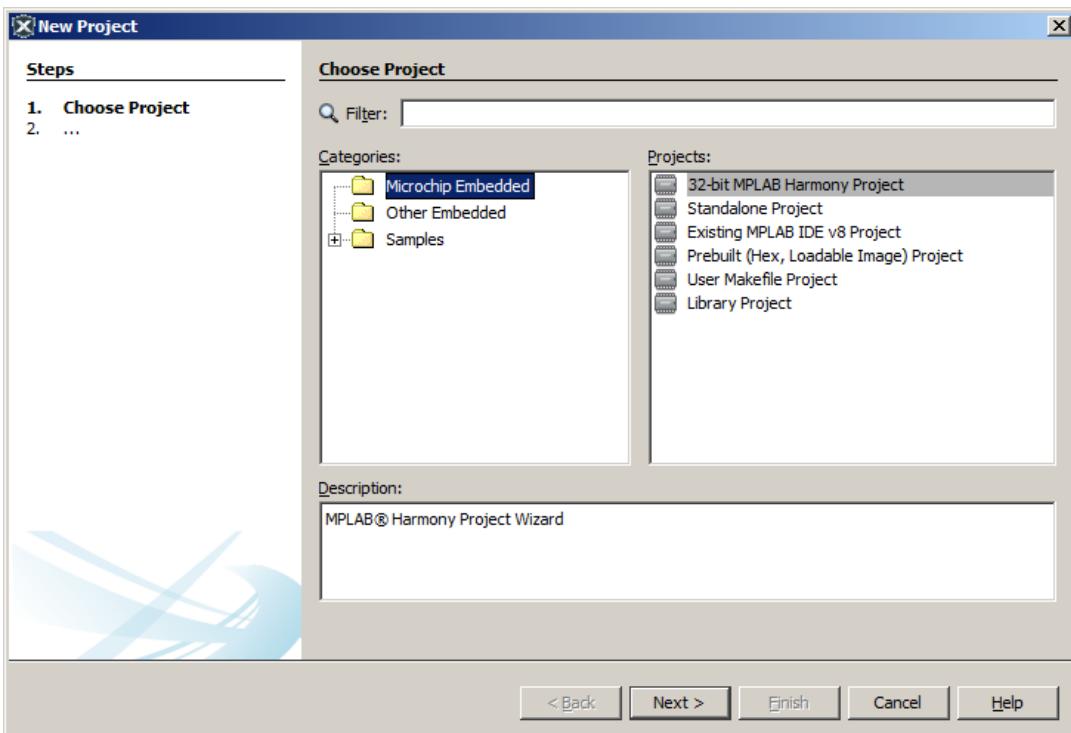
In a case where the available demonstration applications do not meet the end application requirements, a USB device can be created by building a USB Device Library application from scratch. It is recommended to use the Microchip Harmony Configurator (MHC) to generate a USB Device project. Using the MHC ensures that the correct and required MPLAB Harmony framework files are included in the project. The following section outlines the steps to follow to generate the project using the MHC.

Before You Begin

The following process assumes that the MHC plug-in, `com-microchip-mplab-module-mhc.nbm`, which is located in `<install-dir>/utilities/mhc`, is installed in MPLAB X IDE. If the MHC plug-in is not already installed, refer to [Installing a Plug-in Module](#) for information.

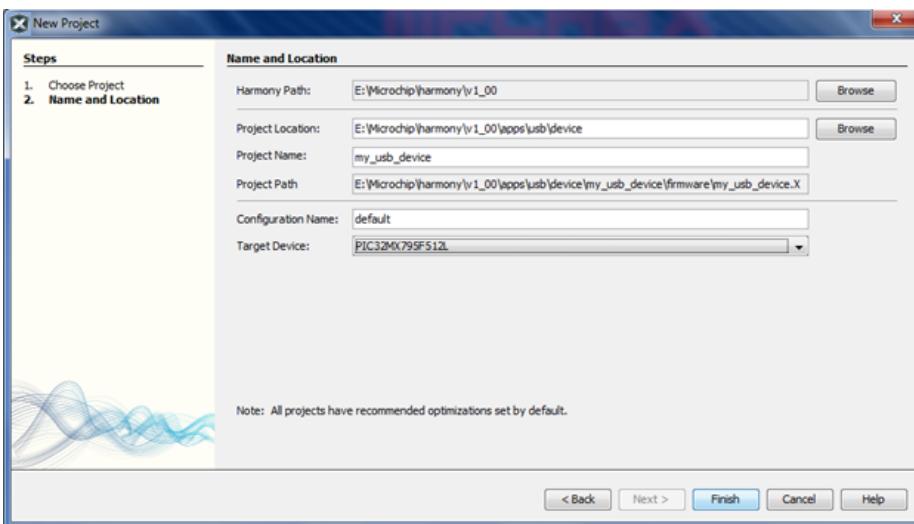
Step 1: Create a New MPLAB X IDE Project

In MPLAB X IDE, select **File > New Project** to open the New Project dialog and in Categories: select **Microchip Embedded** and in Projects: select **MPLAB Harmony Project**, and then click **Next**.



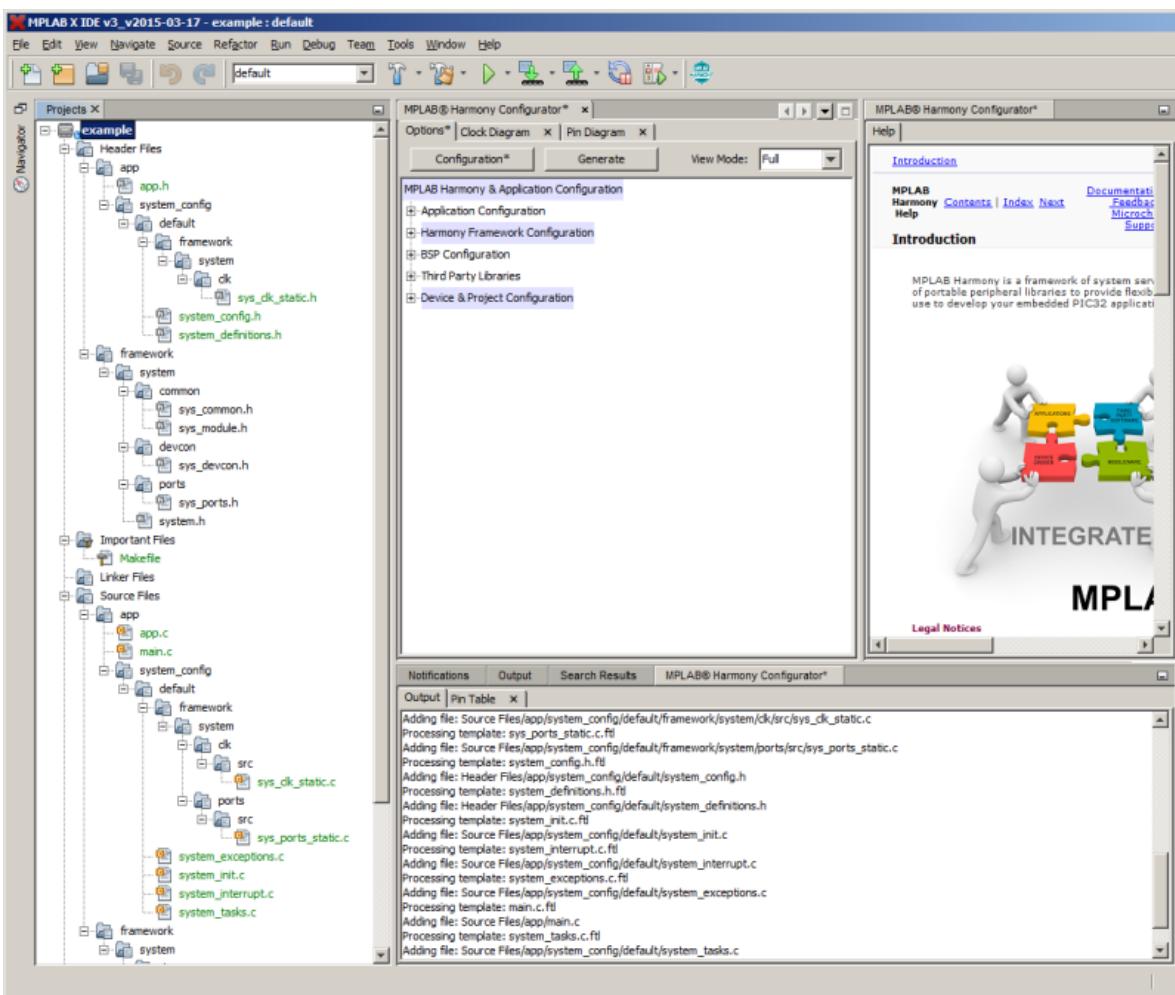
Step 2: Specify Project Name, Location and PIC32 Device Type

In the New Project dialog, specify the Project Location, Project Name, and the Target Device, and then click **Finish**.

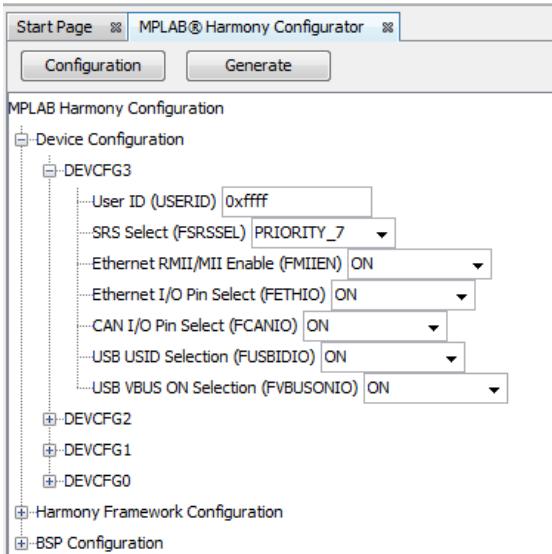


Step 3: Start the MPLAB Harmony Configurator

Open the MHC by selecting *Tools > Embedded > MPLAB Harmony Configurator*. The initial view is shown in the following figure.

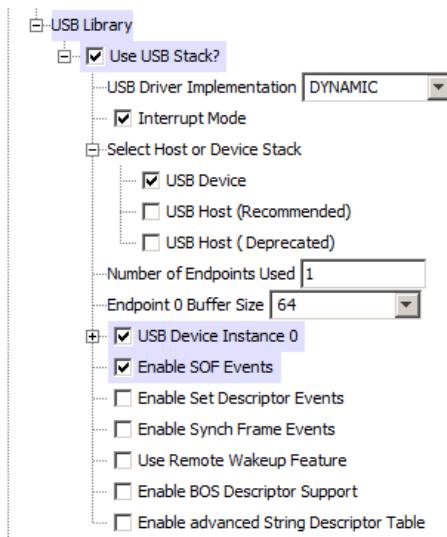


Step 4: Select Device Configuration and configure the fields as required



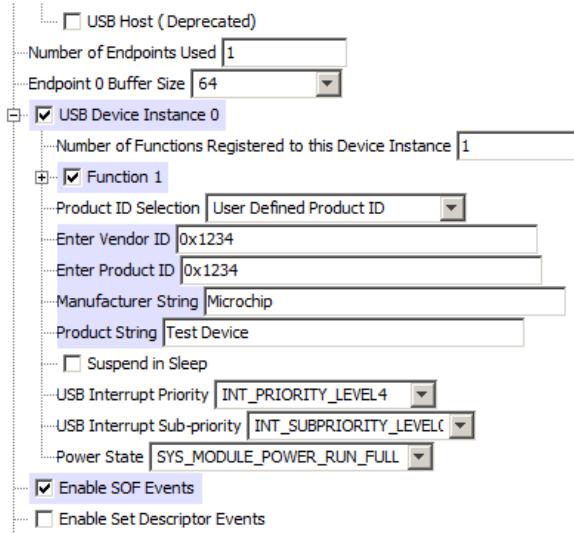
Step 5: MPLAB Harmony Framework Configuration

Expand the MPLAB Harmony Framework Configuration and select **USB Library**, and then select **Use USB Stack**. Doing this will also automatically include other MPLAB Harmony Framework components, that are required by the USB Stack into the project. Select **USB Device** and configure the different parameters as desired.



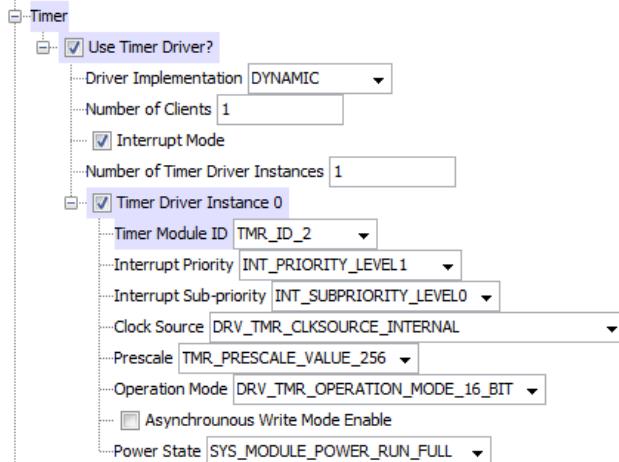
Step 6: MPLAB Harmony Framework Configuration

Expand USB Device Instance 0 and configure parameters as desired.



Step 7: MPLAB Harmony Framework Configuration

For a PIC32MZ device, expand *MPLAB Harmony Framework Configuration >Drivers >Timer* and select the Timer Module ID as TMR_ID_2.



Step 8: Configure Interrupt Priorities

The interrupt priorities of the following modules should be configured as per application needs.

For PIC32MX devices:

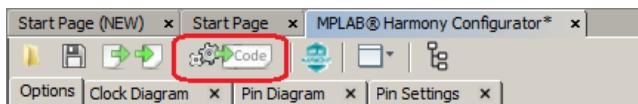
- USB Device Layer

For PIC32MZ devices:

- USB Device Layer
- Timer Driver

Step 9: Generate code

Click **Generate** to generate the code for this configuration.



Step 10: Add application specific information

The following components should be manually added to the generated code:

- USB Descriptors, Function Driver Registration table, USB Device Layer Master Descriptor Table. These data structures should be placed in `system_init.c`. Refer to the [USB Device Layer Library](#) section for more information.
- Application initialization steps should be implemented in `APP_Initialize` function in `app.c`
- Application Tasks routine should be implemented in `APP_Tasks` function in `app.c`

Step 11: Create a USB Device Layer Event Handler

The application should create a Device Layer Event Handler which will handle all the events generated by the Device Layer. The following code shows a list of all possible events that are generated by the Device Layer. The code can be used in the application and updated to meet the application requirements. Refer to the USB Device Layer Library section for more details on the Device Layer Event Handling. The `USB_DEVICE_Attach` function can be called to attach the device on the USB when the `USB_DEVICE_EVENT_POWER_DETECTED` event occurs.

This code example shows how the application can set a Device Layer Event Handler.

```
// Application states
typedef enum
{
    //Application's state machine's initial state.
    APP_STATE_INIT=0,
    APP_STATE_SERVICE_TASKS,
    APP_STATE_WAIT_FOR_CONFIGURATION,
} APP_STATES;

USB_DEVICE_HANDLE usbDeviceHandle;
APP_STATES appState;

// This is the application device layer event handler function.

USB_DEVICE_EVENT_RESPONSE APP_USBDeviceEventHandler
(
    USB_DEVICE_EVENT event,
    void * pData,
    uintptr_t context
)
{
    USB_SETUP_PACKET * setupPacket;
    switch(event)
    {
        case USB_DEVICE_EVENT_POWER_DETECTED:
            // This event is generated when VBUS is detected. Attach the device
            USB_DEVICE_Attach(usbDeviceHandle);
            break;

        case USB_DEVICE_EVENT_POWER_REMOVED:
            // This event is generated when VBUS is removed. Detach the device
            USB_DEVICE_Detach (usbDeviceHandle);
            break;

        case USB_DEVICE_EVENT_CONFIGURED:
            // This event indicates that Host has set Configuration in the Device.
            break;

        case USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST:
            // This event indicates a Control transfer setup stage has been completed.
            setupPacket = (USB_SETUP_PACKET *)pData;

            // Parse the setup packet and respond with a USB_DEVICE_ControlSend(),
            // USB_DEVICE_ControlReceive or USB_DEVICE_ControlStatus().
    }
}
```

```

        break;

    case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT:
        // This event indicates that a Control transfer Data has been sent to Host.
        break;

    case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA RECEIVED:
        // This event indicates that a Control transfer Data has been received from Host.
        break;

    case USB_DEVICE_EVENT_CONTROL_TRANSFER_ABORTED:
        // This event indicates a control transfer was aborted.
        break;

    case USB_DEVICE_EVENT_SUSPENDED:
        break;

    case USB_DEVICE_EVENT_RESUMED:
        break;

    case USB_DEVICE_EVENT_ERROR:
        break;

    case USB_DEVICE_EVENT_RESET:
        break;

    case USB_DEVICE_EVENT_SOF:
        // This event indicates an SOF is detected on the bus. The      USB_DEVICE_SOF_EVENT_ENABLE
        // macro should be defined to get this event.
        break;
    default:
        break;
}
}

```

Step 12: Function Driver Event Handling

The application should create a Function Driver Event Handler which will handle all the events generated by the Function Driver. Refer to the **Event Handling** topic in the applicable function driver documentation for more details on the Function Driver Event Handling.

Step 13: Opening the Device Layer

The application should open the Device Layer in the Application tasks routine and register an event handler with the Device Layer. The Device Layer will generate events when the event handler is registered.

The following code shows an example for how to open the Device Layer.

```

void APP_Tasks ( void )
{
    // Check the application's current state.
    switch ( appState )
    {
        // Application's initial state.
        case APP_STATE_INIT:
            // Open the device layer
            usbDeviceHandle = USB_DEVICE_Open( USB_DEVICE_INDEX_0,
                DRV_IO_INTENT_READWRITE );

            if(usbDeviceHandle != USB_DEVICE_HANDLE_INVALID)
            {
                // Register a callback with device layer to get event notification
                USB_DEVICE_EventHandlerSet(usbDeviceHandle,
                    APP_USBDeviceEventHandler, 0 );
                appState = APP_STATE_WAIT_FOR_CONFIGURATION;
            }
        else
        {
            // The Device Layer is not ready to be opened. We should try
            // gain later.
        }
    }
}

```

```

    case APP_STATE_SERVICE_TASKS:
        break;

        // The default state should never be executed.
    default:
        break;
    }
}

```

USB Device Stack Porting Guide

This section provides information for porting a MLA USB Device to MPLAB Harmony.

Introduction

This topic provides information for porting a MLA USB Device to MPLAB Harmony.

Source Files to Include

This topic provides information on the source files to be included when porting a MLA USB device project to MPLAB Harmony.

Description

The following table lists the source files that must be included for MLA based and MPLAB Harmony based USB device projects. In MLA there is no separate controller driver implementation. Both enumeration functionality and controller driver is implemented in the `usb_device.c` file. In the MPLAB Harmony USB Device Stack, the controller driver is implemented in the `drv_usb.c` and `drv_usb_device.c` files, and enumeration functionality is implemented in the `usb_device.c` file.

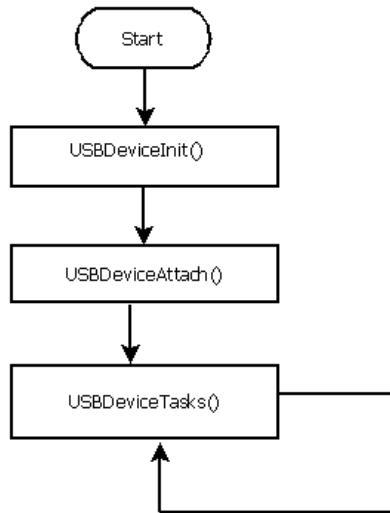
MLA	MPLAB Harmony	Description
<code>usb_device.c</code>	<code>usb_device.c</code>	In MLA, this file contains enumeration functionality and controller driver implementations. In MPLAB Harmony, this file implements enumeration functionality only.
<code>usb_function_xxx.c</code> (Where <code>xxx</code> stands for function driver name such as hid, msd, etc.)	<code>usb_device_xxx.c</code> (Where <code>xxx</code> stands for function driver name such as hid, msd, etc.)	Contains all function driver implementations.
<code>usb_descriptors.c</code>	<code>system_init.c</code>	Contains USB stack configurations.
N/A	<code>drv_usb.c</code> and <code>drv_usb_device.c</code>	USB peripheral driver implementation.

Initializing the USB Device Stack

This topic provides initialization information when porting a MLA USB device project to MPLAB Harmony.

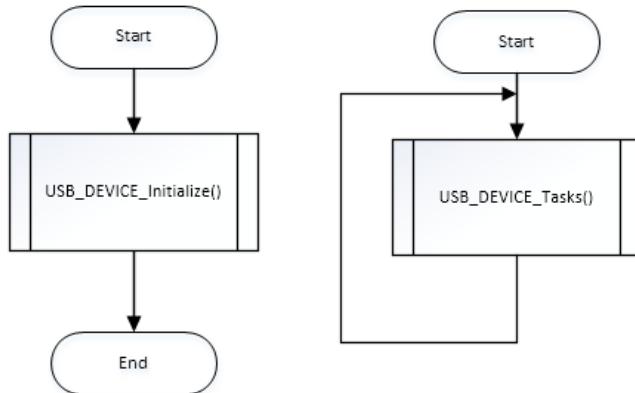
Description

In MLA, initializing the USB device stack consists of calling USB functions in the sequence shown in the following flow chart.



The `USBDDeviceInit` function will initialize the USB Device stack. `USBDDeviceAttach` function will attach the device to bus. `USBDDeviceTasks` function will run the stack task routine.

In the MPLAB Harmony USB Device Stack, initializing the USB device stack consists of calling USB functions in the sequence shown in the following chart.



The `USB_DEVICE_Initialize` function will initialize the USB device stack. On successful initialization, this function returns a valid system object. This system object is passed to the `USB_DEVICE_Tasks` function. This function is called periodically in the MPLAB Harmony `SYS_Tasks` function to maintain the state of the USB Device.

Configuring the Stack

This topic provides stack configuration information when porting a MLA USB device project to MPLAB Harmony.

Description

To configure the stack in MLA, USB descriptors have to be defined in the `usb_descriptors.c` file. The configuration is done in `usb_config.h`. Whereas in MPLAB Harmony, USB descriptors are defined in `system_init.c`. The MPLAB Harmony USB Device Stack configuration is done in `system_config.h`. Aside from descriptors, users must also define a function registration table and a master descriptor table in `system_init.c`. Refer to the specific USB Device section for more information on the function registration table and the master descriptor table.

Calling the Device Layer API

Provides information on calling the Device Layer API functions.

Description

In the Microchip Library of Applications (MLA), the application does not have to explicitly open the Device Layer to invoke the Device Layer API. The Device Layer APIs do not require a client handle.

The MPLAB Harmony USB Device Stack supports multiple instances and therefore requires a client handle. The `USB_DEVICE_Open` function opens a Device Layer instance and returns a application client handle to the instance of the Device Layer that was opened. This handle is then passed to Device Layer API to specify the instance of Device Layer to be accessed.

Event Handling

This topic provides event handling information when porting a MLA USB device project to MPLAB Harmony.

Description

In MLA, users handle events inside the USER_USB_CALLBACK_EVENT_HANDLER function. Whereas in MPLAB Harmony, the user application client will have to register an event handler with the USB device stack using the [USB_DEVICE_EventHandlerSet](#) function. All USB bus events are then forwarded to this user registered event handler function.

The following code shows an example of registering the event handler for the MPAB Harmony stack using the [USB_DEVICE_EventHandlerSet](#) function. In the example, the [USB_DEVICE_EventHandlerSet](#) function is the application event handler.

```
/* Open the Device Layer */
```

```
USB_DEVICE_HANDLE usbDeviceHandle;
```

```
usbDeviceHandle = USB_DEVICE_Open(USB_DEVICE_INDEX_0, DRV_IO_INTENT_READWRITE);
```

```
/* Register an event handler */
```

```
USB_DEVICE_EventHandlerSet(usbDeviceHandle, APP_USBDeviceEventHandler, 0);
```

The following code shows the implementation of the APP_USBDeviceEventHandler function.

```
void APP_USBDeviceEventHandler ( USB_DEVICE_EVENT event, void * eventData, uintptr_t context )
{
    switch ( event )
    {
        case USB_DEVICE_EVENT_SOF:
            break;

        case USB_DEVICE_EVENT_RESET:
            break;

        case USB_DEVICE_EVENT_CONFIGURED:
            break;

        case USB_DEVICE_EVENT_POWER_DETECTED:
            /* VBUS was detected. We can attach the device */
            USB_DEVICE_Attach(appData.deviceHandle);
            break;

        case USB_DEVICE_EVENT_POWER_REMOVED:
            /* VBUS is not available any more. Detach the device. */
            USB_DEVICE_Detach(appData.deviceHandle);
            break;

        case USB_DEVICE_EVENT_SUSPENDED:
            break;

        case USB_DEVICE_EVENT_RESUMED:
            break;

        case USB_DEVICE_EVENT_ERROR:
            break;

        default:
            break;
    }
}
```

Initializing and Communicating with the Endpoint

This topic provides endpoint initialization and communication information when porting a MLA USB device project to MPLAB Harmony.

Description

In MLA, user applications explicitly initialize the required endpoints and can directly read and write to the endpoints. The following code shows how MLA initializes the endpoints in an event handler when the device is configured.

```
*****  
BOOL USER_USB_CALLBACK_EVENT_HANDLER(int event, void *pdata, WORD size)  
{  
    switch(event)  
    {  
        case EVENT_CONFIGURED:  
            //enable the HID endpoint  
            USBEnableEndpoint(HID_EP,  
                USB_IN_ENABLED|USB_OUT_ENABLED|USB_HANDSHAKE_ENABLED|USB_DISALLOW_SETUP);  
            //Re-arm the OUT endpoint for the next packet  
            USBOutHandle = HIDPxFxPacket(HID_EP,(BYTE*)&ReceivedDataBuffer,64);  
            break;  
  
        case EVENT_TRANSFER:  
            //Add application specific callback task or callback function here if desired.  
            break;  
        case EVENT_SOF:  
            USBCB_SOF_Handler();  
            break;  
        case EVENT_SUSPEND:  
            USBCBSuspend();  
    }  
}
```

Endpoint configured in MLA

For MPLAB Harmony, the user application only manages endpoints directly for Generic USB device implementations. For standard USB devices, the USB device layer initializes the endpoint and any communication to an endpoint must occur through function driver APIs. The USB device layer initializes the endpoint based on the information provided by the user in the configurations descriptor. The USB device layer does this by parsing the descriptors table at run time. So the user does not have to explicitly enable or disable the endpoint.

Handling Endpoint 0 (EP0) Packets

This topic provides information for handling Endpoint 0 packets when porting a MLA USB device project to MPLAB Harmony.

Description

Standard device requests are handled by device stack in both MLA and MPLAB Harmony. The difference lies in handling class specific requests. In MLA, the class specific requests are forwarded to the application as EP0 events and application forwards them to appropriate function driver. The following code shows how the application forwards them to appropriate function driver in the event handler.

```
*****  
BOOL USER_USB_CALLBACK_EVENT_HANDLER(int event, void *pdata, WORD size)  
{  
    switch(event)  
    {  
        case EVENT_EP0_REQUEST:  
            USBCheckHIDRequest();  
            break;  
  
        case EVENT_TRANSFER:  
            //Add application specific callback task or callback function here if desi  
            break;  
  
        case EVENT_CONFIGURED:  
            //enable the HID endpoint  
            USBEnableEndpoint(HID_EP,  
                USB_IN_ENABLED|USB_OUT_ENABLED|USB_HANDSHAKE_ENABLED|USB_DISALLOW_SETUP)  
    }  
}
```

EP0 request is forwarded to HID

For the MPLAB Harmony USB Device Stack, EP0 events are generated from the appropriate function driver as meaningful events. For example, the HID function driver in MPLAB Harmony parses the EP0 packet to generate events like USB_DEVICE_HID_EVENT_GET_IDLE, USB_DEVICE_HID_EVENT_SET_REPORT, and USB_DEVICE_HID_EVENT_SET_IDLE, etc.

The following code shows how the event handler callback is registered with the HID function driver using the [USB_DEVICE_HID_EventHandlerSet](#) function. This is done in the Device Layer [USB_DEVICE_EVENT_CONFIGURED](#) event.

```
case USB_DEVICE_EVENT_CONFIGURED:  
    /* Set the flag indicating device is configured. */  
    appData.deviceConfigured = true;  
  
    /* Save the other details for later use. */
```

```

appData.configurationValue = ((USB_DEVICE_EVENT_DATA_CONFIGURED*)
                           eventData)->configurationValue;

/* Register application HID event handler */
USB_DEVICE_HID_EventHandlerSet(USB_DEVICE_HID_INDEX_0, APP_USBDeviceHIDEEventHandler,
                               (uintptr_t)&appData);

```

The following code shows the implementation of the HID event handler function in the application. Note how the HID function driver converts EP0 requests to a HID class-specific control transfer event.

```

USB_DEVICE_HID_EVENT_RESPONSE APP_USBDeviceHIDEEventHandler
(
    USB_DEVICE_HID_INDEX iHID,
    USB_DEVICE_HID_EVENT event,
    void * eventData,
    uintptr_t userData
)
{
    USB_DEVICE_HID_EVENT_DATA_REPORT_SENT * reportSent;
    USB_DEVICE_HID_EVENT_DATA_REPORT RECEIVED * reportReceived;

    /* Check type of event */
    switch (event)
    {

        case USB_DEVICE_HID_EVENT_SET_IDLE:

            /* For now we just accept this request as is. We acknowledge
             * this request using the USB_DEVICE_HID_ControlStatus()
             * function with a USB_DEVICE_CONTROL_STATUS_OK flag */

            USB_DEVICE_ControlStatus(appData.usbDevHandle, USB_DEVICE_CONTROL_STATUS_OK);

            /* Save Idle rate received from Host */
            appData.idleRate = ((USB_DEVICE_HID_EVENT_DATA_SET_IDLE*)eventData)->duration;
            break;
    }
}

```

USB Device Stack Porting Example

Provides information on porting a MLA Device Stack demonstration application to the MPLAB Harmony USB Device Stack.

Description

With the introduction of MPLAB Harmony, PIC32MX USB device applications can now be developed using either the USB Device Stack available in Microchip Application Libraries (MLA) v2013-06-15 or the USB Device Stack in the latest version of MPLAB Harmony.

In a case where an existing or a partially developed MLA-based USB Device application needs to be migrated to use the MPLAB Harmony USB Device Stack, the migration process is not a one-to-one process, as there are API level differences between the MLA and MPLAB Harmony USB Device Stack. This guide provides a detailed step-by-step comparison between a MLA USB Device and a MPLAB Harmony USB Device application, with an objective to enable the application migration process.

An existing v2013-06-15 MLA USB Device Stack example, the "Device – CDC – Basic Demo" will be compared with the MPLAB Harmony USB Device cdc_com_port_single demonstration application. The outline of the comparative analysis is as follows:

- Highlight framework level similarities and differences between the MLA and MPLAB Harmony USB Device Stack
- Analyze and compare the files included in the MLA "Device – CDC – Basic Demo" demonstration application project to the files included in the MPLAB Harmony USB Device Stack cdc_com_port_single demonstration application
- Analyze and compare the PIC32 Device Fuse Configuration macros, USB Device Stack Configuration, USB Device Stack API and application API in the MLA "Device – CDC – Basic Demo" demonstration application source code with their equivalents in the MPLAB Harmony USB Device cdc_com_port_single demonstration application project

The projects to be compared are available in the MLA USB Device and MPLAB Harmony USB Device demonstration applications folders.

Prerequisites

Describes the prerequisites for using this porting example.

Description

Ensure that the following software is installed:

- Microchip Libraries for Applications v2013-06-15
- The latest version of MPLAB Harmony (available by visiting: www.microchip.com/harmony)

In addition, knowledge of common project-related operations and settings in MPLAB X IDE, as well as familiarity with the MLA USB Solutions and

the USB protocol is assumed.

USB Device Stack in MLA and MPLAB Harmony

In this section, the key similarities and differences between the USB Device Stack in MLA and MPLAB Harmony are discussed.

Description

Similarities

The key similarities between the USB Device Stack in MLA and MPLAB Harmony are:

- Both solutions support Interrupt and Polled mode operation
- Both solutions require configuration macros to control/define functional aspects of the device stack
- Both solutions contain module Tasks routines that need to be invoked either in a `while(1)` application superloop and/or in the USB Interrupt Service Routine (ISR)
- In both solutions, the function drivers and Device Layer are implemented as separate modules

Differences

The key differences between the USB Device Stack in MLA and MPLAB Harmony are listed in the following table.

MPLAB Harmony	MLA
In MPLAB Harmony, the complete user application is factored into System and Application code sections. The code sections are implemented in separate source code files. This represents the MPLAB Harmony Application paradigm.	There is no such factoring in MLA.
In MPLAB Harmony, the USB peripheral management is performed by a USB Controller Driver. This is implemented separately.	The MLA USB Device stack combines the implementation of Device Layer and controller driver into one file.
In MPLAB Harmony, the USB Device Layer is a MPLAB Harmony module and requires the application to open it to access the Device Layer functions.	This is not required in MLA.
In MPLAB Harmony, Device Layer API is factored into System and Client functions. The application calls the client functions along with the Device Layer Handle.	There is no such factoring in the MLA Device Layer implementation.
In MPLAB Harmony, function driver API require the specification of an instance index.	This is typically not required in MLA.
In MPLAB Harmony, the USB Device Stack uses System Services to access shared system resources.	The MLA does not contain a formal implementation of such system services.
In MPLAB Harmony, the Device Layer initializes the function driver based on the configuration set by the host. This process does not require application intervention.	In MLA, the application must add code for the Set Configuration Event handling routine to invoke the initialize routine of the required function driver.

MLA and MPLAB Harmony USB Device Stack Files

This section analyzes the files included in the MLA "Device – CDC – Basic Demo" demonstration application MPLAB X IDE project and identifies the equivalent MPLAB Harmony USB Device Stack and Framework files in the MPLAB Harmony USB Device `cdc_com_port_single` demonstration MPLAB X IDE project.

The MLA "Device – CDC – Basic Demo" demonstration application MPLAB X IDE project is located in the following MLA installation path:
`microchip_solutions_v2013-06-15/USB/Device - CDC - Basic Demo/Firmware/MPLAB.X`.

The MPLAB Harmony `cdc_com_port_single` project is located in the
`<install-dir>/apps/usb/device/cdc_com_port_single/firmware/cdc_com_port_single.X`.

Description

The following table lists the source (`.c`) files and header (`.h`) configuration files in the MLA "Device – CDC – Basic Demo" demonstration application project and their MPLAB Harmony counterparts (along with their paths). The parent folder for MPLAB Harmony framework files, unless otherwise specified, is `<install-dir>/framework`. The table also contains files which are exclusively required by the MPLAB Harmony project.

Classification	MLA USB Device Stack Application	MPLAB Harmony USB Device Stack Application
USB Device Layer Implementation	<code>usb_device.c</code>	<code>/usb/usb_device.c</code>
USB Device CDC Function Driver	<code>usb_function_cdc.c</code>	<code>/usb/src/dynamic/usb_device_cdc.c</code> <code>/usb/src/dynamic/usb_device_cdc_acm.c</code>

Application main file	main.c	main.c app.c app.h (see Note)
USB Device Descriptors	usb_descriptors.c	system_init.c (See System Configuration in this table.)
USB Device Stack Configuration	usb_config.h	system_config.h (See System Configuration in this table.)
Hardware Specification	HardwareProfile.h HardwareProfile - PIC32 USB Starter Kit.h	(See Board Support Package in this table.)
Board Support Package	N/A	bsp/pic32mx_usb_sk2/bsp_sys_init.c bsp/pic32mx_usb_sk2/bsp_config.h
System Services	N/A	system/int/src/sys_int_pic32.c system/devcon/src/sys_devcon.c system/devcon/src/pic32mx/sys_devcon_pic32mx.c
PIC32MX USB Driver	N/A	driver/usb/usbfs/src/dynamic/drv_usb.c driver/usb/usbfs/src/dynamic/drv_usb_device.c
System Configuration (see Note)	N/A	system_config.h system_init.c system_interrupt.c system_tasks.c system_definitions.h
Peripheral Library	N/A	<install-dir>/bin/framework/peripheral/PIC32MX795F512I_peripherals.a

 **Note:** These files, which are not a part of the MPLAB Harmony Framework, are application-specific and must be coded to meet the needs of the application.

Source Code Analysis

Provides code comparison information.

Description

In this section, the MLA "Device – CDC – Basic Demo" demonstration application source code will be compared to the MPLAB Harmony cdc_com_port_single demonstration application source code. The MLA "Device – CDC – Basic Demo" demonstration application logic implementation can be classified into the following components

- Fuse configuration and Initialization
- Demonstration Application Logic
- Event Handling
- Invoking the Task Routine
- USB Descriptors
- Device Stack Configuration

The following sections discuss the comparison of these components with the MPLAB Harmony USB Device Stack and MPLAB Harmony application paradigm.

Fuse Configuration and Initialization

Provides information on fuse configuration and initialization.

Description

The microcontroller fuse configuration in the MLA "Device – CDC – Basic Demo" demonstration application is implemented in the project main.c file.

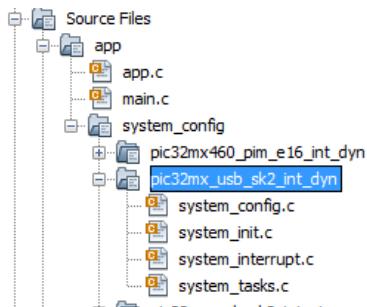
Open the MLA "Device – CDC – Basic Demo" project in MPLAB X IDE and open the main.c file. Browse to line 277 of the file. This code region defines the fuse configuration macros while using the PIC32 USB Starter Kit II. The following screen capture shows this code.

```

277 #elif defined(PIC32_USB_STarter_kIT)
278   #pragma config UPLLLEN = ON      // USB PLL Enabled
279   #pragma config FPLLIMUL = MUL_15 // PLL Multiplier
280   #pragma config UPLLIDIV = DIV_2  // USB PLL Input Divider
281   #pragma config FPLLIDIV = DIV_2  // PLL Input Divider
282   #pragma config FPLLIDIV = DIV_1  // PLL Output Divider
283   #pragma config FBBDIV = DIV_1   // Peripheral Clock divisor
284   #pragma config FWDTEN = OFF     // Watchdog Timer
285   #pragma config WDTPS = PS1      // Watchdog Timer Postscale
286   #pragma config FCKSM = CSDCMD  // Clock Switching & Fail Safe Clock Monitor
287   #pragma config OSCIOFNC = OFF    // CLKO Enable
288   #pragma config POSCMOD = HS     // Primary Oscillator
289   #pragma config IESO = OFF       // Internal/External Switch-over
290   #pragma config FSOSCEN = OFF    // Secondary Oscillator Enable (KLO was off)
291   #pragma config FNOSC = PRIPLL  // Oscillator Selection
292   #pragma config CP = OFF        // Code Protect
293   #pragma config BWP = OFF       // Boot Flash Write Protect
294   #pragma config PWP = OFF       // Program Flash Write Protect
295   #pragma config ICESEL = ICS PGx2 // ICE/ICD Comm Channel Select

```

In the MPLAB Harmony cdc_com_port_single project, locate the system configuration files. These files are located in the <install-dir>/app/system_config logical project folder. Within this folder, expand the pic32mx_usb_sk2_int_dyn configuration folder, as shown in the following figure.



Open `system_init.c` and review the fuse configurations. In MPLAB Harmony projects, the device fuse configuration macros are implemented in the `system_init.c` file. This file is a MPLAB Harmony system configuration file and is configuration-specific.

Continue with analysis of the MLA USB Demo `main.c` file. Navigate to line 488 of this file. This code region contains the application main function. The `InitializeSystem` function call at line 491 initializes the system. Further analysis of the `InitializeSystem` function indicates that this function (see line 648 of the file) configures the PIC32MX Cache and Flash Wait state for maximum performance for 60 MHz operating frequency, calls application specific user initialization function `UserInit` (see line 763) and then initializes the device stack by calling `USBDeviceInit` function (see line 765).

The following table lists the initialization related functions of the MLA USB Demonstration Application and their respective functional equivalent functions in the MPLAB Harmony USB Demonstration Application.

MLA USB Demonstration Application Function	Equivalent MPLAB Harmony USB Demonstration Application Function
<code>InitializeSystem</code>	<code>SYS_Initialize</code>
<code>SYSTEMConfigPerformance</code>	<code>SYS_DEVCON_PerformanceConfig</code>
<code>UserInit</code>	<code>APP_Initialize</code>
<code>mInitAllLEDs</code>	<code>BSP_Initialize</code>
<code>mInitAllSwitches</code>	
<code>USBDeviceInit</code>	<code>USB_DEVICE_Initialize</code>

The `usbDevInitData` and `sysDevconInit` data structures in MPLAB Harmony `system_init.c` are initialization data structures for the MPLAB Harmony USB Device Layer and the Device Control System Service, respectively. Please note that unlike the MLA USB Device Layer, the MPLAB Harmony USB Device Layer requires the specification of an endpoint table (see the `endpointTable` array declaration in `system_init.c`). The size of this table should be `USB_DEVICE_ENDPOINT_TABLE_SIZE`. This table (actually a byte array) should be aligned at 512 bytes address boundary.

The MPLAB Harmony Interrupt System Service functions, `SYS_INT_VectorPrioritySet` and `SYS_INT_Enable`, set the USB Interrupt Vector Priority and enable the global interrupts respectively. The `USB_DEVICE_Initialize` function initializes the USB Device Layer. The `APP_Initialize` function initializes the application data structure.

Demonstration Application Logic

Provides information on the demonstration application logic.

Description

The application logic in the MLA "Device – CDC – Basic Demo" demonstration application is implemented in the `ProcessIO` function (see line 819 of the project `main.c` file). This function is called periodically via the application `while(1)` loop (see line 521 of `main.c`). The `ProcessIO` function

calls the getsUSBUSART, putrsUSBUSART, and putUSBUSART functions to transfer data over the CDC Device Interface. The USBUSARTIsTxTrfReady function checks if the CDC function driver is ready to accept data for transmission. The CDCTxService function maintains the CDC function driver transmit state machine and is called periodically through the ProcessIO function (see line 870 of `main.c`).

In the MPLAB Harmony project, the appData object defined in the `app.h` file serves as a container for all application global data. The APP_StateReset function in the `app.c` file is called by the application and returns true if the device is not configured. This then causes the application state machine to reset.

The MPLAB Harmony application logic state machine (the APP_Tasks function) is implemented in `app.c`. The MPLAB Harmony USB Device Stack requires the application logic to be coded as a state machine that yields to the system periodically. In this state machine, the MPLAB Harmony USB Device Stack application must first open the USB Device Layer to access the functionality of the Device Layer. This is done through the [USB_DEVICE_Open](#) function. This step is not required in the MLA USB Device Stack application. The MPLAB Harmony USB Device Stack application can then wait for the device to be configured.

In the MPLAB Harmony USB Device application, the device is attached to the bus when VBUS power is detected. This is done by calling the [USB_DEVICE_Attach](#) function in the [USB_DEVICE_EVENT_POWER_DETECTED](#) function in APP_USBDeviceEventHandler function. The equivalent MLA USB Device Stack function is the [USBDeviceAttach](#) function. The [USB_DEVICE_CDC_Read](#) and [USB_DEVICE_CDC_Write](#) functions are called to transfer data over the CDC interface. The following table lists some of the MLA USB CDC function driver routines and their equivalents in the MPLAB Harmony CDC Function Driver API.

MLA USB CDC Function Driver Routine	MPLAB Harmony CDC Function Driver Routine
getsUSBUSART	USB_DEVICE_CDC_Read
putUSBUSART	USB_DEVICE_CDC_Write
putrsUSBUSART	USB_DEVICE_CDC_Write
CDCTxService	No equivalent routine. In the MPLAB Harmony USB Device Stack Framework, function driver state machine is updated by the Device Layer.
USBUSARTIsTxTrfReady	No equivalent routine. In the MPLAB Harmony USB Device Stack Framework, all data transfers are scheduled in a queue. The application does not need to check if the transmit state machine is ready. The application must use function driver events to check when a transfer is complete.

In the MLA USB CDC Function Driver, the getsUSBUSART function returns zero if no data has been received. In MPLAB Harmony CDC Function driver, the application must schedule a read with the [USB_DEVICE_CDC_Read](#) function, and then wait for the read complete event.

In the MLA USB CDC Function Driver, the application uses the USBUSARTIsTxTrfReady function to check if the CDC function is ready to transmit data. In the MPLAB Harmony CDC Function driver, the application does not need to check for transmit status. It schedules a write transfer request through the [USB_DEVICE_CDC_Write](#) function. If the request is added successfully to the queue, the application receives a valid transfer handle. The completion the transfer is indicated by an event. The MPLAB Harmony [USB_DEVICE_CDC_Read](#) and the [USB_DEVICE_CDC_Write](#) functions add transfer requests to a queue. The size of the queue is configurable. The MLA USB CDC Function driver does not support queuing.

Event Handling

Provides information on handling Device Layer events.

Description

The MLA USB Device Layer provides events to the application. In the MLA "Device – CDC – Basic Demo" demonstration application, the USB Device Layer application event handle, `USER_USB_CALLBACK_EVENT_HANDLER`, is implemented at line 1427 of the project `main.c` file. The MLA USB CDC function driver does not provide events to the application.

The MPLAB Harmony USB Device Layer Event handler is implemented in the APP_USBDeviceEventHandler function. The following table lists the MLA USB Device Layer Event and the equivalent MPLAB Harmony USB Device Layer Events. The MPLAB Harmony USB Device Layer features additional events that facilitate application device state management. Refer to the [USB Device Layer Library](#) documentation for more details.

MLA USB Device Layer Events	MPLAB Harmony USB Device Layer Events
EVENT_CONFIGURED	USB_DEVICE_EVENT_CONFIGURED
EVENT_SET_DESCRIPTOR	The USB Device Layer implementation in the MPLAB Harmony v0.80.xx installs Set Descriptor requests from the Host. The USB Device Layer implementation in the MPLAB Harmony v1.0 will provide this event to the application as USB_DEVICE_EVENT_SET_DESCRIPTOR event.
EVENT_EP0_REQUEST	USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST
EVENT_ATTACH	USB_DEVICE_EVENT_POWER_DETECTED
EVENT_TRANSFER_TERMINATED	USB_DEVICE_EVENT_CONTROL_TRANSFER_ABORTED
EVENT_TRANSFER	Not Applicable for standard device types. The USB Device Layer implementation in the MPLAB Harmony v1.0 will provide a USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE and a USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE event for Generic device implementations.

EVENT_SOF	USB_DEVICE_EVENT_SOF
EVENT_RESUME	USB_DEVICE_EVENT_RESUMED
EVENT_SUSPEND	USB_DEVICE_EVENT_SUSPENDED
EVENT_BUS_ERROR	USB_DEVICE_EVENT_ERROR
EVENT_TRANSFER_TERMINATED	The USB Device Layer implementation in the MPLAB Harmony does not provide this event.

- The MLA USB Device Layer Event Handler function is fixed and needs to be specified for the code to compile. The MPLAB Harmony USB Device Layer Event Handler is Device Layer instance specific and is set through the [USB_DEVICE_EventHandlerSet](#) function. This function is called after the device layer is opened successfully in the APP_Tasks function.
- The MLA USB Device Layer Event Handler implementation must invoke the required function driver initialization routine in the EVENT_CONFIGURED event. This is not required in the MPLAB Harmony USB Device Layer Event Handler. The MPLAB Harmony USB Device Layer internally initializes all the function drivers associated with the configuration that the USB Host has set. The application should instead register event handlers with all the initialized function drivers.
- The MLA USB Device Layer Event Handler implementation must handle class specific control requests in the EVENT_EP0_REQUEST event response. The MPLAB Harmony USB Device Layer internally routes class specific requests directly to the function drivers. This process does not require application intervention. The [USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST](#) event is generated when device layer receives a control transfer that is cannot be handled by the Device Layer or the Function Driver.

The MLA CDC Function Driver Interface is not event based and hence does not require an event handler. The MPLAB CDC Function Driver Interface is event based and requires a function driver instance specific event handler to be registered. This event handler is registered through the [USB_DEVICE_CDC_EventHandlerSet](#) function. The event handler should be registered in the Device Layer Set Configuration Event

Invoking the Tasks Routine

Provides information on how to invoke the Task Routine.

Description

Refer to line 502 of MLA "Device – CDC – Basic Demo" demonstration application file, `main.c`. This section of the code invokes the MLA USB Device Layer Task Routine, the `USBDeviceTasks` function, in the application `while(1)` superloop if the Device Stack is configured for Polled mode. If the Device Stack is configured for Interrupt mode operation, the `USBDeviceTasks` function is called (see line 664 of the `usb_device.c` file) in the USB Interrupt Service Routine. The MLA USB Device Stack features only one Task routine function.

The USB Device Layer Tasks routine in the MPLAB Harmony USB Device Stack is implemented by the [USB_DEVICE_Tasks](#) function. This function must be always be called in the `SYS_Tasks` function, regardless of whether the USB Device Stack is configured for Polled or Interrupt mode of operation. The MPLAB Harmony USB Device Stack defines an additional Interrupt mode specific task routine, the [USB_DEVICE_Tasks_ISR](#) function, that must be called from the USB Peripheral Interrupt Service Routine (ISR). This function should only be called in the USB Peripheral ISR. The USB Peripheral ISR is implemented in the `system_interrupt.c` file.

USB Device Descriptors

The USB Device Descriptors in the MLA "Device – CDC – Basic Demo" demonstration application are defined in the `usb_descriptors.c` file. This file contains the USB Device, Configuration, and String Descriptors.

Description

The USB Device Descriptors in the MLA "Device – CDC – Basic Demo" demonstration application are defined in the `usb_descriptors.c` file. This file contains the USB Device, Configuration and String Descriptors.

The USB Device, Configuration, and String Descriptors. USB Descriptors in a MPLAB Harmony USB Device Stack Demonstration Application are specified in the `system_init.c` file. Some of the macros (those that define endpoint and direction) used in the "Device – CDC – Basic Demo" demonstration application USB descriptors are not used in the MPLAB Harmony `cdc_com_port_single` USB Descriptors.

The `system_init.c` file also contains additional data structures needed by the MPLAB Harmony USB Device Stack. These are:

- Function Driver registration table (`funcRegistrationTable`), which provisions functions drivers to be included in this USB device
- Master Descriptor table (`usbMasterDescriptor`), which provides information configuration and the string descriptors to the USB Device Layer
- Table of speed specific configuration descriptors (`fullSpeedConfigDescSet`)
- CDC function driver initialization data structure (`cdclInit`). This structure contains the size of the CDC read and write, and the serial state notification send queues.

 **Note:** The MLA USB Device layer does not require the specification of the data structures.

Device Stack Configuration

The USB Device Stack configuration in the MLA "Device – CDC – Basic Demo" demonstration application is defined in the project-specific `usb_config.h` file, located in the Project firmware folder.

Description

The USB Device Stack configuration in the MPLAB Harmony cdc_com_port_single demonstration application needs to be specified in the `system_config.h` file. This includes the configuration for the USB Controller Driver, Device Layer, and CDC Function Driver. The following table shows the MLA USB Device Stack Configuration macros and their equivalent MPLAB Harmony USB Device Stack Configuration Macros.

MLA USB Device Stack Configuration Macros	MPLAB Harmony USB Device Stack Configuration Macros
<code>USB_EP0_BUFF_SIZE</code>	<code>USB_DEVICE_EP0_BUFFER_SIZE</code>
<code>USB_MAX_NUM_INT</code>	The number of the interface used by a function driver instance is specified via an entry in the Function Driver Registration Table (in <code>system_init.c</code>).
<code>USB_USER_DEVICE_DESCRIPTOR</code>	This USB Device Descriptor is specified via the Master Descriptor Table (in <code>system_init.c</code>).
<code>USB_PING_PONG_MODE</code>	Not applicable. The USB Controller is always configured for maximum performance (full ping pong).
<code>USB_POLLING</code> <code>USB_INTERRUPT</code>	<code>DRV_USB_INTERRUPT_MODE</code> . Should be set to true for Interrupt mode and false for Polled mode.
<code>USB_PULLUP_OPTION</code>	Not applicable.
<code>USB_TRANSCEIVER_OPTION</code>	Not applicable.
<code>USB_SPEED_OPTION</code>	This is defined via the <code>USB_DEVICE_INIT</code> data structure at Device Layer Initialization in <code>system_init.c</code> .
<code>USB_ENABLE_STATUS_STAGE_TIMEOUTS</code>	Not applicable.
<code>USB_SUPPORT_DEVICE</code>	Specified via <code>DRV_USB_DEVICE_SUPPORT</code> and <code>DRV_USB_HOST_SUPPORT</code> . These can be set to true to enable the required feature.
<code>USB_ENABLE_ALL_HANDLERS</code>	Not applicable. All Device Layer events are provided via one event handler registered dynamically.
<code>USB_USE_CDC</code>	Not applicable. Function Drivers are provisioned via an entry in the Function Driver Registration Table (in <code>system_init.c</code>).
<code>CDC_COMM_intf_ID</code> <code>CDC_COMM_EP</code> <code>CDC_COMM_IN_EP_SIZE</code> <code>CDC_DATA_intf_ID</code> <code>CDC_DATA_EP</code> <code>CDC_DATA_OUT_EP_SIZE</code> <code>CDC_DATA_IN_EP_SIZE</code>	Not applicable. The number of the interface used by a function driver instance is specified via an entry in the Function Driver Registration Table (in <code>system_init.c</code>). The Device Layer obtains endpoint information for a specific instance of the CDC driver from the USB Configuration Descriptor.
<code>USB_CDC_SUPPORT_ABSTRACT_CONTROL_MANAGEMENT_CAPABILITIES_D1</code>	Not applicable. Features are defined by the CDC Class specific descriptors.

The MPLAB Harmony USB Device Stack requires specification of other configuration macros. Refer to the [USB Device Library](#) and the USB Driver Libraries sections for additional information.

USB Audio 1.0 Device Library

This section describes the USB Audio 1.0 Device Library.

Introduction

This section provides information on library design, configuration, usage and the library interface for the USB Audio 1.0 Device Library.

Description

The MPLAB Harmony USB Audio 1.0 Device Library (also referred to as the Audio 1.0 Function Driver or library) features routines to implement a USB Audio 1.0 Device. Examples of Audio USB Devices include USB Speakers, microphones, and voice telephony. The library provides a convenient abstraction of the USB Audio 1.0 Device specification and simplifies the implementation of USB Audio 1.0 Devices.

Using the Library

This topic describes the basic architecture of the Audio 1.0 Function Driver and provides information and examples on its use.

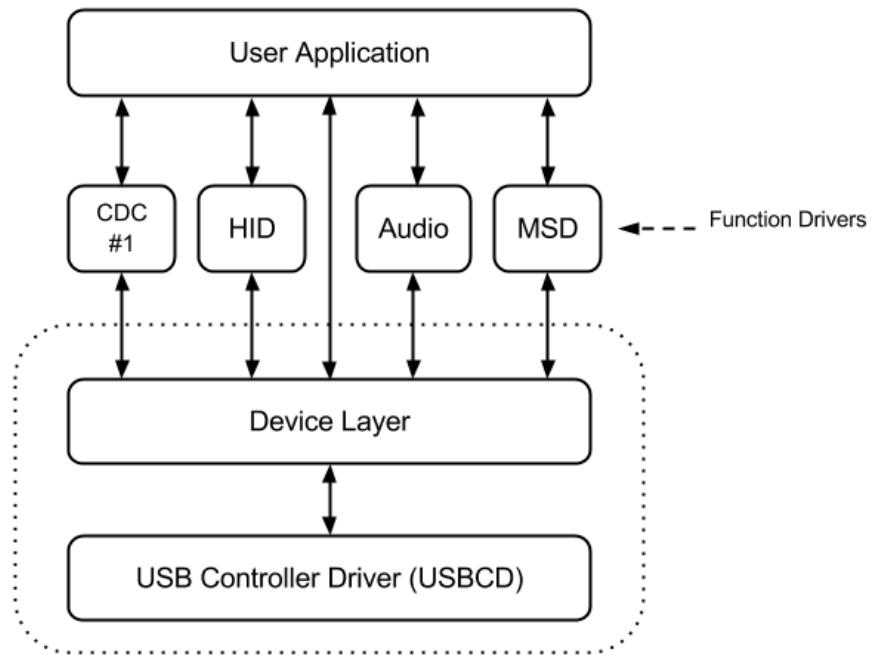
Abstraction Model

Describes the Abstraction Model of the USB Audio 1.0 Device Library.

Description

The Audio 1.0 Function Driver offers various services to the USB Audio 1.0 device to communicate with the host by abstracting USB specification details. It must be used along with the USB Device layer and USB controller to communicate with the USB host. Figure 1 shows a block diagram of the MPLAB Harmony USB Device Stack Architecture and where the Audio 1.0 Function Driver is placed.

Figure 1: USB Device 1.0 Audio Device Driver



The USB controller driver takes the responsibility of managing the USB peripheral on the device. The USB 1.0 Device Layer handles the device enumeration, etc. The USB Device 1.0 Layer forwards all Audio-specific control transfers to the Audio 1.0 Function Driver. The Audio 1.0 Function Driver interprets the control transfers and requests application's intervention through event handlers and a well-defined set of functions. The application must respond to the Audio events either in or out of the event handler. Some of these events are related to Audio 1.0 Device Class specific control transfers. The application must complete these control transfers within the timing constraints defined by USB.

Library Overview

The USB Audio 1.0 Device Library mainly interacts with the system, its clients and function drivers, as shown in the [Abstraction Model](#).

The library interface routines are divided into sub-sections, which address one of the blocks or the overall operation of the USB Audio 1.0 Device Library.

Library Interface Section	Description
Functions	Provides event handler, read/write, and transfer cancellation functions.

How the Library Works

Initializing the Library

Describes how the USB Audio 1.0 Device driver is initialized.

Description

The Audio 1.0 Function Driver instance for a USB device configuration is initialized by the Device Layer when the configuration is set by the host. This process does not require application intervention. Each instance of the Audio 1.0 Function Driver should be registered with the Device layer through the Device Layer Function Driver Registration Table. The Audio 1.0 Function Driver requires a initialization data structure to be specified. This is a [USB_DEVICE_AUDIO_INIT](#) data type that specifies the size of the read and write queues. The funcDriverInit member of the function driver registration table entry of the Audio 1.0 Function Driver instance should point to this initialization data structure. The [USB_DEVICE_AUDIO_FUNCTION_DRIVER](#) object is a global object provided by the Audio 1.0 Function Driver and provides the Device Layer with an entry point into the Audio 1.0 Function Driver. The following code shows an example of how the Audio 1.0 Function Driver can be registered with the Device Layer.

```
/* This code shows an example of how an Audio 1.0 Function Driver instances
 * can be registered with the Device Layer via the Device Layer Function Driver
 * Registration Table. In this case Device Configuration 1 consists of one
 * Audio 1.0 Function Driver instance. */

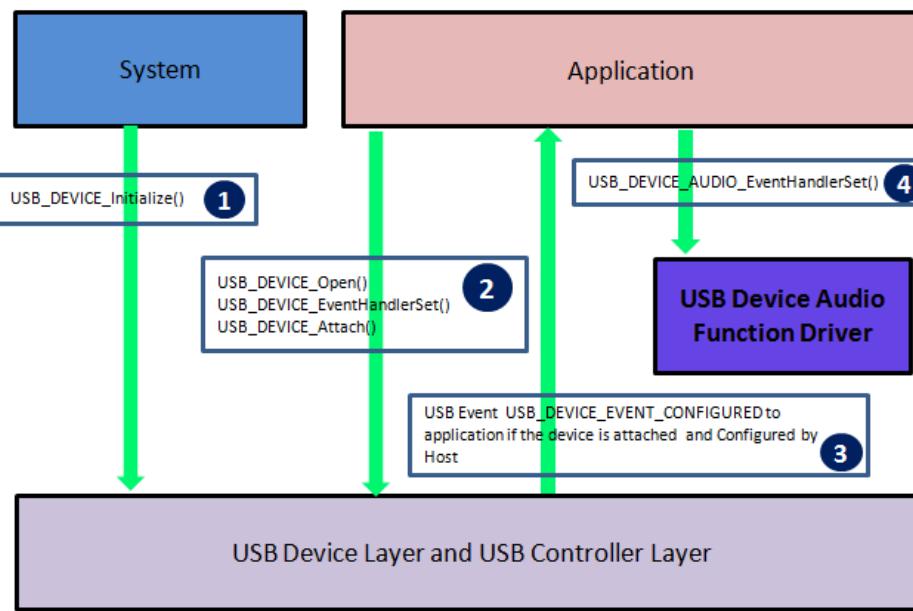
/* The Audio 1.0 Function Driver requires an initialization data structure that
 * specifies the read and write buffer queue sizes. Note that these settings are
 * also affected by the USB_DEVICE_AUDIO_QUEUE_DEPTH_COMBINED configuration
 * macro. */

const USB_DEVICE_AUDIO_INIT audioDeviceInit =
{
    .queueSizeRead = 1,
    .queueSizeWrite = 1
};

const USB_DEVICE_FUNC_REGISTRATION_TABLE funcRegistrationTable[1] =
{
    {
        .speed = USB_SPEED_FULL,                                // Supported speed
        .configurationValue = 1,                               // To be initialized for Configuration 1
        .interfaceNumber = 0,                                 // Starting interface number.
        .numberOfInterfaces = 2,                            // Number of interfaces in this instance
        .funcDriverIndex = 0,                                // Function Driver instance index is 0
        .funcDriverInit = &audioDeviceInit,                  // Function Driver does not need initialization data
        .structure
            .driver = USB_DEVICE_AUDIO_FUNCTION_DRIVER // Pointer to Function Driver - Device Layer interface
        functions
    },
};
}
```

The following figure illustrates the typical sequence that is followed in the application when using the Audio 1.0 Function Driver.

Typical USB Audio 1.0 Device Sequence



1. Call set of APIs to initialize USB Device Layer (refer to the [USB Device Layer Library](#) section for details about these APIs).
2. The Device Layer provides a callback to the application for any USB Device events like attached, powered, configured, etc. The application should receive a callback with an event `USB_DEVICE_EVENT_CONFIGURED` to proceed.
3. Once the Device Layer is configured, the application needs to register a callback function with the Audio 1.0 Function Driver to receive Audio Control transfers, and also other Audio 1.0 Function Driver events. Now the application can use Audio 1.0 Function Driver APIs to communicate with the USB Host.

Event Handling

Describes Audio 1.0 Function Driver event handler registration and event handling.

Description

Registering a Audio 1.0 Function Driver Callback Function

While creating a USB Audio 1.0 Device application, an event handler must be registered with the Device Layer (the Device Layer Event Handler) and every Audio 1.0 Function Driver instance (Audio 1.0 Function Driver Event Handler). The application needs to register the event handler with the Audio 1.0 Function Driver:

- For receiving Audio Control Requests from Host like Volume Control, Mute Control, etc.
- For handling other events from USB Audio 1.0 Device Driver (e.g., Data Write Complete or Data Read Complete)

The event handler should be registered before the USB device layer acknowledges the SET CONFIGURATION request from the USB Host. To ensure this, the callback function should be set in the `USB_DEVICE_EVENT_CONFIGURED` event that is generated by the device layer. The following code example shows how this can be done.

```

/* This a sample Application Device Layer Event Handler
 * Note how the USB Audio 1.0 Device Driver callback function
 * USB_DEVICE_AUDIO_EventHandlerSet()
 * is registered in the USB_DEVICE_EVENT_CONFIGURED event. */

void APP_USBDeviceEventHandler( USB_DEVICE_EVENT event,
                                void * pEventData, uintptr_t context )
{
    switch ( event )
    {
        case USB_DEVICE_EVENT_RESET:
        case USB_DEVICE_EVENT_DECONFIGURED:

            // USB device is reset or device is deconfigured.
            // This means that USB device layer is about to deinitialize
            // all function drivers.

            break;
    }
}

```

```

case USB_DEVICE_EVENT_CONFIGURED:

    /* check the configuration */
    if ( ((USB_DEVICE_EVENT_DATA_CONFIGURED *)  

        (eventData))->configurationValue == 1)
    {

        USB_DEVICE_AUDIO_EventHandlerSet  

        ( USB_DEVICE_AUDIO_INDEX_0,  

            APP_USBDeviceAudioEventHandler ,  

            (uintptr_t)NULL);

        /* mark that set configuration is complete */
        appData.isConfigured = true;

    }
    break;  

case USB_DEVICE_EVENT_SUSPENDED:  

    break;  

case USB_DEVICE_EVENT_RESUMED:  

case USB_DEVICE_EVENT_POWER_DETECTED:  

/* VBUS has been detected */  

USB_DEVICE_Attach(appData.usbDeviceHandle);
break;  

case USB_DEVICE_EVENT_POWER_REMOVED:  

/*VBUS is not available anymore. */  

USB_DEVICE_Detach(appData.usbDeviceHandle);
break;  

case USB_DEVICE_EVENT_ERROR:  

default:
    break;
}
}

```

Event Handling

The Audio 1.0 Function Driver provides events to the application through the event handler function registered by the application. These events indicate:

- Completion of a read or a write data transfer
- Audio Control Interface requests
- Completion of data and the status stages of Audio Control Interface related control transfer

The Audio Control Interface Request events and the related control transfer events typically require the application to respond with the Device Layer Control Transfer routines to complete the control transfer. Based on the generated event, the application may be required to:

- Respond with a [USB_DEVICE_ControlSend](#) function, which completes the data stage of a Control Read Transfer
- Respond with a [USB_DEVICE_ControlReceive](#) function, which provisions the data stage of a Control Write Transfer
- Respond with a [USB_DEVICE_ControlStatus](#) function, which completes the handshake stage of the Control Transfer. The application can either STALL or Acknowledge the handshake stage through the [USB_DEVICE_ControlStatus](#) function.

The following table shows the CDC Function Driver Control Transfer related events and the required application control transfer actions.

Audio 1.0 Function Driver Control Transfer Event	Required Application Action
USB_DEVICE_AUDIO_EVENT_CONTROL_SET_CUR USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MIN USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MAX USB_DEVICE_AUDIO_EVENT_CONTROL_SET_RES USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MEM	Identify the control type using the associated event data. If a data stage is expected, use the USB_DEVICE_ControlReceive function to receive expected data. If a data stage is not required or if the request is not supported, use the USB_DEVICE_ControlStatus function to Acknowledge or Stall the request.
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_CUR USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MIN USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MAX USB_DEVICE_AUDIO_EVENT_CONTROL_GET_RES USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MEM	Identify the control type using the associated event data. Use the USB_DEVICE_ControlSend function to send the expected data. If the request is not supported, use the USB_DEVICE_ControlStatus function to Stall the request.

USB_DEVICE_AUDIO_EVENT_ENTITY_GET_STAT	Identify the entity type using the associated event data. Use the USB_DEVICE_ControlSend function to send the expected data. If the request is not supported, use the USB_DEVICE_ControlStatus function to Stall the request.
USB_DEVICE_AUDIO_CONTROL_TRANSFER_DATA_RECEIVED	Acknowledge or stall using the USB_DEVICE_ControlStatus function.
USB_DEVICE_AUDIO_CONTROL_TRANSFER_DATA_SENT	Action not required.
USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_UNKNOWN	Interpret the setup packet and use the Device layer Control transfer functions to complete the transfer.

The application must analyze the wIndex field of the event data (received with the control transfer event) to identify the entity that is being addressed. The application must be aware of all entities included in the application and their IDs. Once identified, the application can then type cast the event data to entity type the specific control request type. For example, if the Host sends a control request to set the volume of the Audio device, the following occurs in this order:

1. The Audio 1.0 Function Driver will generate a [USB_DEVICE_AUDIO_EVENT_CONTROL_SET_CUR](#) event.
2. The application must type cast the event data to a [USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR](#) type and check the entityID field.
3. The entityID field will be identified by the application as a Feature Unit.
4. The application must now type cast the event data type as a [USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST](#) data type and check the controlSelector field.
5. If the controlSelector field is a [USB_AUDIO_VOLUME_CONTROL](#), the application can then call the [USB_DEVICE_AUDIO_ControlReceive](#) function to receive the new volume settings.

Based on the type of event, the application should analyze the event data parameter of the event handler. This data member should be type cast to an event specific data type. The following table shows the event and the data type to use while type casting. Note that the event data member is not required for all events

Audio 1.0 Function Driver Event	Related Event Data Type
USB_DEVICE_AUDIO_EVENT_CONTROL_SET_CUR	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR *
USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MIN	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MIN *
USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MAX	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MAX *
USB_DEVICE_AUDIO_EVENT_CONTROL_SET_RES	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_RES *
USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MEM	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MEM *
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_CUR	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_CUR *
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MIN	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MIN *
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MAX	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MAX *
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_RES	USB_DEVICE_AUDIO_EVENT_CONTROL_GET_RES
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MEM	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MEM *
USB_DEVICE_AUDIO_EVENT_ENTITY_GET_STAT	USB_DEVICE_AUDIO_EVENT_DATA_ENTITY_GET_STAT *
USB_DEVICE_AUDIO_CONTROL_TRANSFER_DATA_RECEIVED	NULL
USB_DEVICE_AUDIO_CONTROL_TRANSFER_DATA_SENT	NULL
USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_UNKNOWN	USB_SETUP_PACKET *
USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE	USB_DEVICE_AUDIO_EVENT_DATA_WRITE_COMPLETE *
USB_DEVICE_AUDIO_EVENT_READ_COMPLETE	USB_DEVICE_AUDIO_EVENT_DATA_READ_COMPLETE *
USB_DEVICE_AUDIO_EVENT_INTERFACE_SETTING_CHANGED	USB_DEVICE_AUDIO_EVENT_DATA_INTERFACE_SETTING_CHANGED *

Handling Audio Control Requests:

When the Audio 1.0 Function Driver receives an Audio Class Specific Control Transfer Request, it passes this control transfer to the application as a Audio 1.0 Function Driver event. The following code example shows how to handle an Audio Control request.

```
// This code example shows handling Audio Control requests. The following code
// handles a Mute request (both SET and GET) received from a USB Host.
```

```
void APP_USBDeviceAudioEventHandler
(
    USB_DEVICE_AUDIO_INDEX iAudio ,
    USB_DEVICE_AUDIO_EVENT event ,
    void * pData,
    uintptr_t context
```

```

}

{
    USB_DEVICE_AUDIO_EVENT_DATA_INTERFACE_SETTING_CHANGED *interfaceInfo;
    USB_DEVICE_AUDIO_EVENT_DATA_READ_COMPLETE *readEventData;
    uint8_t entityID;
    uint8_t controlSelector;
    if ( iAudio == 0 )
    {
        switch (event)
        {
            case USB_DEVICE_AUDIO_EVENT_INTERFACE_SETTING_CHANGED:

                /* We have received a request from USB host to change the
                 * Interface-Alternate setting. The application should be aware
                 * of the association between alternate settings and the device
                 * features to be enabled. */

                interfaceInfo = (USB_DEVICE_AUDIO_EVENT_DATA_INTERFACE_SETTING_CHANGED *)pData;
                appData.activeInterfaceAlternateSetting = interfaceInfo->interfaceAlternateSetting;
                appData.state = APP_USB_INTERFACE_ALTERNATE_SETTING_RCVD;
                break;

            case USB_DEVICE_AUDIO_EVENT_READ_COMPLETE:
                /* We have received an audio frame from the Host.
                 * Now send this audio frame to Audio Codec for Playback. */
                break;

            case USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE:
                break;

            case USB_DEVICE_AUDIO_EVENT_CONTROL_SET_CUR:

                /* This is an example of handling Audio control request. In this
                 * case the control request is targeted to the Mute Control in
                 * a feature unit entity. This event indicates that the current
                 * value needs to be set. */

                entityID = ((USB_AUDIO_CONTROL_INTERFACE_REQUEST*)pData)->entityID;
                if (entityID == APP_ID_FEATURE_UNIT)
                {
                    controlSelector = ((USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST*)pData)->controlSelector;
                    if (controlSelector == USB_AUDIO_MUTE_CONTROL)
                    {
                        /* It is confirmed that this request is targeted to the
                         * mute control. We schedule a control transfer receive
                         * to get data from the host. */

                        USB_DEVICE_ControlReceive(appData.usbDevHandle, (void *)&(appData.dacMute), 1 );
                        appData.currentAudioControl = APP_USB_AUDIO_MUTE_CONTROL;
                    }
                }
                break;

            case USB_DEVICE_AUDIO_EVENT_CONTROL_GET_CUR:

                /* This event occurs when the host is requesting a current
                 * status of control */

                entityID = ((USB_AUDIO_CONTROL_INTERFACE_REQUEST*)pData)->entityID;
                if (entityID == APP_ID_FEATURE_UNIT)
                {
                    controlSelector = ((USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST*)pData)->controlSelector;
                    if (controlSelector == USB_AUDIO_MUTE_CONTROL)
                    {
                        /* Use the control send function to send the status of
                         * the control to the host */
                        USB_DEVICE_ControlSend(appData.usbDevHandle, (void *)&(appData.dacMute), 1 );
                    }
                }
        }
    }
}

```

```
        break;

    case USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MIN:
    case USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MIN:
    case USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MAX:
    case USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MAX:
    case USB_DEVICE_AUDIO_EVENT_CONTROL_SET_RES:
    case USB_DEVICE_AUDIO_EVENT_CONTROL_GET_RES:
    case USB_DEVICE_AUDIO_EVENT_ENTITY_GET_MEM:

        /* In this example, all of these control requests are not
         * supported. So these are stalled. */
        USB_DEVICE_ControlStatus (appData.usbDevHandle, USB_DEVICE_CONTROL_STATUS_ERROR);
        break;

    case USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_DATA_RECEIVED:

        /* This event occurs when data has been received in a control
         * transfer */

        USB_DEVICE_ControlStatus(appData.usbDevHandle, USB_DEVICE_CONTROL_STATUS_OK );
        if (appData.currentAudioControl == APP_USB_AUDIO_MUTE_CONTROL)
        {
            appData.state = APP_MUTE_AUDIO_PLAYBACK;
            appData.currentAudioControl = APP_USB_CONTROL_NONE;
        }
        break;

    case USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_DATA_SENT:
        break;
    default:
        break;
}
}
}
```

Transferring Data

Describes how to send/receive data to/from USB Host using this USB Audio 1.0 Device Driver.

Description

The USB Audio 1.0 Device Driver provides functions to send and receive data.

Receiving Data

The [USB_DEVICE_AUDIO_Read](#) function schedules a data read. When the host transfers data to the device, the Audio 1.0 Function Driver receives the data and invokes the [USB_DEVICE_AUDIO_EVENT_READ_COMPLETE](#) event. This event indicates that audio data is now available in the application specified buffer.

The Audio 1.0 Function Driver supports buffer queuing. The application can schedule multiple read requests. Each request is assigned a unique buffer handle, which is returned with the [USB_DEVICE_AUDIO_EVENT_READ_COMPLETE](#) event. The application can use the buffer handle to track completion to queued requests. Using this feature allows the application to implement audio buffering schemes such as ping-pong buffering.

Sending Data

The [USB_DEVICE_AUDIO_Write](#) schedules a data write. When the host sends a request for the data, the Audio 1.0 Function Driver transfers the data and invokes the [USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE](#) event.

The Audio 1.0 Function Driver supports buffer queuing. The application can schedule multiple write requests. Each request is assigned a unique buffer handle, which is returned with the [USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE](#) event. The application can use the buffer handle to track completion to queued requests. Using this feature allows the application to implement audio buffering schemes such as ping-pong buffering.

Configuring the Library

Describes how to configure the USB Audio 1.0 Device Driver.

Macros

	Name	Description
	USB_DEVICE_AUDIO_INSTANCES_NUMBER	Specifies the number of Audio Function Driver instances.

	USB_DEVICE_AUDIO_MAX_ALTERNATE_SETTING	Specifies the maximum number of Alternate Settings per streaming interface.
	USB_DEVICE_AUDIO_MAX_STREAMING_INTERFACES	Specifies the maximum number of Audio Streaming interfaces in an Audio Function Driver instance.
	USB_DEVICE_AUDIO_QUEUE_DEPTH_COMBINED	Specifies the combined queue size of all Audio function driver instances.

Description

The application designer must specify the following configuration parameters while using the USB Audio 1.0 Device Driver. The configuration macros that implement these parameters must be located in the `system_config.h` file in the application project and a compiler include path (to point to the folder that contains this file) should be specified.

USB_DEVICE_AUDIO_INSTANCES_NUMBER Macro

Specifies the number of Audio Function Driver instances.

File

`usb_device_audio_v1_0_config_template.h`

C

```
#define USB_DEVICE_AUDIO_INSTANCES_NUMBER
```

Description

USB device Audio Maximum Number of instances

This macro defines the number of instances of the Audio Function Driver. For example, if the application needs to implement two instances of the Audio Function Driver (to create two composite Audio Device) on one USB Device, the macro should be set to 2. Note that implementing a USB Device that features multiple Audio interfaces requires appropriate USB configuration descriptors.

Remarks

None.

USB_DEVICE_AUDIO_MAX_ALTERNATE_SETTING Macro

Specifies the maximum number of Alternate Settings per streaming interface.

File

`usb_device_audio_v1_0_config_template.h`

C

```
#define USB_DEVICE_AUDIO_MAX_ALTERNATE_SETTING
```

Description

Maximum number of Alternate Settings

This macro defines the maximum number of Alternate Settings per streaming interface. If the Audio Device features multiple streaming interfaces, this configuration constant should be equal to the the maximum number of alternate required amongst the streaming interfaces.

Remarks

None.

USB_DEVICE_AUDIO_MAX_STREAMING_INTERFACES Macro

Specifies the maximum number of Audio Streaming interfaces in an Audio Function Driver instance.

File

`usb_device_audio_v1_0_config_template.h`

C

```
#define USB_DEVICE_AUDIO_MAX_STREAMING_INTERFACES
```

Description

Maximum Audio Streaming Interfaces

This macro defines the maximum number of streaming interfaces in an Audio Function Driver instance. In case of multiple Audio Function Driver

instances, this constant should be equal to the maximum number of interfaces amongst the Audio Function Driver instances.

Remarks

None.

USB_DEVICE_AUDIO_QUEUE_DEPTH_COMBINED Macro

Specifies the combined queue size of all Audio function driver instances.

File

`usb_device_audio_v1_0_config_template.h`

C

```
#define USB_DEVICE_AUDIO_QUEUE_DEPTH_COMBINED
```

Description

USB device Audio Combined Queue Size

This macro defines the number of entries in all queues in all instances of the Audio function driver. This value can be obtained by adding up the read and write queue sizes of each Audio Function driver instance. In a simple single instance USB Audio device application, that does not require buffer queuing, the `USB_DEVICE_AUDIO_QUEUE_DEPTH_COMBINED` macro can be set to 2. Consider a case of a Audio function driver instances, with has a read queue size of 2 and write queue size of 3, this macro should be set to 5 (2 + 3).

Remarks

None.

Building the Library

This section lists the files that are available in the USB Audio 1.0 Device Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/usb`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
<code>usb_device_audio_v1_0.h</code>	This header file must be included in every source file that needs to invoke USB Audio 1.0 Device Driver APIs.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
<code>/src/dynamic/usb_device_audio_v1_0.c</code>	This file contains all of functions, macros, definitions, variables, datatypes, etc., that are specific to the USB Audio Specification v1.0 implementation of the Audio 1.0 Function Driver.
<code>/src/dynamic/usb_device_audio_read_write.c</code>	Contains implementation of the Audio 1.0 Function Driver read and write functions.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB Audio 1.0 Device Library depends on the following modules:

- USB Device Library

Library Interface

a) Functions

	Name	Description
≡	USB_DEVICE_AUDIO_EventHandlerSet	This function registers an event handler for the specified Audio function driver instance.
≡	USB_DEVICE_AUDIO_Read	This function requests a data read from the USB Device Audio Function Driver Layer.
≡	USB_DEVICE_AUDIO_TransferCancel	This function cancels a scheduled Audio Device data transfer.
≡	USB_DEVICE_AUDIO_Write	This function requests a data write to the USB Device Audio Function Driver Layer.

b) Data Types and Constants

	Name	Description
	USB_DEVICE_AUDIO_EVENT_RESPONSE_NONE	USB Device Audio Function Driver event handler response type none.
	USB_DEVICE_AUDIO_TRANSFER_HANDLE_INVALID	USB Device Audio Function Driver invalid transfer handle definition.
	USB_DEVICE_AUDIO_EVENT	USB Device Audio Function Driver events.
	USB_DEVICE_AUDIO_EVENT_DATA_READ_COMPLETE	USB Device Audio Function Driver audio read and write complete event data.
	USB_DEVICE_AUDIO_EVENT_DATA_WRITE_COMPLETE	USB Device Audio Function Driver audio read and write complete event data.
	USB_DEVICE_AUDIO_EVENT_HANDLER	USB Device Audio event handler function pointer type.
	USB_DEVICE_AUDIO_EVENT_RESPONSE	USB Device Audio Function Driver event callback response type.
	USB_DEVICE_AUDIO_INDEX	USB Device Audio function driver index.
	USB_DEVICE_AUDIO_RESULT	USB Device Audio Function Driver USB Device Audio result enumeration.
	USB_DEVICE_AUDIO_TRANSFER_HANDLE	USB Device Audio Function Driver transfer handle definition.
	USB_DEVICE_AUDIO_FUNCTION_DRIVER	USB Device Audio Function Driver function pointer.
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_CUR	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_CUR.
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MAX	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MAX.
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MEM	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MEM.
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MIN	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MIN.
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_RES	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_RES.
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR	USB Device Audio Function Driver set and get request data.
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MAX	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MAX.
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MEM	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MEM.
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MIN	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MIN.
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_RES	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_RES.

	USB_DEVICE_AUDIO_EVENT_DATA_INTERFACE_SETTING_CHANGED	USB Device Audio Function Driver alternate interface setting event data.
	USB_DEVICE_AUDIO_INIT	USB Device Audio Function Driver initialization data structure.
	USB_DEVICE_AUDIO_EVENT_DATA_ENTITY_GET_STAT	This is type USB_DEVICE_AUDIO_EVENT_DATA_ENTITY_GET_STAT .
	USB_DEVICE_AUDIO_TRANSFER_ABORT_NOTIFY	USB Audio Transfer abort notification enable

Description

This section describes the Application Programming Interface (API) functions of the USB Device Audio 1.0 Library.

Refer to each section for a detailed description.

a) Functions

[USB_DEVICE_AUDIO_EventHandlerSet Function](#)

This function registers an event handler for the specified Audio function driver instance.

File

[usb_device_audio_v1_0.h](#)

C

```
USB_DEVICE_AUDIO_RESULT USB\_DEVICE\_AUDIO\_EventHandlerSet(USB_DEVICE_AUDIO_INDEX instanceIndex,
USB_DEVICE_AUDIO_EVENT_HANDLER eventHandler, uintptr_t context);
```

Returns

- [USB_DEVICE_AUDIO_RESULT_OK](#) - The operation was successful
- [USB_DEVICE_AUDIO_RESULT_ERROR_INSTANCE_INVALID](#) - The specified instance does not exist.
- [USB_DEVICE_AUDIO_RESULT_ERROR_PARAMETER_INVALID](#) - The eventHandler parameter is NULL

Description

This function registers a event handler for the specified Audio function driver instance. This function should be called by the application when it receives a SET CONFIGURATION event from the device layer. The application must register an event handler with the function driver in order to receive and respond to function driver specific events and control transfers. If the event handler is not registered, the device layer will stall function driver specific commands and the USB device may not function.

Remarks

None.

Preconditions

This function should be called when the function driver has been initialized as a result of a set configuration.

Example

```
// The following code shows an example for registering an event handler. The
// application specifies the context parameter as a pointer to an
// application object (appObject) that should be associated with this
// instance of the Audio function driver.

USB_DEVICE_AUDIO_RESULT result;

USB_DEVICE_AUDIO_EVENT_RESPONSE APP_USBDeviceAUDIOEventHandler
(
    USB_DEVICE_AUDIO_INDEX instanceIndex ,
    USB_DEVICE_AUDIO_EVENT event ,
    void* pData,
    uintptr_t context
)
{
    // Event Handling comes here

    switch(event)
    {
```

```

    ...
}

return(USB_DEVICE_AUDIO_EVENT_RESPONSE_NONE) ;
}

result = USB_DEVICE_AUDIO_EventHandlerSet ( USB_DEVICE_AUDIO_INSTANCE_0 ,
    &APP_USBDeviceAUDIOEventHandler, (uintptr_t) &appObject);

if(USB_DEVICE_AUDIO_RESULT_OK != result)
{
    // Do error handling here
}

```

Parameters

Parameters	Description
instance	Instance of the Audio Function Driver.
eventHandler	A pointer to event handler function.
context	Application specific context that is returned in the event handler.

Function

```

USB_DEVICE_AUDIO_RESULT USB_DEVICE_AUDIO_EventHandlerSet
(
    USB_DEVICE_AUDIO_INDEX instance ,
    USB_DEVICE_AUDIO_EVENT_HANDLER eventHandler ,
    uintptr_t context
);

```

USB_DEVICE_AUDIO_Read Function

This function requests a data read from the USB Device Audio Function Driver Layer.

File

[usb_device_audio_v1_0.h](#)

C

```

USB_DEVICE_AUDIO_RESULT USB_DEVICE_AUDIO_Read(USB_DEVICE_AUDIO_INDEX instanceIndex,
USB_DEVICE_AUDIO_TRANSFER_HANDLE* transferHandle, uint8_t interfaceNumber, void * data, size_t size);

```

Returns

- **USB_DEVICE_AUDIO_RESULT_OK** - The read request was successful. transferHandle contains a valid transfer handle.
- **USB_DEVICE_AUDIO_RESULT_ERROR_TRANSFER_QUEUE_FULL** - internal request queue is full. The read request could not be added.
- **USB_DEVICE_AUDIO_RESULT_ERROR_INSTANCE_NOT_CONFIGURED** - The specified instance is not configured yet.
- **USB_DEVICE_AUDIO_RESULT_ERROR_INSTANCE_INVALID** - The specified instance was not provisioned in the application and is invalid.

Description

This function requests a data read from the USB Device Audio Function Driver Layer. The function places a request with driver, the request will get serviced as data is made available by the USB Host. A handle to the request is returned in the transferHandle parameter. The termination of the request is indicated by the **USB_DEVICE_AUDIO_EVENT_READ_COMPLETE** event. The amount of data read and the transfer handle associated with the request is returned along with the event. The transfer handle expires when event handler for the **USB_DEVICE_AUDIO_EVENT_READ_COMPLETE** exits. If the read request could not be accepted, the function returns an error code and transferHandle will contain the value **USB_DEVICE_AUDIO_TRANSFER_HANDLE_INVALID**.

Remarks

While using the Audio Function Driver with PIC32MZ USB module, the audio buffer provided to the **USB_DEVICE_AUDIO_Read** function should be placed in coherent memory and aligned at a 16 byte boundary. This can be done by declaring the buffer using the **__attribute__((coherent, aligned(16)))** attribute. An example is shown here

```
uint8_t data[256] __attribute__((coherent, aligned(16)));
```

Preconditions

The function driver should have been configured.

Example

```
// Shows an example of how to read. This assumes that
// device had been configured. The example attempts to read
// data from interface 1.

USB_DEVICE_AUDIO_INDEX instanceIndex;
USB_DEVICE_AUDIO_TRANSFER_HANDLE transferHandle;
unit8_t interfaceNumber;
unit8_t rxBuffer[192]; // Use this attribute for PIC32MZ __attribute__((coherent, aligned(16)))
USB_DEVICE_AUDIO_RESULT readRequestResult;

instanceIndex = 0; //specify the Audio Function driver instance number.
interfaceNumber = 1; //Specify the Audio Streaming interface number.

readRequestResult = USB_DEVICE_AUDIO_Read ( instanceIndex, &transferHandle,
                                             interfaceNumber, &rxBuffer, 192);

if(USB_DEVICE_AUDIO_RESULT_OK != readRequestResult)
{
    //Do Error handling here
}

// The completion of the read request will be indicated by the
// USB_DEVICE_AUDIO_EVENT_READ_COMPLETE event. The transfer handle
// and the amount of data read will be returned along with the
// event.
```

Parameters

Parameters	Description
instance	USB Device Audio Function Driver instance.
transferHandle	Pointer to a USB_DEVICE_AUDIO_TRANSFER_HANDLE type of variable. This variable will contain the transfer handle in case the read request was successful.
interfaceNum	The USB Audio streaming interface number on which read request is to placed.
data	pointer to the data buffer where read data will be stored. In case of PIC32MZ device, this buffer should be located in coherent memory and should be aligned a 16 byte boundary.
size	Size of the data buffer. Refer to the description section for more details on how the size affects the transfer.

Function

```
USB_DEVICE_AUDIO_RESULT USB_DEVICE_AUDIO_Read
(
    USB_DEVICE_AUDIO_INDEX instanceIndex ,
    USB_DEVICE_AUDIO_TRANSFER_HANDLE* transferHandle,
    uint8_t interfaceNum ,
    void * data ,
    size_t size
);
```

USB_DEVICE_AUDIO_TransferCancel Function

This function cancels a scheduled Audio Device data transfer.

File

[usb_device_audio_v1_0.h](#)

C

```
USB_DEVICE_AUDIO_RESULT USB_DEVICE_AUDIO_TransferCancel(USB_DEVICE_AUDIO_INDEX instanceIndex,
```

```
USB_DEVICE_AUDIO_TRANSFER_HANDLE transferHandle);
```

Returns

- USB_DEVICE_AUDIO_RESULT_OK - The transfer will be canceled completely or partially.
- USB_DEVICE_AUDIO_RESULT_ERROR - The transfer could not be canceled because it has either completed, the transfer handle is invalid or the last transaction is in progress.

Description

This function cancels a scheduled Audio Device data transfer. The transfer could have been scheduled using the [USB_DEVICE_AUDIO_Read](#), [USB_DEVICE_AUDIO_Write](#), or the [USB_DEVICE_AUDIO_SerialStateNotificationSend](#) functions. If a transfer is still in the queue and its processing has not started, the transfer is canceled completely. A transfer that is in progress may or may not get canceled depending on the transaction that is presently in progress. If the last transaction of the transfer is in progress, the transfer will not be canceled. If it is not the last transaction in progress, the in-progress will be allowed to complete. Pending transactions will be canceled. The first transaction of an in progress transfer cannot be canceled.

Remarks

None.

Preconditions

The USB Device should be in a configured state.

Example

```
// The following code snippet cancels a AUDIO transfer.

USB_DEVICE_AUDIO_TRANSFER_HANDLE transferHandle;
USB_DEVICE_AUDIO_RESULT result;

result = USB_DEVICE_AUDIO_TransferCancel(instanceIndex, transferHandle);

if(USB_DEVICE_AUDIO_RESULT_OK == result)
{
    // The transfer cancellation was either completely or
    // partially successful.
}
```

Parameters

Parameters	Description
instanceIndex	AUDIO Function Driver instance index.
transferHandle	Transfer handle of the transfer to be canceled.

Function

```
USB_DEVICE_AUDIO_RESULT USB_DEVICE_AUDIO_TransferCancel
(
    USB_DEVICE_AUDIO_INDEX instanceIndex,
    USB_DEVICE_AUDIO_TRANSFER_HANDLE transferHandle
);
```

USB_DEVICE_AUDIO_Write Function

This function requests a data write to the USB Device Audio Function Driver Layer.

File

[usb_device_audio_v1_0.h](#)

C

```
USB_DEVICE_AUDIO_RESULT USB_DEVICE_AUDIO_Write(USB_DEVICE_AUDIO_INDEX instanceIndex,
USB_DEVICE_AUDIO_TRANSFER_HANDLE * transferHandle, uint8_t interfaceNumber, void * data, size_t size);
```

Returns

- USB_DEVICE_AUDIO_RESULT_OK - The read request was successful. transferHandle contains a valid transfer handle.
- USB_DEVICE_AUDIO_RESULT_ERROR_TRANSFER_QUEUE_FULL - internal request queue is full. The write request could not be added.

- USB_DEVICE_AUDIO_RESULT_ERROR_INSTANCE_NOT_CONFIGURED - The specified instance is not configured yet.
- USB_DEVICE_AUDIO_RESULT_ERROR_INSTANCE_INVALID - The specified instance was not provisioned in the application and is invalid.

Description

This function requests a data write to the USB Device Audio Function Driver Layer. The function places a request with driver, the request will get serviced as data is requested by the USB Host. A handle to the request is returned in the transferHandle parameter. The termination of the request is indicated by the USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE event. The amount of data written and the transfer handle associated with the request is returned along with the event in writeCompleteData member of the pData parameter in the event handler.

The transfer handle expires when event handler for the USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE exits. If the write request could not be accepted, the function returns an error code and transferHandle will contain the value [USB_DEVICE_AUDIO_TRANSFER_HANDLE_INVALID](#).

Remarks

While the using the Audio Function Driver with the PIC32MZ USB module, the audio buffer provided to the USB_DEVICE_AUDIO_Write function should be placed in coherent memory and aligned at a 16 byte boundary. This can be done by declaring the buffer using the

`__attribute__((coherent, aligned(16)))` attribute. An example is shown here

```
uint8_t data[256] __attribute__((coherent, aligned(16)));
```

Preconditions

The function driver should have been configured.

Example

```
// Shows an example of how to write audio data to the audio streaming
// interface . This assumes that device is configured and the audio
// streaming interface is 1.

USB_DEVICE_AUDIO_INDEX instanceIndex;
USB_DEVICE_AUDIO_TRANSFER_HANDLE transferHandle;
unit8_t interfaceNumber;
unit8_t txBuffer[192]; // Use this attribute for PIC32MZ __attribute__((coherent, aligned(16)))
USB_DEVICE_AUDIO_RESULT writeRequestResult;

instanceIndex = 0; //specify the Audio Function driver instance number.
interfaceNumber = 1; //Specify the Audio Streaming interface number.

writeRequestResult = USB_DEVICE_AUDIO_Write ( instanceIndex, &transferHandle,
                                              interfaceNumber, &txBuffer, 192);

if(USB_DEVICE_AUDIO_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}

// The completion of the write request will be indicated by the
// USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE event. The transfer handle
// and transfer size is provided along with this event.
```

Parameters

Parameters	Description
instance	USB Device Audio Function Driver instance.
transferHandle	Pointer to a USB_DEVICE_AUDIO_TRANSFER_HANDLE type of variable. This variable will contain the transfer handle in case the write request was successful.
interfaceNum	The USB Audio streaming interface number on which the write request is to placed.
data	pointer to the data buffer contains the data to be written. In case of PIC32MZ device, this buffer should be located in coherent memory and should be aligned a 16 byte boundary.
size	Size of the data buffer.

Function

```
USB_DEVICE_AUDIO_RESULT USB_DEVICE_AUDIO_Write
(
    USB_DEVICE_AUDIO_INDEX instance ,
    USB_DEVICE_AUDIO_TRANSFER_HANDLE* transferHandle,
    uint8_t interfaceNum ,
```

```
void * data ,
size_t size
);
```

b) Data Types and Constants

USB_DEVICE_AUDIO_EVENT_RESPONSE_NONE Macro

USB Device Audio Function Driver event handler response type none.

File

[usb_device_audio_v1_0.h](#)

C

```
#define USB_DEVICE_AUDIO_EVENT_RESPONSE_NONE
```

Description

USB Device Audio Function Driver Event Handler Response None

This is the definition of the Audio Function Driver event handler response type none.

Remarks

Intentionally defined to be empty.

USB_DEVICE_AUDIO_TRANSFER_HANDLE_INVALID Macro

USB Device Audio Function Driver invalid transfer handle definition.

File

[usb_device_audio_v1_0.h](#)

C

```
#define USB_DEVICE_AUDIO_TRANSFER_HANDLE_INVALID
```

Description

USB Device Audio Function Driver Invalid Transfer Handle Definition

This definition defines a Invalid USB Device Audio Function Driver Transfer Handle. A Invalid Transfer Handle is returned by the `USB_DEVICE_Audio_Write` , `USB_DEVICE_Audio_Read`, functions when the request was not successful.

Remarks

None.

USB_DEVICE_AUDIO_EVENT Enumeration

USB Device Audio Function Driver events.

File

[usb_device_audio_v1_0.h](#)

C

```
typedef enum {
    USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE,
    USB_DEVICE_AUDIO_EVENT_READ_COMPLETE,
    USB_DEVICE_AUDIO_EVENT_INTERFACE_SETTING_CHANGED,
    USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_DATA_RECEIVED,
    USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_DATA_SENT,
    USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_UNKNOWN,
    USB_DEVICE_AUDIO_EVENT_CONTROL_SET_CUR,
    USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MIN,
    USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MAX,
    USB_DEVICE_AUDIO_EVENT_CONTROL_SET_RES,
    USB_DEVICE_AUDIO_EVENT_ENTITY_SET_MEM,
```

```

USB_DEVICE_AUDIO_EVENT_CONTROL_GET_CUR,
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MIN,
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MAX,
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_RES,
USB_DEVICE_AUDIO_EVENT_ENTITY_GET_MEM,
USB_DEVICE_AUDIO_EVENT_ENTITY_GET_STAT
} USB_DEVICE_AUDIO_EVENT;

```

Members

Members	Description
USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE	This event occurs when a write operation scheduled by calling the USB_DEVICE_AUDIO_Write() function has completed. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE_DATA type.
USB_DEVICE_AUDIO_EVENT_READ_COMPLETE	This event occurs when a read operation scheduled by calling the USB_DEVICE_AUDIO_Read() function has completed. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_READ_COMPLETE_DATA type.
USB_DEVICE_AUDIO_EVENT_INTERFACE_SETTING_CHANGED	This event occurs when the Host requests the Audio USB device to set an alternate setting on an interface present in this audio function. An Audio USB Device will typically feature a default interface setting and one or more alternate interface settings. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_INTERFACE_SETTING_CHANGED type. This contains the index of the interface whose setting must be changed and the index of the alternate setting. The application may enable or disable audio functions based on the interface setting.
USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_DATA_RECEIVED	This event occurs when the data stage of a control write transfer has completed. This would occur after the application would respond with a USB_DEVICE_ControlReceive function, which may possibly have been called in response to a USB_DEVICE_AUDIO_EVENT_ENTITY_SETTINGS_RECEIVED event. This event notifies the application that the data is received from Host and is available at the location passed by the USB_DEVICE_ControlReceive function. If the received data is acceptable to the application, it should acknowledge the data by calling the USB_DEVICE_ControlStatus function with a USB_DEVICE_CONTROL_STATUS_OK flag. The application can reject the received data by calling the USB_DEVICE_ControlStatus function with the USB_DEVICE_CONTROL_STATUS_ERROR flag. The pData parameter will be NULL.
USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_DATA_SENT	This event occurs when the data stage of a control read transfer has completed. This would occur when the application has called the USB_DEVICE_ControlSend function to complete the data stage of a control transfer. The event indicates that the data has been transmitted to the host. The pData parameter will be NULL.
USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_UNKNOWN	This event occurs when the Audio function driver receives a control transfer request that could not be decoded by Audio Function driver. The pData parameter will point to a USB_SETUP_PACKET type containing the SETUP packet. The application must analyze this SETUP packet and use the USB_DEVICE_ControlSend or USB_DEVICE_ControlReceive or the USB_DEVICE_ControlStatus functions to advance the control transfer or complete it.
USB_DEVICE_AUDIO_EVENT_CONTROL_SET_CUR	This event occurs when the Host sends an Audio Control specific Set Current Setting Attribute Control Transfer request to an Audio Device Control. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlStatus and USB_DEVICE_ControlReceive functions.

USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MIN	This event occurs when the Host sends an Audio Control specific Set Minimum Setting Attribute Control Transfer request to an Audio Device Control. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MIN type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlStatus and USB_DEVICE_ControlReceive functions.
USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MAX	This event occurs when the Host sends an Audio Control specific Set Maximum Setting Attribute Control Transfer request to an Audio Device Control. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MAX type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlStatus and USB_DEVICE_ControlReceive functions.
USB_DEVICE_AUDIO_EVENT_CONTROL_SET_RES	This event occurs when the Host sends an Audio Control specific Set Resolution Attribute Control Transfer request to an Audio Device Control. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_RES type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlStatus USB_DEVICE_ControlSend and/or USB_DEVICE_ControlReceive functions.
USB_DEVICE_AUDIO_EVENT_ENTITY_SET_MEM	This event occurs when the Host sends an Audio Entity specific Set Memory Space Attribute Control Transfer request to an Audio Device Entity. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MEM type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlStatus USB_DEVICE_ControlSend and/or USB_DEVICE_ControlReceive functions.
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_CUR	This event occurs when the Host sends an Audio Control specific Get Current Setting Attribute Control Transfer request to an Audio Device Control. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_CUR type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlStatus and USB_DEVICE_ControlSend functions.
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MIN	This event occurs when the Host sends an Audio Control specific Get Minimum Setting Attribute Control Transfer request to an Audio Device Control. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MIN type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlStatus and USB_DEVICE_ControlSend functions.
USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MAX	This event occurs when the Host sends an Audio Control specific Get Maximum Setting Attribute Control Transfer request to an Audio Device Control. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MAX type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlStatus and USB_DEVICE_ControlSend functions.

USB_DEVICE_AUDIO_EVENT_CONTROL_GET_RES	This event occurs when the Host sends an Audio Control specific Get Resolution Setting Attribute Control Transfer request to an Audio Device Control. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_RES type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlStatus and USB_DEVICE_ControlSend functions.
USB_DEVICE_AUDIO_EVENT_ENTITY_GET_MEM	This event occurs when the Host sends an Audio Entity specific Get Memory Space Attribute Control Transfer request to an Audio Device Entity. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MEM type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlStatus or USB_DEVICE_ControlSend functions.
USB_DEVICE_AUDIO_EVENT_ENTITY_GET_STAT	This event occurs when the Host sends a Audio Entity specific Get Status Control Transfer request to an Audio Device Entity. The pData member in the event handler will point to USB_DEVICE_AUDIO_EVENT_DATA_ENTITY_GET_STAT type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlSend and or USB_DEVICE_ControlStatus functions.

Description

USB Device Audio Function Driver Events

These events are specific to a USB Device Audio Function Driver instance. An event may have some data associated with it. This is provided to the event handling function. Each event description contains details about this event data (pData) and other parameters passed along with the event, to the event handler.

Events associated with the Audio Function Driver Specific Control Transfers require application response. The application should respond to these events by using the [USB_DEVICE_ControlReceive](#), [USB_DEVICE_ControlSend](#) and [USB_DEVICE_ControlStatus](#) functions.

Calling the [USB_DEVICE_ControlStatus](#) function with a [USB_DEVICE_CONTROL_STATUS_ERROR](#) will stall the control transfer request. The application would do this if the control transfer request is not supported. Calling the [USB_DEVICE_ControlStatus](#) function with a [USB_DEVICE_CONTROL_STATUS_OK](#) will complete the status stage of the control transfer request. The application would do this if the control transfer request is supported.

The following code shows an example of a possible event handling scheme.

```
// This code example shows all USB Audio Function Driver possible events and
// a possible scheme for handling these events. In this case event responses
// are not deferred.
```

```
void APP_USBDeviceAudioEventHandler
(
    USB_DEVICE_AUDIO_INDEX instanceIndex ,
    USB_DEVICE_AUDIO_EVENT event ,
    void * pData,
    uintptr_t context
)
{
    switch (event)
    {
        case USB_DEVICE_AUDIO_EVENT_READ_COMPLETE:

            // This event indicates that a Audio Read Transfer request
            // has completed. pData should be interpreted as a
            // USB_DEVICE_AUDIO_EVENT_DATA_READ_COMPLETE pointer type.
            // This contains the transfer handle of the read transfer
            // that completed and amount of data that was read.

            break;

        case USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE:

            // This event indicates that a Audio Write Transfer request
            // has completed. pData should be interpreted as a
```

```

// USB_DEVICE_AUDIO_EVENT_DATA_WRITE_COMPLETE pointer type.
// This contains the transfer handle of the write transfer
// that completed and amount of data that was written.

break;

case USB_DEVICE_AUDIO_EVENT_INTERFACE_SETTING_CHANGED:

    // This event occurs when the host sends Set Interface request
    // to the Audio USB Device. pData will be a pointer to a
    // USB_DEVICE_AUDIO_EVENT_DATA_INTERFACE_SETTING_CHANGED. This
    // contains the interface number whose setting was
    // changed and the index of the alternate setting.
    // The application should typically enable the audio function
    // if the interfaceAlternateSetting member of pData is greater
    // than 0.

break;

case USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_UNKNOWN:

    // This event indicates that the Audio function driver has
    // received a control transfer which it cannot decode. pData
    // will be a pointer to USB_SETUP_PACKET type pointer. The
    // application should decode the packet and take the required
    // action using the USB_DEVICE_ControlStatus(),
    // USB_DEVICE_ControlSend() and USB_DEVICE_ControlReceive()
    // functions.

break;

case USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_DATA_SENT:

    // This event indicates the data send request associated with
    // the latest USB_DEVICE_ControlSend() function was
    // completed. pData will be NULL.

case USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_DATA_RECEIVED:

    // This event indicates the data receive request associated with
    // the latest USB_DEVICE_ControlReceive() function was
    // completed. pData will be NULL. The application can either
    // acknowledge the received data or reject it by calling the
    // USB_DEVICE_ControlStatus() function.

break;

case USB_DEVICE_AUDIO_EVENT_CONTROL_SET_CUR:

    // This event indicates that the host is trying to set the
    // current setting attribute of a control. The data type will be
    // USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR type. The
    // application should identify the entity type based on the
    // entity ID. This mapping is application specific. The
    // following example assumes entity type to be a Feature Unit.

if(APP_EntityIdentify(((USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR *)pData)->entityID)
    == APP_AUDIO_ENTITY_FEATURE_UNIT)
{
    // The entity type is a feature unit. Type cast pData as
    // a USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST type and find
    // identify the control selector. This example shows the
    // handling for VOLUME control

    switch((USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST *)pData)->controlSelector)
    {
        case USB_AUDIO_VOLUME_CONTROL:
            // This means the host is trying to set the volume.
            // Use the USB_DEVICE_ControlReceive() function to

```

```

        // receive the volume settings for each channel.

        USB_DEVICE_ControlReceive(usbDeviceHandle, volumeSetting,
            ((USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST *)pData)->wLength);
    default:
        // Only volume control is supported in this example.
        // So everything else is stalled.
        USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_ERROR);
    }
}
break;

case USB_DEVICE_AUDIO_EVENT_CONTROL_GET_CUR:

    // This event indicates that the host is trying to get the
    // current setting attribute of a control. The data type will be
    // USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR type. The
    // application should identify the entity type based on the
    // entity ID. This mapping is application specific. The
    // following example assumes entity type to be a Feature Unit.

    if(APP_EntityIdentify(((USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR *)pData)->entityID)
        == APP_AUDIO_ENTITY_FEATURE_UNIT)
    {
        // The entity type is a feature unit. Type cast pData as
        // a USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST type and find
        // identify the control selector. This example shows the
        // handling for VOLUME control

        switch((USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST *)pData)->controlSelector)
        {
            case USB_AUDIO_VOLUME_CONTROL:
                // This means the host is trying to get the volume.
                // Use the USB_DEVICE_ControlReceive() function to
                // receive the volume settings for each channel.

                USB_DEVICE_ControlSend(usbDeviceHandle, volumeSetting,
                    ((USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST *)pData)->wLength);
            default:
                // Only volume control is supported in this example.
                // So everything else is stalled.
                USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_ERROR);
        }
    }
break;

case USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MAX:
case USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MIN:
case USB_DEVICE_AUDIO_EVENT_CONTROL_SET_RES:
case USB_DEVICE_AUDIO_EVENT_CONTROL_SET_MEM:
case USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MAX:
case USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MIN:
case USB_DEVICE_AUDIO_EVENT_CONTROL_GET_RES:
case USB_DEVICE_AUDIO_EVENT_CONTROL_GET_MEM:
    // In this example these request are not supported and so are
    // stalled.
    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_ERROR);
break;

default:
    break;
}

return(USB_DEVICE_AUDIO_EVENT_RESPONSE_NONE);
}

```

Remarks

The application can defer responses to events triggered by control transfers. In that, the application can respond to the control transfer event after exiting the event handler. This allows the application some time to obtain the response data rather than having to respond to the event

immediately. Note that a USB host will typically wait for an event response for a finite time duration before timing out and canceling the event and associated transactions. Even when deferring response, the application must respond promptly if such time-out have to be avoided.

USB_DEVICE_AUDIO_EVENT_DATA_READ_COMPLETE Structure

USB Device Audio Function Driver audio read and write complete event data.

File

[usb_device_audio_v1_0.h](#)

C

```
typedef struct {
    USB_DEVICE_AUDIO_TRANSFER_HANDLE handle;
    uint16_t length;
    uint8_t interfaceNum;
    USB_DEVICE_AUDIO_RESULT status;
} USB_DEVICE_AUDIO_EVENT_DATA_WRITE_COMPLETE, USB_DEVICE_AUDIO_EVENT_DATA_READ_COMPLETE;
```

Members

Members	Description
USB_DEVICE_AUDIO_TRANSFER_HANDLE handle;	Transfer handle associated with this <ul style="list-style-type: none">• read or write request
uint16_t length;	Indicates the amount of data (in bytes) that was <ul style="list-style-type: none">• read or written
uint8_t interfaceNum;	Interface Number
USB_DEVICE_AUDIO_RESULT status;	Completion status of the transfer

Description

USB Device Audio Function Driver Read and Write Complete Event Data.

This data type defines the data structure returned by the driver along with USB_DEVICE_AUDIO_EVENT_READ_COMPLETE, USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE, events.

Remarks

None.

USB_DEVICE_AUDIO_EVENT_DATA_WRITE_COMPLETE Structure

USB Device Audio Function Driver audio read and write complete event data.

File

[usb_device_audio_v1_0.h](#)

C

```
typedef struct {
    USB_DEVICE_AUDIO_TRANSFER_HANDLE handle;
    uint16_t length;
    uint8_t interfaceNum;
    USB_DEVICE_AUDIO_RESULT status;
} USB_DEVICE_AUDIO_EVENT_DATA_WRITE_COMPLETE, USB_DEVICE_AUDIO_EVENT_DATA_READ_COMPLETE;
```

Members

Members	Description
USB_DEVICE_AUDIO_TRANSFER_HANDLE handle;	Transfer handle associated with this <ul style="list-style-type: none">• read or write request
uint16_t length;	Indicates the amount of data (in bytes) that was <ul style="list-style-type: none">• read or written
uint8_t interfaceNum;	Interface Number
USB_DEVICE_AUDIO_RESULT status;	Completion status of the transfer

Description

USB Device Audio Function Driver Read and Write Complete Event Data.

This data type defines the data structure returned by the driver along with USB_DEVICE_AUDIO_EVENT_READ_COMPLETE, USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE, events.

Remarks

None.

USB_DEVICE_AUDIO_EVENT_HANDLER Type

USB Device Audio event handler function pointer type.

File

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_DEVICE_AUDIO_EVENT_RESPONSE (* USB_DEVICE_AUDIO_EVENT_HANDLER)(USB_DEVICE_AUDIO_INDEX
instanceIndex , USB_DEVICE_AUDIO_EVENT event , void * pData, uintptr_t context);
```

Description

USB Device Audio Event Handler Function Pointer Type.

This data type defines the required function signature USB Device Audio Function Driver event handling callback function. The application must register a pointer to an Audio Function Driver events handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event call backs from the Audio Function Driver. The function driver will invoke this function with event relevant parameters. The description of the event handler function parameters is given here.

instanceIndex - Instance index of the Audio Function Driver that generated the event.

event - Type of event generated.

pData - This parameter should be typecast to an event specific pointer type based on the event that has occurred. Refer to the [USB_DEVICE_AUDIO_EVENT](#) enumeration description for more details.

context - Value identifying the context of the application that registered the event handling function.

Remarks

The event handler function executes in the USB interrupt context when the USB Device Stack is configured for interrupt based operation. It is not advisable to call blocking functions or computationally intensive functions in the event handler. Where the response to a control transfer related event requires extended processing, the response to the control transfer should be deferred and the event handler should be allowed to complete execution.

USB_DEVICE_AUDIO_EVENT_RESPONSE Type

USB Device Audio Function Driver event callback response type.

File

[usb_device_audio_v1_0.h](#)

C

```
typedef void USB_DEVICE_AUDIO_EVENT_RESPONSE;
```

Description

USB Device Audio Function Driver Event Handler Response Type

This is the return type of the Audio Function Driver event handler.

Remarks

None.

USB_DEVICE_AUDIO_INDEX Type

USB Device Audio function driver index.

File

[usb_device_audio_v1_0.h](#)

C

```
typedef uintptr_t USB_DEVICE_AUDIO_INDEX;
```

Description

USB Device Audio Function Driver Index

This definition uniquely identifies a Audio Function Driver instance.

Remarks

None.

USB_DEVICE_AUDIO_RESULT Enumeration

USB Device Audio Function Driver USB Device Audio result enumeration.

File

[usb_device_audio_v1_0.h](#)

C

```
typedef enum {
    USB_DEVICE_AUDIO_RESULT_OK,
    USB_DEVICE_AUDIO_RESULT_ERROR_TRANSFER_QUEUE_FULL,
    USB_DEVICE_AUDIO_RESULT_ERROR_INSTANCE_INVALID,
    USB_DEVICE_AUDIO_RESULT_ERROR_INSTANCE_NOT_CONFIGURED,
    USB_DEVICE_AUDIO_RESULT_ERROR_PARAMETER_INVALID,
    USB_DEVICE_AUDIO_RESULT_ERROR_INVALID_INTERFACE_ID,
    USB_DEVICE_AUDIO_RESULT_ERROR_INVALID_BUFFER,
    USB_DEVICE_AUDIO_RESULT_ERROR_ENDPOINT_HALTED,
    USB_DEVICE_AUDIO_RESULT_ERROR_TERMINATED_BY_HOST,
    USB_DEVICE_AUDIO_RESULT_ERROR
} USB_DEVICE_AUDIO_RESULT;
```

Members

Members	Description
USB_DEVICE_AUDIO_RESULT_OK	The operation was successful
USB_DEVICE_AUDIO_RESULT_ERROR_TRANSFER_QUEUE_FULL	The transfer queue is full and no new transfers can be scheduled
USB_DEVICE_AUDIO_RESULT_ERROR_INSTANCE_INVALID	The specified instance is not provisioned in the system
USB_DEVICE_AUDIO_RESULT_ERROR_INSTANCE_NOT_CONFIGURED	The specified instance is not configured yet
USB_DEVICE_AUDIO_RESULT_ERROR_PARAMETER_INVALID	The event handler provided is NULL
USB_DEVICE_AUDIO_RESULT_ERROR_INVALID_INTERFACE_ID	Interface number passed to the read or write function is invalid.
USB_DEVICE_AUDIO_RESULT_ERROR_INVALID_BUFFER	A NULL buffer was specified in the read or write function
USB_DEVICE_AUDIO_RESULT_ERROR_ENDPOINT_HALTED	Transfer terminated because host halted the endpoint
USB_DEVICE_AUDIO_RESULT_ERROR_TERMINATED_BY_HOST	Transfer terminated by host because of a stall clear
USB_DEVICE_AUDIO_RESULT_ERROR	General Error

Description

USB Device Audio Function Driver USB Device Audio Result enumeration.

This enumeration lists the possible USB Device Audio Function Driver operation results.

Remarks

None.

USB_DEVICE_AUDIO_TRANSFER_HANDLE Type

USB Device Audio Function Driver transfer handle definition.

File

[usb_device_audio_v1_0.h](#)

C

```
typedef uintptr_t USB_DEVICE_AUDIO_TRANSFER_HANDLE;
```

Description

USB Device Audio Function Driver Transfer Handle Definition

This definition defines a USB Device Audio Function Driver Transfer Handle. A Transfer Handle is owned by the application but its value is modified by the [USB_DEVICE_AUDIO_Write](#), [USB_DEVICE_AUDIO_Read](#) functions. The transfer handle is valid for the life time of the transfer and expires when the transfer related event had occurred.

Remarks

None.

USB_DEVICE_AUDIO_FUNCTION_DRIVER Macro

USB Device Audio Function Driver function pointer.

File

[usb_device_audio_v1_0.h](#)

C

```
#define USB_DEVICE_AUDIO_FUNCTION_DRIVER
```

Description

USB Device Audio Function Driver Function Pointer

This is the USB Device Audio Function Driver function pointer. This should registered with the device layer in the function driver registration table.

Remarks

None.

USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_CUR Type

File

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_AUDIO_CONTROL_INTERFACE_REQUEST USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_CUR;
```

Description

This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_CUR.

USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MAX Type

File

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_AUDIO_CONTROL_INTERFACE_REQUEST USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MAX;
```

Description

This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MAX.

USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MEM Type

File

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_AUDIO_CONTROL_INTERFACE_REQUEST USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MEM;
```

Description

This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MEM.

USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MIN Type

File

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_AUDIO_CONTROL_INTERFACE_REQUEST USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MIN;
```

Description

This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MIN.

USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_RES Type**File**

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_AUDIO_CONTROL_INTERFACE_REQUEST USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_RES;
```

Description

This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_RES.

USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR Type

USB Device Audio Function Driver set and get request data.

File

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_AUDIO_CONTROL_INTERFACE_REQUEST USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR;
```

Description

USB Device Audio Function Driver Set and Get request data.

This data type defines the data structure returned by the driver along with the USB_DEVICE_AUDIO_EVENT_CONTROL_SET_XXX, USB_DEVICE_AUDIO_EVENT_ENTITY_SET_MEM, USB_DEVICE_AUDIO_EVENT_CONTROL_GET_XXX and USB_DEVICE_AUDIO_EVENT_ENTITY_GET_MEM events.

Remarks

None.

USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MAX Type**File**

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_AUDIO_CONTROL_INTERFACE_REQUEST USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MAX;
```

Description

This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MAX.

USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MEM Type**File**

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_AUDIO_CONTROL_INTERFACE_REQUEST USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MEM;
```

Description

This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MEM.

USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MIN Type**File**

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_AUDIO_CONTROL_INTERFACE_REQUEST USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MIN;
```

Description

This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MIN.

USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_RES Type**File**

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_AUDIO_CONTROL_INTERFACE_REQUEST USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_RES;
```

Description

This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_RES.

USB_DEVICE_AUDIO_EVENT_DATA_INTERFACE_SETTING_CHANGED Structure

USB Device Audio Function Driver alternate interface setting event data.

File

[usb_device_audio_v1_0.h](#)

C

```
typedef struct {
    uint8_t interfaceNumber;
    uint8_t interfaceAlternateSetting;
} USB_DEVICE_AUDIO_EVENT_DATA_INTERFACE_SETTING_CHANGED;
```

Members

Members	Description
uint8_t interfaceNumber;	Interface number of the interface who setting is to be changed
uint8_t interfaceAlternateSetting;	Alternate setting number

Description

USB Device Audio Function Driver Alternate Interface Setting Event Data.

This data type defines the data structure returned by the driver along with USB_DEVICE_AUDIO_EVENT_DATA_INTERFACE_SETTING_CHANGED.

Remarks

None.

USB_DEVICE_AUDIO_INIT Structure

USB Device Audio Function Driver initialization data structure.

File

[usb_device_audio_v1_0.h](#)

C

```
typedef struct {
    size_t queueSizeRead;
    size_t queueSizeWrite;
} USB_DEVICE_AUDIO_INIT;
```

Members

Members	Description
size_t queueSizeRead;	Size of the read queue for this instance <ul style="list-style-type: none">• of the Audio function driver
size_t queueSizeWrite;	Size of the write queue for this instance <ul style="list-style-type: none">• of the Audio function driver

Description

USB Device Audio Function Driver Initialization Data Structure

This data structure must be defined for every instance of the Audio Function Driver. It is passed to the Audio function driver, by the Device Layer, at the time of initialization. The funcDriverInit member of the Device Layer Function Driver registration table entry must point to this data structure for an instance of the Audio function driver.

Remarks

The queue sizes that are specified in this data structure are also affected by the [USB_DEVICE_AUDIO_QUEUE_DEPTH_COMBINED](#) configuration macro.

USB_DEVICE_AUDIO_EVENT_DATA_ENTITY_GET_STAT Type

File

[usb_device_audio_v1_0.h](#)

C

```
typedef USB_AUDIO_CONTROL_INTERFACE_REQUEST USB_DEVICE_AUDIO_EVENT_DATA_ENTITY_GET_STAT;
```

Description

This is type USB_DEVICE_AUDIO_EVENT_DATA_ENTITY_GET_STAT.

USB_DEVICE_AUDIO_TRANSFER_ABORT_NOTIFY Macro

USB Audio Transfer abort notification enable

File

[usb_device_audio_v1_0_config_template.h](#)

C

```
#define USB_DEVICE_AUDIO_TRANSFER_ABORT_NOTIFY
```

Description

USB Audio Transfer abort notification

This macro enabled USB Audio Transfer abort notifications. Whenever a scheduled transfer request is aborted due to Device Unplug or Host resets the device, the transfer complete event with status as aborted would be send to application's event handler.

Remarks

None.

Files

Files

Name	Description
usb_device_audio_v1_0.h	USB Device Audio function Driver Interface
usb_device_audio_v1_0_config_template.h	USB device Audio Class configuration definitions template

Description

This section lists the source and header files used by the library.

usb_device_audio_v1_0.h

USB Device Audio function Driver Interface

Enumerations

	Name	Description
	USB_DEVICE_AUDIO_EVENT	USB Device Audio Function Driver events.
	USB_DEVICE_AUDIO_RESULT	USB Device Audio Function Driver USB Device Audio result enumeration.

Functions

	Name	Description
≡◊	USB_DEVICE_AUDIO_EventHandlerSet	This function registers an event handler for the specified Audio function driver instance.
≡◊	USB_DEVICE_AUDIO_Read	This function requests a data read from the USB Device Audio Function Driver Layer.
≡◊	USB_DEVICE_AUDIO_TransferCancel	This function cancels a scheduled Audio Device data transfer.
≡◊	USB_DEVICE_AUDIO_Write	This function requests a data write to the USB Device Audio Function Driver Layer.

Macros

	Name	Description
	USB_DEVICE_AUDIO_EVENT_RESPONSE_NONE	USB Device Audio Function Driver event handler response type none.
	USB_DEVICE_AUDIO_FUNCTION_DRIVER	USB Device Audio Function Driver function pointer.
	USB_DEVICE_AUDIO_TRANSFER_HANDLE_INVALID	USB Device Audio Function Driver invalid transfer handle definition.

Structures

	Name	Description
	USB_DEVICE_AUDIO_EVENT_DATA_INTERFACE_SETTING_CHANGED	USB Device Audio Function Driver alternate interface setting event data.
	USB_DEVICE_AUDIO_EVENT_DATA_READ_COMPLETE	USB Device Audio Function Driver audio read and write complete event data.
	USB_DEVICE_AUDIO_EVENT_DATA_WRITE_COMPLETE	USB Device Audio Function Driver audio read and write complete event data.
	USB_DEVICE_AUDIO_INIT	USB Device Audio Function Driver initialization data structure.

Types

	Name	Description
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_CUR	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_CUR .
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MAX	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MAX .
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MEM	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MEM .
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MIN	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MIN .
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_RES	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_RES .
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR	USB Device Audio Function Driver set and get request data.
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MAX	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MAX .
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MEM	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MEM .
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MIN	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MIN .
	USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_RES	This is type USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_RES .
	USB_DEVICE_AUDIO_EVENT_DATA_ENTITY_GET_STAT	This is type USB_DEVICE_AUDIO_EVENT_DATA_ENTITY_GET_STAT .
	USB_DEVICE_AUDIO_EVENT_HANDLER	USB Device Audio event handler function pointer type.
	USB_DEVICE_AUDIO_EVENT_RESPONSE	USB Device Audio Function Driver event callback response type.
	USB_DEVICE_AUDIO_INDEX	USB Device Audio function driver index.
	USB_DEVICE_AUDIO_TRANSFER_HANDLE	USB Device Audio Function Driver transfer handle definition.

Description

USB Device Audio Function Driver Interface

This file describes the USB Device Audio Function Driver interface. This file should be included by the application if it needs to use the Audio Function Driver API.

File Name

`usb_device_audio.h`

Company

Microchip Technology Inc.

`usb_device_audio_v1_0_config_template.h`

USB device Audio Class configuration definitions template

Macros

Name	Description
<code>USB_DEVICE_AUDIO_INSTANCES_NUMBER</code>	Specifies the number of Audio Function Driver instances.
<code>USB_DEVICE_AUDIO_MAX_ALTERNATE_SETTING</code>	Specifies the maximum number of Alternate Settings per streaming interface.
<code>USB_DEVICE_AUDIO_MAX_STREAMING_INTERFACES</code>	Specifies the maximum number of Audio Streaming interfaces in an Audio Function Driver instance.
<code>USB_DEVICE_AUDIO_QUEUE_DEPTH_COMBINED</code>	Specifies the combined queue size of all Audio function driver instances.
<code>USB_DEVICE_AUDIO_TRANSFER_ABORT_NOTIFY</code>	USB Audio Transfer abort notification enable

Description

USB Device Audio Class Configuration Definitions

This file contains configurations macros needed to configure the Audio Function Driver. This file is a template file only. It should not be included by the application. The configuration macros defined in the file should be defined in the configuration specific `system_config.h`.

File Name

`usb_device_audio_v1_0_config_template.h`

Company

Microchip Technology Inc.

USB Audio 2.0 Device Library

This section describes the USB Audio 2.0 Device Library.

Introduction

This section provides information on library design, configuration, usage and the library interface for the USB Audio 2.0 Device Library.

Description

The MPLAB Harmony USB Audio 2.0 Device Library (also referred to as the Audio 2.0 Function Driver or library) features routines to implement a USB Audio 2.0 Device. Examples of Audio USB 2.0 Devices include USB Speakers, microphones, and voice telephony. The library provides a convenient abstraction of the USB Audio 2.0 Device specification and simplifies the implementation of USB Audio 2.0 Devices.

Using the Library

This topic describes the basic architecture of the Audio 2.0 Function Driver and provides information and examples on its use.

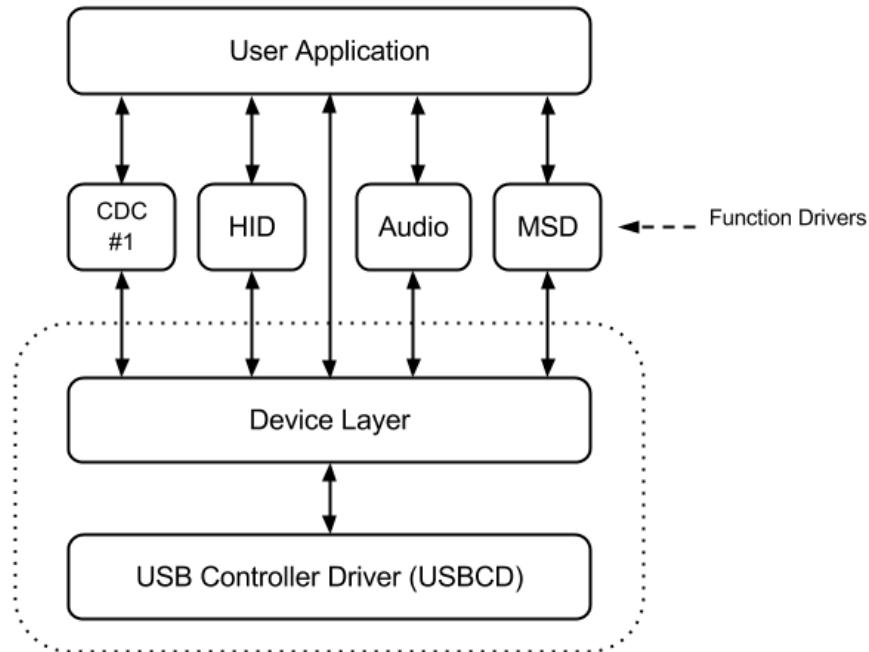
Abstraction Model

Describes the Abstraction Model of the USB Audio 2.0 Device Library.

Description

The Audio 2.0 Function Driver offers various services to the USB Audio 2.0 device to communicate with the host by abstracting USB specification details. It must be used along with the USB Device layer and USB controller to communicate with the USB host. Figure 1 shows a block diagram of the MPLAB Harmony USB Device Stack Architecture and where the Audio 2.0 Function Driver is placed.

Figure 1: USB Device Audio Device Driver



The USB controller driver takes the responsibility of managing the USB peripheral on the device. The USB Device Layer handles the device enumeration, etc. The USB Device Layer forwards all Audio-specific control transfers to the Audio 2.0 Function Driver. The Audio 2.0 Function Driver interprets the control transfers and requests application's intervention through event handlers and a well-defined set of API. The application must respond to the Audio events either in or out of the event handler. Some of these events are related to Audio 2.0 Device Class specific control transfers. The application must complete these control transfers within the timing constraints defined by USB.

Library Overview

The USB Audio 2.0 Device Library mainly interacts with the system, its clients and function drivers, as shown in the [Abstraction Model](#).

The library interface routines are divided into sub-sections, which address one of the blocks or the overall operation of the USB Audio 2.0 Device Library.

Library Interface Section	Description
Functions	Provides event handler, read/write, and transfer cancellation functions.

How the Library Works

Initializing the Library

Describes how the USB Audio 2.0 Device driver is initialized.

Description

The Audio 2.0 Function Driver instance for a USB device configuration is initialized by the Device Layer when the configuration is set by the host. This process does not require application intervention. Each instance of the Audio 2.0 Function Driver should be registered with the Device layer through the Device Layer Function Driver Registration Table. The Audio 2.0 Function Driver requires a initialization data structure to be specified. This is a **USB_DEVICE_AUDIO_2_0_INIT** data type that specifies the size of the read and write queues. The **funcDriverInit** member of the function driver registration table entry of the Audio 2.0 Function Driver instance should point to this initialization data structure. The **USB_DEVICE_AUDIO_2_0_FUNCTION_DRIVER** object is a global object provided by the Audio 2.0 Function Driver and provides the Device Layer with an entry point into the Audio 2.0 Function Driver. The following code shows an example of how the Audio 2.0 Function Driver can be registered with the Device Layer.

```
/* This code shows an example of how an Audio 2.0 function driver instances
 * can be registered with the Device Layer via the Device Layer Function Driver
 * Registration Table. In this case Device Configuration 1 consists of one
```

```

/* Audio 2.0 function driver instance. */

/* The Audio 2.0 Function Driver requires an initialization data structure that
 * specifies the read and write buffer queue sizes. Note that these settings are
 * also affected by the USB_DEVICE_AUDIO_QUEUE_DEPTH_COMBINED configuration
 * macro. */

const USB_DEVICE_AUDIO_2_0_INIT audioDeviceInit =
{
    .queueSizeRead = 1,
    .queueSizeWrite = 1
};

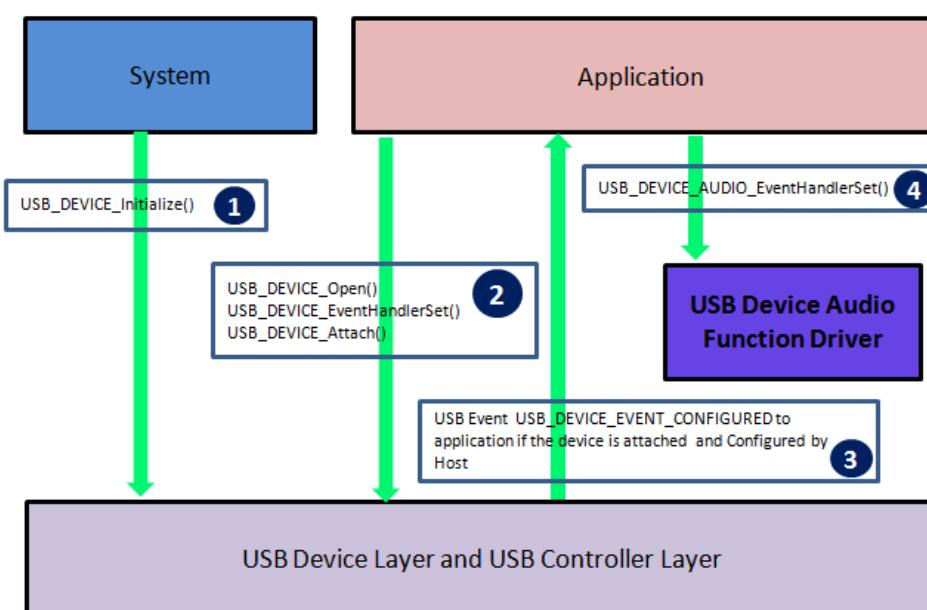
const USB_DEVICE_FUNC_REGISTRATION_TABLE funcRegistrationTable[1] =
{
    {
        .speed = USB_SPEED_FULL,                                     // Supported speed
        .configurationValue = 1,                                     // To be initialized for Configuration 1
        .interfaceNumber = 0,                                       // Starting interface number.
        .numberOfInterfaces = 2,                                     // Number of interfaces in this instance
        .funcDriverIndex = 0,                                       // Function Driver instance index is 0
        .funcDriverInit = &audioDeviceInit,                         // Function Driver does not need initialization
        .dataStructure
            .driver = USB_DEVICE_AUDIO_2_0_FUNCTION_DRIVER        // Pointer to Function Driver - Device Layer
        .interfaceFunctions
    },
};

data structure
    .driver = USB_DEVICE_AUDIO_2_0_FUNCTION_DRIVER        // Pointer to Function Driver - Device Layer
};

The following figure illustrates the typical sequence that is followed in the application when using the Audio 2.0 Function Driver.

```

Typical USB Audio 2.0 Device Sequence



1. Call set of APIs to initialize USB Device Layer (refer to the [USB Device Layer Library](#) section for details about these APIs).
2. The Device Layer provides a callback to the application for any USB Device events like attached, powered, configured, etc. The application should receive a callback with an event **USB_DEVICE_EVENT_CONFIGURED** to proceed.
3. Once the Device Layer is configured, the application needs to register a callback function with the Audio 2.0 Function Driver to receive Audio 2.0 Control transfers, and also other Audio 2.0 Function Driver events. Now the application can use Audio 2.0 Function Driver APIs to communicate with the USB Host.

Event Handling

Describes Audio 2.0 Function Driver event handler registration and event handling.

Description

Registering a Audio 2.0 Function Driver Callback Function

While creating a USB Audio 2.0 Device application, an event handler must be registered with the Device Layer (the Device Layer Event Handler) and every Audio 2.0 Function Driver instance (Audio 2.0 Function Driver Event Handler). The application needs to register the event handler with the Audio 2.0 Function Driver:

- For receiving Audio 2.0 Control Requests from Host like Volume Control, Mute Control, etc.
- For handling other events from USB Audio 2.0 Device Driver (e.g., Data Write Complete or Data Read Complete)

The event handler should be registered before the USB device layer acknowledges the SET CONFIGURATION request from the USB Host. To ensure this, the callback function should be set in the USB_DEVICE_EVENT_CONFIGURED event that is generated by the device layer. The following code example shows how this can be done.

```
/* This a sample Application Device Layer Event Handler
 * Note how the USB Audio 2.0 Device Driver callback function
 * USB_DEVICE_AUDIO_2_0_EventHandlerSet()
 * is registered in the USB_DEVICE_EVENT_CONFIGURED event. */

void APP_UsbDeviceEventCallBack( USB_DEVICE_EVENT event, void * pEventData, uintptr_t context )
{
    uint8_t * configuredEventData;
    switch( event )
    {
        case USB_DEVICE_EVENT_RESET:
            break;
        case USB_DEVICE_EVENT_DECONFIGURED:
            // USB device is reset or device is de-configured.
            // This means that USB device layer is about to de-initialize
            // all function drivers. So close handles to previously opened
            // function drivers.
            break;

        case USB_DEVICE_EVENT_CONFIGURED:
            /* check the configuration */
            /* Initialize the Application */
            configuredEventData = pEventData;
            if(*configuredEventData == 1)
            {
                USB_DEVICE_AUDIO_V2_EventHandlerSet
                (
                    0,
                    APP_USBDeviceAudioEventHandler ,
                    (uintptr_t)NULL
                );
                /* mark that set configuration is complete */
                appData.isConfigured = true;
            }
            break;

        case USB_DEVICE_EVENT_SUSPENDED:
            break;

        case USB_DEVICE_EVENT_POWER_DETECTED:
            /* Attach the device */
            USB_DEVICE_Attach (appData.usbDevHandle);
            break;

        case USB_DEVICE_EVENT_POWER_REMOVED:
            /* VBUS is not available. We can detach the device */
            USB_DEVICE_Detach(appData.usbDevHandle);
            break;

        case USB_DEVICE_EVENT_RESUMED:
        case USB_DEVICE_EVENT_ERROR:
        default:
            break;
    }
}
```

Event Handling

The Audio 2.0 Function Driver provides events to the application through the event handler function registered by the application. These events indicate:

- Completion of a read or a write data transfer
- Audio 2.0 Control Interface requests
- Completion of data and the status stages of Audio 2.0 Control Interface related control transfer

The Audio 2.0 Control Interface Request events and the related control transfer events typically require the application to respond with the Device Layer Control Transfer routines to complete the control transfer. Based on the generated event, the application may be required to:

- Respond with a [USB_DEVICE_ControlSend](#) function, which completes the data stage of a Control Read Transfer
- Respond with a [USB_DEVICE_ControlReceive](#) function, which provisions the data stage of a Control Write Transfer
- Respond with a [USB_DEVICE_ControlStatus](#) function, which completes the handshake stage of the Control Transfer. The application can either STALL or Acknowledge the handshake stage through the [USB_DEVICE_ControlStatus](#) function.

The following table shows the Audio 2.0 Function Driver Control Transfer related events and the required application control transfer actions.

Audio 2.0 Function Driver Control Transfer Event	Required Application Action
USB_DEVICE_AUDIO_V2_CUR_ENTITY_SETTINGS_RECEIVED	Identify the control type using the associated event data. If a data stage is expected, use the USB_DEVICE_ControlReceive function to receive expected data. If a data stage is not required or if the request is not supported, use the USB_DEVICE_ControlStatus function to Acknowledge or Stall the request.
USB_DEVICE_AUDIO_V2_RANGE_ENTITY_SETTINGS_RECEIVED	Identify the control type using the associated event data. If a data stage is expected, use the USB_DEVICE_ControlReceive function to receive expected data. If a data stage is not required or if the request is not supported, use the USB_DEVICE_ControlStatus function to Acknowledge or Stall the request.
USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA_RECEIVED	Acknowledge or stall using the USB_DEVICE_ControlStatus function.
USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA_SENT	Action not required.

The application must analyze the wIndex field of the event data (received with the control transfer event) to identify the entity that is being addressed. The application must be aware of all entities included in the application and their IDs. Once identified, the application can then type cast the event data to entity type the specific control request type. For example, if the Host sends a control request to set the clock source for the Audio 2.0 device, the following occurs in this order:

1. The Audio 2.0 Function Driver will generate a USB_DEVICE_AUDIO_V2_CUR_ENTITY_SETTINGS_RECEIVED event.
2. The application must type cast the event data to a USB_AUDIO_V2_CONTROL_INTERFACE_REQUEST type and check the entityID field.
3. The entityID field will be identified by the application as a Clock Source.
4. The application must now type cast the event data type as a USB_AUDIO_V2_CLOCKSOURCE_CONTROL_REQUEST data type and check the controlSelector field.
5. If the controlSelector field is AUDIO_V2_CS_SAM_FREQ_CONTROL, the application can then call the [USB_DEVICE_ControlReceive](#) function to receive the clock source.

Based on the type of event, the application should analyze the event data parameter of the event handler. This data member should be type cast to an event specific data type. The following table shows the event and the data type to use while type casting. Note that the event data member is not required for all events.

Audio 2.0 Function Driver Event	Related Event Data Type
USB_DEVICE_AUDIO_V2_CUR_ENTITY_SETTINGS_RECEIVED	USB_AUDIO_V2_CONTROL_INTERFACE_REQUEST*
USB_DEVICE_AUDIO_V2_RANGE_ENTITY_SETTINGS_RECEIVED	USB_AUDIO_V2_CONTROL_INTERFACE_REQUEST*
USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA_RECEIVED	NULL
USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA_SENT	NULL
USB_DEVICE_AUDIO_V2_EVENT_INTERFACE_SETTING_CHANGED	USB_DEVICE_AUDIO_V2_EVENT_DATA_SET_ALTERNATE_INTERFACE *
USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE	USB_DEVICE_AUDIO_V2_EVENT_DATA_READ_COMPLETE *
USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE	USB_DEVICE_AUDIO_V2_EVENT_DATA_READ_COMPLETE *

Handling Audio Control Requests:

When the Audio 2.0 Function Driver receives an Audio 2.0 Class Specific Control Transfer Request, it passes this control transfer to the application as a Audio 2.0 Function Driverevent. The following code example shows how to handle an Audio 2.0 Control request.

```

void APP_USBDeviceAudioEventHandler
(
    USB_DEVICE_AUDIO_V2_INDEX iAudio ,
    USB_DEVICE_AUDIO_V2_EVENT event ,
    void * pData,
    uintptr_t context
)
{
    USB_DEVICE_AUDIO_V2_EVENT_DATA_SET_ALTERNATE_INTERFACE * interfaceInfo;
    USB_DEVICE_AUDIO_V2_EVENT_DATA_READ_COMPLETE * readEventData;
    USB_DEVICE_AUDIO_V2_EVENT_DATA_WRITE_COMPLETE * writeEventData;
    USB_AUDIO_V2_CONTROL_INTERFACE_REQUEST* controlRequest;
    if ( iAudio == 0 )
    {
        switch (event)
        {
            case USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE:
                readEventData = (USB_DEVICE_AUDIO_V2_EVENT_DATA_READ_COMPLETE *)pData;
                //We have received an audio frame from the Host.
                //Now send this audio frame to Audio Codec for Playback.

                break;

            case USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE:
                break;

            case USB_DEVICE_AUDIO_V2_EVENT_INTERFACE_SETTING_CHANGED:
                //We have received a request from USB host to change the Interface-
                 //Alternate setting.
                interfaceInfo = (USB_DEVICE_AUDIO_V2_EVENT_DATA_SET_ALTERNATE_INTERFACE *)pData;
                appData.activeInterfaceAlternateSetting = interfaceInfo->interfaceAlternateSetting;
                appData.state = APP_USB_INTERFACE_ALTERNATE_SETTING_RCVD;

                break;
            case USB_DEVICE_AUDIO_V2_CUR_ENTITY_SETTINGS RECEIVED:
                controlRequest = (USB_AUDIO_V2_CONTROL_INTERFACE_REQUEST*) setupPkt;
                USB_AUDIO_V2_CLOCKSOURCE_CONTROL_REQUEST* clockSourceRequest;

                switch(controlRequest->entityID)
                {
                    case APP_ID_CLOCK_SOURCE:
                        clockSourceRequest = (USB_AUDIO_V2_CLOCKSOURCE_CONTROL_REQUEST*) controlRequest;
                        switch(clockSourceRequest->controlSelector)
                        {
                            case AUDIO_V2_CS_SAM_FREQ_CONTROL:
                                {
                                    if ((controlRequest->bmRequestType & 0x80) == 0)
                                    {
                                        //A control write transfer received from Host. Now receive data from Host.
                                        USB_DEVICE_ControlReceive(appData.usbDevHandle, (void *)&(appData.clockSource),
4 );
                                        appData.currentAudioControl = APP_USB_AUDIO_CLOCKSOURCE_CONTROL;
                                    }
                                }
                            else
                            {
                                /*Handle Get request*/
                                USB_DEVICE_ControlSend(appData.usbDevHandle, (void *)&(appData.clockSource), 4 );
                                appData.currentAudioControl = APP_USB_CONTROL_NONE;
                            }
                        }
                }
            }
        }
    }
}
break;

```

```

case USB_DEVICE_AUDIO_V2_RANGE_ENTITY_SETTINGS RECEIVED:
    break;

case USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA RECEIVED:
    USB_DEVICE_ControlStatus(appData.usbDevHandle, USB_DEVICE_CONTROL_STATUS_OK );
    switch (appData.currentAudioControl)
    {
        case APP_USB_AUDIO_MUTE_CONTROL:
        {
            appData.state = APP_MUTE_AUDIO_PLAYBACK;
            appData.currentAudioControl = APP_USB_CONTROL_NONE;
        }
        break;
        case APP_USB_AUDIO_CLOCKSOURCE_CONTROL:
        {
            // Handle Clock Source Control here.
            appData.state = APP_CLOCKSOURCE_SET;
            appData.currentAudioControl = APP_USB_CONTROL_NONE;
        }
        break;
        case APP_USB_AUDIO_CLOCKSELECT_CONTROL:
        {
            // Handle Clock Source Control here.
            appData.currentAudioControl = APP_USB_CONTROL_NONE;
        }
        break;
    }
    break;
case USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA SENT:
    break;
default:
    SYS_ASSERT ( false , "Invalid callback" );
    break;
} //end of switch ( callback )
}//end of if if ( iAudio == 0 )
}//end of function APP_AudioEventCallback

```

Transferring Data

Describes how to send/receive data to/from USB Host using this USB Audio 2.0 Device Driver.

Description

The USB Audio 2.0 Device Driver provides functions to send and receive data.

Receiving Data

The [USB_DEVICE_AUDIO_V2_Read](#) function schedules a data read. When the host transfers data to the device, the Audio 2.0 Function Driver receives the data and invokes the [USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE](#) event. This event indicates that audio data is now available in the application specified buffer.

The Audio 2.0 Function Driver supports buffer queuing. The application can schedule multiple read requests. Each request is assigned a unique buffer handle, which is returned with the [USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE](#) event. The application can use the buffer handle to track completion to queued requests. Using this feature allows the application to implement audio buffering schemes such as ping-pong buffering.

Sending Data

The [USB_DEVICE_AUDIO_V2_Write](#) schedules a data write. When the host sends a request for the data, the Audio 2.0 Function Driver transfers the data and invokes the [USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE](#) event.

The Audio 2.0 Function Driver supports buffer queuing. The application can schedule multiple write requests. Each request is assigned a unique buffer handle, which is returned with the [USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE](#) event. The application can use the buffer handle to track completion to queued requests. Using this feature allows the application to implement audio buffering schemes such as ping-pong buffering.

Configuring the Library

Macros

	Name	Description
	USB_DEVICE_AUDIO_V2_INSTANCES_NUMBER	Specifies the number of Audio 2.0 Function Driver instances.
	USB_DEVICE_AUDIO_V2_MAX_ALTERNATE_SETTING	Specifies the maximum number of Alternate Settings per streaming interface.
	USB_DEVICE_AUDIO_V2_MAX_STREAMING_INTERFACES	Specifies the maximum number of Audio 2.0 Streaming interfaces in an Audio 2.0 Function Driver instance.
	USB_DEVICE_AUDIO_V2_QUEUE_DEPTH_COMBINED	Specifies the combined queue size of all Audio 2.0 function driver instances.

Description

The application designer must specify the following configuration parameters while using the USB Audio 2.0 Device Driver. The configuration macros that implement these parameters must be located in the `system_config.h` file in the application project and a compiler include path (to point to the folder that contains this file) should be specified.

USB_DEVICE_AUDIO_V2_INSTANCES_NUMBER Macro

Specifies the number of Audio 2.0 Function Driver instances.

File

`usb_device_audio_v2_0_config_template.h`

C

```
#define USB_DEVICE_AUDIO_V2_INSTANCES_NUMBER
```

Description

USB device Audio 2.0 Maximum Number of instances

This macro defines the number of instances of the Audio 2.0 Function Driver. For example, if the application needs to implement two instances of the Audio 2.0 Function Driver (to create two composite Audio Device) on one USB Device, the macro should be set to 2. Note that implementing a USB Device that features multiple Audio 2.0 interfaces requires appropriate USB configuration descriptors.

Remarks

None.

USB_DEVICE_AUDIO_V2_MAX_ALTERNATE_SETTING Macro

Specifies the maximum number of Alternate Settings per streaming interface.

File

`usb_device_audio_v2_0_config_template.h`

C

```
#define USB_DEVICE_AUDIO_V2_MAX_ALTERNATE_SETTING
```

Description

Maximum number of Alternate Settings

This macro defines the maximum number of Alternate Settings per streaming interface. If the Audio 2.0 Device features multiple streaming interfaces, this configuration constant should be equal to the the maximum number of alternate required amongst the streaming interfaces.

Remarks

None.

USB_DEVICE_AUDIO_V2_MAX_STREAMING_INTERFACES Macro

Specifies the maximum number of Audio 2.0 Streaming interfaces in an Audio 2.0 Function Driver instance.

File

`usb_device_audio_v2_0_config_template.h`

C

```
#define USB_DEVICE_AUDIO_V2_MAX_STREAMING_INTERFACES
```

Description

Maximum Audio 2.0 Streaming Interfaces

This macro defines the maximum number of streaming interfaces in an Audio 2.0 Function Driver instance. In case of multiple Audio 2.0 Function Driver instances, this constant should be equal to the maximum number of interfaces amongst the Audio 2.0 Function Driver instances.

Remarks

None.

USB_DEVICE_AUDIO_V2_QUEUE_DEPTH_COMBINED Macro

Specifies the combined queue size of all Audio 2.0 function driver instances.

File

[usb_device_audio_v2_0_config_template.h](#)

C

```
#define USB_DEVICE_AUDIO_V2_QUEUE_DEPTH_COMBINED
```

Description

USB device Audio 2.0 Combined Queue Size

This macro defines the number of entries in all queues in all instances of the Audio 2.0 function driver. This value can be obtained by adding up the read and write queue sizes of each Audio Function driver instance. In a simple single instance USB Audio 2.0 device application, that does not require buffer queuing, the [USB_DEVICE_AUDIO_QUEUE_DEPTH_COMBINED](#) macro can be set to 2. Consider a case of a Audio 2.0 function driver instances, with has a read queue size of 2 and write queue size of 3, this macro should be set to 5 (2 + 3).

Remarks

None.

Building the Library

This section lists the files that are available in the USB Audio 2.0 Device Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/usb.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
usb_device_audio_v2_0.h	This header file must be included in every source file that needs to invoke USB Audio 2.0 Device Driver APIs.
usb_audio_v2_0.h	This header file must be included when the audio 2.0 descriptor macros are used.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/usb_device_audio_v2_0.c	This file contains all of functions, macros, definitions, variables, datatypes, etc., that are specific to the USB Audio v2.0 Specification implementation of the Audio 2.0 Function Driver.
/src/dynamic/usb_device_audio2_read_write.c	Contains implementation of the audio 2.0 function driver read and write functions.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB Audio 2.0 Device Library depends on the following modules:

- [USB Device Library](#)

Library Interface

This section describes the Application Programming Interface (API) functions of the USB Device Audio Library

Refer to each section for a detailed description.

a) Functions

Functions

	Name	Description
≡◊	USB_DEVICE_AUDIO_V2_EventHandlerSet	This function registers an event handler for the specified Audio function driver instance.
≡◊	USB_DEVICE_AUDIO_V2_Read	This function requests a data read from the USB Device Audio v2.0 Function Driver Layer.
≡◊	USB_DEVICE_AUDIO_V2_TransferCancel	This function cancels a scheduled Audio v2.0 Device data transfer.
≡◊	USB_DEVICE_AUDIO_V2_Write	This function requests a data write to the USB Device Audio v2.0 Function Driver Layer.

Description

[USB_DEVICE_AUDIO_V2_EventHandlerSet Function](#)

This function registers an event handler for the specified Audio function driver instance.

File

[usb_device_audio_v2_0.h](#)

C

```
USB_DEVICE_AUDIO_V2_RESULT USB_DEVICE_AUDIO_V2_EventHandlerSet(USB_DEVICE_AUDIO_V2_INDEX instanceIndex,
USB_DEVICE_AUDIO_V2_EVENT_HANDLER eventHandler, uintptr_t context);
```

Returns

- [USB_DEVICE_AUDIO_V2_RESULT_OK](#) - The operation was successful
- [USB_DEVICE_AUDIO_V2_RESULT_ERROR_INSTANCE_INVALID](#) - The specified instance does not exist.
- [USB_DEVICE_AUDIO_V2_RESULT_ERROR_PARAMETER_INVALID](#) - The eventHandler parameter is NULL

Description

This function registers a event handler for the specified Audio function driver instance. This function should be called by the application when it receives a SET CONFIGURATION event from the device layer. The application must register an event handler with the function driver in order to receive and respond to function driver specific events and control transfers. If the event handler is not registered, the device layer will stall function driver specific commands and the USB device may not function.

Remarks

None.

Preconditions

This function should be called when the function driver has been initialized as a result of a set configuration.

Example

```
// The following code shows an example registering an event handler. The
// application specifies the context parameter as a pointer to an
// application object (appObject) that should be associated with this
// instance of the Audio function driver.

USB_DEVICE_AUDIO_V2_RESULT result;

USB_DEVICE_AUDIO_V2_EVENT_RESPONSE APP_USBDeviceAUDIOEventHandler
(
    USB_DEVICE_AUDIO_V2_INDEX instanceIndex ,
    USB_DEVICE_AUDIO_V2_EVENT event ,
    void* pData,
    uintptr_t context
)
{
    // Event Handling comes here

    switch(event)
    {
        ...
    }

    return(USB_DEVICE_AUDIO_V2_EVENT_RESPONSE_NONE);
}

result = USB_DEVICE_AUDIO_V2_EventHandlerSet ( USB_DEVICE_AUDIO_V2_INSTANCE_0 ,
    &APP_USBDeviceAUDIOEventHandler, (uintptr_t) &appObject);

if(USB_DEVICE_AUDIO_V2_RESULT_OK != result)
{
    // Do error handling here
}
}
```

Parameters

Parameters	Description
instance	Instance of the Audio v2.0 Function Driver.
eventHandler	A pointer to event handler function.
context	Application specific context that is returned in the event handler.

Function

```
USB_DEVICE_AUDIO_V2_RESULT USB_DEVICE_AUDIO_V2_EventHandlerSet
(
    USB_DEVICE_AUDIO_V2_INDEX instance ,
    USB_DEVICE_AUDIO_V2_EVENT_HANDLER eventHandler ,
    uintptr_t context
);
```

USB_DEVICE_AUDIO_V2_Read Function

This function requests a data read from the USB Device Audio v2.0 Function Driver Layer.

File

[usb_device_audio_v2_0.h](#)

C

```
USB_DEVICE_AUDIO_V2_RESULT USB_DEVICE_AUDIO_V2_Read(USB_DEVICE_AUDIO_V2_INDEX instanceIndex,
    USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE* transferHandle, uint8_t interfaceNumber, void * data, size_t size);
```

Returns

- `USB_DEVICE_AUDIO_V2_RESULT_OK` - The read request was successful. transferHandle contains a valid transfer handle.
- `USB_DEVICE_AUDIO_V2_RESULT_ERROR_TRANSFER_QUEUE_FULL` - internal request queue is full. The read request could not be

added.

- **USB_DEVICE_AUDIO_V2_RESULT_ERROR_INSTANCE_NOT_CONFIGURED** - The specified instance is not configured yet.
- **USB_DEVICE_AUDIO_V2_RESULT_ERROR_INSTANCE_INVALID** - The specified instance was not provisioned in the application and is invalid.

Description

This function requests a data read from the USB Device Audio v2.0 Function Driver Layer. The function places a request with the driver, the request will get serviced as data is made available by the USB Host. A handle to the request is returned in the transferHandle parameter. The termination of the request is indicated by the **USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE** event. The amount of data read and the transfer handle associated with the request is returned along with the event. The transfer handle expires when event handler for the **USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE** exits. If the read request could not be accepted, the function returns an error code and transferHandle will contain the value **USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE_INVALID**.

Remarks

While using the Audio Function Driver with PIC32MZ USB module, the audio buffer provided to the **USB_DEVICE_AUDIO_V2_Read** function should be placed in coherent memory and aligned at a 16 byte boundary. This can be done by declaring the buffer using the **_attribute_((coherent, aligned(16)))** attribute, as shown in the following example:

```
uint8_t data[256] __attribute__((coherent, aligned(16)));
```

Preconditions

The function driver should have been configured.

Example

```
// Shows an example of how to read. This assumes that
// device had been configured. The example attempts to read
// data from interface 1.

USB_DEVICE_AUDIO_V2_INDEX instanceIndex;
USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE transferHandle;
unit8_t interfaceNumber;
// Use this attribute for PIC32MZ __attribute__((coherent, aligned(16)))
unit8_t rxBuffer[192];
USB_DEVICE_AUDIO_V2_RESULT readRequestResult;

instanceIndex = 0; //specify the Audio v2.0 Function driver instance number.
interfaceNumber = 1; //Specify the Audio v2.0 Streaming interface number.

readRequestResult = USB_DEVICE_AUDIO_V2_Read ( instanceIndex, &transferHandle,
                                              interfaceNumber, &rxBuffer, 192);

if(USB_DEVICE_AUDIO_V2_RESULT_OK != readRequestResult)
{
    //Do Error handling here
}

// The completion of the read request will be indicated by the
// USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE event. The transfer handle
// and the amount of data read will be returned along with the
// event.
```

Parameters

Parameters	Description
instance	USB Device Audio v2.0 Function Driver instance.
transferHandle	Pointer to a USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE type of variable. This variable will contain the transfer handle in case the read request was successful.
interfaceNum	The USB Audio v2.0 streaming interface number on which read request is to be placed.
data	pointer to the data buffer where read data will be stored. In case of PIC32MZ device, this buffer should be located in coherent memory and should be aligned a 16 byte boundary.
size	Size of the data buffer. Refer to the description section for more details on how the size affects the transfer.

Function

```
USB_DEVICE_AUDIO_V2_RESULT USB_DEVICE_AUDIO_V2_Read
(
    USB_DEVICE_AUDIO_V2_INDEX instanceIndex ,
```

```
USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE* transferHandle,
uint8_t interfaceNum ,
void * data ,
size_t size
);
```

USB_DEVICE_AUDIO_V2_TransferCancel Function

This function cancels a scheduled Audio v2.0 Device data transfer.

File

[usb_device_audio_v2_0.h](#)

C

```
USB_DEVICE_AUDIO_V2_RESULT USB_DEVICE_AUDIO_V2_TransferCancel(USB_DEVICE_AUDIO_V2_INDEX instanceIndex,
USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE transferHandle);
```

Returns

- `USB_DEVICE_AUDIO_V2_RESULT_OK` - The transfer will be canceled completely or partially.
- `USB_DEVICE_AUDIO_V2_RESULT_ERROR` - The transfer could not be canceled because it has either completed, the transfer handle is invalid or the last transaction is in progress.

Description

This function cancels a scheduled Audio v2.0 Device data transfer. The transfer could have been scheduled using the `USB_DEVICE_AUDIO_V2_Read`, `USB_DEVICE_AUDIO_V2_Write`, or the `USB_DEVICE_AUDIO_V2_SerialStateNotificationSend` function. If a transfer is still in the queue and its processing has not started, the transfer is canceled completely. A transfer that is in progress may or may not get canceled depending on the transaction that is presently in progress. If the last transaction of the transfer is in progress, the transfer will not be canceled. If it is not the last transaction in progress, the in-progress will be allowed to complete. Pending transactions will be canceled. The first transaction of an in progress transfer cannot be canceled.

Remarks

None.

Preconditions

The USB Device should be in a configured state.

Example

```
// The following code example cancels an Audio transfer.

USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE transferHandle;
USB_DEVICE_AUDIO_V2_RESULT result;

result = USB_DEVICE_AUDIO_V2_TransferCancel(instanceIndex, transferHandle);

if(USB_DEVICE_AUDIO_V2_RESULT_OK == result)
{
    // The transfer cancellation was either completely or
    // partially successful.
}
```

Parameters

Parameters	Description
<code>instanceIndex</code>	AUDIO v2.0 Function Driver instance index.
<code>transferHandle</code>	Transfer handle of the transfer to be canceled.

Function

```
USB_DEVICE_AUDIO_V2_RESULT USB_DEVICE_AUDIO_V2_TransferCancel
(
    USB_DEVICE_AUDIO_V2_INDEX instanceIndex,
    USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE transferHandle
);
```

USB_DEVICE_AUDIO_V2_Write Function

This function requests a data write to the USB Device Audio v2.0 Function Driver Layer.

File

[usb_device_audio_v2_0.h](#)

C

```
USB_DEVICE_AUDIO_V2_RESULT USB_DEVICE_AUDIO_V2_Write(USB_DEVICE_AUDIO_V2_INDEX instanceIndex,
USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE * transferHandle, uint8_t interfaceNumber, void * data, size_t size);
```

Returns

- USB_DEVICE_AUDIO_V2_RESULT_OK - The read request was successful. transferHandle contains a valid transfer handle.
- USB_DEVICE_AUDIO_V2_RESULT_ERROR_TRANSFER_QUEUE_FULL - internal request queue is full. The write request could not be added.
- USB_DEVICE_AUDIO_V2_RESULT_ERROR_INSTANCE_NOT_CONFIGURED - The specified instance is not configured yet.
- USB_DEVICE_AUDIO_V2_RESULT_ERROR_INSTANCE_INVALID - The specified instance was not provisioned in the application and is invalid.

Description

This function requests a data write to the USB Device Audio v2.0 Function Driver Layer. The function places a request with the driver, the request will get serviced as data is requested by the USB Host. A handle to the request is returned in the transferHandle parameter. The termination of the request is indicated by the USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE event. The amount of data written and the transfer handle associated with the request is returned along with the event in writeCompleteData member of the pData parameter in the event handler.

The transfer handle expires when event handler for the USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE exits. If the write request could not be accepted, the function returns an error code and transferHandle will contain the value

[USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE_INVALID](#).

Remarks

While using the Audio Function Driver with the PIC32MZ USB module, the audio buffer provided to the USB_DEVICE_AUDIO_V2_Write function should be placed in coherent memory and aligned at a 16 byte boundary. This can be done by declaring the buffer using the

`__attribute__((coherent, aligned(16)))` attribute. An example is shown here

```
uint8_t data[256] __attribute__((coherent, aligned(16)));
```

Preconditions

The function driver should have been configured.

Example

```
// Shows an example of how to write audio data to the audio streaming
// interface. This assumes that device is configured and the audio
// streaming interface is 1.

USB_DEVICE_AUDIO_V2_INDEX instanceIndex;
USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE transferHandle;
unit8_t interfaceNumber;
unit8_t txBuffer[192]; // Use this attribute for PIC32MZ __attribute__((coherent, aligned(16)))
USB_DEVICE_AUDIO_V2_RESULT writeRequestResult;

instanceIndex = 0; //specify the Audio Function driver instance number.
interfaceNumber = 1; //Specify the Audio Streaming interface number.

writeRequestResult = USB_DEVICE_AUDIO_V2_Write ( instanceIndex, &transferHandle,
                                                interfaceNumber, &txBuffer, 192);

if(USB_DEVICE_AUDIO_V2_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}

// The completion of the write request will be indicated by the
// USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE event. The transfer handle
// and transfer size is provided along with this event.
```

Parameters

Parameters	Description
instance	USB Device Audio Function Driver instance.
transferHandle	Pointer to a USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE type of variable. This variable will contain the transfer handle in case the write request was successful.
interfaceNum	The USB Audio streaming interface number on which the write request is to placed.
data	pointer to the data buffer contains the data to be written. In case of PIC32MZ device, this buffer should be located in coherent memory and should be aligned a 16 byte boundary.
size	Size of the data buffer.

Function

```
USB\_DEVICE\_AUDIO\_V2\_RESULT USB_DEVICE_AUDIO_V2_Write
(
    USB\_DEVICE\_AUDIO\_V2\_INDEX instance ,
    USB\_DEVICE\_AUDIO\_V2\_TRANSFER\_HANDLE* transferHandle,
    uint8_t interfaceNum ,
    void * data ,
    size_t size
);
```

b) Data Types and Constants

Enumerations

	Name	Description
	USB_DEVICE_AUDIO_V2_EVENT	USB Device Audio v2.0 Function Driver events.
	USB_DEVICE_AUDIO_V2_RESULT	USB Device Audio Function Driver USB Device Audio v2.0 result enumeration.

Macros

	Name	Description
	USB_DEVICE_AUDIO_V2_EVENT_RESPONSE_NONE	USB Device Audio v2.0 Function Driver event handler response type none.
	USB_DEVICE_AUDIO_V2_FUNCTION_DRIVER	USB Device Audio v2.0 Function Driver function pointer.
	USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE_INVALID	USB Device Audio v2.0 Function Driver Invalid Transfer Handle Definition.

Structures

	Name	Description
	USB_DEVICE_AUDIO_V2_EVENT_DATA_READ_COMPLETE	USB Device Audio Function Driver Audio v2.0 read and write complete event data.
	USB_DEVICE_AUDIO_V2_EVENT_DATA_SET_ALTERNATE_INTERFACE	USB Device Audio v2.0 Function Driver alternate interface setting event data.
	USB_DEVICE_AUDIO_V2_EVENT_DATA_WRITE_COMPLETE	USB Device Audio Function Driver Audio v2.0 read and write complete event data.
	USB_DEVICE_AUDIO_V2_INIT	USB Device Audio v2.0 Function Driver initialization data structure.

Types

	Name	Description
	USB_DEVICE_AUDIO_V2_EVENT_HANDLER	USB Device Audio v2.0 Event Handler Function Pointer Type.
	USB_DEVICE_AUDIO_V2_EVENT_RESPONSE	USB Device Audio v2.0 Function Driver event callback response type.
	USB_DEVICE_AUDIO_V2_INDEX	USB Device Audio v2.0 Function Driver index.
	USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE	USB Device Audio v2.0 Function Driver Transfer Handle Definition.

Description

USB_DEVICE_AUDIO_V2_EVENT Enumeration

USB Device Audio v2.0 Function Driver events.

File

[usb_device_audio_v2_0.h](#)

C

```
typedef enum {
    USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE,
    USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE,
    USB_DEVICE_AUDIO_V2_EVENT_INTERFACE_SETTING_CHANGED,
    USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA_RECEIVED,
    USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA_SENT,
    USB_DEVICE_AUDIO_V2_CUR_ENTITY_SETTINGS_RECEIVED,
    USB_DEVICE_AUDIO_V2_RANGE_ENTITY_SETTINGS_RECEIVED,
    USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_UNKNOWN
} USB_DEVICE_AUDIO_V2_EVENT;
```

Members

Members	Description
USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE	This event occurs when a write operation scheduled by calling the USB_DEVICE_AUDIO_V2_Write function has completed. The pData member in the event handler will point to USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE_DATA type.
USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE	This event occurs when a read operation scheduled by calling the USB_DEVICE_AUDIO_V2_Read function has completed. The pData member in the event handler will point to USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE_DATA type.
USB_DEVICE_AUDIO_V2_EVENT_INTERFACE_SETTING_CHANGED	This event occurs when the Host requests the Audio v2.0 USB Device to set an alternate setting on an interface present in this audio function. An Audio v2.0 USB Device will typically feature a default interface setting and one or more alternate interface settings. The pData member in the event handler will point to the USB_DEVICE_AUDIO_V2_EVENT_DATA_INTERFACE_SETTING_CHANGED type. This contains the index of the interface whose setting must be changed and the index of the alternate setting. The application may enable or disable audio functions based on the interface setting.
USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA_RECEIVED	This event occurs when the data stage of a control write transfer has completed. This would occur after the application would respond with a USB_DEVICE_ControlReceive function, which may possibly have been called in response to a USB_DEVICE_AUDIO_V2_EVENT_ENTITY_SETTINGS_RECEIVED event. This event notifies the application that the data is received from Host and is available at the location passed by the USB_DEVICE_ControlReceive function. If the received data is acceptable to the application, it should acknowledge the data by calling the USB_DEVICE_ControlStatus function with a USB_DEVICE_CONTROL_STATUS_OK flag. The application can reject the received data by calling the USB_DEVICE_ControlStatus function with the USB_DEVICE_CONTROL_STATUS_ERROR flag. The pData parameter will be NULL.
USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA_SENT	This event occurs when the data stage of a control read transfer has completed. This would occur when the application has called the USB_DEVICE_ControlSend function to complete the data stage of a control transfer. The event indicates that the data has been transmitted to the host. The pData parameter will be NULL.

USB_DEVICE_AUDIO_V2_CUR_ENTITY_SETTINGS RECEIVED	This event occurs when the Host sends an Audio 2.0 Control specific Set Current Setting Attribute Control Transfer request to an Audio Device Control. The pData member in the event handler will point to type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with the USB_DEVICE_ControlStatus and USB_DEVICE_ControlReceive functions.
USB_DEVICE_AUDIO_V2_RANGE_ENTITY_SETTINGS RECEIVED	This event occurs when the Host sends an Audio 2.0 Control specific Set Range Setting Attribute Control Transfer request to an Audio Device Control. The pData member in the event handler will point to type. The application must use the entityID, interface, endpoint and the wValue field in the event data to determine the entity and control type and then respond to the control transfer with a USB_DEVICE_ControlStatus and USB_DEVICE_ControlReceive functions.
USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_UNKNOWN	This event occurs when the Audio v2.0 function driver receives a control transfer request that could not be decoded by Audio Function driver. The pData parameter will point to a USB_SETUP_PACKET type containing the setup packet. The application must analyze this Setup packet and use the USB_DEVICE_ControlSend , USB_DEVICE_ControlReceive , or the USB_DEVICE_ControlStatus function to advance the control transfer or complete it.

Description

USB Device Audio v2.0 Function Driver Events

These events are specific to a USB Device Audio v2.0 Function Driver instance. An event may have some data associated with it. This is provided to the event handling function. Each event description contains details about this event data (pData) and other parameters passed along with the event, to the event handler.

Events associated with the Audio v2.0 Function Driver Specific Control Transfers require application response. The application should respond to these events by using the [USB_DEVICE_ControlReceive\(\)](#), [USB_DEVICE_ControlSend\(\)](#) and [USB_DEVICE_ControlStatus\(\)](#) functions.

Calling the [USB_DEVICE_ControlStatus\(\)](#) function with a [USB_DEVICE_CONTROL_STATUS_ERROR](#) will stall the control transfer request. The application would do this if the control transfer request is not supported. Calling the [USB_DEVICE_ControlStatus\(\)](#) function with a [USB_DEVICE_CONTROL_STATUS_OK](#) will complete the status stage of the control transfer request. The application would do this if the control transfer request is supported.

The following code shows an example of a possible event handling scheme.

```
// This code example shows all USB Audio v2.0 Function Driver possible
// events and a possible scheme for handling these events.
// In this case event responses are not deferred.

void APP_USBDeviceAudioEventHandler
(
    USB_DEVICE_AUDIO_V2_INDEX instanceIndex ,
    USB_DEVICE_AUDIO_V2_EVENT event ,
    void * pData,
    uintptr_t context
)
{
    switch (event)
    {
        case USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE:

            // This event indicates that a Audio v2.0 Read Transfer request
            // has completed. pData should be interpreted as a
            // USB\_DEVICE\_AUDIO\_V2\_EVENT\_DATA\_READ\_COMPLETE pointer type.
            // This contains the transfer handle of the read transfer
            // that completed and amount of data that was read.

            break;

        case USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE:

            // This event indicates that a Audio v2.0 Write Transfer request
            // has completed. pData should be interpreted as a
            // USB\_DEVICE\_AUDIO\_V2\_EVENT\_DATA\_WRITE\_COMPLETE pointer type.
            // This contains the transfer handle of the write transfer
            // that completed and amount of data that was written.
    }
}
```

```

break;

case USB_DEVICE_AUDIO_V2_EVENT_INTERFACE_SETTING_CHANGED:

    // This event occurs when the host sends Set Interface request
    // to the Audio v2.0 USB Device. pData will be a pointer to a
    // USB_DEVICE_AUDIO_V2_EVENT_DATA_INTERFACE_SETTING_CHANGED. This
    // contains the interface number whose setting was
    // changed and the index of the alternate setting.
    // The application should typically enable the audio function
    // if the interfaceAlternateSetting member of pData is greater
    // than 0.

break;

case USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_UNKNOWN:

    // This event indicates that the Audio v2.0 function driver has
    // received a control transfer which it cannot decode. pData
    // will be a pointer to USB_SETUP_PACKET type pointer. The
    // application should decode the packet and take the required
    // action using the USB_DEVICE_ControlStatus(),
    // USB_DEVICE_ControlSend() and USB_DEVICE_ControlReceive()
    // functions.

break;

case USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA_SENT:

    // This event indicates the data send request associated with
    // the latest USB_DEVICE_ControlSend() function was
    // completed. pData will be NULL.

case USB_DEVICE_AUDIO_V2_EVENT_CONTROL_TRANSFER_DATA_RECEIVED:

    // This event indicates the data receive request associated with
    // the latest USB_DEVICE_ControlReceive() function was
    // completed. pData will be NULL. The application can either
    // acknowledge the received data or reject it by calling the
    // USB_DEVICE_ControlStatus() function.

break;

case USB_DEVICE_AUDIO_V2_CUR_ENTITY_SETTINGS RECEIVED:
    // This event indicates the Current entity request has been
    // received.
    USB_AUDIO_CONTROL_INTERFACE_REQUEST* controlRequest;
    controlRequest = (USB_AUDIO_CONTROL_INTERFACE_REQUEST*) setupPkt;
    switch(controlRequest->entityID)
    {
        case APP_ID_CLOCK_SOURCE:
            USB_AUDIO_CLOCKSOURCE_CONTROL_REQUEST*
                clockSourceRequest;
            clockSourceRequest =
                (USB_AUDIO_CLOCKSOURCE_CONTROL_REQUEST*) controlRequest;

            if (clockSourceRequest->bRequest == CUR)
            {
                switch(clockSourceRequest->controlSelector)
                {
                    case CS_SAM_FREQ_CONTROL:
                    {
                        if (((controlRequest->bmRequestType & 0x80)
                            == 0)

                        {
                            //A control write transfer received
                            //from Host. Now receive data from Host.
                            USB_DEVICE_ControlReceive(

```

```

                appData.usbDevHandle,
        void *) &(appData.clockSource),
        4 );
    appData.currentAudioControl =
        APP_USB_AUDIO_CLOCKSOURCE_CONTROL;
}
else
{
    //Handle Get request
    USB_DEVICE_ControlSend(
        appData.usbDevHandle,
        (void *)&(appData.clockSource),
        4 );
    appData.currentAudioControl =
        APP_USB_CONTROL_NONE;
}
}
break;

case CS_CLOCK_VALID_CONTROL:
{
    if ((controlRequest->bmRequestType & 0x80)
        == 0x80)
    {
        //Handle Get request
        USB_DEVICE_ControlSend(
            appData.usbDevHandle,
            (void *)&(appData.clockValid),
            1);
    }
    else
    {
        USB_DEVICE_ControlStatus(
            appData.usbDevHandle,
            USB_DEVICE_CONTROL_STATUS_ERROR);

    }
}
break;

default:
    //This USB Audio Speaker application does
    //not support any other feature unit request
    //from Host. So Stall if any other feature
    //unit request received from Host.
    USB_DEVICE_ControlStatus (
        appData.usbDevHandle,
        USB_DEVICE_CONTROL_STATUS_ERROR);
break;
}

case USB_DEVICE_AUDIO_V2_RANGE_ENTITY_SETTINGS RECEIVED:
// This event indicates the Range entity request has been
// received.
USB_AUDIO_CONTROL_INTERFACE_REQUEST* controlRequest;
controlRequest = (USB_AUDIO_CONTROL_INTERFACE_REQUEST*)setupPkt;
switch(controlRequest->entityID)
{
    case APP_ID_CLOCK_SOURCE:
        USB_AUDIO_CLOCKSOURCE_CONTROL_REQUEST* clockSourceRequest;
        clockSourceRequest =
        (USB_AUDIO_CLOCKSOURCE_CONTROL_REQUEST*) controlRequest;
        if (clockSourceRequest->bRequest == RANGE)
        {
            switch(clockSourceRequest->controlSelector)
            {
                case CS_SAM_FREQ_CONTROL:

```

```

    {
        if ((controlRequest->bmRequestType & 0x80)
            == 0x80)
        {
            //A control read transfer received from
            // Host. Now send data to Host.
            USB_DEVICE_ControlSend(
                appData.usbDevHandle,
                void * )&(appData.clockSourceRange),
                sizeof(appData.clockSourceRange));
        }
        else
        {
            //Handle Get request
            // USB_DEVICE_ControlReceive(
                appData.usbDevHandle,
                (void *)&(appData.clockSourceRange[0]),
                sizeof(appData.clockSourceRange) );
            USB_DEVICE_ControlStatus(
                appData.usbDevHandle,
                USB_DEVICE_CONTROL_STATUS_ERROR);
        }
        break;
    }
    default:
        //This USB Audio Speaker application does
        // not support any other feature unit
        // request from Host. So Stall if any other
        // feature unit request received from Host.
        USB_DEVICE_ControlStatus (
            appData.usbDevHandle,
            USB_DEVICE_CONTROL_STATUS_ERROR);
        break;
    }
}
}
}

```

Remarks

The application can defer responses to events triggered by control transfers. In that, the application can respond to the control transfer event after exiting the event handler. This allows the application some time to obtain the response data rather than having to respond to the event immediately. Note that a USB host will typically wait for an event response for a finite time duration before timing out and canceling the event and associated transactions. Even when deferring response, the application must respond promptly if such time-out have to be avoided.

USB_DEVICE_AUDIO_V2_EVENT_DATA_READ_COMPLETE Structure

USB Device Audio Function Driver Audio v2.0 read and write complete event data.

File

[usb_device_audio_v2_0.h](#)

C

```

typedef struct {
    USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE handle;
    uint16_t length;
    uint8_t interfaceNum;
    USB_DEVICE_AUDIO_V2_RESULT status;
} USB_DEVICE_AUDIO_V2_EVENT_DATA_WRITE_COMPLETE, USB_DEVICE_AUDIO_V2_EVENT_DATA_READ_COMPLETE;

```

Members

Members	Description
USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE handle;	Transfer handle associated with this • read or write request

uint16_t length;	Indicates the amount of data (in bytes) that was <ul style="list-style-type: none">• read or written
uint8_t interfaceNum;	Interface Number
USB_DEVICE_AUDIO_V2_RESULT status;	Completion status of the transfer

Description

USB Device Audio v2.0 Function Driver Read and Write Complete Event Data.

This data type defines the data structure returned by the driver along with USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE, USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE, events.

Remarks

None.

USB_DEVICE_AUDIO_V2_EVENT_DATA_SET_ALTERNATE_INTERFACE Structure

USB Device Audio v2.0 Function Driver alternate interface setting event data.

File

[usb_device_audio_v2_0.h](#)

C

```
typedef struct {
    uint8_t interfaceNumber;
    uint8_t interfaceAlternateSetting;
} USB_DEVICE_AUDIO_V2_EVENT_DATA_SET_ALTERNATE_INTERFACE;
```

Members

Members	Description
uint8_t interfaceNumber;	Interface number of the interface who setting is to be changed
uint8_t interfaceAlternateSetting;	Alternate setting number

Description

USB Device Audio v2.0 Function Driver Alternate Interface Setting Event Data.

This data type defines the data structure returned by the driver along with USB_DEVICE_AUDIO_V2_EVENT_DATA_INTERFACE_SETTING_CHANGED.

Remarks

None.

USB_DEVICE_AUDIO_V2_EVENT_DATA_WRITE_COMPLETE Structure

USB Device Audio Function Driver Audio v2.0 read and write complete event data.

File

[usb_device_audio_v2_0.h](#)

C

```
typedef struct {
    USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE handle;
    uint16_t length;
    uint8_t interfaceNum;
    USB_DEVICE_AUDIO_V2_RESULT status;
} USB_DEVICE_AUDIO_V2_EVENT_DATA_WRITE_COMPLETE, USB_DEVICE_AUDIO_V2_EVENT_DATA_READ_COMPLETE;
```

Members

Members	Description
USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE handle;	Transfer handle associated with this <ul style="list-style-type: none">• read or write request
uint16_t length;	Indicates the amount of data (in bytes) that was <ul style="list-style-type: none">• read or written
uint8_t interfaceNum;	Interface Number
USB_DEVICE_AUDIO_V2_RESULT status;	Completion status of the transfer

Description

USB Device Audio v2.0 Function Driver Read and Write Complete Event Data.

This data type defines the data structure returned by the driver along with USB_DEVICE_AUDIO_V2_EVENT_READ_COMPLETE, USB_DEVICE_AUDIO_V2_EVENT_WRITE_COMPLETE, events.

Remarks

None.

USB_DEVICE_AUDIO_V2_EVENT_HANDLER Type

USB Device Audio v2.0 Event Handler Function Pointer Type.

File

[usb_device_audio_v2_0.h](#)

C

```
typedef USB_DEVICE_AUDIO_V2_EVENT_RESPONSE (* USB_DEVICE_AUDIO_V2_EVENT_HANDLER)(USB_DEVICE_AUDIO_V2_INDEX instanceIndex , USB_DEVICE_AUDIO_V2_EVENT event , void * pData, uintptr_t context);
```

Description

USB Device Audio v2.0 Event Handler Function Pointer Type.

This data type defines the required function signature USB Device Audio Function Driver event handling callback function. The application must register a pointer to an Audio Function Driver events handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive event call backs from the Audio Function Driver. The function driver will invoke this function with event relevant parameters. The description of the event handler function parameters is given here.

instanceIndex - Instance index of the Audio v2.0 Function Driver that generated the event.

event - Type of event generated.

pData - This parameter should be type casted to an event specific pointer type based on the event that has occurred. Refer to the [USB_DEVICE_AUDIO_V2_EVENT](#) enumeration description for more details.

context - Value identifying the context of the application that registered the event handling function.

Remarks

The event handler function executes in the USB interrupt context when the USB Device Stack is configured for interrupt based operation. It is not advisable to call blocking functions or computationally intensive functions in the event handler. Where the response to a control transfer related event requires extended processing, the response to the control transfer should be deferred and the event handler should be allowed to complete execution.

USB_DEVICE_AUDIO_V2_EVENT_RESPONSE Type

USB Device Audio v2.0 Function Driver event callback response type.

File

[usb_device_audio_v2_0.h](#)

C

```
typedef void USB_DEVICE_AUDIO_V2_EVENT_RESPONSE;
```

Description

USB Device Audio v2.0 Function Driver Event Handler Response Type

This is the return type of the Audio Function Driver event handler.

Remarks

None.

USB_DEVICE_AUDIO_V2_INDEX Type

USB Device Audio v2.0 Function Driver index.

File

[usb_device_audio_v2_0.h](#)

C

```
typedef uintptr_t USB_DEVICE_AUDIO_V2_INDEX;
```

Description

USB Device Audio v2.0 Function Driver Index

This definition uniquely identifies a Audio v2.0 Function Driver instance.

Remarks

None.

USB_DEVICE_AUDIO_V2_INIT Structure

USB Device Audio v2.0 Function Driver initialization data structure.

File

[usb_device_audio_v2_0.h](#)

C

```
typedef struct {
    size_t queueSizeRead;
    size_t queueSizeWrite;
} USB_DEVICE_AUDIO_V2_INIT;
```

Members

Members	Description
size_t queueSizeRead;	Size of the read queue for this instance <ul style="list-style-type: none"> of the Audio function driver
size_t queueSizeWrite;	Size of the write queue for this instance <ul style="list-style-type: none"> of the Audio function driver

Description

USB Device Audio v2.0 Function Driver Initialization Data Structure

This data structure must be defined for every instance of the Audio function driver. It is passed to the Audio v2.0 function driver, by the Device Layer, at the time of initialization. The funcDriverInit member of the Device Layer Function Driver registration table entry must point to this data structure for an instance of the Audio function driver.

Remarks

The queue sizes that are specified in this data structure are also affected by the [USB_DEVICE_AUDIO_V2_QUEUE_DEPTH_COMBINED](#) configuration macro.

USB_DEVICE_AUDIO_V2_RESULT Enumeration

USB Device Audio Function Driver USB Device Audio v2.0 result enumeration.

File

[usb_device_audio_v2_0.h](#)

C

```
typedef enum {
    USB_DEVICE_AUDIO_V2_RESULT_OK,
    USB_DEVICE_AUDIO_V2_RESULT_ERROR_TRANSFER_QUEUE_FULL,
    USB_DEVICE_AUDIO_V2_RESULT_ERROR_INSTANCE_INVALID,
    USB_DEVICE_AUDIO_V2_RESULT_ERROR_INSTANCE_NOT_CONFIGURED,
    USB_DEVICE_AUDIO_V2_RESULT_ERROR_PARAMETER_INVALID,
    USB_DEVICE_AUDIO_V2_RESULT_ERROR_INVALID_INTERFACE_ID,
    USB_DEVICE_AUDIO_V2_RESULT_ERROR_INVALID_BUFFER,
    USB_DEVICE_AUDIO_V2_RESULT_ERROR_ENDPOINT_HALTED,
    USB_DEVICE_AUDIO_V2_RESULT_ERROR_TERMINATED_BY_HOST,
    USB_DEVICE_AUDIO_V2_RESULT_ERROR
} USB_DEVICE_AUDIO_V2_RESULT;
```

Members

Members	Description
USB_DEVICE_AUDIO_V2_RESULT_OK	The operation was successful
USB_DEVICE_AUDIO_V2_RESULT_ERROR_TRANSFER_QUEUE_FULL	The transfer queue is full and no new transfers can be scheduled
USB_DEVICE_AUDIO_V2_RESULT_ERROR_INSTANCE_INVALID	The specified instance is not provisioned in the system
USB_DEVICE_AUDIO_V2_RESULT_ERROR_INSTANCE_NOT_CONFIGURED	The specified instance is not configured yet
USB_DEVICE_AUDIO_V2_RESULT_ERROR_PARAMETER_INVALID	The event handler provided is NULL
USB_DEVICE_AUDIO_V2_RESULT_ERROR_INVALID_INTERFACE_ID	Interface number passed to the read or write function is invalid.
USB_DEVICE_AUDIO_V2_RESULT_ERROR_INVALID_BUFFER	A NULL buffer was specified in the read or write function
USB_DEVICE_AUDIO_V2_RESULT_ERROR_ENDPOINT_HALTED	Transfer terminated because host halted the endpoint
USB_DEVICE_AUDIO_V2_RESULT_ERROR_TERMINATED_BY_HOST	Transfer terminated by host because of a stall clear
USB_DEVICE_AUDIO_V2_RESULT_ERROR	General Error

Description

USB Device Audio v2.0 Function Driver USB Device Audio v2.0 Result enumeration.

This enumeration lists the possible USB Device Audio v2.0 Function Driver operation results.

Remarks

None.

USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE Type

USB Device Audio v2.0 Function Driver Transfer Handle Definition.

File

[usb_device_audio_v2_0.h](#)

C

```
typedef uintptr_t USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE;
```

Description

USB Device Audio v2.0 Function Driver Transfer Handle Definition

This definition defines a USB Device Audio v2.0 Function Driver Transfer Handle. A Transfer Handle is owned by the application but its value is modified by the [USB_DEVICE_AUDIO_V2_Write](#) and [USB_DEVICE_AUDIO_V2_Read](#) functions. The transfer handle is valid for the life time of the transfer and expires when the transfer related event had occurred.

Remarks

None.

USB_DEVICE_AUDIO_V2_EVENT_RESPONSE_NONE Macro

USB Device Audio v2.0 Function Driver event handler response type none.

File

[usb_device_audio_v2_0.h](#)

C

```
#define USB_DEVICE_AUDIO_V2_EVENT_RESPONSE_NONE
```

Description

USB Device Audio v2.0 Function Driver Event Handler Response None

This is the definition of the Audio v2.0 Function Driver event handler response type none.

Remarks

Intentionally defined to be empty.

USB_DEVICE_AUDIO_V2_FUNCTION_DRIVER Macro

USB Device Audio v2.0 Function Driver function pointer.

File

[usb_device_audio_v2_0.h](#)

C

```
#define USB_DEVICE_AUDIO_V2_FUNCTION_DRIVER
```

Description

USB Device Audio v2.0 Function Driver Function Pointer

This is the USB Device Audio v2.0 Function Driver Function pointer. This should registered with the device layer in the function driver registration table.

Remarks

None.

USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE_INVALID Macro

USB Device Audio v2.0 Function Driver Invalid Transfer Handle Definition.

File

[usb_device_audio_v2_0.h](#)

C

```
#define USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE_INVALID
```

Description

USB Device Audio v2.0 Function Driver Invalid Transfer Handle Definition

This definition defines a USB Device Audio v2.0 Function Driver Invalid Transfer Handle. A Invalid Transfer Handle is returned by the `USB_DEVICE_Audio_V2_Write` and `USB_DEVICE_Audio_V2_Read` functions when the request was not successful.

Remarks

None.

Files**Files**

Name	Description
usb_device_audio_v2_0.h	USB Device Audio v2.0 Function Driver interface .
usb_device_audio_v2_0_config_template.h	USB device Audio 2.0 Class configuration definitions template

Description

This section lists the source and header files used by the library.

usb_device_audio_v2_0.h

USB Device Audio v2.0 Function Driver interface .

Enumerations

	Name	Description
	USB_DEVICE_AUDIO_V2_EVENT	USB Device Audio v2.0 Function Driver events.
	USB_DEVICE_AUDIO_V2_RESULT	USB Device Audio Function Driver USB Device Audio v2.0 result enumeration.

Functions

	Name	Description
	USB_DEVICE_AUDIO_V2_EventHandlerSet	This function registers an event handler for the specified Audio function driver instance.

	USB_DEVICE_AUDIO_V2_Read	This function requests a data read from the USB Device Audio v2.0 Function Driver Layer.
	USB_DEVICE_AUDIO_V2_TransferCancel	This function cancels a scheduled Audio v2.0 Device data transfer.
	USB_DEVICE_AUDIO_V2_Write	This function requests a data write to the USB Device Audio v2.0 Function Driver Layer.

Macros

Name	Description
USB_DEVICE_AUDIO_V2_EVENT_RESPONSE_NONE	USB Device Audio v2.0 Function Driver event handler response type none.
USB_DEVICE_AUDIO_V2_FUNCTION_DRIVER	USB Device Audio v2.0 Function Driver function pointer.
USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE_INVALID	USB Device Audio v2.0 Function Driver Invalid Transfer Handle Definition.

Structures

Name	Description
USB_DEVICE_AUDIO_V2_EVENT_DATA_READ_COMPLETE	USB Device Audio Function Driver Audio v2.0 read and write complete event data.
USB_DEVICE_AUDIO_V2_EVENT_DATA_SET_ALTERNATE_INTERFACE	USB Device Audio v2.0 Function Driver alternate interface setting event data.
USB_DEVICE_AUDIO_V2_EVENT_DATA_WRITE_COMPLETE	USB Device Audio Function Driver Audio v2.0 read and write complete event data.
USB_DEVICE_AUDIO_V2_INIT	USB Device Audio v2.0 Function Driver initialization data structure.

Types

Name	Description
USB_DEVICE_AUDIO_V2_EVENT_HANDLER	USB Device Audio v2.0 Event Handler Function Pointer Type.
USB_DEVICE_AUDIO_V2_EVENT_RESPONSE	USB Device Audio v2.0 Function Driver event callback response type.
USB_DEVICE_AUDIO_V2_INDEX	USB Device Audio v2.0 Function Driver index.
USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE	USB Device Audio v2.0 Function Driver Transfer Handle Definition.

Description

USB Device Audio Function Driver Interface

This file describes the USB Device Audio v2.0 Function Driver interface. This file should be included by the application if it needs to use the Audio v2.0 Function Driver API.

File Name

usb_device_audio_v2.0.h

Company

Microchip Technology Inc.

usb_device_audio_v2_0_config_template.h

USB device Audio 2.0 Class configuration definitions template

Macros

Name	Description
USB_DEVICE_AUDIO_V2_INSTANCES_NUMBER	Specifies the number of Audio 2.0 Function Driver instances.
USB_DEVICE_AUDIO_V2_MAX_ALTERNATE_SETTING	Specifies the maximum number of Alternate Settings per streaming interface.
USB_DEVICE_AUDIO_V2_MAX_STREAMING_INTERFACES	Specifies the maximum number of Audio 2.0 Streaming interfaces in an Audio 2.0 Function Driver instance.
USB_DEVICE_AUDIO_V2_QUEUE_DEPTH_COMBINED	Specifies the combined queue size of all Audio 2.0 function driver instances.

Description

USB Device Audio 2.0 Class Configuration Definitions

This file contains configurations macros needed to configure the Audio 2.0 Function Driver. This file is a template file only. It should not be included

by the application. The configuration macros defined in the file should be defined in the configuration specific system_config.h.

File Name

usb_device_audio_v2_0_config_template_0_config_template.h

Company

Microchip Technology Inc.

USB Device Layer Library

This section describes the USB Device Layer Library.

Introduction

Introduces the MPLAB Harmony USB Device Layer Library.

Description

The MPLAB Harmony USB Device Layer Library (also referred to as the Device Layer) is part of the MPLAB Harmony USB Device Stack. Within the USB Device Stack, the Device Layer responds to enumeration requests from the Hosts. It receives control transfers from the Host and responds to these control transfers in case of standard device requests. It dispatches function driver and application specific control transfers to the respective function drivers and to the application. It provides the application and function drivers with API routines that allow them to respond and complete a control transfer, with a possibility of deferring such responses. The Device Layer also provides the application with events and functions that allow the application to track the state of the device.

The Device Layer plays the role of a system in the MPLAB Harmony USB Device Stack. In that, it initializes the USB Driver, the device function drivers and maintains their state machines by invoking the Tasks routines of these modules. The Device Layer thus treats the USB Driver and function drivers as sub modules.

The device layer features the following:

- Supports both USB Full-Speed and Hi-Speed operation
- Based on a modular and event-driven architecture
- Supports the PIC32MX and PIC32MZ families of microcontrollers
- Supports composite USB devices
- Supports different types of function drivers (i.e., CDC, HID, MSD, etc.)
- Supports non-blocking operation and is RTOS friendly
- Designed to integrate readily with other MPLAB Harmony middleware
- Supports both interrupt and polling operation
- Reduces the required application intervention in maintaining the USB state of the device
- Allows implementation of a multi-configuration USB device
- Functions to implement Generic USB Devices

Using the Library

This topic describes the basic architecture of the USB Device Layer Library and provides information and examples on its use.

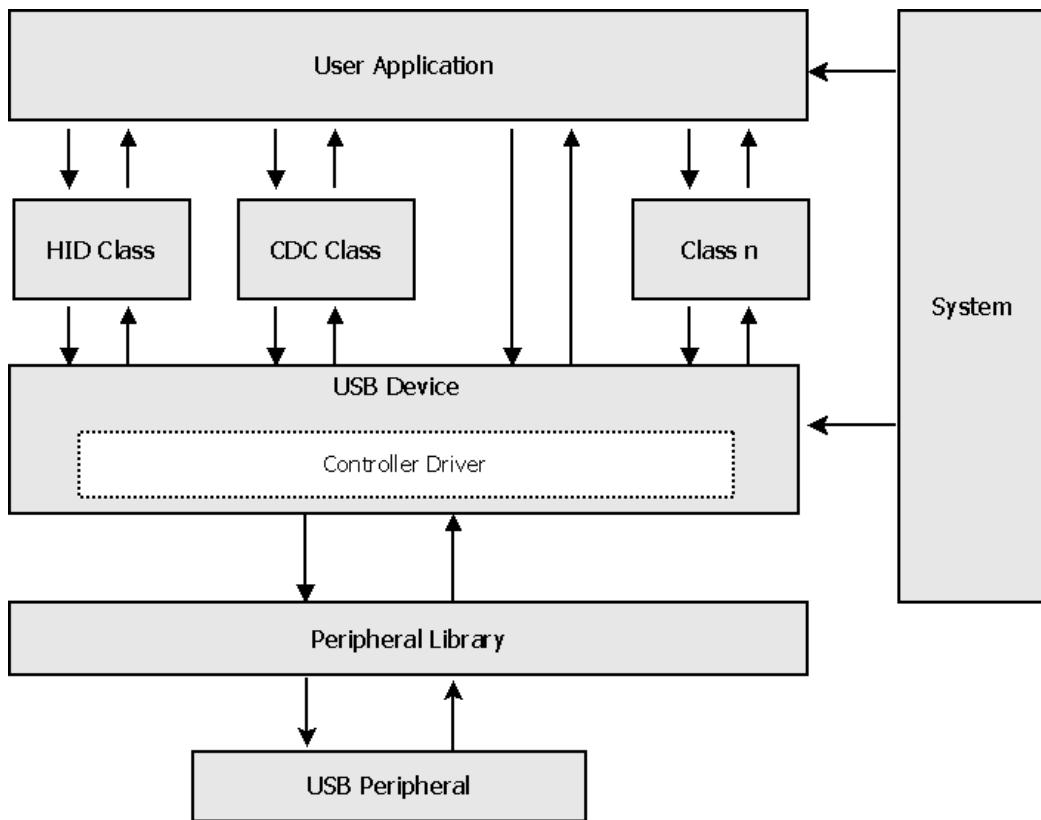
Abstraction Model

Describes the Abstraction Model of the USB Device Layer.

Description

The block diagram shows USB Device Layer interaction with USB controller driver, function drivers, user application and the system.

USB Device Layer Software Abstraction Block Diagram



System Interaction

The system is responsible for initializing and deinitializing the device layer. It is also responsible for calling the USB Device Layer task routine.

Function Driver Interaction

The USB Device Layer interacts with a function driver for following reasons:

- Initializes the function driver when device is configured by the Host. This can happen when the Host issues a set configuration request to the device. The device layer initializes only those function drivers that are valid for the selected configuration.
- Deinitializes the function driver when the Host issues a bus reset or when device is detached from the host or when the Host unconfigures the device by setting configuration value to '0'.
- Function driver and USB Controller Driver task routines are run by the device layer. This means that these task routines run at the same priority as a device layer task.
- Forwards class/interface specific setup requests from host to function drivers for processing. The function drivers can use device layer APIs to read and write data to Endpoint 0.

All of these interactions are initiated by the device layer, and therefore, it is required for a function driver to register a set of standard APIs with the device layer for initializing/deinitializing the function driver, for handling control transfers and for running the task routines. Registering of these callback functions with the device layer is a compile time step and is done using the function driver registration table. Function driver registration is explained in subsequent sections.

User Application (Client) Interaction

User application clients can register a callback function with the device layer to get USB device events. Apart from device events, the clients can interact with USB device layer to determine other status such as USB speed and remote wake-up. The Device Layer will forward Control Transfers whose Recipient field is set to Other, to the application. The application must use the Device Layer Control Transfer Routines to complete the control transfer.

Library Overview

The USB device layer mainly interacts with the system, its clients and function drivers, as shown in the [Abstraction Model](#).

The library interface routines are divided into sub-sections, which address one of the blocks or the overall operation of the USB Device Layer Library.

Library Interface Section	Description
System Interaction Functions	Provides system module interfaces, device initialization, deinitialization, reinitialization and task functions.
Client Core Functions	Provides function to register callbacks, mechanism to pass events to clients and functions to know the status.

Device Power State Management Functions	These functions manage the power state of the device (self or bus powered) and remote wake-up.
Endpoint Management Functions	These functions allow the application to manage endpoints (enable, disable and stall).
Device Management Functions	These functions allow the application to manage the state of the device (attach, detach etc.).
Control Transfer Functions	These functions allow the application to respond to complete control transfers.

How the Library Works

Library Initialization

Describes how the USB Device Layer must be initialized.

Description

The Device Layer initialization process requires the following components:

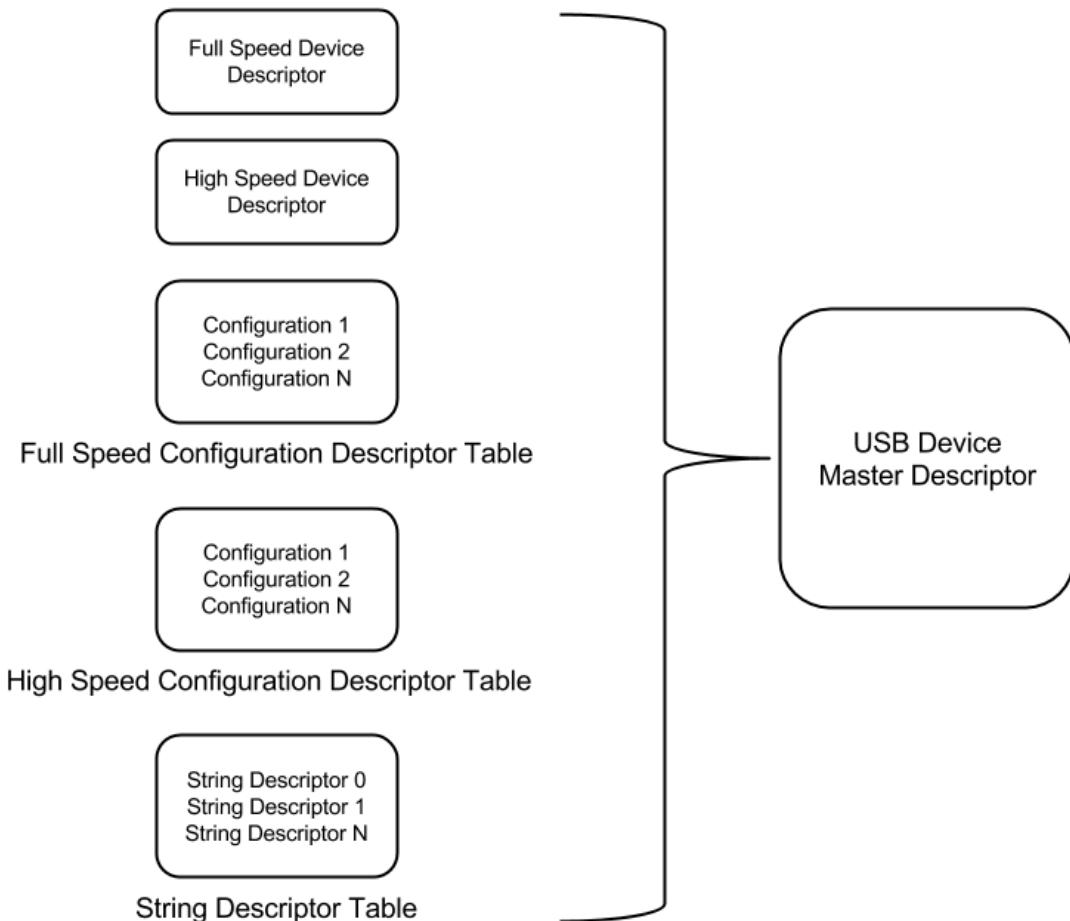
- USB Standard Descriptors that define the device functionality. The definitions of these descriptors are defined by the USB 2.0 and Device Class specification.
- Device Master Descriptor Table
- Function Driver Registration Table

The USB Standard Descriptors that define the device functionality are discussed in detail in the USB 2.0 and Device Class Specifications. The reader is encouraged to refer to these specifications for a detailed understanding of this topic.

Master Descriptor Table

Describes the USB Device Layer Master Descriptor Table.

Description



As seen in the figure, the Device Master Descriptor Table (specified by the `USB_DEVICE_MASTER_DESCRIPTOR` data type) is a container for all descriptor related information that is needed by the Device Layer for its operation. This table contains the following information:

- Pointer to the Full-Speed and High-Speed Device Descriptor
- Number of Full-Speed and High-Speed Configurations
- Pointers to Table of Full-Speed and High-Speed Configuration Descriptors
- Number of String Descriptors
- Pointer to a Table of String Descriptors
- Pointers to Full-Speed and High-Speed Device Qualifier

In a case where a particular item in the Device Master Descriptor Table is not applicable, that entry can be either set to '0' or NULL as applicable. For example for a Full-Speed-only device, the number of High Speed Configuration should be set to '0' and the pointer to the table of High-Speed Configuration Descriptors should be set to NULL.

The following code shows an example of a USB Device Master Descriptor design for a Full-Speed USB HID Keyboard.

```
*****
 * USB Device Layer Master Descriptor Table
 ****
const USB_DEVICE_MASTER_DESCRIPTOR usbMasterDescriptor =
{
    &fullSpeedDeviceDescriptor,          /* Full-speed descriptor */
    1,                                  /* Total number of full-speed configurations available */
    &fullSpeedConfigDescSet[0],          /* Pointer to array of full-speed configurations descriptors */

    NULL,                               /* High-speed device descriptor is not supported*/
    0,                                  /* Total number of high-speed configurations available */
    NULL,                               /* Pointer to array of high-speed configurations descriptors. Not
supported */

    3,                                  /* Total number of string descriptors available */
    stringDescriptors,                  /* Pointer to array of string descriptors */

    NULL,                               /* Pointer to full-speed device qualifier. Not supported */
}
```

```

        NULL,                                /* Pointer to high-speed device qualifier. Not supported */
};

The following code shows an example of a USB Device Master Descriptor design for a Full Speed/High Speed USB HID Keyboard.

/*****
 * USB Device Layer Master Descriptor Table
 *****/
const USB_DEVICE_MASTER_DESCRIPTOR usbMasterDescriptor =
{
    &fullSpeedDeviceDescriptor,           /* Full-speed descriptor */
    1,                                    /* Total number of full-speed configurations available */
    &fullSpeedConfigDescSet[0],          /* Pointer to array of full-speed configurations descriptors */

    &highSpeedDeviceDescriptor,          /* High-speed descriptor */
    1,                                    /* Total number of high-speed configurations available */
    &highSpeedConfigDescSet[0],          /* Pointer to array of high-speed configurations descriptors */

    3,                                    /* Total number of string descriptors available */
    stringDescriptors,                  /* Pointer to array of string descriptors */

    &deviceQualifierDescriptor1,         /* Pointer to full-speed device qualifier. */
    NULL,                                /* Pointer to high-speed device qualifier. Not supported */
};

```

The USB Device Layer Master Descriptor table can be placed in the data or program memory of the PIC32 device. The contents of this table could be modified while the application is running. Doing this will affect the operation of the Device Stack. A typical USB device application will not need to change the contents of this table while the application is running.

Function Driver Registration Table

This section explains how function drivers can be registered with the USB Device Layer using the Function Registration Table.

Description

The Function Driver Registration Table (defined by the [USB_DEVICE_FUNCTION_REGISTRATION_TABLE](#) data type) contains information about the function drivers that are present in the application. The Device Layer needs this information to establish the intended functionality of the USB Device and then manage the operation of the device.

The Function Driver Registration Table contains an entry for every function driver instance contained in the application. Each entry is configuration specific. Hence in a case of device that features multiple configurations, the Function Driver Registration Table will contain an entry for every function driver in each configuration. Entries are instance and configuration specific. Hence if a configuration contains two instances of the same function driver type, the Function Driver Registration Table will contain two entries to for the same function driver but with different instance indexes. A description of each member of the Function Driver Registration Table entry is as follows:

- The configurationValue member of the entry specifies to which configuration this entry belongs. The Device Layer will process this entry when the configurationValue configuration is set.
- The driver member of the entry should be set to Function Driver – Device Layer Interface Functions Object exported by the function driver. This object is provided by the function driver. In case of the CDC function driver, this is [USB_DEVICE_CDC_FUNCTION_DRIVER](#). In case of HID function driver, this is [USB_DEVICE_HID_FUNCTION_DRIVER](#). Refer to the "Library Initialization" topic in Function Driver Specific help section for more details.
- The funcDriverIndex member of the entry specifies the instance of the function driver that this entry relates to. The Device Layer will use this instance when communicating with the function driver. In a case where there are multiple instances of the same function driver in a configuration, the funcDriverIndex allows the Device Layer to uniquely identify the function driver.
- The funcDriverInit member of the entry must point to the function driver instance specific initialization data structure. Function Drivers typically require an initialization data structure to be specified. The Device Layer passes the pointer to the initialization data structure when the function driver is initialized. Refer to the "Library Initialization" topic in Function Driver Specific help section for more details.
- The interfaceNumber member of the entry must contain the interface number of the first interface that is owned by this function driver instance. The information is available from the Device Configuration Descriptor.
- The numberOfInterfaces member of the entry must contain the number of interfaces following the interfaceNumber interface that is owned by this function driver instance. For example, a CDC Device requires two interfaces. The interfaceNumber member of Function Driver Registration Table entry for this function driver would be 0 and the numberOfInterfaces member would be 2. This indicates that Interface 0 and Interface 1 in the Device Configuration Descriptor are owned by this function driver.
- The speed member of the entry specifies the device speeds for which this function driver should be initialized. This can be set to either [USB_SPEED_FULL](#), [USB_SPEED_HIGH](#) or a logical OR combination of both. The Device Layer will initialize the function if the device attach speed matches the speed mention in the speed member of the entry.

The following code shows an example of Function Driver Registration Table for one function driver. The CDC Function Driver in this case has two interfaces.

```

/*****
 * USB Device Layer Function Driver Registration
 * Table
 *****/
const USB_DEVICE_FUNCTION_REGISTRATION_TABLE funcRegistrationTable[1] =

```

```
{
{
    .configurationValue = 1 , // Configuration descriptor index
    .driver = USB_DEVICE_CDC_FUNCTION_DRIVER, // CDC APIs exposed to the device layer
    .funcDriverIndex = 0 , // Instance index of CDC function driver
    .funcDriverInit = (void *)&cdcInit, // CDC init data
    .interfaceNumber = 0 , // Start interface number of this instance
    .numberOfInterfaces = 2 , // Total number of interfaces contained in this instance
    .speed = USB_SPEED_FULL|USB_SPEED_HIGH // USB Speed
}
};
```

The following code shows an example of Function Driver Registration Table for two function drivers. This example demonstrates a Composite (CDC + MSD) device. The CDC Function Driver uses two interfaces starting from interface 0. The MSD Function Driver has one interface starting from interface 2.

```
*****
* Function Driver Registration Table
*****
USB_DEVICE_FUNCTION_REGISTRATION_TABLE funcRegistrationTable[2] =
{
{
    .speed = USB_SPEED_FULL|USB_SPEED_HIGH, // Speed at which this device can operate
    .configurationValue = 1, // Configuration number to which this device belongs
    .interfaceNumber = 1, // Starting interface number for this function driver
    .numberOfInterfaces = 2, // Number of interfaces that this function driver owns.
    .funcDriverIndex = 0, // Function Driver index
    .funcDriverInit = &cdcInit, // Function Driver initialization data structure
    .driver = USB_DEVICE_CDC_FUNCTION_DRIVER // CDC Function Driver.
},
{
    .speed = USB_SPEED_FULL|USB_SPEED_HIGH, // Speed at which this device can operate
    .configurationValue = 1, // Configuration number to which this device belongs
    .interfaceNumber = 0, // Starting interface number for this function driver
    .numberOfInterfaces = 1, // Number of interfaces that this function driver owns.
    .funcDriverIndex = 0, // Function Driver index
    .funcDriverInit = &msdInit, // Function Driver initialization data structure
    .driver = USB_DEVICE_MSD_FUNCTION_DRIVER // MSD Function Driver.
},
};

};

The USB Device Layer Function Driver registration table can be placed in the data or program memory of the PIC32 device. The contents of this table could be modified while the application is running. Doing this will affect the operation of the device stack. A typical USB device application will not need to change the contents of this table while the application is running.
```

Initializing the Device Layer

This section describes the USB Device Layer initialization.

Description

With the USB Device Master Descriptor and the Function Driver Registration Table available, the application can now create the Device Layer Initialization Data structure. This data structure is a [USB_DEVICE_INIT](#) type and contains the information need to initialize the Device Layer and the USB Controller Driver. The Device Layer uses the USB Controller Driver initialization data to initialize the USB Controller driver. The actual initialization is performed by calling the [USB_DEVICE_Initialize](#) function. This function returns a Device Layer System Module Object which must be used with the other Device Layer System Routines (such as the [USB_DEVICE_Tasks](#) and [USB_DEVICE_Tasks_ISR](#) functions). In case of PIC32MX device, the Device Layer requires the allocation of an endpoint table. This table (an array of type `uint8_t`) should aligned at a 512 byte aligned address boundary. Its size should be `USB_DEVICE_ENDPOINT_TABLE_SIZE`. The table is not required for PIC32MZ devices. The value of `USB_DEVICE_ENDPOINT_TABLE_SIZE` while designing for PIC32MZ device will be automatically set to '0'.

The following code shows an example of initializing the Device Layer.

```
*****
* Endpoint Table needed by the Device Layer.
*****
uint8_t __attribute__((aligned(512))) endpointTable[USB_DEVICE_ENDPOINT_TABLE_SIZE];

*****
* USB Device Layer Initialization.
*****
USB_DEVICE_INIT usbDevInitData =
```

```
{
    /* System module initialization */
    .moduleInit = {SYS_MODULE_POWER_RUN_FULL} ,

    /* Identifies peripheral (PLIB-level) ID */
    .usbID = USB_ID_1,

    /* Stop in idle */
    .stopInIdle = false,

    /* Stop in sleep */
    .suspendInSleep = false,

    /* Interrupt source */
    .interruptSource = INT_SOURCE_USB_1,

    /* Endpoint Table */
    .endpointTable = endpointTable,

    /* Number of function drivers registered to this instance of the
     * USB device layer */
    .registeredFuncCount = 2,

    /* Function driver table registered to this instance of the USB device layer*/
    .registeredFunctions = (USB_DEVICE_FUNCTION_REGISTRATION_TABLE*)funcRegistrationTable,

    /* Pointer to USB Descriptor structure */
    .usbMasterDescriptor = (USB_DEVICE_MASTER_DESCRIPTOR*)&usbMasterDescriptor
};

/*****************
 * System Initialization Routine
 *****************/
void SYS_Initialize ( void * data )
{
    /* Set up cache and wait states for maximum performance. */
    SYS_DEVCON_Initialize(SYS_DEVCON_INDEX_0, (SYS_MODULE_INIT *)&devconInit);
    SYS_DEVCON_PerformanceConfig(80000000);

    /* Initialize the BSP */
    BSP_Initialize( );

    /* Initialize the USB device layer */
    sysObjects.usbDevObject = USB_DEVICE_Initialize (USB_DEVICE_INDEX_0 ,
                                                    (SYS_MODULE_INIT* ) & usbDevInitData);

    /* check if the object returned by the device layer is valid */
    SYS_ASSERT((SYS_MODULE_OBJ_INVALID != sysObjects.usbDevObject), "Invalid USB DEVICE object");

    /* Initialize the Application */
    APP_Initialize ( );

    /* Initialize the interrupt system */
    SYS_INT_Initialize();

    /* set priority for USB interrupt source */
    SYS_INT_VectorPrioritySet(INT_VECTOR_USB, INT_PRIORITY_LEVEL3);

    /* set sub-priority for USB interrupt source */
    SYS_INT_VectorSubprioritySet(INT_VECTOR_USB, INT_SUBPRIORITY_LEVEL3);

    /* Initialize the global interrupts */
    SYS_INT_Enable();
}
}
```

Device Layer Task Routines

Describes the Device Layer task routines.

Description

In a case where the USB Controller driver is configured for interrupt mode operation (DRV_USB_INTERRUPT_MODE is true), the [USB_DEVICE_Tasks_ISR\(\)](#) function should be called in the USB module interrupt associated with the Device Layer. The following code shows an example of this.

```
*****
 * Example of instantiating the USB Interrupt
 * and call the Device Layer ISR Tasks Routines.
*****
```

```
void __ISR ( _USB_1_VECTOR, ipl3 ) _InterruptHandler_USB_stub ( void )
{
    USB_DEVICE_Tasks_ISR(sysObjects.usbDevObject);
}
```

If the USB Controller driver is configured for polled mode, the Device Layer will automatically invoke the [USB_DEVICE_Tasks_ISR\(\)](#) functions from the [USB_DEVICE_Tasks\(\)](#) function. Irrespective of whether the USB Controller driver is configured for polled or interrupt mode, the [USB_DEVICE_Tasks\(\)](#) function should be called from [SYS_Tasks\(\)](#) function. This will ensure that this function is called periodically. The [USB_DEVICE_Tasks\(\)](#) function in turn calls the tasks routines of the applicable functions drivers and the USB Controller Driver (when the driver is configured for polled mode). The following code shows an example of how the [USB_DEVICE_Tasks\(\)](#) function is called in the [SYS_Tasks\(\)](#) function.

```
void SYS_Tasks ( void )
{
    /* Device layer tasks routine. Function Driver tasks gets called
     * from device layer tasks */

    USB_DEVICE_Tasks(sysObjects.usbDevObject);

    /* Call the application's tasks routine */
    APP_Tasks ();
}
```

Application Client Interaction

Describes application client interaction.

Description

Once initialized, Device Layer becomes ready for operation. The application must open the Device Layer by calling the [USB_DEVICE_Open\(\)](#) function. Opening the Device Layer makes the application a Device Layer client. The Device Layer returns a valid Device Layer Handle when opened successfully. It will return an invalid Device Layer Handle when the open function fails. The application in this case should try opening the Device Layer again. The application can use a valid Device Layer handle to access the Device Layer functionality.

The client must now register a Device Layer Event Handler with the Device Layer. This is a mandatory step that enables USB Device Layer Events. The application must use the [USB_DEVICE_EventHandlerSet](#) function to register the event handler. The Application Event Handler should be of the type [USB_DEVICE_EVENT_HANDLER](#). The Device Layer, when an event needs to be generated, calls this event handler function with the event type and event relevant information. The application must register an event handler for proper functioning of the USB Device. Not registering an event handler may cause the USB Device to malfunction and become non-compliant.

With an event handler register, the client can now attach the USB Device on the bus. The application must attach the in response to the [USB_DEVICE_EVENT_POWER_DETECTED](#) event. Attaching the device on the bus makes the device visible to the host (if it is already attached to the bus) and will cause the host to interact with the device.

The following code shows an example of the application opening the Device Layer, registering the event handler and then attaching the device on the bus.

```
*****
 * Here the application tries to open the Device Layer
 * and then register an event handler and then attach
 * the device on the bus.
*****
```

```
void APP_Tasks(void)
{
    switch(appData.state)
    {
        case APP_STATE_INIT:

            /* Open the device layer */
            appData.deviceHandle = USB_DEVICE_Open( USB_DEVICE_INDEX_0,
                DRV_IO_INTENT_READWRITE );

            if(appData.deviceHandle != USB_DEVICE_HANDLE_INVALID)
```

```

    {
        /* Register a callback with device layer to get event notification */
        USB_DEVICE_EventHandlerSet(appData.deviceHandle,
            APP_USBDeviceEventCallBack, 0);

        appData.state = APP_STATE_WAIT_FOR_CONFIGURATION;
    }
    else
    {
        /* The Device Layer is not ready to be opened. We should try
         * again later. */
    }

    break;
}

```

Event Handling

Describes USB Device Layer event handling.

Description

The Device Layer generates events to let the application client know about the state of the bus. Some of these events require the application to respond in a specific manner. Not doing so, could cause the USB device to malfunction and become non-compliant. Code inside the event handler executes in an interrupt context when the Device Layer is configured for interrupt mode operation. The application must avoid calling computationally intensive functions or blocking functions in the event handler.

The following table shows a summary of the events that the Device Layer generates and the required Application client response.

Event	Required Application Response
USB_DEVICE_EVENT_POWER_DETECTED	Attach the device.
USB_DEVICE_EVENT_POWER_REMOVED	Detach the device.
USB_DEVICE_EVENT_RESET	No response required.
USB_DEVICE_EVENT_SUSPENDED	No response required.
USB_DEVICE_EVENT_RESUMED	No response required.
USB_DEVICE_EVENT_ERROR	The application can try detaching the device and reattaching. This should be done after exiting from the event handler.
USB_DEVICE_EVENT_SOF	No response required.
USB_DEVICE_EVENT_CONFIGURED	No response required.
USB_DEVICE_EVENT_DECONFIGURED	No response required.
USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST	Application must either respond with a USB_DEVICE_ControlSend() to send data, USB_DEVICE_ControlReceive() to receive data or stall or acknowledge the control request by calling the USB_DEVICE_ControlStatus() function.
USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT	No response required.
USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA RECEIVED	Application must either or stall or acknowledge the control request by calling the USB_DEVICE_ControlStatus() function.
USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_ABORTED	No response required.

The Device Layer generates events with event relevant data. The pData parameter in the event handler functions points to this event specific data. The application can access this data by type casting the pData parameter of the event handler to a event specific data type. The following table shows a summary of the USB Device Layer events and the event data generated along with the event.

Event	Related pData Type
USB_DEVICE_EVENT_POWER_DETECTED	NULL
USB_DEVICE_EVENT_POWER_REMOVED	NULL
USB_DEVICE_EVENT_RESET	NULL
USB_DEVICE_EVENT_SUSPENDED	NULL
USB_DEVICE_EVENT_RESUMED	NULL
USB_DEVICE_EVENT_ERROR	NULL

USB_DEVICE_EVENT_SOFTWARE_RESET	NULL
USB_DEVICE_EVENT_CONFIGURED	USB_DEVICE_EVENT_DATA_CONFIGURED *
USB_DEVICE_EVENT_DECONFIGURED	NULL
USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST	USB_SETUP_PACKET *
USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT	NULL
USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_RECEIVED	NULL

A detailed description of each Device Layer event along with the required application client response, the likely follow-up event (if applicable) and the event specific data is provided here.

USB_DEVICE_EVENT_POWER_DETECTED

Application Response: This event indicates that the device has detected a valid VBUS to the host. The device is yet to be enumerated and configured. The application should not access the function drivers at this point. The application can use this event to attach the device on the bus.

Event Specific Data(pData): The pData parameter will be NULL.

Likely Follow Up Event: None.

USB_DEVICE_EVENT_POWER_REMOVED

Application Response: This event is an indication to the application client that the device is detached from the host. The application can use this event to detach the device.

Event Specific Data(pData): The pData parameter will be NULL.

Likely Follow Up Event: None.

USB_DEVICE_EVENT_SUSPENDED

Application Response: This event is an indication to the application client that device is suspended and it can put the device to sleep mode if required. Power saving routines should not be called in the event handler.

Event Specific Data: The pData parameter will be NULL.

Likely Follow Up Event: None.

USB_DEVICE_EVENT_RESET

Application Response: USB bus reset occurred. This event is an indication to the application client that device layer has deinitialized all function drivers. The application should not use the function drivers in this state.

Event Specific Data: The pData parameter will be NULL.

Likely Follow Up Event: None.

USB_DEVICE_EVENT_RESUMED

Application Response: This event indicates that device has resumed from suspended state. The application can use this event to resume the operational state of the device.

Event Specific Data: The pData parameter will be NULL.

Likely Follow Up Event: None.

USB_DEVICE_EVENT_ERROR

Application Response: This event is an indication to the application client that an error occurred on the USB bus. The application can try detaching and reattaching the device.

Event Specific Data: The pData parameter will be NULL.

Likely Follow Up Event: None.

USB_DEVICE_EVENT_SOFTWARE_RESET

Application Response: This event occurs when the device receives a Start Of Frame packet. The application can use this event for synchronizing purposes. This event will be received every 1 millisecond for Full Speed USB and every one 125 micro seconds for High Speed USB. No application response is required.

Event Specific Data: Will point to [USB_DEVICE_EVENT_DATA_SOFTWARE_RESET](#) data type containing the frame number

Likely Follow Up Event: None.

USB_DEVICE_CONFIGURED

Application Response: This event is an indication to the application client that device layer has initialized all function drivers. The application can check the configuration set by the host. The application should use the event to register event handlers with the function drivers that are

provisioned in the system.

Event Specific Data: The pData parameter will point to a [USB_DEVICE_EVENT_DATA_CONFIGURED](#) data type that contains configuration set by the host

Likely Follow Up Event: None.

USB_DEVICE_DECONFIGURED

Application Response: The host has deconfigured the device. This happens when the host sends a Set Configuration request with configuration number 0. The device layer will deinitialize all function drivers and then generate this event. No application response is required.

Event Specific Data: The pData parameter will be NULL

Likely Follow Up Event: None.

USB_DEVICE_CONTROL_TRANSFER_ABORTED

Application Response: An on-going control transfer was aborted. The application can use this event to reset its control transfer state machine.

Event Specific Data: The pData parameter will be NULL

Likely Follow Up Event: None.

USB_DEVICE_CONTROL_TRANSFER_DATA RECEIVED

Application Response: The data stage of a Control write transfer has completed. This event occurs after the application has used the [USB_DEVICE_ControlReceive\(\)](#) function to receive data in the control transfer (in response to the [USB_DEVICE_CONTROL_TRANSFER_SETUP_REQUEST](#) event). The application can inspect the received data and stall or acknowledge the control transfer by calling the [USB_DEVICE_ControlStatus\(\)](#) function with the [USB_DEVICE_CONTROL_STATUS_ERROR](#) flag or [USB_DEVICE_CONTROL_STATUS_OK](#) flag respectively. The application can call the [USB_DEVICE_ControlStatus\(\)](#) function in the event handler or after exiting the event handler.

Event Specific Data: The pData parameter will be NULL

Likely Follow Up Event: None.

USB_DEVICE_CONTROL_TRANSFER_SETUP REQUEST

Application Response: A setup packet of a control transfer has been received. The recipient field of the received setup packet is Other. The application can initiate the data stage by using the [USB_DEVICE_ControlReceive\(\)](#) and [USB_DEVICE_ControlSend\(\)](#) functions. It can end the control transfer by calling the [USB_DEVICE_ControlStatus\(\)](#) function.

Event Specific Data: The pData parameter in the event handler will point to [USB_SETUP_PACKET](#) data type.

Likely Follow Up Event: [USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT](#) if the [USB_DEVICE_ControlSend\(\)](#) function was called to send data to the host. [USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA RECEIVED](#) if the [USB_DEVICE_ControlReceive\(\)](#) function was called to receive data from the host.

USB_DEVICE_CONTROL_TRANSFER_DATA SENT

Application Response: The data stage of a Control Read transfer has completed. This event occurs after the application has used the [USB_DEVICE_ControlSend\(\)](#) function to send data in the control transfer. No application response is required.

Event Specific Data: The pData parameter will be NULL

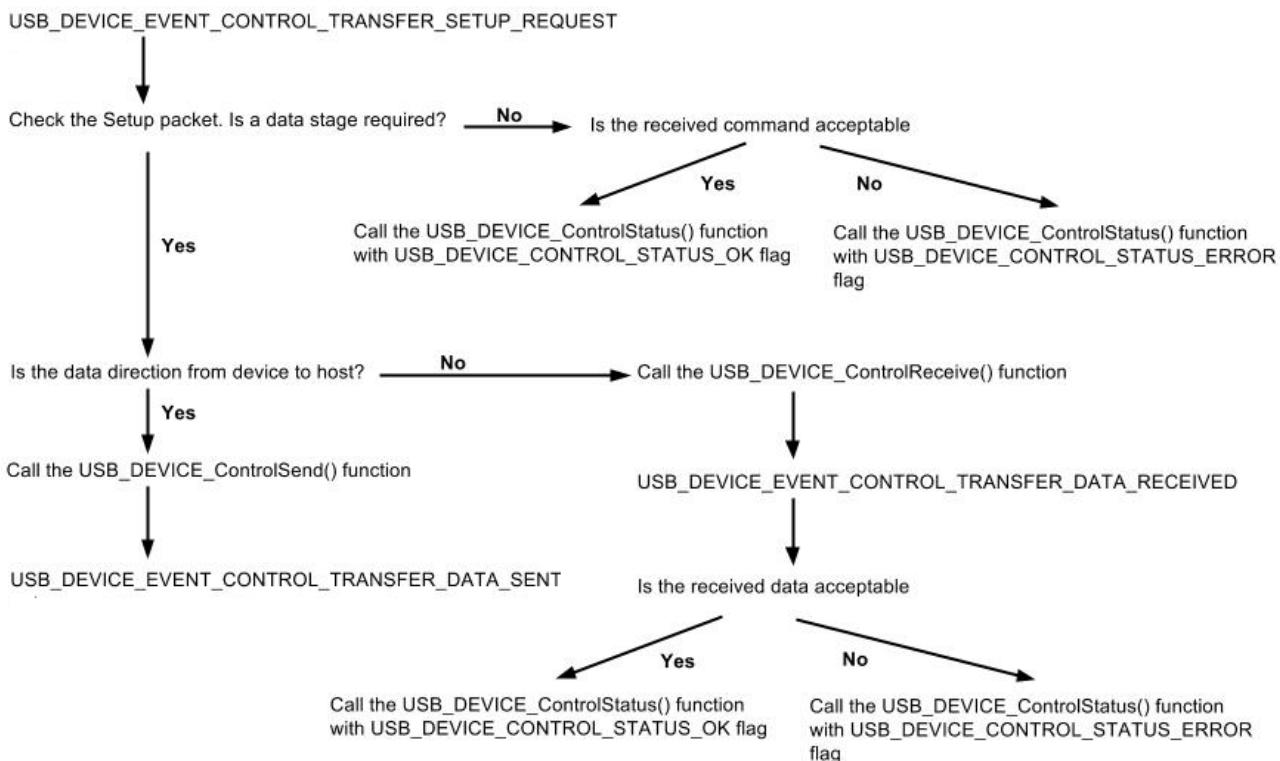
Likely Follow Up Event: None.

Device Layer Control Transfers

Describes USB Device Layer control transfers.

Description

The USB Device Layer forwards control transfer setup packets, where the Recipient field in the Setup packet is "Other", to the application for handling. The application must respond appropriately to this event. The following flow chart shows the possible sequences of events and application responses.



The Device Layer provides the [USB_DEVICE_ControlReceive\(\)](#), [USB_DEVICE_ControlSend\(\)](#) and [USB_DEVICE_ControlStatus\(\)](#) functions to complete the control transfers. These functions should be called only in response to the [USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST](#) event. In response to this event, the application can use the [USB_DEVICE_ControlReceive\(\)](#) function to receive data in the data stage of a Control Write transfer. The reception of data is indicated by the [USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_RECEIVED](#) event. The application can then complete the Control Write transfer by either:

- accepting the received data and acknowledging the Status Stage of the Control transfer. This is done by calling the [USB_DEVICE_ControlStatus\(\)](#) function with the [USB_DEVICE_CONTROL_STATUS_OK](#) flag.
- rejecting the received data and stalling the Status Stage of the Control transfer. This is done by calling the [USB_DEVICE_ControlStatus\(\)](#) function with the [USB_DEVICE_CONTROL_STATUS_ERROR](#) flag.

The application can use the [USB_DEVICE_ControlSend\(\)](#) function to send data in the data stage of a Control Read transfer. The transmission of data is indicated by the [USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT](#) event.

In a case where the Control Transfer does not contain a data stage or if the application does not support the Setup Request, the application can end the Control Transfer by calling the [USB_DEVICE_ControlStatus\(\)](#) function in response to the [USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST](#) event. Here the application can

- accepting the command by acknowledging the Status Stage of the Zero Data Stage of the Control transfer. This is done by calling the [USB_DEVICE_ControlStatus\(\)](#) function with the [USB_DEVICE_CONTROL_STATUS_OK](#) flag.
- rejecting the Setup Request and stalling the Status Stage of the Control transfer. This is done by calling the [USB_DEVICE_ControlStatus\(\)](#) function with the [USB_DEVICE_CONTROL_STATUS_ERROR](#) flag.

The application can also defer the response to Control transfer events. In that, the application does not have to respond to Control Transfer Events in the event handler. This may be needed in cases where resources required to respond to the Control Transfer Events are not readily available. The application, even while deferring the response, must however complete the Control Transfer in a time fashion. Failing to do so, will cause the host to cancel and retry the control transfer. This could also cause the USB device to malfunction and become non-compliant.

The following code shows an example of handling Device Layer events.

```

USB_DEVICE_EVENT_RESPONSE APP_USBDeviceEventHandler
{
    USB_DEVICE_EVENT event,
    void * pData,
    uintptr_t context
}
{
    uint8_t      activeConfiguration;
    uint16_t     frameNumber;
    USB_SPEED   attachSpeed;
    USB_SETUP_PACKET * setupEventData;

    // Handling of each event
    switch(event)
    {

```

```
case USB_DEVICE_EVENT_POWER_DETECTED:

    // This means the device detected a valid VBUS voltage
    // and is attached to the USB if the device is bus powered.
    break;

case USB_DEVICE_EVENT_POWER_REMOVED:

    // This means the device is not attached to the USB.
    break;

case USB_DEVICE_EVENT_SUSPENDED:

    // The bus is idle. There was no activity detected.
    // The application can switch to a low power mode after
    // exiting the event handler.
    break;

case USB_DEVICE_EVENT_SOF:

    // A start of frame was received. This is a periodic
    // event and can be used the application for time
    // related activities. pData will point to a USB_DEVICE_EVENT_DATA_SOF type data
    // containing the frame number.

    frameNumber = ((USB_DEVICE_EVENT_DATA_SOF *) (pData)) ->frameNumber;
    break;

case USB_DEVICE_EVENT_RESET :

    // Reset signalling was detected on the bus. The
    // application can find out the attach speed.

    attachedSpeed = USB_DEVICE_ActiveSpeedGet(usbDeviceHandle);
    break;

case USB_DEVICE_EVENT_DECONFIGURED :

    // This indicates that host has deconfigured the device i.e., it
    // has set the configuration as 0. All function driver instances
    // would have been deinitialized.

    break;

case USB_DEVICE_EVENT_ERROR :

    // This means an unknown error has occurred on the bus.
    // The application can try detaching and attaching the
    // device again.
    break;

case USB_DEVICE_EVENT_CONFIGURED :

    // This means that device is configured and the application can
    // start using the device functionality. The application must
    // register function driver event handlers within this event.
    // The pData parameter will be a pointer to a USB_DEVICE_EVENT_DATA_CONFIGURED data type
    // that contains the active configuration number.

    activeConfiguration = ((USB_DEVICE_EVENT_DATA_CONFIGURED *) (pData)) ->configurationValue;
    break;

case USB_DEVICE_EVENT_RESUMED:

    // This means that the resume signalling was detected on the
    // bus. The application can bring the device out of power
    // saving mode.

    break;
```

```

case USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST:

    // This means that the setup stage of the control transfer is in
    // progress and a setup packet has been received. The pData
    // parameter will point to a USB_SETUP_PACKET data type. The
    // application can process the command and update its control
    // transfer state machine. The application for example could call
    // the USB_DEVICE_ControlReceive() function (as shown here) to
    // submit the buffer that would receive data in case of a
    // control read transfer.

    setupPacket = (USB_SETUP_PACKET *)pData;

    // Submit a buffer to receive 32 bytes in the control write transfer.
    USB_DEVICE_ControlReceive(usbDeviceHandle, data, 32);
break;

case USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_RECEIVED:

    // This means that data in the data stage of the control write
    // transfer has been received. The application can either accept
    // the received data by calling the USB_DEVICE_ControlStatus()
    // function with USB_DEVICE_CONTROL_STATUS_OK flag (as shown in
    // this example) or it can reject it by calling the
    // USB_DEVICE_ControlStatus() function with
    // USB_DEVICE_CONTROL_STATUS_ERROR flag.

    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);
break;

case USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_SENT:

    // This means that data in the data stage of the control
    // read transfer has been sent. The application would typically
    // end the control transfer by calling the
    // USB_DEVICE_ControlStatus() function with
    // USB_DEVICE_CONTROL_STATUS_OK flag (as shown in this example).

    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);
break;

case USB_DEVICE_CONTROL_TRANSFER_EVENT_ABORTED:

    // This means the host has aborted the control transfer. The
    // application can reset its control transfer state machine.

    break;

default:
    break;
}

return USB_DEVICE_EVENT_RESPONSE_NONE;
}

```

String Descriptor Table

Describes the String Descriptor Table o the device layer.

Description

The Device Layer allows the application to specify string descriptors via a String Descriptor Table. When the USB Host requests for a string by its index and language ID, the Device Layer looks for the corresponding string descriptor in the String Descriptor Table. There are two possible methods of specifying this String Descriptor Table, Basic and Advanced. These methods are discussed here.

Basic String Descriptor Table

The Basic String Descriptor Table should be used when the USB Device Application has equal number of string descriptors for each language

string indexes exist without any intervals. This method is the default method of specifying the String Descriptor Table. Each entry in the table contains the following information

- The size of the entry
- The descriptor type, which is always set to USB_DESCRIPTOR_STRING
- The array containing the string

The first entry in the String Descriptor Table, at index 0 of the table, will always contain the Lang ID string. This string specifies the one language ID of the String Descriptor that this application intends to support. The subsequent entries in the String Descriptor Table contain the actual string descriptor. Each language must have an equal set of the string descriptors. The Device layer will associate each set of string descriptors with language ID specified in the language ID string descriptor. The following code shows an example of a Basic String Descriptor table.

Example:

```
/* This code shows an example of a Basic String Descriptor Table. In
 * this example, the table contains five entries. The first entry is the
 * language ID string. The second entry in the manufacturer string and the third
 * entry is the product string for language ID 0x0409. The fourth and the fifth
 * entry is the manufacture and product string, respectively for the language ID
 * 0x040C. */

*****  

* Language ID string descriptor. Note that this contains two Language IDs.  

*****  

const struct  

{  

    uint8_t bLength;  

    uint8_t bDscType;  

    uint16_t string[1];  

}  

sd000 =  

{  

    sizeof(sd000),           // Size of this descriptor in bytes  

    USB_DESCRIPTOR_STRING,   // STRING descriptor type  

    {0x0409, 0x040C}         // Language ID
};  

*****  

* Manufacturer string descriptor  

*****  

const struct  

{  

    uint8_t bLength;          // Size of this descriptor in bytes  

    uint8_t bDscType;         // STRING descriptor type  

    uint16_t string[25];     // String
}  

sd001 =  

{  

    sizeof(sd001),  

    USB_DESCRIPTOR_STRING,  

    {'M','i','c','r','o','s','o','f','t','\0',  

     'T','e','c','h','n','o','l','o','g','y','\0','I','n','c','.\0'}
};  

*****  

* Product string descriptor  

*****  

const struct  

{  

    uint8_t bLength;          // Size of this descriptor in bytes  

    uint8_t bDscType;         // STRING descriptor type  

    uint16_t string[22];     // String
}  

sd002 =  

{  

    sizeof(sd002),  

    USB_DESCRIPTOR_STRING,  

    {'S','i','m','p','l','e','\0','C','D','C','\0','D','e','v','i','c','e','\0','D','e','m','o','\0'}
};  

*****  

* Manufacturer string descriptor
```

```
*****
const struct
{
    uint8_t bLength;           // Size of this descriptor in bytes
    uint8_t bDscType;          // STRING descriptor type
    uint16_t string[25];      // String
}
sd003 =
{
    sizeof(sd003),
    USB_DESCRIPTOR_STRING,
    {'M','i','c','r','o','c','h','i','p',' ','
     'T','e','c','h','n','o','l','o','g','y',' ','I','n','c','.'}
};

*****
* Product string descriptor
*****
const struct
{
    uint8_t bLength;           // Size of this descriptor in bytes
    uint8_t bDscType;          // STRING descriptor type
    uint16_t string[22];      // String
}
sd004 =
{
    sizeof(sd004),
    USB_DESCRIPTOR_STRING,
    {'S','i','m','p','l','e','C','D','C','D','e','v','i','c','e','D','e','m','o'}
};

*****
* Array of string descriptors
*****
USB_DEVICE_STRING_DESCRIPTORS_TABLE stringDescriptors[3]=
{
    /* This is the language ID string */
    (const uint8_t *const)&sd000,

    /* This string descriptor at index 1 will be returned when the host request
     * for a string descriptor with index 1 and language ID 0x0409. */
    (const uint8_t *const)&sd001,

    /* This string descriptor at index 2 will be returned when the host request
     * for a string descriptor with index 2 and language ID 0x0409. */
    (const uint8_t *const)&sd002,

    /* This string descriptor at index 3 will be returned when the host request
     * for a string descriptor with index 1 and language ID 0x040C. */
    (const uint8_t *const)&sd003,

    /* This string descriptor at index 4 will be returned when the host request
     * for a string descriptor with index 2 and language ID 0x040C. */
    (const uint8_t *const)&sd004
};
```

Advanced String Descriptor Table

The Advanced String Descriptor Table should be used when the application needs to specify string descriptors with string indexes that are not continuous. One such example is the Microsoft OS String Descriptor. The index of this string descriptor is 0xEE. If the application were to use the Basic String Descriptor Table , this would require the String Descriptor Table to have at least 0xED entries (valid or invalid) before the entry for the Microsoft OS String Descriptor. This may result in waste of RAM. Using the Ad Advanced String Descriptor Table mitigates this problem. The Advanced String Descriptor Table format is enabled only when [USB_DEVICE_STRING_DESCRIPTOR_TABLE_ADVANCED_ENABLE](#) configuration option is specified in the `system_config.h`. Each entry in the Advanced String Descriptor Table contains the following information:

- The index of the string descriptor
- The language ID of the string descriptor
- The size of the entry, which is two more than the length of the string
- The descriptor type, which is always set to `USB_DESCRIPTOR_STRING`

- The array containing the string

The first such entry in the Advanced String Descriptor Table specifies the language ID string. The string index and the language ID of this entry should be zero. This first entry is then followed by the actual string descriptors. Unlike the Basic String Descriptor Table, the index of an entry in the Advanced String Descriptor Table is not the same as the String Descriptor Index that the host uses to identify the string. In the Advanced String Descriptor Table, the index of the string is specified by the stringIndex member of the Advanced String Descriptor Table table entry. The following code shows an example of the Advanced String Descriptor table.

Example:

```
/* This code shows an example of an Advanced String Descriptor Table.
 * The Advanced String Descriptor table should be used when multiple languages
 * are needed to be supported. In this example, two languages are supported*/

*****  

* Language ID string descriptor. Note that stringIndex and
* language ID are always 0 for this descriptor.
*****  

const struct __attribute__ ((packed))
{
    uint8_t stringIndex;      // Index of the string descriptor
    uint16_t languageID ;    // Language ID of this string.
    uint8_t bLength;          // Size of this descriptor in bytes
    uint8_t bDscType;         // STRING descriptor type
    uint16_t string[2];       // String
}
sd000 =
{
    0,                         // Index of this string is 0
    0,                         // This field is always blank for String Index 0
    sizeof(sd000)- 3,        // Should always be set to this.
    USB_DESCRIPTOR_STRING,
    {0x0409, 0x040C}           // Language ID
};

*****  

* Manufacturer string descriptor for language 0x0409
*****  

const struct __attribute__ ((packed))
{
    uint8_t stringIndex;      // Index of the string descriptor
    uint16_t languageID ;    // Language ID of this string.
    uint8_t bLength;          // Size of this descriptor in bytes
    uint8_t bDscType;         // STRING descriptor type
    uint16_t string[25];      // String
}
sd001 =
{
    1,                         // Index of this string descriptor is 1.
    0x0409, // Language ID of this string descriptor is 0x0409 (English)
    sizeof(sd001) - 3,
    USB_DESCRIPTOR_STRING,
    {'M','i','c','r','o','c','h','i','p',' ',' ','T','e','c','h','n','o','l','o','g','y',' ',' ','I','n','d','u','s','t','r','y',' ',' ','.'}
};

*****  

* Manufacturer string descriptor for language 0x040C
*****  

const struct __attribute__ ((packed))
{
    uint8_t stringIndex;      // Index of the string descriptor
    uint16_t languageID ;    // Language ID of this string.
    uint8_t bLength;          // Size of this descriptor in bytes
    uint8_t bDscType;         // STRING descriptor type
    uint16_t string[25];      // String
}
sd002 =
{
    1,                         // Index of this string descriptor is 1.
    0x040C, // Language ID of this string descriptor is 0x040C (French)
```

```

sizeof(sd001) - 3,
USB_DESCRIPTOR_STRING,
{'M','i','c','r','o','c','h','i','p',' ',',
'T','e','c','h','n','o','l','o','g','y',' ','I','n','c','.'}
};

<***** Product string descriptor for language 0x409 *****/
const struct __attribute__ ((packed))
{
    uint8_t stringIndex;      // Index of the string descriptor
    uint16_t languageID ;    // Language ID of this string.
    uint8_t bLength;         // Size of this descriptor in bytes
    uint8_t bDscType;        // STRING descriptor type
    uint16_t string[22];     // String
}
sd003 =
{
    2,           // Index of this string descriptor is 2.
    0x0409,     // Language ID of this string descriptor is 0x0409 (English)
    sizeof(sd002) - 3,
    USB_DESCRIPTOR_STRING,
    {'S','i','m','p','l','e',' ', 'C','D','C',' ', 'D','e','v','i','c','e',' ', 'D','e','m','o' }
};

<***** Product string descriptor for language 0x40C *****/
const struct __attribute__ ((packed))
{
    uint8_t stringIndex;      // Index of the string descriptor
    uint16_t languageID ;    // Language ID of this string.
    uint8_t bLength;         // Size of this descriptor in bytes
    uint8_t bDscType;        // STRING descriptor type
    uint16_t string[22];     // String
}
sd004 =
{
    2,           // Index of this string descriptor is 2.
    0x0409,     // Language ID of this string descriptor is 0x040C (French)
    sizeof(sd002) - 3,
    USB_DESCRIPTOR_STRING,
    {'S','i','m','p','l','e',' ', 'C','D','C',' ', 'D','e','v','i','c','e',' ', 'D','e','m','o' }
};

<***** Array of string descriptors. The entry order does not matter. *****/
USB_DEVICE_STRING_DESCRIPTORS_TABLE stringDescriptors[5]=
{
    (const uint8_t *const)&sd000,
    (const uint8_t *const)&sd001,    // Manufacturer string for language 0x0409
    (const uint8_t *const)&sd002,    // Manufacturer string for language 0x040C
    (const uint8_t *const)&sd003,    // Product string for language 0x0409
    (const uint8_t *const)&sd004,    // Product string for language 0x040C
};

```

BOS Descriptor Support

Provides information on the BOS descriptor.

Description

The USB 3.0 and the USB 2.0 LPM specifications define a new descriptor called the Binary Device Object Store (BOS) descriptor. This descriptor contains information of the capability of the device. When the bcdUSB value in the Device Descriptor is greater than 0x0200, the USB Host Operating System may request for the BOS descriptor.

The MPLAB Harmony USB Device Library allows the application to support the BOS descriptor requests. This support is enabled by adding the [USB_DEVICE_BOS_DESCRIPTOR_SUPPORT_ENABLE](#) configuration macro in `system_config.h`. The application must set the

bosDescriptor member of the `USB_DEVICE_INIT` data structure (this data structure is passed in the `USB_DEVICE_Initialize` function) to point to the data to be returned in the data stage of the BOS descriptor request.

If the `USB_DEVICE_BOS_DESCRIPTOR_SUPPORT_ENABLE` configuration macro is not specified, the Device Layer will stall the Host request BOS descriptor.

Configuring the Library

Describes how to configure the USB Device Library.

Macros

	Name	Description
	<code>USB_DEVICE_INSTANCES_NUMBER</code>	Number of Device Layer instances to provisioned in the application.
	<code>USB_DEVICE_ENDPOINT_QUEUE_DEPTH_COMBINED</code>	Specifies the combined endpoint queue depth in case of a vendor USB device implementation.
	<code>USB_DEVICE_SET_DESCRIPTOR_EVENT_ENABLE</code>	Enables the Device Layer Set Descriptor Event.
	<code>USB_DEVICE_SOF_EVENT_ENABLE</code>	Enables the Device Layer SOF event.
	<code>USB_DEVICE_SYNCH_FRAME_EVENT_ENABLE</code>	Enables the Device Layer Synch Frame Event.
	<code>USB_DEVICE_EP0_BUFFER_SIZE</code>	Buffer Size in Bytes for Endpoint 0.
	<code>USB_DEVICE_MICROSOFT_OS_DESCRIPTOR_SUPPORT_ENABLE</code>	Specifies if the USB Device stack should support Microsoft OS Descriptor.

Description

The USB Device Layer initializes and configures the USB Controller Driver (the driver that manages the USB peripheral when operating as device) and maintains its task routine. For completeness, the following table lists the configuration macros that are needed by the USB Controller Driver. These macros should be defined in `system_config.h` file along with the Device Layer Configuration macros.

USB_DEVICE_INSTANCES_NUMBER Macro

Number of Device Layer instances to provisioned in the application.

File

`usb_device_config_template.h`

C

```
#define USB_DEVICE_INSTANCES_NUMBER 1
```

Description

Number of Device Layer Instances.

This configuration macro defines the number of Device Layer instances in the application. In cases of microcontrollers that feature multiple USB peripherals, this allows the application to run multiple instances of the Device Layer. As an example, for a microcontroller containing two USB peripherals, setting `USB_DEVICE_INSTANCES_NUMBER` to 2 will cause 2 instances of the Device Layer to execute.

Remarks

Setting this value to more than the number of USB peripheral will result if unused RAM consumption.

USB_DEVICE_ENDPOINT_QUEUE_DEPTH_COMBINED Macro

Specifies the combined endpoint queue depth in case of a vendor USB device implementation.

File

`usb_device_config_template.h`

C

```
#define USB_DEVICE_ENDPOINT_QUEUE_DEPTH_COMBINED 2
```

Description

USB Device Layer Combined Endpoint Queue Depth

This configuration constant specifies the combined endpoint queue depth in a case where the endpoint read and endpoint write functions are used to implement a vendor USB device. This constant should be used in conjunction with the `usb_device_endpoint_functions.c` file.

This macro defines the number of entries in all IN and OUT endpoint queues in all instances of the USB Device Layer. This value can be obtained

by adding up the endpoint read and write queue sizes of each USB Device Layer instance . In a simple single instance USB Device Layer, that requires only one read and one write endpoint with one buffer each, the `USB_DEVICE_ENDPOINT_QUEUE_DEPTH_COMBINED` macro can be set to 2. Consider a case with one Device Layer instance using 2 IN and 2 OUT endpoints, each requiring 2 buffers, this macro should be set to 8 (2 + 2 + 2 + 2).

Remarks

This constant needs to be specified only if a Vendor USB Device is to be implemented and the `usb_device_endpoint_functions.c` file is included in the project. This constant does not have any effect on queues of other standard function drivers that are included in the USB device implementation.

`USB_DEVICE_SET_DESCRIPTOR_EVENT_ENABLE` Macro

Enables the Device Layer Set Descriptor Event.

File

[usb_device_config_template.h](#)

C

```
#define USB_DEVICE_SET_DESCRIPTOR_EVENT_ENABLE
```

Description

USB Device Layer Set Descriptor Event Enable

Specifying this configuration macro will cause the USB Device Layer to generate the `USB_DEVICE_EVENT_SET_DESCRIPTOR` event when a Set Descriptor request is received. If this macro is not defined, the USB Device Layer will stall the Set Descriptor control transfer request.

Remarks

None.

`USB_DEVICE_SOF_EVENT_ENABLE` Macro

Enables the Device Layer SOF event.

File

[usb_device_config_template.h](#)

C

```
#define USB_DEVICE_SOF_EVENT_ENABLE
```

Description

USB Device Layer SOF Event Enable

Specifying this configuration macro will cause the USB Device Layer to generate the `USB_DEVICE_EVENT_SOF` event. On Full Speed USB Devices, this event will be generated every 1 millisecond. On High Speed USB devices, this event will be generated every 125 microsecond.

Remarks

None.

`USB_DEVICE_SYNCH_FRAME_EVENT_ENABLE` Macro

Enables the Device Layer Synch Frame Event.

File

[usb_device_config_template.h](#)

C

```
#define USB_DEVICE_SYNCH_FRAME_EVENT_ENABLE
```

Description

USB Device Layer Synch Frame Event Enable

Specifying this configuration macro will cause the USB Device Layer to generate the `USB_DEVICE_EVENT_SYNCH_FRAME` event when a Synch Frame control transfer request is received. If this macro is not defined, the USB Device Layer will stall the control transfer request associated with this event.

Remarks

None.

USB_DEVICE_EP0_BUFFER_SIZE Macro

Buffer Size in Bytes for Endpoint 0.

File

[usb_device_config_template.h](#)

C

```
#define USB_DEVICE_EP0_BUFFER_SIZE
```

Description

Endpoint 0 Buffer Size

This number defines the size (in bytes) of Endpoint 0. For High Speed USB Devices, this number should be 64. For Full Speed USB Devices, this number can be 8, 16, 32 or 64 bytes. This number will be applicable to all USB Device Stack instances.

Remarks

None.

USB_DEVICE_MICROSOFT_OS_DESCRIPTOR_SUPPORT_ENABLE Macro

Specifies if the USB Device stack should support Microsoft OS Descriptor.

File

[usb_device_config_template.h](#)

C

```
#define USB_DEVICE_MICROSOFT_OS_DESCRIPTOR_SUPPORT_ENABLE
```

Description

USB Device Layer Microsoft OS Descriptor Support Enable

This macro needs to be defined to enable Microsoft OS descriptor support. If this macro is defined all Vendor Interface requests are forwarded to client unconditionally and Device layer does not validate the recipient interface field in a Control transfer.

Device and Class Control Requests are not affected by this configuration. The Device Layer will validate the recipient interface in Device and Class Control Requests, irrespective of this configuration constant, and will stall these requests if the interface is provisioned in the Function Driver Registration Table.

If this macro is not defined, then USB Device Layer will validate the interface number in a Vendor Interface request and stall the request if Interface number is not available in the Function registration table.

Remarks

None.

Building the Library

This section lists the files that are available in the USB Device Layer Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/usb.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
usb_device.h	This header file should be included in any .c file that accesses the USB Device Layer API.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/usb_device.c	This file implements the USB Device Layer interface and should be included in project if USB Device mode operation is desired.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB Device Layer Library depends on the following modules:

- USB Driver Library (Device mode files)

Library Interface

a) System Interaction Functions

	Name	Description
≡	USB_DEVICE_Initialize	Creates and initializes an instance of the USB device layer.
≡	USB_DEVICE_Deinitialize	De-initializes the specified instance of the USB device layer.
≡	USB_DEVICE_Status	Provides the current status of the USB device layer
≡	USB_DEVICE_Tasks	USB Device layer calls all other function driver tasks in this function. It also generates and forwards events to its clients.
≡	USB_DEVICE_Tasks_ISR	USB Device Layer Tasks routine to be called in the USB Interrupt Service Routine.
≡	USB_DEVICE_Tasks_ISR_USBDMA	This is function USB_DEVICE_Tasks_ISR_USBDMA.

b) Client Core Functions

	Name	Description
≡	USB_DEVICE_Open	Opens the specified USB device layer instance and returns a handle to it.
≡	USB_DEVICE_Close	Closes an opened handle to an instance of the USB device layer.
≡	USB_DEVICE_ClientStatusGet	Returns the client specific status.
≡	USB_DEVICE_EventHandlerSet	USB Device Layer Event Handler Callback Function set function.

c) Device Power State Management Functions

	Name	Description
≡	USB_DEVICE_PowerStateSet	Sets power state of the device.
≡	USB_DEVICE_RemoteWakeUpStatusGet	Gets the "Remote wake-up" status of the device.
≡	USB_DEVICE_IsSuspended	Returns true if the device is in a suspended state.
≡	USB_DEVICE_RemoteWakeUpStart	This function will start the resume signaling.
≡	USB_DEVICE_RemoteWakeUpStartTimed	This function will start a self timed Remote Wake-up.
≡	USB_DEVICE_RemoteWakeUpStop	This function will stop the resume signaling.

d) Device Management Functions

	Name	Description
≡	USB_DEVICE_StateGet	Returns the current state of the USB device.
≡	USB_DEVICE_Attach	This function will attach the device to the USB.
≡	USB_DEVICE_Detach	This function will detach the device from the USB.
≡	USB_DEVICE_ActiveConfigurationGet	Informs the client of the current USB device configuration set by the USB host.
≡	USB_DEVICE_ActiveSpeedGet	Informs the client of the current operation speed of the USB bus.

e) Endpoint Management Functions

	Name	Description
≡	USB_DEVICE_EndpointIsStalled	This function returns the stall status of the specified endpoint and direction.
≡	USB_DEVICE_EndpointStall	This function stalls an endpoint in the specified direction.
≡	USB_DEVICE_EndpointStallClear	This function clears the stall on an endpoint in the specified direction.
≡	USB_DEVICE_EndpointDisable	Disables a device endpoint.
≡	USB_DEVICE_EndpointEnable	Enables a device endpoint.
≡	USB_DEVICE_EndpointIsEnabled	Returns true if the endpoint is enabled.
≡	USB_DEVICE_EndpointRead	Reads data received from host on the requested endpoint.
≡	USB_DEVICE_EndpointTransferCancel	This function cancels a transfer scheduled on an endpoint.
≡	USB_DEVICE_EndpointWrite	This function requests a data write to a USB Device Endpoint.

f) Control Transfer Functions

	Name	Description
≡	USB_DEVICE_ControlReceive	Receives data stage of the control transfer from host to device.
≡	USB_DEVICE_ControlSend	Sends data stage of the control transfer from device to host.
≡	USB_DEVICE_ControlStatus	Initiates status stage of the control transfer.

g) Data Types and Constants

	Name	Description
	USB_DEVICE_INDEX_0	USB device layer index definitions.
	USB_DEVICE_INDEX_1	This is macro USB_DEVICE_INDEX_1.
	USB_DEVICE_INDEX_2	This is macro USB_DEVICE_INDEX_2.
	USB_DEVICE_INDEX_3	This is macro USB_DEVICE_INDEX_3.
	USB_DEVICE_INDEX_4	This is macro USB_DEVICE_INDEX_4.
	USB_DEVICE_INDEX_5	This is macro USB_DEVICE_INDEX_5.
	USB_DEVICE_CONTROL_STATUS	USB Device Layer Control Transfer Status Stage flags.
	USB_DEVICE_CONTROL_TRANSFER_RESULT	Enumerated data type identifying results of a control transfer.
	USB_DEVICE_EVENT	USB Device Layer Events.
	USB_DEVICE_HANDLE	Data type for USB device handle.
	USB_DEVICE_INIT	USB Device Initialization Structure
	USB_DEVICE_POWER_STATE	Enumerated data type that identifies if the device is self powered or bus powered .
	USB_DEVICE_REMOTE_WAKEUP_STATUS	Enumerated data type that identifies if the remote wakeup status of the device.
	USB_DEVICE_EVENT_DATA_CONFIGURED	USB Device Set Configuration Event Data type.
	USB_DEVICE_HANDLE_INVALID	Constant that defines the value of an Invalid Device Handle.
	USB_DEVICE_EVENT_RESPONSE_NONE	Device Layer Event Handler Function Response Type.
	USB_DEVICE_CLIENT_STATUS	Enumerated data type that identifies the USB Device Layer Client Status.
	USB_DEVICE_CONFIGURATION_DESCRIPTOR_TABLE	Pointer to an array that contains pointer to configuration descriptors.
	USB_DEVICE_EVENT_HANDLER	USB Device Layer Event Handler Function Pointer Type
	USB_DEVICE_EVENT_RESPONSE	Device Layer Event Handler function return type.
	USB_DEVICE_FUNCTION_REGISTRATION_TABLE	USB Device Function Registration Structure
	USB_DEVICE_MASTER_DESCRIPTOR	USB Device Master Descriptor Structure.
	USB_DEVICE_STRING_DESCRIPTOR_TABLE	Pointer to an array that contains pointer to string descriptors.
	USB_DEVICE_EVENT_DATA_ENDPOINT_READ_COMPLETE	USB Device Layer Endpoint Read and Write Complete Event Data type.
	USB_DEVICE_EVENT_DATA_ENDPOINT_WRITE_COMPLETE	USB Device Layer Endpoint Read and Write Complete Event Data type.
	USB_DEVICE_EVENT_DATA_SET_DESCRIPTOR	USB Device Set Descriptor Event Data type.
	USB_DEVICE_EVENT_DATA_SOFTWARE	USB Device Start Of Frame Event Data Type
	USB_DEVICE_EVENT_DATA_SYNCH_FRAME	USB Device Synch Frame Event Data type.

	USB_DEVICE_RESULT	USB Device Layer Results Enumeration
	USB_DEVICE_TRANSFER_FLAGS	Enumerated data type that identifies the USB Device Layer Transfer Flags.
	USB_DEVICE_TRANSFER_HANDLE	Data type for USB Device Endpoint Data Transfer Handle.
	USB_DEVICE_TRANSFER_HANDLE_INVALID	Constant that defines the value of an Invalid Device Endpoint Data Transfer Handle.
◆	_USB_DEVICE_IRP	This structure defines the USB Device Mode IRP data structure.
◆	_USB_HOST_IRP	This structure defines the USB Host Mode IRP data structure.
◆	_USB_HOST_IRP_STATUS	Enumerates the possible status options of USB Host IRP.
	USB_DEVICE_IRP	This structure defines the USB Device Mode IRP data structure.
	USB_DEVICE_IRP_FLAG	USB Device IRP flags enumeration
	USB_DEVICE_IRP_STATUS	Enumerates the possible status options of USB Device IRP.
	USB_ENDPOINT	Defines a type to store Endpoint and Direction. The MSB defines the direction. The lower 4 bits defines the endpoint.
	USB_ERROR	Enumeration of all possible error codes that are returned by various components functions in the USB Stack.
	USB_HOST_IRP	This structure defines the USB Host Mode IRP data structure.
	USB_HOST_IRP_FLAG	USB Host IRP flags enumeration
	USB_HOST_IRP_STATUS	Enumerates the possible status options of USB Host IRP.
	USB_SPEED	Provides enumeration of USB 2.0 speeds.
	USB_ENDPOINT_AND_DIRECTION	This macro helps in setting up the USB_ENDPOINT type.
	USB_DATA_DIRECTION	Defines the communication direction
	USB_DEVICE_BOS_DESCRIPTOR_SUPPORT_ENABLE	Specifies if the Device Layer should process a Host request for a BOS descriptor.
	USB_DEVICE_DRIVER_INITIALIZE_EXPLICIT	Specifies if the USB Controller Driver must be initialized explicitly as opposed to being initialized by the Device Layer.
	USB_DEVICE_STRING_DESCRIPTOR_TABLE_ADVANCED_ENABLE	Specifying this macro enables the Advanced String Descriptor Table Entry Format.

Description

This section describes the Application Programming Interface (API) functions of the USB Device Layer Library.

Refer to each section for a detailed description.

a) System Interaction Functions

USB_DEVICE_Initialize Function

Creates and initializes an instance of the USB device layer.

File

[usb_device.h](#)

C

```
SYS_MODULE_OBJ USB_DEVICE_Initialize(const SYS_MODULE_INDEX instanceIndex, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to a device layer object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

Description

This function initializes an instance of USB device layer, making it ready for clients to open and use it. The number of instances is limited by the value of macro [USB_DEVICE_MAX_INSTANCES](#) defined in [system_config.h](#) file.

Remarks

This routine must be called before any other USB Device Layer routine is called and after the initialization of USB Device Driver. This routine should only be called once during system initialization.

Preconditions

None.

Example

```
// This code example shows the initialization of the
// the USB Device Layer. Note how an endpoint table is
// created and assigned.

USB_DEVICE_INIT deviceLayerInit;
SYS_MODULE_OBJ usbDeviceObj;
uint8_t __attribute__((aligned(512))) endpointTable[USB_DEVICE_ENDPOINT_TABLE_SIZE];

// System module initialization
deviceLayerInit.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;

// Identifies peripheral (PLIB-level) ID
deviceLayerInit.usbID = USB_ID_1;

// Boolean flag: true -> Stop USB module in Idle Mode
deviceLayerInit.stopInIdle = false;

// Boolean flag: true -> Suspend USB in Sleep Mode
deviceLayerInit.suspendInSleep = false;

// Interrupt Source for USB module
deviceLayerInit.interruptSource = INT_SOURCE_USB_1;

// Number of function drivers registered to this instance of the
// USB device layer
deviceLayerInit.registeredFuncCount = 1;

// Function driver table registered to this instance of the USB device layer
deviceLayerInit.registeredFunctions = funcRegistrationTable;

// Pointer to USB Descriptor structure
deviceLayerInit.usbMasterDescriptor = &usbMasterDescriptor;

// Pointer to an endpoint table.
deviceLayerInit.endpointTable = endpointTable;

// USB device initialization
usbDeviceObj = USB_DEVICE_Initialize(USB_DEVICE_INDEX_0, &deviceLayerInit);

if (SYS_MODULE_OBJ_INVALID == usbDeviceObj)
{
    // Handle error
}
```

Parameters

Parameters	Description
instanceIndex	In case of microcontrollers with multiple USB peripherals, user can create multiple instances of USB device layer. Parameter instanceIndex identifies this instance.
init	Pointer to a data structure containing any data necessary to initialize the USB device layer

Function

```
SYS_MODULE_OBJ USB_DEVICE_Initialize
(
const SYS_MODULE_INDEX instanceIndex,
const SYS_MODULE_INIT * const init
)
```

USB_DEVICE_Deinitialize Function

De-initializes the specified instance of the USB device layer.

File

[usb_device.h](#)

C

```
void USB_DEVICE_Deinitialize(SYS_MODULE_OBJ usbDeviceObj);
```

Returns

None.

Description

This function deinitializes the specified instance of the USB device layer, disabling its operation (and any hardware) and invalidates all of the internal data.

Remarks

Once the Initialize operation has been called, the deinitialize operation must be called before the Initialize operation can be called again.

Preconditions

Function [USB_DEVICE_Initialize](#) must have been called before calling this routine and a valid SYS_MODULE_OBJ must have been returned.

Example

```
// This code example shows how the USB
// Device Layer can be deinitialized. It is assumed the
// USB Device Layer was already initialized.
```

```
SYS_MODULE_OBJ usbDeviceobj;
```

```
USB_DEVICE_Deinitialize(usbDeviceobj);
```

Parameters

Parameters	Description
object	USB device layer object handle, returned by USB_DEVICE_Initialize

Function

```
void USB_DEVICE_Deinitialize ( SYS_MODULE_OBJ usbDeviceobj )
```

USB_DEVICE_Status Function

Provides the current status of the USB device layer

File

[usb_device.h](#)

C

```
SYS_STATUS USB_DEVICE_Status(SYS_MODULE_OBJ object);
```

Returns

SYS_STATUS_READY - Indicates that the device is busy with a previous system level operation and cannot start another

SYS_STATUS_UNINITIALIZED - Indicates that the device layer is in a deinitialized state

Description

This function provides the current status of the USB device layer.

Remarks

None.

Preconditions

The [USB_DEVICE_Initialize](#) function must have been called before calling this function.

Example

```
// This code example shows how the USB_DEVICE_Status function
// can be used to check if the USB Device Layer is ready
// for client operations.

SYS_MODULE_OBJ      object;      // Returned from DRV_USB_Initialize
SYS_STATUS          status;

status = USB_DEVICE_Status(object);

if (SYS_STATUS_READY != status)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Driver object handle, returned from USB_DEVICE_Initialize

Function

SYS_STATUS USB_DEVICE_Status (SYS_MODULE_OBJ object)

USB_DEVICE_Tasks Function

USB Device layer calls all other function driver tasks in this function. It also generates and forwards events to its clients.

File

[usb_device.h](#)

C

```
void USB_DEVICE_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function must be periodically called by the user application. The USB Device layer calls all other function driver tasks in this function. It also generates and forwards events to its clients.

Remarks

None.

Preconditions

Device layer must have been initialized by calling [USB_DEVICE_Initialize](#).

Example

```
// The USB_DEVICE_Tasks() function should be placed in the
// SYS_Tasks() function of a MPLAB Harmony application.

SYS_MODULE_OBJ usbDeviceLayerObj; // Returned by USB_DEVICE_Initialize().

void SYS_Tasks(void)
{
    USB_DEVICE_Tasks(usbDeviceLayerObj);
}
```

Parameters

Parameters	Description
devLayerObj	Pointer to the Device Layer Object that is returned from USB_DEVICE_Initialize

Function

void USB_DEVICE_Tasks(SYS_MODULE_OBJ devLayerObj)

USB_DEVICE_Tasks_ISR Function

USB Device Layer Tasks routine to be called in the USB Interrupt Service Routine.

File

[usb_device.h](#)

C

```
void USB_DEVICE_Tasks_ISR(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function must be called in the USB Interrupt Service Routine if the Device Stack is configured for interrupt mode. In case the Device Stack is configured for polling mode, this function is automatically called from the [USB_DEVICE_Tasks](#) function. devLayerObj must be the system module object associated with the USB module generating the interrupt.

Remarks

None.

Preconditions

Device layer must have been initialized.

Example

```
// devLayerObj is returned while initializing the USB1 module.  
// The USB_DEVICE_Tasks_ISR function should be placed in the  
// USB1 module ISR.  
  
SYS_MODULE_OBJ devLayerObj; // Returned by USB_DEVICE_Initialize().  
  
void __ISR(_USB_1_VECTOR, ipl4) USB1InterruptHandle(void)  
{  
    USB_DEVICE_Tasks_ISR(devLayerObj);  
}
```

Parameters

Parameters	Description
devLayerObj	Pointer to the Device Layer Object that is returned from USB_DEVICE_Initialize

Function

```
void USB_DEVICE_Tasks_ISR( SYS_MODULE_OBJ devLayerObj )
```

USB_DEVICE_Tasks_ISR_USBDMA Function

File

[usb_device.h](#)

C

```
void USB_DEVICE_Tasks_ISR_USBDMA( SYS_MODULE_OBJ devLayerObj );
```

Description

This is function USB_DEVICE_Tasks_ISR_USBDMA.

b) Client Core Functions

USB_DEVICE_Open Function

Opens the specified USB device layer instance and returns a handle to it.

File

[usb_device.h](#)

C

```
USB_DEVICE_HANDLE USB_DEVICE_Open(const SYS_MODULE_INDEX instanceIndex, const DRV_IO_INTENT intent);
```

Returns

If successful, returns a valid device layer handle. Otherwise, it returns [USB_DEVICE_HANDLE_INVALID](#).

Description

This function opens the USB device layer instance specified by instance index and returns a handle. This handle must be provided to all other client operations to identify the caller and the instance of the USB device layer. An instance of the Device Layer can be opened only once. Trying to open the Device Layer more than once will return a invalid device handle.

Remarks

None.

Preconditions

This function must be called after USB device driver initialization and after the initialization of USB Device Layer.

Example

```
// This code example shows how the
// USB Device Layer can be opened.

USB_DEVICE_HANDLE usbDeviceHandle;

// Before opening a handle, USB device must have been initialized
// by calling USB_DEVICE_Initialize().

usbDeviceHandle = USB_DEVICE_Open( USB_DEVICE_INDEX_0,
                                  DRV_IO_INTENT_READWRITE );

if(USB_DEVICE_INVALID == usbDeviceHandle)
{
    //Failed to open handle.
}
```

Parameters

Parameters	Description
instanceIndex	USB device layer instance index
intent	This parameter is ignored. The Device Layer will always open in read/write and exclusive mode.

Function

```
USB_DEVICE_HANDLE USB_DEVICE_Open
(
    const SYS_MODULE_INDEX instanceIndex,
    const DRV_IO_INTENT intent
)
```

[USB_DEVICE_Close Function](#)

Closes an opened handle to an instance of the USB device layer.

File

[usb_device.h](#)

C

```
void USB_DEVICE_Close(USB_DEVICE_HANDLE usbDeviceHandle);
```

Returns

None

Description

This function closes an opened handle to an instance of the USB device layer, invalidating the handle.

Remarks

After calling this routine, the handle passed in "usbDevHandle" must not be used with any of the remaining driver routines. A new handle must be obtained by calling [USB_DEVICE_Open\(\)](#) before the client may use the device layer again.

Preconditions

The [USB_DEVICE_Initialize](#) function must have been called for the specified device layer instance. [USB_DEVICE_Open](#) must have been called to obtain a valid opened device handle.

Example

```
USB_DEVICE_HANDLE usbDeviceHandle;

// Before opening a handle, USB device must have been initialized
// by calling USB_DEVICE_Initialize().
usbDeviceHandle = USB_DEVICE_Open( USB_DEVICE_INDEX_0 );

if(USB_DEVICE_HANDLE_INVALID == usbDeviceHandle)
{
    //Failed to open handle.
}

.....
.....
// User's code
.....
.....
// Close handle
USB_DEVICE_Close( usbDevHandle );
```

Parameters

Parameters	Description
handle	A valid open-instance handle, returned from USB_DEVICE_Open

Function

```
void USB_DEVICE_Close( USB_DEVICE_HANDLE usbDeviceHandle )
```

[**USB_DEVICE_ClientStatusGet Function**](#)

Returns the client specific status.

File

[usb_device.h](#)

C

```
USB_DEVICE_CLIENT_STATUS USB_DEVICE_ClientStatusGet(USB_DEVICE_HANDLE usbDeviceHandle);
```

Returns

[USB_DEVICE_CLIENT_STATUS](#) type of client status.

Description

This function returns the status of the client (ready or closed). The application can use this function to query the present state of a client. Some of the USB Device Layer functions do not have any effect if the client handle is invalid. The [USB_DEVICE_ClientStatusGet](#) function in such cases can be used for debugging or trouble shooting.

Remarks

The application may ordinarily not find the need to use this function. It can be used for troubleshooting or debugging purposes.

Preconditions

The USB device layer must have been initialized and opened before calling this function.

Example

```
// This code example shows usage of the
// USB_DEVICE_ClientStatusGet function.

if(USB_DEVICE_CLIENT_STATUS_READY == USB_DEVICE_ClientStatusGet(usbDeviceHandle))
{
    // Client handle is valid.
    if(USB_DEVICE_IsSuspended(usbDeviceHandle))
    {
        // Device is suspended. Do something here.
    }
}
```

Parameters

Parameters	Description
usbDeviceHandle	Pointer to the device layer handle that is returned from USB_DEVICE_Open

Function

```
USB_DEVICE_CLIENT_STATUS USB_DEVICE_ClientStatusGet
(
    USB_DEVICE_HANDLE usbDeviceHandle
)
```

USB_DEVICE_EventHandlerSet Function

USB Device Layer Event Handler Callback Function set function.

File

[usb_device.h](#)

C

```
void USB_DEVICE_EventHandlerSet(USB_DEVICE_HANDLE usbDeviceHandle, const USB_DEVICE_EVENT_HANDLER
callBackFunc, uintptr_t context);
```

Returns

None.

Description

This is the USB Device Layer Event Handler Callback Set function. A client can receive USB Device Layer event by using this function to register and event handler callback function. The client can additionally specify a specific context which will be returned with the event handler callback function.

Remarks

None.

Preconditions

The device layer must have been initialized by calling [USB_DEVICE_Initialize](#) and a valid handle to the instance must have been obtained by calling [USB_DEVICE_Open](#).

Example

```
// This code example shows how the application can set
// a Device Layer Event Handler.

// Application states
typedef enum
{
    //Application's state machine's initial state.
    APP_STATE_INIT=0,
    APP_STATE_SERVICE_TASKS,
    APP_STATE_WAIT_FOR_CONFIGURATION,
} APP_STATES;

USB_DEVICE_HANDLE usbDeviceHandle;
```

```
APP_STATES gameState;

// This is the application device layer event handler function.

USB_DEVICE_EVENT_RESPONSE APP_USBDeviceEventHandler
(
    USB_DEVICE_EVENT event,
    void * pData,
    uintptr_t context
)
{
    USB_SETUP_PACKET * setupPacket;
    switch(event)
    {
        case USB_DEVICE_EVENT_POWER_DETECTED:
            // This event is generated when VBUS is detected. Attach the device
            USB_DEVICE_Attach(usbDeviceHandle);
            break;

        case USB_DEVICE_EVENT_POWER_REMOVED:
            // This event is generated when VBUS is removed. Detach the device
            USB_DEVICE_Detach (usbDeviceHandle);
            break;

        case USB_DEVICE_EVENT_CONFIGURED:
            // This event indicates that Host has set Configuration in the Device.
            break;

        case USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST:
            // This event indicates a Control transfer setup stage has been completed.
            setupPacket = (USB_SETUP_PACKET *)pData;

            // Parse the setup packet and respond with a USB_DEVICE_ControlSend(),
            // USB_DEVICE_ControlReceive or USB_DEVICE_ControlStatus().

            break;

        case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT:
            // This event indicates that a Control transfer Data has been sent to Host.
            break;

        case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA RECEIVED:
            // This event indicates that a Control transfer Data has been received from Host.
            break;

        case USB_DEVICE_EVENT_CONTROL_TRANSFER_ABORTED:
            // This event indicates a control transfer was aborted.
            break;

        case USB_DEVICE_EVENT_SUSPENDED:
            break;

        case USB_DEVICE_EVENT_RESUMED:
            break;

        case USB_DEVICE_EVENT_ERROR:
            break;

        case USB_DEVICE_EVENT_RESET:
            break;

        case USB_DEVICE_EVENT_SOF:
            // This event indicates an SOF is detected on the bus. The USB_DEVICE_SOF_EVENT_ENABLE
            // macro should be defined to get this event.
            break;
        default:
            break;
    }
}
```

```

void APP_Tasks ( void )
{
    // Check the application's current state.
    switch ( appState )
    {
        // Application's initial state.
        case APP_STATE_INIT:
            // Open the device layer
            usbDeviceHandle = USB_DEVICE_Open( USB_DEVICE_INDEX_0,
                DRV_IO_INTENT_READWRITE );

            if(usbDeviceHandle != USB_DEVICE_HANDLE_INVALID)
            {
                // Register a callback with device layer to get event notification
                USB_DEVICE_EventHandlerSet(usbDeviceHandle,
                    APP_USBDeviceEventHandler, 0);
                appState = APP_STATE_WAIT_FOR_CONFIGURATION;
            }
            else
            {
                // The Device Layer is not ready to be opened. We should try
                // gain later.
            }
            break;

        case APP_STATE_SERVICE_TASKS:
            break;

            // The default state should never be executed.
        default:
            break;
    }
}

```

Parameters

Parameters	Description
usbDeviceHandle	Pointer to the device layer handle that is returned from USB_DEVICE_Open
callBackFunc	Pointer to the call back function. The device layer calls notifies the client about bus event by calling this function.
context	Client specific context

Function

```

void USB_DEVICE_EventHandlerSet
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    USB_DEVICE_EVENT_HANDLER *callBackFunc,
    uintptr_t context
)

```

c) Device Power State Management Functions

USB_DEVICE_PowerStateSet Function

Sets power state of the device.

File

[usb_device.h](#)

C

```

void USB_DEVICE_PowerStateSet(USB_DEVICE_HANDLE usbDeviceHandle, USB_DEVICE_POWER_STATE powerState);

```

Returns

None.

Description

Application clients can use this function to set the power state of the device. A USB device can be bus powered or self powered. Based on hardware configuration, this power state may change while the device is on operation. The application can call this function to update the Device Layer on the present power status of the device.

Remarks

By default, the device is bus powered.

Preconditions

The device layer should have been initialized and opened.

Example

```
// The following code example shows how the application can
// change the power state of the device. In this case the application checks
// if a battery is charged and if so, the application set the device power
// state to self-powered.

if(APP_BATTERY_IS_CHARGED == APP_BatteryChargeStatusGet())
{
    // The application switches if power source.

    APP_PowerSourceSwitch(APP_POWER_SOURCE_BATTERY);
    USB_DEVICE_PowerStateSet(usbDeviceHandle, USB_DEVICE_POWER_STATE_SELF_POWERED);
}
else
{
    // The battery is still not charged. The application uses the USB power.

    USB_DEVICE_PowerStateSet(usbDeviceHandle, USB_DEVICE_POWER_STATE_BUS_POWERED);
}
```

Parameters

Parameters	Description
usbDeviceHandle	USB device handle returned by USB_DEVICE_Open() .
powerState	USB_DEVICE_POWER_STATE_BUS_POWERED / USB_DEVICE_POWER_STATE_SELF_POWERED

Function

```
void USB_DEVICE_PowerStateSet
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    USB_DEVICE_POWER_STATE powerState
)
```

[USB_DEVICE_RemoteWakeupStatusGet Function](#)

Gets the "Remote wake-up" status of the device.

File

[usb_device.h](#)

C

```
USB_DEVICE_REMOTE_WAKEUP_STATUS USB_DEVICE_RemoteWakeupStatusGet(USB_DEVICE_HANDLE usbDeviceHandle);
```

Returns

[USB_DEVICE_REMOTE_WAKEUP_ENABLED](#) - Remote wakeup is enabled. [USB_DEVICE_REMOTE_WAKEUP_DISABLED](#) - Remote wakeup is disabled.

Description

This function returns the present "Remote Wake-up" status of the device. If the device supports remote wake-up, the host may enable or disable this feature. The client can use this function to find out the status of this feature.

Remarks

None.

Preconditions

The device layer should have been initialized and opened.

Example

```
// This code example checks if the host has enabled the remote wake-up
// feature and then starts resume signaling. It is assumed
// that the device is in suspended mode.

USB_DEVICE_HANDLE usbDeviceHandle;

if(USB_DEVICE_RemoteWakeupStatusGet(usbDeviceHandle))
{
    // Start resume signaling.

    USB_DEVICE_RemoteWakeupStart(usbDeviceHandle);
}
```

Parameters

Parameters	Description
usbDeviceHandle	USB device handle returned by USB_DEVICE_Open() .

Function

```
USB_DEVICE_REMOTE_WAKEUP_STATUS USB_DEVICE_RemoteWakeupStatusGet
(
    USB_DEVICE_HANDLE usbDeviceHandle
)
```

USB_DEVICE_IsSuspended Function

Returns true if the device is in a suspended state.

File

[usb_device.h](#)

C

```
bool USB_DEVICE_IsSuspended(USB_DEVICE_HANDLE usbDeviceHandle);
```

Returns

Returns true if the device is suspended.

Description

This function returns true if the device is presently in suspended state. The application can use this function in conjunction with the [USB_DEVICE_StateGet](#) function to obtain the detailed state of the device (such as addressed and suspended, configured and suspended etc.). The Device Layer also provides the macro [USB_DEVICE_EVENT_SUSPENDED](#) event to indicate entry into suspend state.

Remarks

None.

Preconditions

The USB Device Layer must have been initialized and opened before calling this function.

Example

```
// This code example shows how the application
// can find out if the device is in a configured but suspended state.
```

```

if(USB_DEVICE_IsSuspended(usbDeviceHandle))
{
    // Device is in a suspended state.

    if(USB_DEVICE_STATE_CONFIGURED == USB_DEVICE_StateGet(usbDeviceHandle))
    {
        // This means the device is in configured and suspended state.

    }
}

```

Parameters

Parameters	Description
usbDeviceHandle	Pointer to the Device Layer Handle that is returned from USB_DEVICE_Open

Function

bool USB_DEVICE_IsSuspended([USB_DEVICE_HANDLE](#) usbDeviceHandle)

[USB_DEVICE_RemoteWakeupStart Function](#)

This function will start the resume signaling.

File

[usb_device.h](#)

C

```
void USB_DEVICE_RemoteWakeupStart(USB\_DEVICE\_HANDLE usbDeviceHandle);
```

Returns

None.

Description

This function will start the resume signaling on the bus. The client calls this function after it has detected a idle bus (through the [USB_DEVICE_EVENT_SUSPENDED](#) event). The remote wake-up feature should have been enabled by the host, before the client can call this function. The client can use the [USB_DEVICE_RemoteWakeupStatusGet](#) function to check if the host has enabled the remote wake-up feature.

Remarks

None.

Preconditions

Client handle should be valid. The remote wake-up feature should have been enabled by the host.

Example

```

// This code example shows how the device can enable and disable
// Resume signaling on the bus. These function should only be called if the
// device support remote wakeup and the host has enabled this
// feature.

USB\_DEVICE\_HANDLE usbDeviceHandle;

// Start resume signaling.
USB_DEVICE_RemoteWakeupStart(usbDeviceHandle);

// As per section 7.1.7.7 of the USB specification, device can
// drive resume signaling for at least 1 millisecond but no
// more than 15 milliseconds.

APP_DelayMilliseconds(10);

// Stop resume signaling.
USB_DEVICE_RemoteWakeupStop(usbDeviceHandle);

```

Parameters

Parameters	Description
usbDeviceHandle	Client's driver handle (returned from USB_DEVICE_Open)

Function

```
void USB_DEVICE_RemoteWakeupStart( USB\_DEVICE\_HANDLE usbDeviceHandle )
```

[USB_DEVICE_RemoteWakeupStartTimed Function](#)

This function will start a self timed Remote Wake-up.

File

[usb_device.h](#)

C

```
void USB_DEVICE_RemoteWakeupStartTimed(USB_DEVICE_HANDLE usbDeviceHandle);
```

Returns

None.

Description

This function will start a self timed Remote Wake-up sequence. The function will cause the device to generate resume signaling for 10 milliseconds. The resume signaling will stop after 10 milliseconds. The application can use this function instead of the [USB_DEVICE_RemoteWakeupStart](#) and [USB_DEVICE_RemoteWakeupStop](#) functions, which require the application to manually start, maintain duration and then stop the resume signaling.

Remarks

None.

Preconditions

Client handle should be valid. The host should have enabled the Remote Wake-up feature for this device.

Example

```
// This code example shows how the device can use the
// USB\_DEVICE\_RemoteWakeupStartTimed function to drive resume signaling
// on the bus for 10 milliseconds.

USB_DEVICE_HANDLE usbDeviceHandle;

// Check if host has enabled remote wake-up for the device.
if(USB_DEVICE_REMOTE_WAKEUP_ENABLED == USB_DEVICE_RemoteWakeupStatusGet(usbDeviceHandle))
{
    // Remote wake-up is enabled

    USB_DEVICE_RemoteWakeupStartTimed(usbDeviceHandle);
}
```

Parameters

Parameters	Description
usbDeviceHandle	Client's driver handle (returned from USB_DEVICE_Open)

Function

```
void USB_DEVICE_RemoteWakeupStartTimed( USB\_DEVICE\_HANDLE usbDeviceHandle )
```

[USB_DEVICE_RemoteWakeupStop Function](#)

This function will stop the resume signaling.

File

[usb_device.h](#)

C

```
void USB_DEVICE_RemoteWakeupStop(USB_DEVICE_HANDLE usbDeviceHandle);
```

Returns

None.

Description

This function will stop the resume signaling. This function should be called after the client has called the [USB_DEVICE_RemoteWakeupStart\(\)](#) function.

Remarks

None.

Preconditions

Client handle should be valid. The host should have enabled the Remote Wakeup feature for this device.

Example

```
// This code example shows how the device can enable and disable
// Resume signaling on the bus. These function should only be called if the
// device support remote wake-up and the host has enabled this
// feature.

USB_DEVICE_HANDLE usbDeviceHandle;

// Start resume signaling.
USB_DEVICE_RemoteWakeupStart(usbDeviceHandle);

// As per section 7.1.7.7 of the USB specification, device must
// drive resume signaling for at least 1 millisecond but no
// more than 15 milliseconds.

APP_DelayMilliseconds(10);

// Stop resume signaling.
USB_DEVICE_RemoteWakeupStop(usbDeviceHandle);
```

Parameters

Parameters	Description
usbDeviceHandle	Client's driver handle (returned from USB_DEVICE_Open)

Function

```
void USB_DEVICE_RemoteWakeupStop ( USB_DEVICE_HANDLE usbDeviceHandle )
```

d) Device Management Functions**[USB_DEVICE_StateGet Function](#)**

Returns the current state of the USB device.

File

[usb_device.h](#)

C

```
USB_DEVICE_STATE USB_DEVICE_StateGet(USB_DEVICE_HANDLE usbDeviceHandle);
```

Returns

USB_DEVICE_STATE_DETACHED - Device is not in any of the known states

USB_DEVICE_STATE_ATTACHED - Device is attached to the USB, but is not powered

USB_DEVICE_STATE_POWERED - Device is attached to the USB and powered, but has not been reset

USB_DEVICE_STATE_DEFAULT - Device is attached to the USB and powered and has been reset, but has not been assigned a unique address

USB_DEVICE_STATE_ADDRESS - Device is attached to the USB, powered, has been reset, and a unique device address has been assigned

`USB_DEVICE_STATE_CONFIGURED` - Device is attached to the USB, powered, has been reset, has a unique address, is configured, and is not suspended

Description

This function returns the current state of the USB device, as described in Chapter 9 of the USB 2.0 Specification.

Remarks

None.

Preconditions

The USB device layer must have been initialized and opened before calling this function.

Example

```
USB_DEVICE_STATE usbDevState;

// Get USB Device State.
usbDevState = USB_DEVICE_StateGet( usbDeviceHandle );

switch(usbDevState)
{
    case USB_DEVICE_STATE_ATTACHED:
        // Add code here
        break;

    case USB_DEVICE_STATE_POWERED:
        // Add code here
        break;

    default:
        break;
}
```

Parameters

Parameters	Description
usbDeviceHandle	Pointer to the device layer handle that is returned from USB_DEVICE_Open

Function

`USB_DEVICE_STATE USB_DEVICE_StateGet(USB_DEVICE_HANDLE usbDeviceHandle)`

USB_DEVICE_Attach Function

This function will attach the device to the USB.

File

[usb_device.h](#)

C

```
void USB_DEVICE_Attach(USB_DEVICE_HANDLE usbDeviceHandle);
```

Returns

None.

Description

This function will attach the device to the USB. It does this by enabling the pull up resistors on the D+ or D- lines. This function should be called after the USB device layer has generated the `USB_DEVICE_EVENT_POWER_DETECTED` event.

Remarks

None.

Preconditions

Client handle should be valid. The device layer should have been initialized and an device layer event handler function should have been assigned.

Example

```
// This code example shows the set
```

```

// of steps to follow before attaching the
// device on the bus. It is assumed that the
// device layer is already initialized.

USB_DEVICE_HANDLE usbDeviceHandle;

// Get an handle to the USB device layer.
usbDeviceHandle = USB_DEVICE_Open( USB_DEVICE_INDEX_0,
                                    DRV_IO_INTENT_READWRITE );

if(USB_DEVICE_INVALID == usbDeviceHandle)
{
    // Failed to open handle.
    // Handle error.
}

// Register an event handler call back function with device layer
// so that we are ready to receive events when the device is
// attached to the bus.

USB_DEVICE_EventHandlerSet(usbDeviceHandle, APP_USBDeviceEventHandler, NULL);

// Now, connect device to USB
USB_DEVICE_Attach(usbDeviceHandle);

```

Parameters

Parameters	Description
usbDeviceHandle	Client's USB device layer handle (returned from USB_DEVICE_Open)

Function

```
void USB_DEVICE_Attach( USB_DEVICE_HANDLE usbDeviceHandle )
```

USB_DEVICE_Detach Function

This function will detach the device from the USB.

File

[usb_device.h](#)

C

```
void USB_DEVICE_Detach(USB_DEVICE_HANDLE usbDeviceHandle);
```

Returns

None.

Description

This function will detach the device from the USB. It does this by disabling the pull up resistors on the D+ or D- lines. This function should be called when the application wants to disconnect the device from the bus (typically to implement a soft detach or switch to host mode operation). It should be called when the Device Layer has generated the USB_DEVICE_EVENT_POWER_REMOVED event.

Remarks

None.

Preconditions

The device layer should have been initialized and opened.

Example

```

USB_DEVICE_HANDLE usbDeviceHandle;

// Detach the device from the USB
USB_DEVICE_Detach( usbDeviceHandle );

```

Parameters

Parameters	Description
usbDeviceHandle	Client's driver handle (returned from USB_DEVICE_Open)

Function

```
void USB_DEVICE_Detach( USB\_DEVICE\_HANDLE usbDeviceHandle );
```

[USB_DEVICE_ActiveConfigurationGet Function](#)

Informs the client of the current USB device configuration set by the USB host.

File

[usb_device.h](#)

C

```
uint8_t USB\_DEVICE\_ActiveConfigurationGet(USB\_DEVICE\_HANDLE usbDeviceHandle);
```

Returns

Present active configuration.

Description

This function returns the current active USB device configuration.

Remarks

None.

Preconditions

The USB Device Layer must have been initialized and opened before calling this function.

Example

```
// This code example shows how the
// USB\_DEVICE\_ActiveConfigurationGet function can be called to obtain
// the configuration that has been set by the host. Note that this
// information is also available in the macro USB\_DEVICE\_EVENT\_CONFIGURED.
// -----
// This example shows how to call the USB\_DEVICE\_ActiveConfigurationGet function.
// It first initializes the USB Device Layer, then opens a device handle,
// and finally calls the function to get the active configuration.
// -----
// This example shows how to call the USB\_DEVICE\_ActiveConfigurationGet function.
// It first initializes the USB Device Layer, then opens a device handle,
// and finally calls the function to get the active configuration.
```

Parameters

Parameters	Description
usbDeviceHandle	Pointer to the Device Layer Handle that is returned from USB_DEVICE_Open

Function

```
uint8_t USB_DEVICE_ActiveConfigurationGet( USB\_DEVICE\_HANDLE usbDeviceHandle )
```

[USB_DEVICE_ActiveSpeedGet Function](#)

Informs the client of the current operation speed of the USB bus.

File

[usb_device.h](#)

C

```
USB\_SPEED USB\_DEVICE\_ActiveSpeedGet(USB\_DEVICE\_HANDLE usbDeviceHandle);
```

Returns

[USB_SPEED_LOW](#) - USB module is at low-speed
[USB_SPEED_FULL](#) - USB module is at full-speed
[USB_SPEED_HIGH](#) - USB module is at high-speed

Description

The USB device stack supports both high speed and full speed operations. This function returns the current operation speed of the USB bus. This function should be called after the USB_DEVICE_EVENT_RESET event has occurred.

Remarks

None.

Preconditions

The USB device layer must have been initialized and a valid handle to USB device layer must have been opened.

Example

```
// This code example shows how the
// USB_DEVICE_GetDeviceSpeed function can be called to obtain
// the current device speed. This information is also
// available in the USB_DEVICE_EVENT_CONFIGURED event.

if(USB_DEVICE_ActiveSpeedGet(usbDeviceHandle) == USB_SPEED_FULL)
{
    // This means the device attached at full speed.
}
else if(USB_DEVICE_ActiveSpeedGet(usbDeviceHandle) == USB_SPEED_HIGH)
{
    // This means the device attached at high speed.
}
```

Parameters

Parameters	Description
usbDeviceHandle	Pointer to device layer handle that is returned from USB_DEVICE_Open

Function

[USB_SPEED](#) `USB_DEVICE_ActiveSpeedGet(USB_DEVICE_HANDLE usbDeviceHandle)`

e) Endpoint Management Functions

`USB_DEVICE_EndpointIsStalled` Function

This function returns the stall status of the specified endpoint and direction.

File

[usb_device.h](#)

C

```
bool USB_DEVICE_EndpointIsStalled(USB_DEVICE_HANDLE usbDeviceHandle, USB_ENDPOINT_ADDRESS endpoint);
```

Returns

Returns true if endpoint is stalled, false otherwise.

Description

This function returns the stall status of the specified endpoint and direction.

Remarks

None.

Preconditions

The USB Device should be in a configured state.

Example

```
// This code example shows of how the
// USB_DEVICE_EndpointIsStalled function can be used to obtain the
// stall status of the endpoint 1 and IN direction.
```

```

USB_ENDPOINT_ADDRESS ep;

ep = 0x1 | USB_EP_DIRECTION_IN;

if(true == USB_DEVICE_EndpointIsStalled (handle, ep))
{
    // Endpoint stall is enabled. Clear the stall.

    USB_DEVICE_EndpointStallClear(handle, ep);

}

```

Parameters

Parameters	Description
usbDeviceHandle	USB device handle returned by USB_DEVICE_Open
endpoint	Specifies the endpoint and direction

Function

```

bool USB_DEVICE_EndpointIsStalled
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    USB_ENDPOINT_ADDRESS endpoint
)

```

[USB_DEVICE_EndpointStall Function](#)

This function stalls an endpoint in the specified direction.

File

[usb_device.h](#)

C

```
void USB_DEVICE_EndpointStall(USB_DEVICE_HANDLE usbDeviceHandle, USB_ENDPOINT_ADDRESS endpoint);
```

Returns

None.

Description

This function stalls an endpoint in the specified direction.

Remarks

The application may typically not find the need to stall an endpoint. Stalling an endpoint erroneously could potentially make the device non-compliant.

Preconditions

Client handle should be valid.

Example

```
// This code example shows how to stall an endpoint. In
// this case, endpoint 1 IN direction is stalled.
```

```

USB_ENDPOINT_ADDRESS ep;

ep = 0x1 | USB_EP_DIRECTION_IN;

USB_DEVICE_EndpointStall(usbDeviceHandle, ep);

```

Parameters

Parameters	Description
usbDeviceHandle	USB device handle returned by USB_DEVICE_Open
endpoint	Specifies the endpoint and direction

Function

```
void USB_DEVICE_EndpointStall
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    USB_ENDPOINT_ADDRESS endpoint
)
```

USB_DEVICE_EndpointStall Clear Function

This function clears the stall on an endpoint in the specified direction.

File

[usb_device.h](#)

C

```
void USB_DEVICE_EndpointStallClear(USB_DEVICE_HANDLE usbDeviceHandle, USB_ENDPOINT_ADDRESS endpoint);
```

Returns

None.

Description

This function clear the stall on an endpoint in the specified direction.

Remarks

None.

Preconditions

Client handle should be valid.

Example

```
// This code example shows how to clear a stall on an
// endpoint. In this case, the stall on endpoint 1 IN direction is
// cleared.

USB_ENDPOINT_ADDRESS ep;

ep = USB_ENDPOINT_AND_DIRECTION(USB_DATA_DIRECTION_DEVICE_TO_HOST, 1);

USB_DEVICE_EndpointStallClear(usbDeviceHandle, ep);
```

Parameters

Parameters	Description
usbDeviceHandle	USB device handle returned by USB_DEVICE_Open() .
endpoint	Specifies the endpoint and direction.

Function

```
void USB_DEVICE_EndpointStallClear
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    USB_ENDPOINT_ADDRESS endpoint
)
```

USB_DEVICE_EndpointDisable Function

Disables a device endpoint.

File

[usb_device.h](#)

C

```
USB_DEVICE_RESULT USB_DEVICE_EndpointDisable(USB_DEVICE_HANDLE usbDeviceHandle, USB_ENDPOINT_ADDRESS endpoint);
```

Returns

USB_DEVICE_RESULT_OK - The endpoint was enabled successfully.

USB_DEVICE_RESULT_ERROR_ENDPOINT_INVALID - The specified instance was not provisioned in the application and is invalid.

Description

This function disables a device endpoint. The application may need to disable the endpoint when it wants to change the endpoint characteristics. This could happen when the device features interfaces with multiple alternate settings. In such cases, the host may request the device to switch to specific alternate setting by sending the Set Interface request. The device application must then disable the endpoint (if it was enabled) before re-enabling it with the new settings. The application can use the [USB_DEVICE_EndpointIsEnabled](#) function to check the status of the endpoint and [USB_DEVICE_EndpointEnable](#) function to enable the endpoint.

Remarks

None.

Preconditions

The device should have been configured.

Example

```
// The following code example checks if an Set Alternate request has
// been received and changes the endpoint characteristics based on the
// alternate setting. Endpoint is 1 and direction is device to host.
// Assume that endpoint size was 32 bytes in alternate setting 0.

if(setAlternateRequest)
{
    if(alternateSetting == 1)
    {
        // Check if the endpoint is already enabled.
        if(USB_DEVICE_EndpointIsEnabled(usbDeviceHandle, (0x1|USB_EP_DIRECTION_IN)))
        {
            // Disable the endpoint.
            USB_DEVICE_EndpointDisable(usbDeviceHandle, (0x1|USB_EP_DIRECTION_IN));
        }

        // Re-enable the endpoint with new settings
        USB_DEVICE_EndpointEnable(usbDeviceHandle, (0x1|USB_EP_DIRECTION_IN)
                                  USB_TRANSFER_TYPE_BULK, 64);
    }
}
```

Parameters

Parameters	Description
usbDeviceHandle	USB Device Layer Handle.
endpoint	Endpoint to disable.

Function

```
USB_DEVICE_RESULT USB_DEVICE_EndpointDisable
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    USB_ENDPOINT_ADDRESS endpoint,
);
```

USB_DEVICE_EndpointEnable Function

Enables a device endpoint.

File

[usb_device.h](#)

C

```
USB_DEVICE_RESULT USB_DEVICE_EndpointEnable(USB_DEVICE_HANDLE usbDeviceHandle, uint8_t interface,  
USB_ENDPOINT_ADDRESS endpoint, USB_TRANSFER_TYPE transferType, size_t size);
```

Returns

USB_DEVICE_RESULT_OK - The endpoint was enabled successfully.

USB_DEVICE_RESULT_ERROR_ENDPOINT_INVALID - The specified instance was not provisioned in the application and is invalid.

Description

This function enables a device endpoint for the specified transfer type and size. A Vendor specific device application may typically call this function in response to a Set Interface request from the host. Note that Device Layer will enable endpoints contained in Alternate Setting 0 of an interface, when the host configures the device. If there is only one alternate setting in an interface, the application may not need to call the `USB_DEVICE_EndpointEnable` function.

If the device supports multiple alternate settings in an Interface, the device application must then disable an endpoint (if it was enabled) before re-enabling it with the new settings. The application can use the `USB_DEVICE_EndpointIsEnabled` function to check the status of the endpoint and `USB_DEVICE_EndpointDisable` function to disable the endpoint.

Remarks

None.

Preconditions

The device should have been configured.

Example

```
// The following code example checks if an Set Alternate request has  
// been received and changes the endpoint characteristics based on the  
// alternate setting. Endpoint is 1 and direction is device to host.  
// Assume that endpoint size was 32 bytes in alternate setting 0.  
  
if(setAlternateRequest)  
{  
    if(alternateSetting == 1)  
    {  
        // Check if the endpoint is already enabled.  
        if(USB_DEVICE_EndpointIsEnabled(usbDeviceHandle, (0x1|USB_EP_DIRECTION_IN)) )  
        {  
            // Disable the endpoint.  
            USB_DEVICE_EndpointDisable(usbDeviceHandle, (0x1|USB_EP_DIRECTION_IN));  
        }  
  
        // Re-enable the endpoint with new settings  
        USB_DEVICE_EndpointEnable(usbDeviceHandle, 0, (0x1|USB_EP_DIRECTION_IN)  
                                USB_TRANSFER_TYPE_BULK, 64);  
    }  
}
```

Parameters

Parameters	Description
usbDeviceHandle	USB Device Layer Handle.
interface	This parameter is ignored in the PIC32 USB Device Stack implementation.
endpoint	Endpoint to enable.
transferType	Type of transfer that this endpoint will support. This should match the type reported to the host
size	Maximum endpoint size. This should match the value reported to the host.

Function

```
USB_DEVICE_RESULT USB_DEVICE_EndpointEnable  
(  
    USB_DEVICE_HANDLE usbDeviceHandle,  
    uint8_t interface,
```

```
USB_ENDPOINT_ADDRESS endpoint,
USB_TRANSFER_TYPE transferType
size_t size
);
```

USB_DEVICE_EndpointIsEnabled Function

Returns true if the endpoint is enabled.

File

[usb_device.h](#)

C

```
bool USB_DEVICE_EndpointIsEnabled(USB_DEVICE_HANDLE usbDeviceHandle, USB_ENDPOINT_ADDRESS endpoint);
```

Returns

true - The endpoint is enabled.

false - The endpoint is not enabled or the specified endpoint is not provisioned in the system and is invalid.

Description

This function returns true if the endpoint is enabled. The application can use this function when handling Set Interface requests in case of Vendor or Custom USB devices.

Remarks

None.

Preconditions

The device should have been configured.

Example

```
// The following code example checks if an Set Alternate request has
// been received and changes the endpoint characteristics based on the
// alternate setting. Endpoint is 1 and direction is device to host.
// Assume that endpoint size was 32 bytes in alternate setting 0.

if(setAlternateRequest)
{
    if(alternateSetting == 1)
    {
        // Check if the endpoint is already enabled.
        if(USB_DEVICE_EndpointIsEnabled(usbDeviceHandle, (0x1|USB_EP_DIRECTION_IN)))
        {
            // Disable the endpoint.
            USB_DEVICE_EndpointDisable(usbDeviceHandle, (0x1|USB_EP_DIRECTION_IN));
        }

        // Re-enable the endpoint with new settings
        USB_DEVICE_EndpointEnable(usbDeviceHandle, (0x1|USB_EP_DIRECTION_IN)
                                  USB_TRANSFER_TYPE_BULK, 64);
    }
}
```

Parameters

Parameters	Description
usbDeviceHandle	USB Device Layer Handle.
endpoint	Endpoint to disable.

Function

```
bool USB_DEVICE_EndpointIsEnabled
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    USB_ENDPOINT_ADDRESS endpoint,
```

```
);
```

USB_DEVICE_EndpointRead Function

Reads data received from host on the requested endpoint.

File

[usb_device.h](#)

C

```
USB_DEVICE_RESULT USB_DEVICE_EndpointRead(USB_DEVICE_HANDLE usbDeviceHandle, USB_DEVICE_TRANSFER_HANDLE * transferHandle, USB_ENDPOINT_ADDRESS endpoint, void * buffer, size_t bufferSize);
```

Returns

USB_DEVICE_RESULT_OK - The read request was successful. transferHandle contains a valid transfer handle.

USB_DEVICE_RESULT_ERROR_TRANSFER_QUEUE_FULL - internal request queue is full. The write request could not be added.

USB_DEVICE_RESULT_ERROR_TRANSFER_SIZE_INVALID - The specified transfer size was not a multiple of endpoint size or is 0.

USB_DEVICE_RESULT_ERROR_ENDPOINT_NOT_CONFIGURED - The specified endpoint is not configured yet and is not ready for data transfers.

USB_DEVICE_RESULT_ERROR_ENDPOINT_INVALID - The specified instance was not provisioned in the application and is invalid.

Description

This function requests a endpoint data read from the USB Device Layer. The function places a requests with driver, the request will get serviced as data is made available by the USB Host. A handle to the request is returned in the transferHandle parameter. The termination of the request is indicated by the USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE event. The amount of data read and the transfer handle associated with the request is returned along with the event in the pData parameter of the event handler. The transfer handle expires when event handler for the USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE exits. If the read request could not be accepted, the function returns an error code and transferHandle will contain the value [USB_DEVICE_TRANSFER_HANDLE_INVALID](#).

If the size parameter is not a multiple of maxPacketSize or is 0, the function returns [USB_DEVICE_TRANSFER_HANDLE_INVALID](#) in transferHandle and returns an error code as a return value. If the size parameter is a multiple of maxPacketSize and the host sends less than maxPacketSize data in any transaction, the transfer completes and the function driver will issue a

USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE event along with the

USB_DEVICE_EVENT_DATA_ENDPOINT_READ_COMPLETE_DATA data structure. If the size parameter is a multiple of maxPacketSize and the host sends maxPacketSize amount of data, and total data received does not exceed size, then the function driver will wait for the next packet.

Remarks

While the using the device layer with PIC32MZ USB module, the receive buffer provided to the USB_DEVICE_EndpointRead should be placed in coherent memory and aligned at a 16 byte boundary. This can be done by declaring the buffer using the `__attribute__((coherent, aligned(16)))` attribute. An example is shown here

```
uint8_t data[256] __attribute__((coherent, aligned(16)));
```

Preconditions

The device should have been configured.

Example

```
// Shows an example of how to read. This assumes that
// driver was opened successfully. Note how the endpoint
// is specified. The most significant bit is cleared while
// the lower nibble specifies the endpoint (which is 1).

USB_DEVICE_TRANSFER_HANDLE transferHandle;
USB_DEVICE_RESULT readRequestResult;
USB_DEVICE_HANDLE usbDeviceHandle;
USB_ENDPOINT_ADDRESS endpoint = 0x01;

readRequestResult = USB_DEVICE_EndpointRead(usbDeviceHandle,
                                            &transferHandle, endpoint, data, 128);

if(USB_DEVICE_RESULT_OK != readRequestResult)
{
    //Do Error handling here
}

// The completion of the read request will be indicated by the
```

```
// USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE event.
```

Parameters

Parameters	Description
usbDeviceHandle	USB Device Layer Handle.
transferHandle	Pointer to a USB_DEVICE_TRANSFER_HANDLE type of variable. This variable will contain the transfer handle in case the read request was successful.
endpoint	Endpoint from which the data should be read.
data	pointer to the data buffer where read data will be stored.
size	Size of the data buffer. Refer to the description section for more details on how the size affects the transfer.

Function

```
USB_DEVICE_RESULT USB_DEVICE_EndpointRead
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    USB_DEVICE_TRANSFER_HANDLE * transferHandle,
    USB_ENDPOINT_ADDRESS endpoint,
    void * buffer,
    size_t bufferSize
);
```

USB_DEVICE_EndpointTransferCancel Function

This function cancels a transfer scheduled on an endpoint.

File

[usb_device.h](#)

C

```
USB_DEVICE_RESULT USB_DEVICE_EndpointTransferCancel(USB_DEVICE_HANDLE usbDeviceHandle, USB_ENDPOINT_ADDRESS
    endpoint, USB_DEVICE_TRANSFER_HANDLE transferHandle);
```

Returns

USB_DEVICE_RESULT_OK - The transfer will be canceled completely or partially.

USB_DEVICE_RESULT_ERROR - The transfer could not be canceled because it has either completed, the transfer handle is invalid or the last transaction is in progress.

Description

This function cancels a transfer scheduled on an endpoint using the [USB_DEVICE_EndpointRead](#) and [USB_DEVICE_EndpointWrite](#) functions. If a transfer is still in the queue and its processing has not started, the transfer is canceled completely. A transfer that is in progress may or may not get canceled depending on the transaction that is presently in progress. If the last transaction of the transfer is in progress, the transfer will not be canceled. If it is not the last transaction in progress, the in progress transfer will be allowed to complete. Pending transactions will be canceled. The first transaction of an in progress transfer cannot be canceled.

Remarks

None.

Preconditions

The USB Device should be in a configured state.

Example

```
// The following code example cancels a transfer on endpoint 1, IN direction.

USB_DEVICE_TRANSFER_HANDLE transferHandle;
USB_DEVICE_RESULT result;

result = USB_DEVICE_EndpointTransferCancel(usbDeviceHandle,
    (0x1|USB_EP_DIRECTION_IN), transferHandle);

if(USB_DEVICE_RESULT_OK == result)
{
```

```
// The transfer cancellation was either completely or
// partially successful.
}
```

Parameters

Parameters	Description
usbDeviceHandle	USB Device Layer Handle.
endpoint	Endpoint of which the transfer needs to be canceled.
handle	Transfer handle of the transfer to be canceled.

Function

```
USB_DEVICE_RESULT USB_DEVICE_EndpointTransferCancel
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    USB_ENDPOINT_ADDRESS endpoint,
    USB_DEVICE_TRANSFER_HANDLE handle
);
```

USB_DEVICE_EndpointWrite Function

This function requests a data write to a USB Device Endpoint.

File

[usb_device.h](#)

C

```
USB_DEVICE_RESULT USB_DEVICE_EndpointWrite(USB_DEVICE_HANDLE usbDeviceHandle, USB_DEVICE_TRANSFER_HANDLE * transferHandle, USB_ENDPOINT_ADDRESS endpoint, const void * data, size_t size, USB_DEVICE_TRANSFER_FLAGS flags);
```

Returns

USB_DEVICE_RESULT_OK - The write request was successful. transferHandle contains a valid transfer handle.
 USB_DEVICE_RESULT_ERROR_TRANSFER_QUEUE_FULL - Internal request queue is full. The write request could not be added.
 USB_DEVICE_RESULT_ERROR_ENDPOINT_INVALID - Endpoint is not provisioned in the system.
 USB_DEVICE_RESULT_ERROR_TRANSFER_SIZE_INVALID - The combination of the transfer size and the specified flag is invalid.
 USB_DEVICE_RESULT_ERROR_ENDPOINT_NOT_CONFIGURED - Endpoint is not enabled because device is not configured.
 USB_DEVICE_RESULT_ERROR_PARAMETER_INVALID - Device Layer handle is not valid.

Description

This function requests a data write to the USB Device Endpoint. The function places a request with Device layer, the request will get serviced as and when the data is requested by the USB Host. A handle to the request is returned in the transferHandle parameter. The termination of the request is indicated by the USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE event. The amount of data written and the transfer handle associated with the request is returned along with the event in length member of the pData parameter in the event handler. The transfer handle expires when event handler for the USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE exits. If the write request could not be accepted, the function returns an error code and transferHandle will contain the value [USB_DEVICE_TRANSFER_HANDLE_INVALID](#).

The behavior of the write request depends on the flags and size parameter. If the application intends to send more data in a request, then it should use the [USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING](#) flag. If there is no more data to be sent in the request, the application must use the [USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE](#) flag. This is explained in more detail here:

- If size is a multiple of maxPacketSize and flag is set as [USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE](#), the write function will append a Zero Length Packet (ZLP) to complete the transfer.
- If size is a multiple of maxPacketSize and flag is set as [USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING](#), the write function will not append a ZLP and hence will not complete the transfer.
- If size is greater than but not a multiple of maxPacketSize and flags is set as [USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE](#), the write function complete the transfer without appending a ZLP.

- If size is greater than but not a multiple of maxPacketSize and flags is set as `USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING`, the write function returns an error code and sets the transferHandle parameter to `USB_DEVICE_TRANSFER_HANDLE_INVALID`.
- If size is less than maxPacketSize and flag is set as `USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE`, the write function schedules one packet.
- If size is less than maxPacketSize and flag is set as `USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING`, the write function returns an error code and sets the transferHandle parameter to `USB_DEVICE_TRANSFER_HANDLE_INVALID`.

Remarks

While the using the device layer with PIC32MZ USB module, the transmit buffer provided to the `USB_DEVICE_EndpointWrite` should be placed in coherent memory and aligned at a 16 byte boundary. This can be done by declaring the buffer using the `__attribute__((coherent, aligned(16)))` attribute. An example is shown here

```
uint8_t data[256] __attribute__((coherent, aligned(16)));
```

Preconditions

The USB Device should be in a configured state.

Example

```
// Below is a set of examples showing various conditions trying to
// send data with the Write command.
//
// This assumes that Device Layer was opened successfully.
// Assume maxPacketSize is 64.

USB_DEVICE_TRANSFER_HANDLE transferHandle;
USB_DEVICE_RESULT writeRequestHandle;
USB_DEVICE_HANDLE usbDeviceHandle;

//-----
// In this example we want to send 34 bytes only.

writeRequestResult = USB_DEVICE_EndpointWrite(usbDeviceHandle,
                                              &transferHandle, data, 34,
                                              USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE);

if(USB_DEVICE_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}

//-----
// In this example we want to send 64 bytes only.
// This will cause a ZLP to be sent.

writeRequestResult = USB_DEVICE_EndpointWrite(usbDeviceHandle,
                                              &transferHandle, data, 64,
                                              USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE);

if(USB_DEVICE_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}

//-----
// This example will return an error because size is less
// than maxPacketSize and the flag indicates that more
// data is pending.

writeRequestResult = USB_DEVICE_EndpointWrite(usbDeviceHandle,
                                              &transferHandle, data, 32,
                                              USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING);

//-----
```

```

// In this example we want to place a request for a 70 byte transfer.
// The 70 bytes will be sent out in a 64 byte transaction and a 6 byte
// transaction completing the transfer.

writeRequestResult = USB_DEVICE_EndpointWrite(usbDeviceHandle,
                                              &transferHandle, data, 70,
                                              USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE);

if(USB_DEVICE_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}

-----
// This example would result in an error because the transfer size is
// not an exact multiple of the endpoint size and the
// USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING flag indicate that the
// transfer should continue.

writeRequestResult = USB_DEVICE_EndpointWrite(usbDeviceHandle,
                                              &transferHandle, data, 70,
                                              USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING);

if(USB_DEVICE_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}

// The completion of the write request will be indicated by the
// USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE event.

```

Parameters

Parameters	Description
instance	Handle to the device layer.
transferHandle	Pointer to a USB_DEVICE_TRANSFER_HANDLE type of variable. This variable will contain the transfer handle in case the write request was successful.
endpoint	Endpoint to which the data should be written. Note that is a combination of direction and the endpoint number. Refer to the description of USB_ENDPOINT_ADDRESS for more details.
data	Pointer to the data buffer that contains the data to written.
size	Size of the data buffer. Refer to the description section for more details on how the size affects the transfer.
flags	Flags that indicate whether the transfer should continue or end. Refer to the description for more details.

Function

```

USB_DEVICE_RESULT USB_DEVICE_EndpointWrite
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    USB_DEVICE_TRANSFER_HANDLE * transferHandle,
    USB_ENDPOINT_ADDRESS endpoint,
    const void * data,
    size_t size,
    USB_DEVICE_TRANSFER_FLAGS flag
);

```

f) Control Transfer Functions

[USB_DEVICE_ControlReceive Function](#)

Receives data stage of the control transfer from host to device.

File

[usb_device.h](#)

C

```
USB_DEVICE_CONTROL_TRANSFER_RESULT USB_DeviceControlReceive(USB_DEVICE_HANDLE usbDeviceHandle, void *  
data, size_t length);
```

Returns

USB_DEVICE_CONTROL_TRANSFER_RESULT_FAILED - If control transfer failed due to host aborting the previous control transfer.

USB_DEVICE_CONTROL_TRANSFER_RESULT_SUCCESS - The request was submitted successfully.

Description

This function allows the application to specify the data buffer that would be needed to receive the data stage of a control write transfer. It should be called when the application receives the USB_DEVICE_CONTROL_TRANSFER_EVENT_SETUP_REQUEST event and has identified this setup request as the setup stage of a control write transfer. The function can be called in the Application Control Transfer Event handler or can be called after the application has returned from the control transfer event handler.

Calling this function after returning from the event handler defers the response to the event. This allows the application to prepare the data buffer out of the event handler context, especially if the data buffer to receive the data is not readily available. Note however, that there are timing considerations when responding to the control transfer. Exceeding the response time will cause the host to cancel the control transfer and may cause USB host to reject the device.

Remarks

None.

Preconditions

Client handle should be valid.

Example

```
// The following code example shows an example of how the  

// USB_DeviceControlReceive function is called in response to the  

// USB_DEVICE_CONTROL_TRANSFER_EVENT_SETUP_REQUEST event to enable a control  

// write transfer.  
  

void APP_USBDeviceControlTransferEventHandler  
(  
    USB_DEVICE_EVENT event,  
    void * pData,  
    uintptr_t context  
)  
{  
    uint8_t * setupPkt;  
  
    switch(event)  
    {  
        case USB_DEVICE_CONTROL_TRANSFER_EVENT_SETUP_REQUEST:  
  
            setupPkt = (uint8_t *)pData;  
  
            // Submit a buffer to receive 32 bytes in the control write transfer.  
            USB_DeviceControlReceive(usbDeviceHandle, data, 32);  
            break;  
  
        case USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_RECEIVED:  
  
            // This means that data in the data stage of the control  
            // write transfer has been received. The application can either  
            // accept the received data by calling the  
            // USB_DeviceControlStatus function with  
            // USB_DEVICE_CONTROL_STATUS_OK flag (as shown in this example)  
            // or it can reject it by calling the USB_DeviceControlStatus()  
            // function with USB_DEVICE_CONTROL_STATUS_ERROR flag.  
  
            USB_DeviceControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);  
            break;  
    }  
}
```

```

case USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_SENT:

    // This means that data in the data stage of the control
    // read transfer has been sent. The application would typically
    // end the control transfer by calling the
    // USB_DEVICE_ControlStatus function with
    // USB_DEVICE_CONTROL_STATUS_OK flag (as shown in this example).

    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);
    break;

case USB_DEVICE_CONTROL_TRANSFER_EVENT_ABORTED:

    // This means the host has aborted the control transfer. The
    // application can reset its control transfer state machine.
    break;
}

}

```

Parameters

Parameters	Description
usbDeviceHandle	USB device handle returned by USB_DEVICE_Open
data	Pointer to buffer that holds data
length	Size in bytes

Function

```

USB_DEVICE_CONTROL_TRANSFER_RESULT USB_DEVICE_ControlReceive
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    void * data,
    size_t length
)

```

USB_DEVICE_ControlSend Function

Sends data stage of the control transfer from device to host.

File

[usb_device.h](#)

C

```

USB_DEVICE_CONTROL_TRANSFER_RESULT USB_DEVICE_ControlSend(USB_DEVICE_HANDLE usbDeviceHandle, void * data,
size_t length);

```

Returns

USB_DEVICE_CONTROL_TRANSFER_RESULT_FAILED - If control transfer failed due to host aborting the previous control transfer.

USB_DEVICE_CONTROL_TRANSFER_RESULT_SUCCESS - The request was submitted successfully.

Description

This function allows the application to specify the data that would be sent to host in the data stage of a control read transfer. It should be called when the application has received the **USB_DEVICE_CONTROL_TRANSFER_EVENT_SETUP_REQUEST** event and has identified this setup request as the setup stage of a control read transfer. The Device Layer will generate a **USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_SENT** event when the data stage has completed. The function can be called in the Application Control Transfer Event handler or can be called after the application has returned from the control transfer event handler.

Calling this function after returning from the event handler defers the response to the event. This allows the application to prepare the data buffer out of the event handler context, especially if the data buffer to receive the data is not readily available. Note however, that there are timing considerations when responding to the control transfer. Exceeding the response time will cause the host to cancel the control transfer and may cause USB host to reject the device.

Remarks

None.

Preconditions

Client handle should be valid.

Example

```
// The following code example shows an example of how the
// USB_DEVICE_ControlSend() function is called in response to the
// USB_DEVICE_CONTROL_TRANSFER_EVENT_SETUP_REQUEST event to enable a control
// read transfer.

void APP_USBDeviceEventHandler(
{
    USB_DEVICE_EVENT event,
    void * pData,
    uintptr_t context
}
{
    USB_SETUP_PACKET * setupPkt;

    switch(event)
    {
        case USB_DEVICE_CONTROL_TRANSFER_EVENT_SETUP_REQUEST:

            setupPkt = (USB_SETUP_PACKET *)pData;

            // Submit a buffer to send 32 bytes in the control read transfer.
            USB_DEVICE_ControlSend(usbDeviceHandle, data, 32);
            break;

        case USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_RECEIVED:

            // This means that data in the data stage of the control
            // write transfer has been received. The application can either
            // accept the received data by calling the
            // USB_DEVICE_ControlStatus function with
            // USB_DEVICE_CONTROL_STATUS_OK flag (as shown in this example)
            // or it can reject it by calling the USB_DEVICE_ControlStatus
            // function with USB_DEVICE_CONTROL_STATUS_ERROR flag.

            USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);
            break;

        case USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_SENT:

            // This means that data in the data stage of the control
            // read transfer has been sent. The application would typically
            // end the control transfer by calling the
            // USB_DEVICE_ControlStatus function with
            // USB_DEVICE_CONTROL_STATUS_OK flag (as shown in this example).

            USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);
            break;

        case USB_DEVICE_CONTROL_TRANSFER_EVENT_ABORTED:

            // This means the host has aborted the control transfer. The
            // application can reset its control transfer state machine.
            break;
    }
}
```

Parameters

Parameters	Description
usbDeviceHandle	USB device handle returned by USB_DEVICE_Open
data	Pointer to buffer that holds data
length	Size in bytes

Function

```
USB_DEVICE_CONTROL_TRANSFER_RESULT USB_DEVICE_ControlSend
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    void * data,
    size_t length
)
```

USB_DEVICE_ControlStatus Function

Initiates status stage of the control transfer.

File

`usb_device.h`

C

```
USB_DEVICE_CONTROL_TRANSFER_RESULT USB_DEVICE_ControlStatus(USB_DEVICE_HANDLE usbDeviceHandle,
    USB_DEVICE_CONTROL_STATUS status);
```

Returns

`USB_DEVICE_CONTROL_TRANSFER_RESULT_FAILED` - If control transfer failed due to host aborting the previous control transfer.
`USB_DEVICE_CONTROL_TRANSFER_RESULT_SUCCESS` - The request was submitted successfully.

Description

This function allows the application to complete the status stage of the of an on-going control transfer. The application must call this function when the data stage of the control transfer is complete or when a Setup Request has been received (in case of a zero data stage control transfer). The application can either accept the data stage/setup command or reject it. Calling this function with status set to `USB_DEVICE_CONTROL_STATUS_OK` will acknowledge the status stage of the control transfer. The control transfer can be stalled by setting the status parameter to `USB_DEVICE_CONTROL_STATUS_ERROR`.

The function can be called in the Application Control Transfer event handler or can be called after returning from this event handler. Calling this function after returning from the control transfer event handler defers the response to the event. This allows the application to analyze the event response outside the event handler. Note however, that there are timing considerations when responding to the control transfer. Exceeding the response time will cause the host to cancel the control transfer and may cause USB host to reject the device.

The application must be aware of events and associated control transfers that do or do not require data stages. Incorrect usage of the `USB_DEVICE_ControlStatus` function could cause the device function to be non-compliant.

Remarks

None.

Preconditions

Client handle should be valid. This function should only be called to complete an on-going control transfer.

Example

```
// The following code example shows an example of how the
// USB_DeviceControlStatus() function is called in response to the
// USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_RECEIVED and
// USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_SENT event to complete the control
// transfer. Here, the application code acknowledges the status stage of the
// control transfer.

void APP_USBDeviceControlTransferEventHandler
(
    USB_DEVICE_EVENT event,
    void * data,
    uintptr_t context
)
{
    USB_SETUP_PACKET * setupPkt;

    switch(event)
    {
        case USB_DEVICE_CONTROL_TRANSFER_EVENT_SETUP_REQUEST:
```

```

setupPkt = (USB_SETUP_PACKET *)pData;

// Submit a buffer to receive 32 bytes in the control write transfer.
USB_DEVICE_ControlReceive(usbDeviceHandle, data, 32);
break;

case USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_RECEIVED:

    // This means that data in the data stage of the control
    // write transfer has been received. The application can either
    // accept the received data by calling the
    // USB_DEVICE_ControlStatus function with
    // USB_DEVICE_CONTROL_STATUS_OK flag (as shown in this example)
    // or it can reject it by calling the USB_DEVICE_ControlStatus
    // function with USB_DEVICE_CONTROL_STATUS_ERROR flag.

    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);
    break;

case USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_SENT:

    // This means that data in the data stage of the control
    // read transfer has been sent. The application would typically
    // end the control transfer by calling the
    // USB_DEVICE_ControlStatus function with
    // USB_DEVICE_CONTROL_STATUS_OK flag (as shown in this example).

    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);
    break;

case USB_DEVICE_CONTROL_TRANSFER_EVENT_ABORTED:

    // This means the host has aborted the control transfer. The
    // application can reset its control transfer state machine.
    break;
}
}

```

Parameters

Parameters	Description
usbDeviceHandle	USB device handle returned by USB_DEVICE_Open
status	USB_DEVICE_CONTROL_STATUS_OK to acknowledge the status stage. USB_DEVICE_CONTROL_STATUS_ERROR to stall the status stage.

Function

```

USB_DEVICE_CONTROL_TRANSFER_RESULT USB_DEVICE_ControlStatus
(
    USB_DEVICE_HANDLE usbDeviceHandle,
    USB_DEVICE_CONTROL_STATUS status
)

```

g) Data Types and Constants

USB_DEVICE_INDEX_0 Macro

USB device layer index definitions.

File

[usb_device.h](#)

C

```
#define USB_DEVICE_INDEX_0 0
```

Description

USB Device Layer Index Numbers

These constants provide USB device layer index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals. These values should be passed into the [USB_DEVICE_Initialize](#) and [USB_DEVICE_Open](#) routines to identify the device layer instance in use.

USB_DEVICE_INDEX_1 Macro

File

[usb_device.h](#)

C

```
#define USB_DEVICE_INDEX_1 1
```

Description

This is macro USB_DEVICE_INDEX_1.

USB_DEVICE_INDEX_2 Macro

File

[usb_device.h](#)

C

```
#define USB_DEVICE_INDEX_2 2
```

Description

This is macro USB_DEVICE_INDEX_2.

USB_DEVICE_INDEX_3 Macro

File

[usb_device.h](#)

C

```
#define USB_DEVICE_INDEX_3 3
```

Description

This is macro USB_DEVICE_INDEX_3.

USB_DEVICE_INDEX_4 Macro

File

[usb_device.h](#)

C

```
#define USB_DEVICE_INDEX_4 4
```

Description

This is macro USB_DEVICE_INDEX_4.

USB_DEVICE_INDEX_5 Macro

File

[usb_device.h](#)

C

```
#define USB_DEVICE_INDEX_5 5
```

Description

This is macro USB_DEVICE_INDEX_5.

USB_DEVICE_CONTROL_STATUS Enumeration

USB Device Layer Control Transfer Status Stage flags.

File

[usb_device.h](#)

C

```
typedef enum {
    USB_DEVICE_CONTROL_STATUS_OK,
    USB_DEVICE_CONTROL_STATUS_ERROR
} USB_DEVICE_CONTROL_STATUS;
```

Members

Members	Description
USB_DEVICE_CONTROL_STATUS_OK	Using this flag acknowledges the Control transfer. A Zero Length Packet will be transmitted in the status stage of the control transfer.
USB_DEVICE_CONTROL_STATUS_ERROR	Using this flag stalls the control transfer. This flag should be used if the control transfer request needs to be declined.

Description

Control Transfer Status Stage Flags

This enumeration defines the flags to be used with the [USB_DEVICE_ControlStatus](#) function.

Remarks

None.

USB_DEVICE_CONTROL_TRANSFER_RESULT Enumeration

Enumerated data type identifying results of a control transfer.

File

[usb_device.h](#)

C

```
typedef enum {
    USB_DEVICE_CONTROL_TRANSFER_RESULT_FAILED,
    USB_DEVICE_CONTROL_TRANSFER_RESULT_SUCCESS
} USB_DEVICE_CONTROL_TRANSFER_RESULT;
```

Members

Members	Description
USB_DEVICE_CONTROL_TRANSFER_RESULT_FAILED	Control transfer failed. This could be because the control transfer handle is no more valid since the control transfer was aborted by host by sending a new setup packet
USB_DEVICE_CONTROL_TRANSFER_RESULT_SUCCESS	Control transfer was successful

Description

USB Device Layer Control Transfer Result Enumeration

These enumerated values are the possible return values for control transfer operations. These values are returned by the [USB_DEVICE_ControlStatus](#), [USB_DEVICE_ControlSend](#) and the [USB_DEVICE_ControlReceive](#) functions.

Remarks

None.

USB_DEVICE_EVENT Enumeration

USB Device Layer Events.

File

[usb_device.h](#)

C

```
typedef enum {
    USB_DEVICE_EVENT_RESET,
    USB_DEVICE_EVENT_SUSPENDED,
    USB_DEVICE_EVENT_RESUMED,
    USB_DEVICE_EVENT_ERROR,
    USB_DEVICE_EVENT_SOF,
    USB_DEVICE_EVENT_CONFIGURED,
    USB_DEVICE_EVENT_DECONFIGURED,
    USB_DEVICE_EVENT_CONTROL_TRANSFER_ABORTED,
    USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_RECEIVED,
    USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST,
    USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT,
    USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE,
    USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE,
    USB_DEVICE_EVENT_SET_DESCRIPTOR,
    USB_DEVICE_EVENT_SYNCH_FRAME
} USB_DEVICE_EVENT;
```

Members

Members	Description
USB_DEVICE_EVENT_RESET	USB bus reset occurred. This event is an indication to the application client that device layer has deinitialized all function drivers. The application should not use the function drivers in this state. The pData parameter in the event handler function will be NULL.
USB_DEVICE_EVENT_SUSPENDED	This event is an indication to the application client that device is suspended and it can put the device to sleep mode if required. Power saving routines should not be called in the event handler. The pData parameter in the event handler function will be NULL.
USB_DEVICE_EVENT_RESUMED	This event indicates that device has resumed from suspended state. The pData parameter in the event handler function will be NULL.
USB_DEVICE_EVENT_ERROR	This event is an indication to the application client that an error occurred on the USB bus. The pData parameter in the event handler function will be NULL.
USB_DEVICE_EVENT_SOF	This event is generated at every start of frame detected on the bus. Application client can use this SOF event for general time based house keeping activities. The pData parameter in the event handler function will point to a USB_DEVICE_EVENT_DATA_SOF type that contains the frame number.
USB_DEVICE_EVENT_CONFIGURED	This event is an indication to the application client that device layer has initialized all function drivers and application can set the event handlers for the function drivers. The pData parameter will point to a USB_DEVICE_EVENT_DATA_CONFIGURED data type that contains configuration set by the host.
USB_DEVICE_EVENT_DECONFIGURED	The host has deconfigured the device. This happens when the host sends a Set Configuration request with configuration number 0. The device layer will deinitialize all function drivers and then generate this event.
USB_DEVICE_EVENT_CONTROL_TRANSFER_ABORTED	An on-going control transfer was aborted. The application can use this event to reset its control transfer state machine. The pData parameter in the event handler function will be NULL.
USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_RECEIVED	The data stage of a Control write transfer has completed. This event occurs after the application has used the USB_DEVICE_ControlReceive function to receive data in the control transfer. The pData parameter in the event handler function will be NULL.
USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST	A setup packet of a control transfer has been received. The recipient field of the received setup packet is Other. The application can initiate the data stage using the USB_DEVICE_ControlReceive and USB_DEVICE_ControlSend function. It can end the control transfer by calling the USB_DEVICE_ControlStatus function. The pData parameter in the event handler will point to USB_SETUP_PACKET data type.

USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT	The data stage of a Control read transfer has completed. This event occurs after the application has used the USB_DEVICE_ControlSend function to send data in the control transfer. The pData parameter in the event handler function will be NULL.
USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE	This event occurs when a endpoint read transfer scheduled using the USB_DEVICE_EndpointRead function has completed. The pData parameter in the event handler function will be a pointer to a USB_DEVICE_EVENT_DATA_ENDPOINT_READ_COMPLETE data type.
USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE	This event occurs when a endpoint write transfer scheduled using the USB_DEVICE_EndpointWrite function has completed. The pData parameter in the event handler function will be a pointer to a USB_DEVICE_EVENT_DATA_ENDPOINT_WRITE_COMPLETE data type.
USB_DEVICE_EVENT_SET_DESCRIPTOR	A SET_DESCRIPTOR request is received. This event occurs when Host sends a SET_DESCRIPTOR request. The pData parameter in the event handler function will be a pointer to a USB_DEVICE_EVENT_DATA_SET_DESCRIPTOR data type. The application should initiate the data stage using the USB_DEVICE_ControlReceive function. In the PIC32 USB Device Stack, this event is generated if USB_DEVICE_EVENT_ENABLE_SET_DESCRIPTOR is defined in the system configuration.
USB_DEVICE_EVENT_SYNCH_FRAME	A SYNCH_FRAME request is received. This event occurs when Host sends a SYNCH_FRAME request. The pData parameter in the event handler function will be a pointer to a USB_DEVICE_EVENT_DATA_SYNCH_FRAME data type. The application should initiate the data stage using the USB_DEVICE_ControlSend function. In the PIC32 USB Device Stack, this event is generated if USB_DEVICE_EVENT_ENABLE_SYNCH_FRAME is defined in the system configuration.

Description

USB Device Layer Events.

This enumeration lists the possible events that the device layer can generate. The client should register an event handler of the type [USB_DEVICE_EVENT_HANDLER](#) to receive device layer events. The contents of pData in the event handler depends on the generated event. Refer to the description of the event for details on data provided along with that event. The events generated are device layer instance specific.

The client will receive control transfers for handling from the device layer, where the recipient field of the Control Transfer Setup packet is set to Other. The client can use the control transfer events and the Device Layer control transfer functions to complete such control transfers.

It is not mandatory for the client application to handle the control transfer event within the event handler. Indeed, it may be possible that the data stage of the control transfer requires extended processing. Because the event handler executes in an interrupt context, it is recommended to keep the processing in the event handler to a minimum. The client application can call the [USB_DEVICE_ControlSend](#), [USB_DEVICE_ControlReceive](#) and [USB_DEVICE_ControlStatus](#) functions after returning from the event handler, thus deferring the control transfer event handling and responses.

Note that a USB host will typically wait for control transfer response for a finite time duration before timing out and canceling the transfer and associated transactions. Even when deferring response, the application must respond promptly if such timeouts have to be avoided.

The client must use the [USB_DEVICE_EventHandlerSet](#) function to register the event handler call back function. The following code example shows the handling of the USB Device Layer Events.

```
USB_DEVICE_EVENT_RESPONSE APP_USBDeviceEventHandler
(
    USB_DEVICE_EVENT event,
    void * pData,
    uintptr_t context
)
{
    uint8_t      activeConfiguration;
    uint16_t     frameNumber;
    USB_SPEED    attachSpeed;
    USB_SETUP_PACKET * setupEventData;

    // Handling of each event
    switch(event)
    {
        case USB_DEVICE_EVENT_POWER_DETECTED:

            // This means the device detected a valid VBUS voltage and is
            // attached to the USB. The application can now call
            // USB_DEVICE_Attach() function to enable D+/D- pull up
            // resistors.
            break;
    }
}
```

```
case USB_DEVICE_EVENT_POWER_REMOVED:  
  
    // This means the device is not attached to the USB.  
    // The application should now call the USB_DEVICE_Detach()  
    // function.  
    break;  
  
case USB_DEVICE_EVENT_SUSPENDED:  
  
    // The bus is idle. There was no activity detected.  
    // The application can switch to a low power mode after  
    // exiting the event handler.  
    break;  
  
case USB_DEVICE_EVENT_SOF:  
  
    // A start of frame was received. This is a periodic event and  
    // can be used by the application for timing related activities.  
    // pData will point to a USB_DEVICE_EVENT_DATA_SOF type data  
    // containing the frame number. In PIC32 USB Device Stack, this  
    // event is generated if USB_DEVICE_SOF_EVENT_ENABLE is  
    // defined in System Configuration.  
  
    frameNumber = ((USB_DEVICE_EVENT_DATA_SOF *)pData)->frameNumber;  
    break;  
  
case USB_DEVICE_EVENT_RESET :  
  
    // Reset signaling was detected on the bus. The  
    // application can find out the attach speed.  
  
    attachedSpeed = USB_DEVICE_ActiveSpeedGet(usbDeviceHandle);  
    break;  
  
case USB_DEVICE_EVENT_DECONFIGURED :  
  
    // This indicates that host has deconfigured the device i.e., it  
    // has set the configuration as 0. All function driver instances  
    // would have been deinitalized.  
  
    break;  
  
case USB_DEVICE_EVENT_ERROR :  
  
    // This means an unknown error has occurred on the bus.  
    // The application can try detaching and attaching the  
    // device again.  
    break;  
  
case USB_DEVICE_EVENT_CONFIGURED :  
  
    // This means that device is configured and the application can  
    // start using the device functionality. The application must  
    // register function driver event handlersI have one device  
    // level event. The pData parameter will be a pointer to a  
    // USB_DEVICE_EVENT_DATA_CONFIGURED data type that contains the  
    // active configuration number.  
  
    activeConfiguration = ((USB_DEVICE_EVENT_DATA_CONFIGURED *)pData)->configurationValue;  
    break;  
  
case USB_DEVICE_EVENT_RESUMED:  
  
    // This means that the resume signaling was detected on the  
    // bus. The application can bring the device out of power  
    // saving mode.  
  
    break;
```

```
case USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST:

    // This means that the setup stage of the control transfer is in
    // progress and a setup packet has been received. The pData
    // parameter will point to a USB_SETUP_PACKET data type. The
    // application can process the command and update its control
    // transfer state machine. The application for example could call
    // the USB_DEVICE_ControlReceive function (as shown here) to
    // submit the buffer that would receive data in case of a
    // control read transfer.

    setupPacket = (USB_SETUP_PACKET *)pData;

    // Submit a buffer to receive 32 bytes in the control write transfer.
    USB_DEVICE_ControlReceive(usbDeviceHandle, data, 32);
break;

case USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_RECEIVED:

    // This means that data in the data stage of the control write
    // transfer has been received. The application can either accept
    // the received data by calling the USB_DEVICE_ControlStatus
    // function with USB_DEVICE_CONTROL_STATUS_OK flag (as shown in
    // this example) or it can reject it by calling the
    // USB_DEVICE_ControlStatus function with
    // USB_DEVICE_CONTROL_STATUS_ERROR flag.

    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);
break;

case USB_DEVICE_CONTROL_TRANSFER_EVENT_DATA_SENT:

    // This means that data in the data stage of the control
    // read transfer has been sent.

break;

case USB_DEVICE_CONTROL_TRANSFER_EVENT_ABORTED:

    // This means the host has aborted the control transfer. The
    // application can reset its control transfer state machine.

break;

case USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE:

    // This means schedule endpoint read operation has completed.
    // The application should interpret pData as a pointer to
    // a USB_DEVICE_EVENT_DATA_ENDPOINT_READ_COMPLETE type.

break;

case USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE:

    // This means schedule endpoint write operation has completed.
    // The application should interpret pData as a pointer to
    // a USB_DEVICE_EVENT_DATA_ENDPOINT_WRITE_COMPLETE type.

break;

case USB_DEVICE_EVENT_SET_DESCRIPTOR:

    // This means the Host has sent a Set Descriptor request. The
    // application should interpret pData as a
    // USB_DEVICE_EVENT_DATA_SET_DESCRIPTOR pointer type containing the
    // details of the Set Descriptor request. In PIC32 USB Device
    // Stack, this event is generated if
    // USB_DEVICE_SET_DESCRIPTOR_EVENT_ENABLE is defined in the
```

```

// system configuration. The application can use
// USB_DEVICE_ControlSend, USB_DEVICE_ControlReceive and/or
// the USB_DEVICE_ControlStatus functions to complete the
// control transfer.

break;

case USB_DEVICE_EVENT_SYNCH_FRAME:

    // This means the host has sent a Sync Frame Request. The
    // application should interpret pData as a
    // USB_DEVICE_EVENT_DATA_SYNCH_FRAME pointer type. In PIC32 USB Device
    // Stack, this event is generated if
    // USB_DEVICE_SYNCH_FRAME_EVENT_ENABLE is defined in the
    // system configuration. The application should respond by
    // sending the 2 byte frame number using the
    // USB_DEVICE_ControlSend function.

    USB_DEVICE_ControlSend(usbDeviceHandle, &frameNumber, 2);
    break;

default:
    break;
}

return USB_DEVICE_EVENT_RESPONSE_NONE;
}

```

Remarks

Generation of some events required the definition of configuration macros. Refer to the event specific description for more details.

USB_DEVICE_HANDLE Type

Data type for USB device handle.

File

[usb_device.h](#)

C

```
typedef uintptr_t USB_DEVICE_HANDLE;
```

Description

Data type for USB device handle.

The data type of the handle that is returned from [USB_DEVICE_Open](#) function.

Remarks

None.

USB_DEVICE_INIT Structure

USB Device Initialization Structure

File

[usb_device.h](#)

C

```
typedef struct {
    SYS_MODULE_INIT moduleInit;
    unsigned int usbID;
    bool stopInIdle;
    bool suspendInSleep;
    INT_SOURCE interruptSource;
    INT_SOURCE interruptSourceUSBDma;
    void * endpointTable;
    uint16_t registeredFuncCount;
    USB_DEVICE_FUNCTION_REGISTRATION_TABLE * registeredFunctions;
}
```

```

USB_DEVICE_MASTER_DESCRIPTOR * usbMasterDescriptor;
USB_SPEED deviceSpeed;
uint16_t queueSizeEndpointRead;
uint16_t queueSizeEndpointWrite;
SYS_MODULE_INDEX driverIndex;
void * usbDriverInterface;
} USB_DEVICE_INIT;

```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
unsigned int usbID;	Identifies peripheral (PLIB-level) ID. The use of this parameter is <ul style="list-style-type: none"> • deprecated.
bool stopInIdle;	If true, USB module will stop when CPU enters Idle Mode. The use of this <ul style="list-style-type: none"> • parameter is deprecated.
bool suspendInSleep;	If true, USB module will suspend when the microcontroller enters sleep <ul style="list-style-type: none"> • mode. The use of this parameter is deprecated.
INT_SOURCE interruptSource;	Interrupt Source for USB module. The use of this parameter is deprecated.
INT_SOURCE interruptSourceUSBDma;	Interrupt Source for USB DMA module. The use of this parameter is <ul style="list-style-type: none"> • deprecated.
void * endpointTable;	Pointer to an byte array whose size is USB_DEVICE_ENDPOINT_TABLE_SIZE and who start address is aligned at a 512 bytes address boundary. The <ul style="list-style-type: none"> • use of this parameter is deprecated.
uint16_t registeredFuncCount;	Number of function drivers registered to this instance of the USB device layer
USB_DEVICE_FUNCTION_REGISTRATION_TABLE * registeredFunctions;	Function driver table registered to this instance of the USB device layer
USB_DEVICE_MASTER_DESCRIPTOR * usbMasterDescriptor;	Pointer to USB Descriptor structure
USB_SPEED deviceSpeed;	Specify the speed at which this device will attempt to connect to the host. PIC32MX devices support Full Speed only. PIC32MZ devices support Full Speed and High Speed. Selecting High Speed will allow the device to work at both Full Speed and High Speed.
uint16_t queueSizeEndpointRead;	Enter Endpoint Read queue size. Application can place this many Endpoint Read requests in the queue. Each Endpoint Read queue element would consume 36 Bytes of RAM. Value of this field should be at least 1. This is applicable only for applications using Endpoint Read/Write functions like USB Vendor Device.
uint16_t queueSizeEndpointWrite;	Enter Endpoint Write queue size. Application can place this many Endpoint Read requests in the queue. Each Endpoint Write queue element would consume 36 Bytes of RAM. Value of this field should be at least 1. This is applicable only for applications using Endpoint Read/Write functions like USB Vendor Device.
SYS_MODULE_INDEX driverIndex;	System Module Index of the driver that this device layer should open
void * usbDriverInterface;	Interface to the USB Driver that this Device Layer should use

Description

USB Device Initialization Structure

This data type defines the USB Device Initialization data structure. A data structure of this type should be initialized and provided to [USB_DEVICE_Initialize](#).

Remarks

This type is specific to the PIC32 implementation of the USB Device Stack API.

USB_DEVICE_POWER_STATE Enumeration

Enumerated data type that identifies if the device is self powered or bus powered .

File

[usb_device.h](#)

C

```

typedef enum {
    USB_DEVICE_POWER_STATE_BUS_POWERED,
    USB_DEVICE_POWER_STATE_SELF_POWERED
} USB_DEVICE_POWER_STATE;

```

Members

Members	Description
USB_DEVICE_POWER_STATE_BUS_POWERED	Device is bus powered
USB_DEVICE_POWER_STATE_SELF_POWERED	Device is self powered

Description

Device Power state

This enumeration defines the possible power states of the device. The application specifies this state to the device layer (through the [USB_DEVICE_PowerStateSet](#) function) to let the device layer know if this USB Device is presently bus or self powered.

Remarks

None.

USB_DEVICE_REMOTE_WAKEUP_STATUS Enumeration

Enumerated data type that identifies if the remote wakeup status of the device.

File

[usb_device.h](#)

C

```
typedef enum {
    USB_DEVICE_REMOTE_WAKEUP_DISABLED,
    USB_DEVICE_REMOTE_WAKEUP_ENABLED
} USB_DEVICE_REMOTE_WAKEUP_STATUS;
```

Members

Members	Description
USB_DEVICE_REMOTE_WAKEUP_DISABLED	Remote wakeup is disabled
USB_DEVICE_REMOTE_WAKEUP_ENABLED	Remote wakeup is enabled

Description

Remote Wakeup Status

This enumeration defines the possible status of the remote wake up capability. These values are returned by the [USB_DEVICE_RemoteWakeUpStatusGet](#) function.

Remarks

None.

USB_DEVICE_EVENT_DATA_CONFIGURED Structure

USB Device Set Configuration Event Data type.

File

[usb_device.h](#)

C

```
typedef struct {
    uint8_t configurationValue;
} USB_DEVICE_EVENT_DATA_CONFIGURED;
```

Members

Members	Description
uint8_t configurationValue;	The configuration that was set

Description

USB Device Set Configuration Event Data type.

This data type defines the type of data that is returned by the Device Layer along with the [USB_DEVICE_EVENT_CONFIGURED](#) event.

Remarks

None.

USB_DEVICE_HANDLE_INVALID Macro

Constant that defines the value of an Invalid Device Handle.

File

[usb_device.h](#)

C

```
#define USB_DEVICE_HANDLE_INVALID
```

Description

USB Device Layer Invalid Handle

This constant is returned by the [USB_DEVICE_Open\(\)](#) function when the function fails.

Remarks

None.

USB_DEVICE_EVENT_RESPONSE_NONE Macro

Device Layer Event Handler Function Response Type.

File

[usb_device.h](#)

C

```
#define USB_DEVICE_EVENT_RESPONSE_NONE
```

Description

Device Layer Event Handler Function Response Type None.

This is the definition of the Device Layer Event Handler Response Type None.

Remarks

Intentionally defined to be empty.

USB_DEVICE_CLIENT_STATUS Enumeration

Enumerated data type that identifies the USB Device Layer Client Status.

File

[usb_device.h](#)

C

```
typedef enum {
    USB_DEVICE_CLIENT_STATUS_CLOSED,
    USB_DEVICE_CLIENT_STATUS_READY
} USB_DEVICE_CLIENT_STATUS;
```

Members

Members	Description
USB_DEVICE_CLIENT_STATUS_CLOSED	Client is closed or the specified handle is invalid
USB_DEVICE_CLIENT_STATUS_READY	Client is ready

Description

USB Device Layer Client Status

This enumeration defines the possible status of the USB Device Layer Client. It is returned by the [USB_DEVICE_ClientStatusGet](#) function.

Remarks

None.

USB_DEVICE_CONFIGURATION_DESCRIPTOR_TABLE Type

Pointer to an array that contains pointer to configuration descriptors.

File

[usb_device.h](#)

C

```
typedef const uint8_t * const USB_DEVICE_CONFIGURATION_DESCRIPTOR_TABLE;
```

Description

Configuration descriptors pointer

This type defines a pointer to an array that contains pointers to configuration descriptors. This data type is used in [USB_DEVICE_MASTER_DESCRIPTOR](#) data type to point to the table of configuration descriptors.

Remarks

This type is specific to the PIC32 implementation of the USB Device Stack API.

USB_DEVICE_EVENT_HANDLER Type

USB Device Layer Event Handler Function Pointer Type

File

[usb_device.h](#)

C

```
typedef USB_DEVICE_EVENT_RESPONSE (* USB_DEVICE_EVENT_HANDLER)(USB_DEVICE_EVENT event, void * eventData, uintptr_t context);
```

Description

USB Device Layer Event Handler Function Pointer Type

This data type defines the required function signature of the USB Device Layer Event handling callback function. The application must register a pointer to a Device Layer Event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event call backs from the Device Layer. The Device Layer will invoke this function with event relevant parameters. The description of the event handler function parameters is given here.

event - Type of event generated.

pData - This parameter should be type cast to an event specific pointer type based on the event that has occurred. Refer to the [USB_DEVICE_EVENT](#) enumeration description for more details.

context - Value identifying the context of the application that was registered along with the event handling function.

Remarks

None.

USB_DEVICE_EVENT_RESPONSE Type

Device Layer Event Handler function return type.

File

[usb_device.h](#)

C

```
typedef void USB_DEVICE_EVENT_RESPONSE;
```

Description

Device Layer Event Handler function return type.

This data type defines the return type of the Device Layer event handler function.

Remarks

None.

USB_DEVICE_FUNCTION_REGISTRATION_TABLE Structure

USB Device Function Registration Structure

File[usb_device.h](#)**C**

```
typedef struct {
    USB_SPEED speed;
    uint8_t configurationValue;
    uint8_t interfaceNumber;
    uint8_t numberOfInterfaces;
    uintptr_t funcDriverIndex;
    void * funcDriverInit;
    void * driver;
} USB_DEVICE_FUNCTION_REGISTRATION_TABLE;
```

Members

Members	Description
USB_SPEED speed;	Type of speed (high, full or low speed)
uint8_t configurationValue;	Configuration Value to which the function driver has to be tied
uint8_t interfaceNumber;	Interface number to which this function driver has to be tied
uint8_t numberOfInterfaces;	Number of interfaces used by the function
uintptr_t funcDriverIndex;	Function driver instance index
void * funcDriverInit;	Pointer to a structure that contains function driver initialization data
void * driver;	Pointer to a standard structure that exposes function driver APIs to USB device layer

Description

USB Device Function Registration Structure

This data type defines the USB Device Function Registration Structure. A table containing entries for each function driver instance should be registered with device layer.

Remarks

This type is specific to the PIC32 implementation of the USB Device Stack API.

USB_DEVICE_MASTER_DESCRIPTOR Structure

USB Device Master Descriptor Structure.

File[usb_device.h](#)**C**

```
typedef struct {
    const USB_DEVICE_DESCRIPTOR * deviceDescriptor;
    uint8_t configDescriptorCount;
    USB_DEVICE_CONFIGURATION_DESCRIPTORS_TABLE * configDescriptorTable;
    const USB_DEVICE_DESCRIPTOR * highSpeedDeviceDescriptor;
    uint8_t highSpeedConfigDescriptorCount;
    USB_DEVICE_CONFIGURATION_DESCRIPTORS_TABLE * highSpeedConfigDescriptorTable;
    uint8_t stringDescCount;
    USB_DEVICE_STRING_DESCRIPTORS_TABLE * stringDescriptorTable;
    const USB_DEVICE_QUALIFIER * fullSpeedDeviceQualifier;
    const USB_DEVICE_QUALIFIER * highSpeedDeviceQualifier;
    const uint8_t * bosDescriptor;
} USB_DEVICE_MASTER_DESCRIPTOR;
```

Members

Members	Description
const USB_DEVICE_DESCRIPTOR * deviceDescriptor;	Pointer to standard device descriptor (for low/full speed)
uint8_t configDescriptorCount;	Total number configurations available (for low/full speed)

<code>USB_DEVICE_CONFIGURATION_DESCRIPTOR_TABLE</code>	Pointer to array of configurations descriptor pointers (for low/full speed)
<code>* configDescriptorTable;</code>	
<code>const USB_DEVICE_DESCRIPTOR * highSpeedDeviceDescriptor;</code>	Pointer to array of high speed standard Device descriptor. Assign this to NULL if not supported.
<code>uint8_t highSpeedConfigDescriptorCount;</code>	Total number of high speed configurations available. Set this to zero if not supported
<code>USB_DEVICE_CONFIGURATION_DESCRIPTOR_TABLE</code>	Pointer to array of high speed configurations descriptor pointers. Set this to NULL if not supported
<code>* highSpeedConfigDescriptorTable;</code>	
<code>uint8_t stringDescCount;</code>	Total number of string descriptors available (common to all speeds)
<code>USB_DEVICE_STRING_DESCRIPTOR_TABLE * stringDescriptorTable;</code>	Pointer to array of string Descriptor pointers (common to all speeds)
<code>const USB_DEVICE_QUALIFIER * fullSpeedDeviceQualifier;</code>	Pointer to full speed device_qualifier descriptor. Device responds with this descriptor when it is operating at high speed
<code>const USB_DEVICE_QUALIFIER * highSpeedDeviceQualifier;</code>	Pointer to high speed device_qualifier descriptor. Device responds with this descriptor when it is operating at full speed
<code>const uint8_t * bosDescriptor;</code>	Pointer to BOS descriptor for this Device. Device responds with this descriptor when Host sends a GET_DESCRIPTOR request for BOS descriptor

Description

USB Device Master Descriptor Structure.

This data type defines the structure of the USB Device Master Descriptor. The application must provide such a structure for each instance of the device layer.

Remarks

This type is specific to the PIC32 implementation of the USB Device Stack API.

`USB_DEVICE_STRING_DESCRIPTOR_TABLE` Type

Pointer to an array that contains pointer to string descriptors.

File

[usb_device.h](#)

C

```
typedef const uint8_t * const USB_DEVICE_STRING_DESCRIPTOR_TABLE;
```

Description

String Descriptors Pointer

This type defines a pointer to an array that contains pointers to string descriptors. This data type is used in `USB_DEVICE_MASTER_DESCRIPTOR` data type to point to the table of string descriptors.

Remarks

This type is specific to the PIC32 implementation of the USB Device Stack API.

`USB_DEVICE_EVENT_DATA_ENDPOINT_READ_COMPLETE` Structure

USB Device Layer Endpoint Read and Write Complete Event Data type.

File

[usb_device.h](#)

C

```
typedef struct {
    USB_DEVICE_TRANSFER_HANDLE transferHandle;
    size_t length;
    USB_DEVICE_RESULT status;
} USB_DEVICE_EVENT_DATA_ENDPOINT_READ_COMPLETE, USB_DEVICE_EVENT_DATA_ENDPOINT_WRITE_COMPLETE;
```

Members

Members	Description
<code>USB_DEVICE_TRANSFER_HANDLE transferHandle;</code>	Transfer Handle

size_t length;	Size of transferred data
USB_DEVICE_RESULT status;	Completion status of the transfer

Description

USB Device Layer Endpoint Read and Write Complete Event Data type.

This data type defines the type of data that is returned by the Device Layer along with the USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE and USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE events.

Remarks

None.

USB_DEVICE_EVENT_DATA_ENDPOINT_WRITE_COMPLETE Structure

USB Device Layer Endpoint Read and Write Complete Event Data type.

File

[usb_device.h](#)

C

```
typedef struct {
    USB_DEVICE_TRANSFER_HANDLE transferHandle;
    size_t length;
    USB_DEVICE_RESULT status;
} USB_DEVICE_EVENT_DATA_ENDPOINT_WRITE_COMPLETE;
```

Members

Members	Description
USB_DEVICE_TRANSFER_HANDLE transferHandle;	Transfer Handle
size_t length;	Size of transferred data
USB_DEVICE_RESULT status;	Completion status of the transfer

Description

USB Device Layer Endpoint Read and Write Complete Event Data type.

This data type defines the type of data that is returned by the Device Layer along with the USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE and USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE events.

Remarks

None.

USB_DEVICE_EVENT_DATA_SET_DESCRIPTOR Structure

USB Device Set Descriptor Event Data type.

File

[usb_device.h](#)

C

```
typedef struct {
    uint8_t descriptorIndex;
    uint8_t descriptorType;
    uint16_t languageID;
    uint16_t descriptorLength;
} USB_DEVICE_EVENT_DATA_SET_DESCRIPTOR;
```

Description

USB Device Set Descriptor Event Data type.

This data type defines the type of data that is returned by the Device Layer along with the USB_DEVICE_EVENT_SET_DESCRIPTOR event.

Remarks

None.

USB_DEVICE_EVENT_DATA_SOF Structure

USB Device Start Of Frame Event Data Type

File

[usb_device.h](#)

C

```
typedef struct {
    uint16_t frameNumber;
} USB_DEVICE_EVENT_DATA_SOF;
```

Members

Members	Description
uint16_t frameNumber;	The Start Of Frame number

Description

USB Device Start Of Frame Event Data Type

This data type defines the type of data that is returned by the Device Layer along with the USB_DEVICE_EVENT_SOF event.

Remarks

None.

USB_DEVICE_EVENT_DATA_SYNCH_FRAME Structure

USB Device Synch Frame Event Data type.

File

[usb_device.h](#)

C

```
typedef struct {
    USB_ENDPOINT_ADDRESS endpoint;
} USB_DEVICE_EVENT_DATA_SYNCH_FRAME;
```

Members

Members	Description
USB_ENDPOINT_ADDRESS endpoint;	Endpoint for which the Synch Frame number is requested

Description

USB Device Synch Frame Event Data type.

This data type defines the type of data that is returned by the Device Layer along with the USB_DEVICE_EVENT_SYNCH_FRAME event.

Remarks

None.

USB_DEVICE_RESULT Enumeration

USB Device Layer Results Enumeration

File

[usb_device.h](#)

C

```
typedef enum {
    USB_DEVICE_RESULT_ERROR_TRANSFER_QUEUE_FULL,
    USB_DEVICE_RESULT_OK,
    USB_DEVICE_RESULT_ERROR_ENDPOINT_NOT_CONFIGURED,
    USB_DEVICE_RESULT_ERROR_ENDPOINT_INVALID,
    USB_DEVICE_RESULT_ERROR_PARAMETER_INVALID,
    USB_DEVICE_RESULT_ERROR_DEVICE_HANDLE_INVALID,
    USB_DEVICE_RESULT_ERROR_ENDPOINT_HALTED,
```

```

USB_DEVICE_RESULT_ERROR_TERMINATED_BY_HOST,
USB_DEVICE_RESULT_ERROR
} USB_DEVICE_RESULT;

```

Members

Members	Description
USB_DEVICE_RESULT_ERROR_TRANSFER_QUEUE_FULL	Queue is full
USB_DEVICE_RESULT_OK	No Error
USB_DEVICE_RESULT_ERROR_ENDPOINT_NOT_CONFIGURED	Endpoint not configured
USB_DEVICE_RESULT_ERROR_ENDPOINT_INVALID	Endpoint not provisioned in the system
USB_DEVICE_RESULT_ERROR_PARAMETER_INVALID	One or more parameter/s of the function is invalid
USB_DEVICE_RESULT_ERROR_DEVICE_HANDLE_INVALID	Device Handle passed to the function is invalid
USB_DEVICE_RESULT_ERROR_ENDPOINT_HALTED	Transfer terminated because host halted the endpoint
USB_DEVICE_RESULT_ERROR_TERMINATED_BY_HOST	Transfer terminated by host because of a stall clear
USB_DEVICE_RESULT_ERROR	An unspecified error has occurred

Description

USB Device Result Enumeration

This enumeration lists the possible USB Device Endpoint operation results. These values are returned by USB Device Endpoint functions.

Remarks

None.

USB_DEVICE_TRANSFER_FLAGS Enumeration

Enumerated data type that identifies the USB Device Layer Transfer Flags.

File

[usb_device.h](#)

C

```

typedef enum {
    USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE,
    USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING
} USB_DEVICE_TRANSFER_FLAGS;

```

Members

Members	Description
USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE	This flag indicates there is no further data to be sent in this transfer and that the transfer should end. If the size of the transfer is a multiple of the maximum packet size for related endpoint configuration, the device layer will send a zero length packet to indicate end of the transfer to the host.
USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING	This flag indicates there is more data to be sent in this transfer. If the size of the transfer is a multiple of the maximum packet size for the related endpoint configuration, the device layer will not send a zero length packet. This flags should not be specified if the size of the transfer is is not a multiple of the maximum packet size or if the transfer is less than maximum packet size.

Description

USB Device Layer Transfer Flags

This enumeration defines the possible USB Device Layer Transfer Flags. These flags are specified in [USB_DEVICE_EndpointWrite\(\)](#) function to specify the handling of the transfer. Please refer to the description of the [USB_DEVICE_EndpointWrite](#) function for examples.

Remarks

None.

USB_DEVICE_TRANSFER_HANDLE Type

Data type for USB Device Endpoint Data Transfer Handle.

File

[usb_device.h](#)

C

```
typedef uintptr_t USB_DEVICE_TRANSFER_HANDLE;
```

Description

Data type for USB Device Endpoint Data Transfer Handle.

The data type of the handle that is returned by the [USB_DEVICE_EndpointRead\(\)](#) and [USB_DEVICE_EndpointWrite\(\)](#) functions.

Remarks

None.

USB_DEVICE_TRANSFER_HANDLE_INVALID Macro

Constant that defines the value of an Invalid Device Endpoint Data Transfer Handle.

File

[usb_device.h](#)

C

```
#define USB_DEVICE_TRANSFER_HANDLE_INVALID
```

Description

USB Device Layer Invalid Endpoint Data Transfer Handle

This constant defines the value that is returned by the [USB_DEVICE_EndpointRead\(\)](#) and [USB_DEVICE_EndpointWrite\(\)](#) functions, as a transfer handle, when the function is not successful.

Remarks

None.

USB_DEVICE_IRP Structure

This structure defines the USB Device Mode IRP data structure.

File

[usb_common.h](#)

C

```
typedef struct _USB_DEVICE_IRP {
    void * data;
    unsigned int size;
    USB_DEVICE_IRP_STATUS status;
    void (* callback)(struct _USB_DEVICE_IRP * irp);
    USB_DEVICE_IRP_FLAG flags;
    uintptr_t userData;
    uint32_t privateData[3];
} USB_DEVICE_IRP;
```

Members

Members	Description
void * data;	Pointer to the data buffer
unsigned int size;	Size of the data buffer
USB_DEVICE_IRP_STATUS status;	Status of the IRP
void (* callback)(struct _USB_DEVICE_IRP * irp);	IRP Callback. If this is NULL, <ul style="list-style-type: none"> • then there is no callback generated
USB_DEVICE_IRP_FLAG flags;	Request specific flags
uintptr_t userData;	User data
uint32_t privateData[3];	The following members should not be modified by the client

Description

USB Device Mode I/O Request Packet

This structure defines the USB Device Mode IRP data structure.

Remarks

None.

USB_DEVICE_IRP_FLAG Enumeration

USB Device IRP flags enumeration

File

[usb_common.h](#)

C

```
typedef enum {
    USB_DEVICE_IRP_FLAG_DATA_COMPLETE = 0x1,
    USB_DEVICE_IRP_FLAG_DATA_PENDING = 0x2
} USB_DEVICE_IRP_FLAG;
```

Members

Members	Description
USB_DEVICE_IRP_FLAG_DATA_COMPLETE = 0x1	Using this flag indicates that there is no <ul style="list-style-type: none"> more data pending in the IRP. When data moves from device to host, and if the IRP size is a multiple of endpoint size, specifying this flag sends the ZLP. The size is not a multiple of endpoint size, no ZLP will be sent.
USB_DEVICE_IRP_FLAG_DATA_PENDING = 0x2	In case of data moving from device to host, and if the <ul style="list-style-type: none"> size parameter of the IRP is an exact multiple of the endpoint maximum packet size, specifying this flag, does send the ZLP. If the size is less than endpoint size, specifying this flag will return an error. If the size is more than endpoint size but not a multiple, only endpoint multiple size of data is sent.

Description

USB Device IRP Flags

This enumeration defines the possible flags that can be specified while adding the IRP.

Remarks

Not all flags are applicable in all conditions. Refer to API documentation for more details

USB_DEVICE_IRP_STATUS Enumeration

Enumerates the possible status options of USB Device IRP.

File

[usb_common.h](#)

C

```
typedef enum {
    USB_DEVICE_IRP_STATUS_TERMINATED_BY_HOST = -4,
    USB_DEVICE_IRP_STATUS_ABORTED_ENDPOINT_HALT = -3,
    USB_DEVICE_IRP_STATUS_ABORTED = -2,
    USB_DEVICE_IRP_STATUS_ERROR = -1,
    USB_DEVICE_IRP_STATUS_COMPLETED = 0,
    USB_DEVICE_IRP_STATUS_COMPLETED_SHORT = 1,
    USB_DEVICE_IRP_STATUS_SETUP = 2,
    USB_DEVICE_IRP_STATUS_PENDING = 3,
    USB_DEVICE_IRP_STATUS_IN_PROGRESS = 4
} USB_DEVICE_IRP_STATUS;
```

Members

Members	Description
USB_DEVICE_IRP_STATUS_TERMINATED_BY_HOST = -4	The IRP was aborted because the host cleared the stall on the endpoint
USB_DEVICE_IRP_STATUS_ABORTED_ENDPOINT_HALT = -3	IRP was aborted because the endpoint halted
USB_DEVICE_IRP_STATUS_ABORTED = -2	USB Device IRP was aborted by the function driver
USB_DEVICE_IRP_STATUS_ERROR = -1	An error occurred on the bus when the IRP was being <ul style="list-style-type: none"> • processed
USB_DEVICE_IRP_STATUS_COMPLETED = 0	The IRP was completed
USB_DEVICE_IRP_STATUS_COMPLETED_SHORT = 1	The IRP was completed but the amount of <ul style="list-style-type: none"> • data received was less than the requested • size
USB_DEVICE_IRP_STATUS_SETUP = 2	The IRP was completed and the received <ul style="list-style-type: none"> • token was a SETUP token. This is applicable • to IRP scheduled on CONTROL endpoint
USB_DEVICE_IRP_STATUS_PENDING = 3	The IRP is pending in the queue
USB_DEVICE_IRP_STATUS_IN_PROGRESS = 4	The IRP is currently being processed

Description

USB Device IRP Status Enumeration

Enumerates the possible status options of USB Device IRP.

Remarks

The application that schedules the IRP can check the status member of the USB Device IRP at any time to obtain current status of the IRP.

USB_ENDPOINT Type

Defines a type to store Endpoint and Direction. The MSB defines the direction. The lower 4 bits defines the endpoint.

File

[usb_common.h](#)

C

```
typedef uint8_t USB_ENDPOINT;
```

Description

USB Endpoint and Direction Type

Defines a type to store Endpoint and Direction. The MSB defines the direction. The lower 4 bits defines the endpoint.

Remarks

None.

USB_ERROR Enumeration

Enumeration of all possible error codes that are returned by various components functions in the USB Stack.

File

[usb_common.h](#)

C

```
typedef enum {
    USB_ERROR_IRP_QUEUE_FULL = SCHAR_MIN,
    USB_ERROR_OSAL_FUNCTION,
    USB_ERROR_IRP_SIZE_INVALID,
    USB_ERROR_PARAMETER_INVALID,
    USB_ERROR_DEVICE_ENDPOINT_INVALID,
    USB_ERROR_DEVICE_IRP_IN_USE,
    USB_ERROR_CLIENT_NOT_READY,
    USB_ERROR_IRP_OBJECTS_UNAVAILABLE,
    USB_ERROR_DEVICE_FUNCTION_INSTANCE_INVALID,
}
```

```

USB_ERROR_DEVICE_FUNCTION_NOT_CONFIGURED,
USB_ERROR_ENDPOINT_NOT_CONFIGURED,
USB_ERROR_DEVICE_CONTROL_TRANSFER_FAILED,
USB_ERROR_HOST_DEVICE_INSTANCE_INVALID,
USB_ERROR_HOST_DRIVER_NOT_READY,
USB_ERROR_HOST_DRIVER_NOT_FOUND,
USB_ERROR_HOST_ENDPOINT_INVALID,
USB_ERROR_HOST_PIPE_INVALID,
USB_ERROR_HOST_ARGUMENTS_INVALID,
USB_ERROR_HOST_HEADERSIZE_INVALID,
USB_ERROR_HOST_MAX_INTERFACES_INVALID,
USB_ERROR_HOST_ENDPOINT_DESC_SIZE_INVALID,
USB_ERROR_HOST_DESCRIPTOR_INVALID,
USB_ERROR_HOST_MAX_ENDPOINT_INVALID,
USB_ERROR_HOST_ALT_SETTING_INVALID,
USB_ERROR_HOST_BUSY,
USB_HOST_OBJ_INVALID,
USB_ERROR_HOST_POINTER_INVALID,
USB_ERROR_HOST_ENDPOINT_NOT_FOUND,
USB_ERROR_HOST_DRIVER_INSTANCE_INVALID,
USB_ERROR_HOST_INTERFACE_NOT_FOUND,
USB_ERROR_ENDPOINT_HALTED,
USB_ERROR_TRANSFER_TERMINATED_BY_HOST,
USB_ERROR_NONE = 0
} USB_ERROR;

```

Members

Members	Description
USB_ERROR_IRP_QUEUE_FULL = SCHAR_MIN	IRP Queue Full Error
USB_ERROR_OSAL_FUNCTION	OSAL Function fails
USB_ERROR_IRP_SIZE_INVALID	IRP Size parameter invalid
USB_ERROR_PARAMETER_INVALID	Some function parameter was not valid
USB_ERROR_DEVICE_ENDPOINT_INVALID	Device endpoint is not valid
USB_ERROR_DEVICE_IRP_IN_USE	IRP is already in use
USB_ERROR_CLIENT_NOT_READY	Client is not ready
USB_ERROR_IRP_OBJECTS_UNAVAILABLE	Free IRP object unavailable
USB_ERROR_DEVICE_FUNCTION_INSTANCE_INVALID	Function Driver instance was not provisioned
USB_ERROR_DEVICE_FUNCTION_NOT_CONFIGURED	Function Driver instance is not configured
USB_ERROR_ENDPOINT_NOT_CONFIGURED	Endpoint is not configured
USB_ERROR_DEVICE_CONTROL_TRANSFER_FAILED	Device Control Transfer Failed
USB_ERROR_HOST_DEVICE_INSTANCE_INVALID	Host device instance invalid
USB_ERROR_HOST_DRIVER_NOT_READY	Host driver not ready for communication
USB_ERROR_HOST_DRIVER_NOT_FOUND	Host driver not found
USB_ERROR_HOST_ENDPOINT_INVALID	Host endpoint invalid
USB_ERROR_HOST_PIPE_INVALID	Host pipe invalid
USB_ERROR_HOST_ARGUMENTS_INVALID	Invalid arguments
USB_ERROR_HOST_HEADERSIZE_INVALID	Header size invalid
USB_ERROR_HOST_MAX_INTERFACES_INVALID	Max interface Number
USB_ERROR_HOST_ENDPOINT_DESC_SIZE_INVALID	Endpoint descriptor size is invalid
USB_ERROR_HOST_DESCRIPTOR_INVALID	Invalid Descriptor
USB_ERROR_HOST_MAX_ENDPOINT_INVALID	Invalid number of endpoints
USB_ERROR_HOST_ALT_SETTING_INVALID	Host alternate setting is invalid
USB_ERROR_HOST_BUSY	Host is busy
USB_HOST_OBJ_INVALID	USB host invalid
USB_ERROR_HOST_POINTER_INVALID	Pointer is invalid
USB_ERROR_HOST_ENDPOINT_NOT_FOUND	Could not find endpoint
USB_ERROR_HOST_DRIVER_INSTANCE_INVALID	Driver Instance Invalid
USB_ERROR_HOST_INTERFACE_NOT_FOUND	Could not find endpoint
USB_ERROR_ENDPOINT_HALTED	Transfer terminated because endpoint was halted
USB_ERROR_TRANSFER_TERMINATED_BY_HOST	Transfer terminated by host because of a stall clear
USB_ERROR_NONE = 0	No Error, Operation was successful

Description

USB Error Codes

Enumeration of all possible error codes that are returned by various components functions in the USB Stack.

Remarks

None.

USB_HOST_IRP Structure

This structure defines the USB Host Mode IRP data structure.

File

[usb_common.h](#)

C

```
typedef struct _USB_HOST_IRP {
    void * setup;
    void * data;
    unsigned int size;
    USB_HOST_IRP_STATUS status;
    USB_HOST_IRP_FLAG flags;
    uintptr_t userData;
    void (* callback)(struct _USB_HOST_IRP * irp);
    uintptr_t privateData[7];
} USB_HOST_IRP;
```

Members

Members	Description
void * setup;	Points to the 8 byte setup command <ul style="list-style-type: none"> • packet in case this is a IRP is • scheduled on a CONTROL pipe. Should • be NULL otherwise
void * data;	Pointer to data buffer
unsigned int size;	Size of the data buffer
USB_HOST_IRP_STATUS status;	Status of the IRP
USB_HOST_IRP_FLAG flags;	Request specific flags
uintptr_t userData;	User data
void (* callback)(struct _USB_HOST_IRP * irp);	Pointer to function to be called <ul style="list-style-type: none"> • when IRP is terminated. Can be • NULL, in which case the function • will not be called.
uintptr_t privateData[7];	These members of the IRP should not be modified by client

Description

USB Host Mode I/O Request Packet

This structure defines the USB Host Mode IRP data structure.

Remarks

None.

USB_HOST_IRP_FLAG Enumeration

USB Host IRP flags enumeration

File

[usb_common.h](#)

C

```
typedef enum {
    USB_HOST_IRP_FLAG_NONE = 0,
    USB_HOST_IRP_FLAG_SEND_ZLP,
```

```

    USB_HOST_IRP_WAIT_FOR_ZLP
} USB_HOST_IRP_FLAG;

```

Members

Members	Description
USB_HOST_IRP_FLAG_NONE = 0	Does not do anything
USB_HOST_IRP_FLAG_SEND_ZLP	In case of data moving from host to device, and if the <ul style="list-style-type: none"> • size parameter of the IRP is an exact multiple of the • endpoint maximum packet size, specifying this flag sends • a Zero Length Packet before the IRP is completed.
USB_HOST_IRP_WAIT_FOR_ZLP	In case of data moving device to host, and if the <ul style="list-style-type: none"> • size parameter of the IRP is an exact multiple of • the endpoint maximum packet size, specifying this • flag will cause the IRP to completed only when the • a ZLP was requested and acknowledged and the amount • of data was a multiple of endpoint maximum packet size.

Description

USB Host IRP Flags

This enumeration defines the possible flags that can be specified while adding the IRP.

Remarks

Not all flags are applicable in all conditions. Refer to API documentation for more details

USB_HOST_IRP_STATUS Enumeration

Enumerates the possible status options of USB Host IRP.

File

[usb_common.h](#)

C

```

typedef enum _USB_HOST_IRP_STATUS {
    USB_HOST_IRP_STATUS_ERROR_UNKNOWN = -6,
    USB_HOST_IRP_STATUS_ABORTED = -5,
    USB_HOST_IRP_STATUS_ERROR_BUS = -4,
    USB_HOST_IRP_STATUS_ERROR_DATA = -3,
    USB_HOST_IRP_STATUS_ERROR_NAK_TIMEOUT = -2,
    USB_HOST_IRP_STATUS_ERROR_STALL = -1,
    USB_HOST_IRP_STATUS_COMPLETED = 0,
    USB_HOST_IRP_STATUS_COMPLETED_SHORT = 1,
    USB_HOST_IRP_STATUS_PENDING = 2,
    USB_HOST_IRP_STATUS_IN_PROGRESS = 3
} USB_HOST_IRP_STATUS;

```

Members

Members	Description
USB_HOST_IRP_STATUS_ERROR_UNKNOWN = -6	IRP was terminated due to an unknown error
USB_HOST_IRP_STATUS_ABORTED = -5	IRP was terminated by the application
USB_HOST_IRP_STATUS_ERROR_BUS = -4	IRP was terminated due to a bus error
USB_HOST_IRP_STATUS_ERROR_DATA = -3	IRP was terminated due to data error
USB_HOST_IRP_STATUS_ERROR_NAK_TIMEOUT = -2	IRP was terminated because of a NAK timeout
USB_HOST_IRP_STATUS_ERROR_STALL = -1	IRP was terminated because of a STALL
USB_HOST_IRP_STATUS_COMPLETED = 0	IRP has been completed
USB_HOST_IRP_STATUS_COMPLETED_SHORT = 1	IRP has been completed but the <ul style="list-style-type: none"> • amount of data processed was less • than requested.
USB_HOST_IRP_STATUS_PENDING = 2	IRP is waiting in queue
USB_HOST_IRP_STATUS_IN_PROGRESS = 3	IRP is currently being processed

Description

USB Host IRP Status Enumeration

Enumerates the possible status options of USB Host IRP.

Remarks

The application that schedules the IRP can check the status member of the USB Host IRP at any time to obtain current status of the IRP.

USB_SPEED Enumeration

Provides enumeration of USB 2.0 speeds.

File

[usb_common.h](#)

C

```
typedef enum {
    USB_SPEED_ERROR = 0,
    USB_SPEED_HIGH = 1,
    USB_SPEED_FULL = 2,
    USB_SPEED_LOW = 3
} USB_SPEED;
```

Members

Members	Description
USB_SPEED_ERROR = 0	Error in obtaining USB module speed
USB_SPEED_HIGH = 1	USB module is at high speed
USB_SPEED_FULL = 2	USB module is at full speed
USB_SPEED_LOW = 3	USB module is at low speed

Description

USB 2.0 Speeds Enumeration

Provides enumeration of USB 2.0 speeds.

Remarks

None.

USB_ENDPOINT_AND_DIRECTION Macro

This macro helps in setting up the [USB_ENDPOINT](#) type.

File

[usb_common.h](#)

C

```
#define USB_ENDPOINT_AND_DIRECTION(direction, endpoint) ((uint8_t)((direction << 7) | endpoint))
```

Description

USB Endpoint and Direction helper macro

This macro helps in setting up the [USB_ENDPOINT](#) type. Here x is the direction and can be either USB_DATA_DIRECTION_HOST_TO_DEVICE or USB_DATA_DIRECTION_DEVICE_TO_HOST. y is the endpoint.

Remarks

None.

USB_DATA_DIRECTION Enumeration

Defines the communication direction

File

[usb_common.h](#)

C

```
typedef enum {
    USB_DATA_DIRECTION_DEVICE_TO_HOST = 1,
    USB_DATA_DIRECTION_HOST_TO_DEVICE = 0
} USB_DATA_DIRECTION;
```

Members

Members	Description
USB_DATA_DIRECTION_DEVICE_TO_HOST = 1	Data moves from device to host
USB_DATA_DIRECTION_HOST_TO_DEVICE = 0	Data moves from host to device

Description

USB Communication direction definitions

This definitions define the communication direction and can be used to specify direction while using the DRV_USB_ENDPOINT type.

Remarks

None.

USB_DEVICE_BOS_DESCRIPTOR_SUPPORT_ENABLE Macro

Specifies if the Device Layer should process a Host request for a BOS descriptor.

File

[usb_device_config_template.h](#)

C

```
#define USB_DEVICE_BOS_DESCRIPTOR_SUPPORT_ENABLE
```

Description

USB Device Layer BOS Descriptor Support Enable

Specifying this configuration macro will enable support for BOS request. When the request is received, the device layer will transfer the data pointed to by the bosDescriptor member of the [USB_DEVICE_INIT](#) data structure. If this configuration macro is not specified, request for a BOS descriptor is stalled.

Remarks

The USB Host will request for a BOS descriptor when the bcdVersion field in the Device Descriptor is greater than 0x0200.

USB_DEVICE_DRIVER_INITIALIZE_EXPLICIT Macro

Specifies if the USB Controller Driver must be initialized explicitly as opposed to being initialized by the Device Layer.

File

[usb_device_config_template.h](#)

C

```
#define USB_DEVICE_DRIVER_INITIALIZE_EXPLICIT
```

Description

USB Device Layer USB Controller Driver Explicit Initialize

Specifying this macro indicates that the USB Controller Driver will be initialized explicitly in the SYS_Initialize() function. The Device Layer will not initialize the USB Controller Driver. All releases of the USB Device Layer starting from v1.04 of MPALB Harmony will specify this macro i.e the controller driver will be initialized explicitly.

If this macro is not specified, then the USB Device Layer will initialize the controller driver and run its tasks routines.

Remarks

None.

USB_DEVICE_STRING_DESCRIPTOR_TABLE_ADVANCED_ENABLE Macro

Specifying this macro enables the Advanced String Descriptor Table Entry Format.

File

[usb_device_config_template.h](#)

C

```
#define USB_DEVICE_STRING_DESCRIPTOR_TABLE_ADVANCED_ENABLE
```

Description

USB Device Layer Advanced String Descriptor Table Entry Format Enable.

Specifyin this macro enables the Advanced String Descriptor Table Entry Format. The advanced format allows the application to specify the String Index and the language in the entry itself. In the basic format, this information is obtained by the virtue of the entry index of the String Descriptor in the String Descriptor Table. Using the Advanced format allows the application to specify strings with arbitrary strings indexes. In basic format, the string indexes are forced to be contiguous.

Remarks

The basic string descriptor entry format is selected by default. The advanced format must be enabled explicitly by specifying this macro.

Files**Files**

Name	Description
usb_device.h	USB Device Layer Interface Header
usb_common.h	USB Common Definitions File
usb_device_config_template.h	USB device configuration template header file.

Description

This section lists the source and header files used by the library.

usb_device.h

USB Device Layer Interface Header

Enumerations

	Name	Description
	USB_DEVICE_CLIENT_STATUS	Enumerated data type that identifies the USB Device Layer Client Status.
	USB_DEVICE_CONTROL_STATUS	USB Device Layer Control Transfer Status Stage flags.
	USB_DEVICE_CONTROL_TRANSFER_RESULT	Enumerated data type identifying results of a control transfer.
	USB_DEVICE_EVENT	USB Device Layer Events.
	USB_DEVICE_POWER_STATE	Enumerated data type that identifies if the device is self powered or bus powered .
	USB_DEVICE_REMOTE_WAKEUP_STATUS	Enumerated data type that identifies if the remote wakeup status of the device.
	USB_DEVICE_RESULT	USB Device Layer Results Enumeration
	USB_DEVICE_TRANSFER_FLAGS	Enumerated data type that identifies the USB Device Layer Transfer Flags.

Functions

	Name	Description
≡◊	USB_DEVICE_ActiveConfigurationGet	Informs the client of the current USB device configuration set by the USB host.
≡◊	USB_DEVICE_ActiveSpeedGet	Informs the client of the current operation speed of the USB bus.
≡◊	USB_DEVICE_Attach	This function will attach the device to the USB.
≡◊	USB_DEVICE_ClientStatusGet	Returns the client specific status.
≡◊	USB_DEVICE_Close	Closes an opened handle to an instance of the USB device layer.
≡◊	USB_DEVICE_ControlReceive	Receives data stage of the control transfer from host to device.
≡◊	USB_DEVICE_ControlSend	Sends data stage of the control transfer from device to host.
≡◊	USB_DEVICE_ControlStatus	Initiates status stage of the control transfer.

USB_DEVICE_Deinitialize	De-initializes the specified instance of the USB device layer.
USB_DEVICE_Detach	This function will detach the device from the USB.
USB_DEVICE_EndpointDisable	Disables a device endpoint.
USB_DEVICE_EndpointEnable	Enables a device endpoint.
USB_DEVICE_EndpointsEnabled	Returns true if the endpoint is enabled.
USB_DEVICE_EndpointIsStalled	This function returns the stall status of the specified endpoint and direction.
USB_DEVICE_EndpointRead	Reads data received from host on the requested endpoint.
USB_DEVICE_EndpointStall	This function stalls an endpoint in the specified direction.
USB_DEVICE_EndpointStallClear	This function clears the stall on an endpoint in the specified direction.
USB_DEVICE_EndpointTransferCancel	This function cancels a transfer scheduled on an endpoint.
USB_DEVICE_EndpointWrite	This function requests a data write to a USB Device Endpoint.
USB_DEVICE_EventHandlerSet	USB Device Layer Event Handler Callback Function set function.
USB_DEVICE_Initialize	Creates and initializes an instance of the USB device layer.
USB_DEVICE_IsSuspended	Returns true if the device is in a suspended state.
USB_DEVICE_Open	Opens the specified USB device layer instance and returns a handle to it.
USB_DEVICE_PowerStateSet	Sets power state of the device.
USB_DEVICE_RemoteWakeUpStart	This function will start the resume signaling.
USB_DEVICE_RemoteWakeUpStartTimed	This function will start a self timed Remote Wake-up.
USB_DEVICE_RemoteWakeUpStatusGet	Gets the "Remote wake-up" status of the device.
USB_DEVICE_RemoteWakeUpStop	This function will stop the resume signaling.
USB_DEVICE_StateGet	Returns the current state of the USB device.
USB_DEVICE_Status	Provides the current status of the USB device layer
USB_DEVICE_Tasks	USB Device layer calls all other function driver tasks in this function. It also generates and forwards events to its clients.
USB_DEVICE_Tasks_ISR	USB Device Layer Tasks routine to be called in the USB Interrupt Service Routine.
USB_DEVICE_Tasks_ISR_USBDMA	This is function USB_DEVICE_Tasks_ISR_USBDMA .

Macros

	Name	Description
	USB_DEVICE_EVENT_RESPONSE_NONE	Device Layer Event Handler Function Response Type.
	USB_DEVICE_HANDLE_INVALID	Constant that defines the value of an Invalid Device Handle.
	USB_DEVICE_INDEX_0	USB device layer index definitions.
	USB_DEVICE_INDEX_1	This is macro USB_DEVICE_INDEX_1 .
	USB_DEVICE_INDEX_2	This is macro USB_DEVICE_INDEX_2 .
	USB_DEVICE_INDEX_3	This is macro USB_DEVICE_INDEX_3 .
	USB_DEVICE_INDEX_4	This is macro USB_DEVICE_INDEX_4 .
	USB_DEVICE_INDEX_5	This is macro USB_DEVICE_INDEX_5 .
	USB_DEVICE_TRANSFER_HANDLE_INVALID	Constant that defines the value of an Invalid Device Endpoint Data Transfer Handle.

Structures

	Name	Description
	USB_DEVICE_EVENT_DATA_CONFIGURED	USB Device Set Configuration Event Data type.
	USB_DEVICE_EVENT_DATA_ENDPOINT_READ_COMPLETE	USB Device Layer Endpoint Read and Write Complete Event Data type.
	USB_DEVICE_EVENT_DATA_ENDPOINT_WRITE_COMPLETE	USB Device Layer Endpoint Read and Write Complete Event Data type.
	USB_DEVICE_EVENT_DATA_SET_DESCRIPTOR	USB Device Set Descriptor Event Data type.
	USB_DEVICE_EVENT_DATA_SOF	USB Device Start Of Frame Event Data Type
	USB_DEVICE_EVENT_DATA_SYNCH_FRAME	USB Device Synch Frame Event Data type.
	USB_DEVICE_FUNCTION_REGISTRATION_TABLE	USB Device Function Registration Structure
	USB_DEVICE_INIT	USB Device Initialization Structure
	USB_DEVICE_MASTER_DESCRIPTOR	USB Device Master Descriptor Structure.

Types

	Name	Description
	USB_DEVICE_CONFIGURATION_DESCRIPTOR_TABLE	Pointer to an array that contains pointer to configuration descriptors.
	USB_DEVICE_EVENT_HANDLER	USB Device Layer Event Handler Function Pointer Type
	USB_DEVICE_EVENT_RESPONSE	Device Layer Event Handler function return type.
	USB_DEVICE_HANDLE	Data type for USB device handle.
	USB_DEVICE_STRING_DESCRIPTOR_TABLE	Pointer to an array that contains pointer to string descriptors.
	USB_DEVICE_TRANSFER_HANDLE	Data type for USB Device Endpoint Data Transfer Handle.

Description

USB Device Layer Interface Definition

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the USB device layer. This application should include this file if it needs to use the USB Device Layer API.

File Name

usb_device.h

Company

Microchip Technology Inc.

usb_common.h

USB Common Definitions File

Enumerations

	Name	Description
	_USB_HOST_IRP_STATUS	Enumerates the possible status options of USB Host IRP.
	USB_DATA_DIRECTION	Defines the communication direction
	USB_DEVICE_IRP_FLAG	USB Device IRP flags enumeration
	USB_DEVICE_IRP_STATUS	Enumerates the possible status options of USB Device IRP.
	USB_ERROR	Enumeration of all possible error codes that are returned by various components functions in the USB Stack.
	USB_HOST_IRP_FLAG	USB Host IRP flags enumeration
	USB_HOST_IRP_STATUS	Enumerates the possible status options of USB Host IRP.
	USB_SPEED	Provides enumeration of USB 2.0 speeds.

Macros

	Name	Description
	USB_ENDPOINT_AND_DIRECTION	This macro helps in setting up the USB_ENDPOINT type.

Structures

	Name	Description
	_USB_DEVICE_IRP	This structure defines the USB Device Mode IRP data structure.
	_USB_HOST_IRP	This structure defines the USB Host Mode IRP data structure.
	USB_DEVICE_IRP	This structure defines the USB Device Mode IRP data structure.
	USB_HOST_IRP	This structure defines the USB Host Mode IRP data structure.

Types

	Name	Description
	USB_ENDPOINT	Defines a type to store Endpoint and Direction. The MSB defines the direction. The lower 4 bits defines the endpoint.

Description

USB Common Definitions File

This file contains definitions that are used by various components of the USB stack. This file is included by the USB Device and Host stack files. The application may typically not need to include this file directly.

File Name

usb_common.h

Company

Microchip Technology Inc.

usb_device_config_template.h

USB device configuration template header file.

Macros

Name	Description
USB_DEVICE_BOS_DESCRIPTOR_SUPPORT_ENABLE	Specifies if the Device Layer should process a Host request for a BOS descriptor.
USB_DEVICE_DRIVER_INITIALIZE_EXPLICIT	Specifies if the USB Controller Driver must be initialized explicitly as opposed to being initialized by the Device Layer.
USB_DEVICE_ENDPOINT_QUEUE_DEPTH_COMBINED	Specifies the combined endpoint queue depth in case of a vendor USB device implementation.
USB_DEVICE_EP0_BUFFER_SIZE	Buffer Size in Bytes for Endpoint 0.
USB_DEVICE_INSTANCES_NUMBER	Number of Device Layer instances to provisioned in the application.
USB_DEVICE_MICROSOFT_OS_DESCRIPTOR_SUPPORT_ENABLE	Specifies if the USB Device stack should support Microsoft OS Descriptor.
USB_DEVICE_SET_DESCRIPTOR_EVENT_ENABLE	Enables the Device Layer Set Descriptor Event.
USB_DEVICE_SOF_EVENT_ENABLE	Enables the Device Layer SOF event.
USB_DEVICE_STRING_DESCRIPTOR_TABLE_ADVANCED_ENABLE	Specifying this macro enables the Advanced String Descriptor Table Entry Format.
USB_DEVICE_SYNCH_FRAME_EVENT_ENABLE	Enables the Device Layer Synch Frame Event.

Description

USB Device Layer Compile Time Options

This file contains USB device layer compile time options (macros) that are to be configured by the user. This file is a template file and must be used as an example only. This file must not be directly included in the project.

File Name

usb_device_config_template.h

Company

Microchip Technology Inc.

USB CDC Device Library

This section describes the USB CDC Device Library.

Introduction

This help section provides information on library design, configuration, usage and the Library Interface for the USB Communications Device Class (CDC) Device Library.

Description

The MPLAB Harmony USB Communications Device Class (CDC) Device Library (also referred to as the CDC function driver or library) provides functions and methods that allow application designers to implement a USB CDC Device. The current version of the library supports the Abstract Control Model (ACM) of the CDC specification revision 1.2 and specifically implements a subset of the AT250 command set. This library must be used in conjunction with the MPLAB Harmony USB Device Layer.

Using the Library

This topic describes the basic architecture of the USB CDC Device Library and provides information and examples on its use.

Library Overview

The USB CDC Device Library mainly interacts with the system, its clients and function drivers, as shown in the [Abstraction Model](#).

The library interface routines are divided into sub-sections, which address one of the blocks or the overall operation of the USB CDC Device Library.

Library Interface Section	Description
Functions	Provides event handler, read/write, and serial state notification functions.

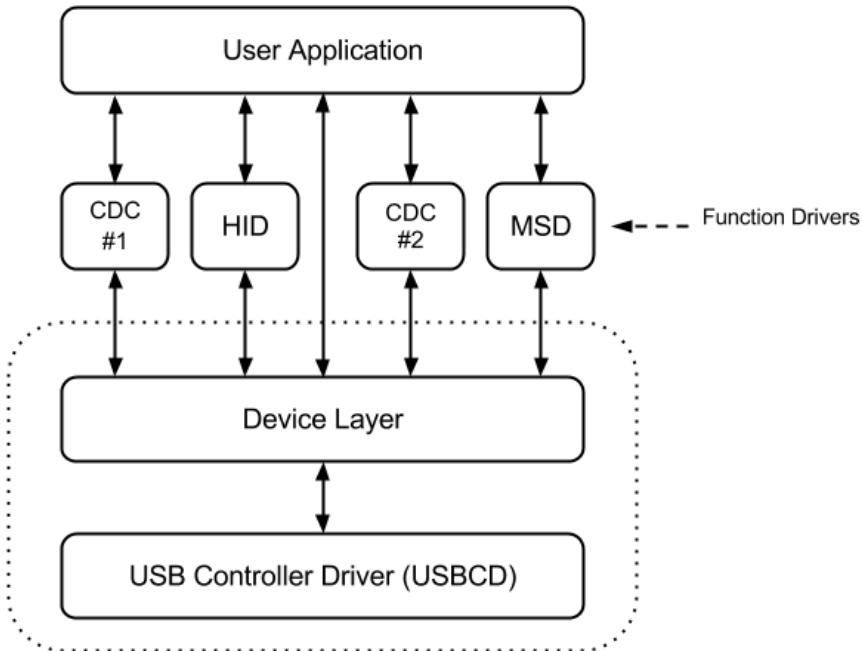
Abstraction Model

Provides an architectural overview of the CDC Function Driver.

Description

The CDC Function Driver offers services to a USB CDC Device to communicate with the host by abstracting the USB specification details. It must be used along with the USB Device Layer and USB controller to communicate with the USB Host. Figure 1 shows a block diagram of the MPLAB Harmony USB Architecture and where the CDC Function Driver is placed.

Figure 1: CDC Function Driver



As shown in Figure 1, the USB Controller Driver takes the responsibility of managing the USB peripheral on the device. The USB Device Layer handles the device enumeration, etc. The USB Device layer forwards all USB CDC specific control transfers to the CDC Function Driver. The CDC Function Driver ACM sub-layer interprets the control transfers and requests application's intervention through event handlers and well defined set of API. The application must register a event handler with the CDC Function Driver in the Device Layer Set Configuration Event. The application should respond to CDC ACM events. Response to CDC ACM event that require control transfer response can be deferred by responding to the event after returning from the event handler. The application interacts directly with the CDC Function Driver to send/receive data and to send serial state notifications.

As per the CDC specification,a USB CDC Device is a collection of the following interfaces:

- Communication Interface (Device Management) on Endpoint 0
- Optional Communication Interface (Notification) on an interrupt endpoint
- Optional Data Interface (either a bulk or isochronous endpoint)

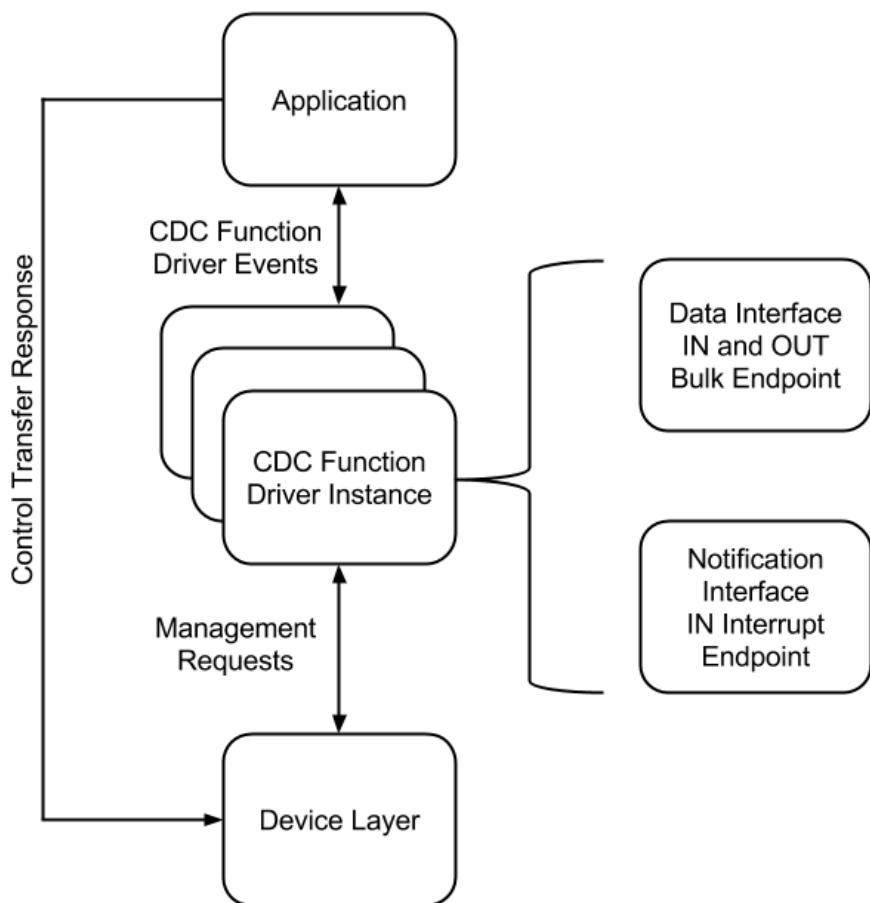


Figure 2: CDC Function Driver Architecture

Figure 2 shows the architecture of the CDC Function Driver. The device management on Endpoint 0 is handled by the device library(class specific requests are routed to the CDC Function Driver by the USB Device Layer). An instance of the CDC Function Driver actually consists of a data interface and a notification interface. The library is implemented in two .c files. The `usb_device_cdc.c` file implements the CDC data and serial state notification, while the `usb_device_cdc_acm.c` file implements the control transfer interpretation and event generation. The application must respond to control transfer related CDC ACM events by directly calling the Device Layer control transfer routines.

Abstract Control Model (ACM)

Describes the various Abstract Control Model (ACM) commands supported by this CDC Function Driver implementation.

Description

One of the basic supported models for communication by CDC is POTS (Plain Old Telephone Service). The POTS model is for devices that communicate via ordinary phone lines and generic COM port devices. The USB CDC specification refers to this basic model as PSTN (Public Switched Telephone Network).

Depending on the amount of data processing the device is responsible for POTS/PSTN is divided into several models. The processing of data can include modulation, demodulation, error correction and data compression.

Of the supported PSTN models, this CDC Function Driver implements ACM. In the ACM the device handles modulation, demodulation and handles V.25ter (AT) commands. This model (ACM) also supports requests and notifications to get and set RS-232 status, control, and asynchronous port part parameters. Virtual COM port devices use ACM.

The following sections describe the management requests and notifications supported by the CDC Function Driver ACM layer.

Management Requests

The Host requests/sends some information in the form of management requests on the bidirectional Endpoint 0. The following table shows the CDC specification ACM sub class management requests and how these request are handled by the CDC Function Driver.

Request Code	Required/Optional	Comments
<code>SEND_ENCAPSULATED_COMMAND</code>	Required	Implemented by the CDC Function Driver ACM layer. This request is stalled.
<code>GET_ENCAPSULATED_RESPONSE</code>	Required	Implemented by the CDC Function Driver ACM layer. This request is stalled.

SET_COMM_FEATURE	Optional	Not Implemented.
GET_COMM_FEATURE	Optional	Not Implemented.
CLEAR_COMM_FEATURE	Optional	Not Implemented.
SET_LINE_CODING	Optional	Implemented by the CDC Function Driver ACM layer. Requires application response.
GET_LINE_CODING	Optional	Implemented by the CDC Function Driver ACM layer. Requires application response.
SET_CONTROL_LINE_STATE	Optional	Implemented by the CDC Function Driver ACM layer. Requires application response.
SEND_BREAK	Optional	Implemented by the CDC Function Driver ACM layer. Requires application response.

How the Library Works

Library Initialization

Describes how the CDC Function Driver is initialized.

Description

The CDC Function Driver instance for a USB device configuration is initialized by the Device Layer when the configuration is set by the host. This process does not require application intervention. Each instance of the CDC Function Driver should be registered with the Device layer through the [Device Layer Function Driver Registration Table](#). The CDC Function Driver does require a initialization data structure to be defined for each instance of the function driver. This initialization data structure should be of the type `USB_DEVICE_CDC_INIT`. This data structure specifies the read and write queue sizes. The `funcDriverInit` member of the function driver registration table entry for the CDC Function Driver instance should be set to point to the corresponding initialization data structure. The `USB_DEVICE_CDC_FUNCTION_DRIVER` object is a global object provided by the CDC Function Driver and points to the CDC Function Driver - Device Layer interface functions, which are required by the Device Layer. The following code an example of how multiple instances of CDC Function Driver can registered with the Device Layer.

```
/* This code shows an example of how two CDC function
 * driver instances can be registered with the Device Layer
 * via the Device Layer Function Driver Registration Table.
 * In this case Device Configuration 1 consists of two CDC
 * function driver instances. */

/* Define the CDC initialization data structure for CDC instance 0.
 * Set read queue size to 2 and write queue size to 3 */

const USB_DEVICE_CDC_INIT cdcInit0 = { .queueSizeRead = 2, .queueSizeWrite = 3 };

/* Define the CDC initialization data structure for CDC instance 1.
 * Set read queue size to 4 and write queue size to 1 */

const USB_DEVICE_CDC_INIT cdcInit1 = { .queueSizeRead = 4, .queueSizeWrite = 1 };
const USB_DEVICE_FUNC_REGISTRATION_TABLE funcRegistrationTable[2] =
{
    /* This is the first instance of the CDC Function Driver */
    {
        .speed = USB_SPEED_FULL|USB_SPEED_HIGH,           // Supported speed
        .configurationValue = 1,                          // To be initialized for Configuration 1
        .interfaceNumber = 0,                            // Starting interface number.
        .numberOfInterfaces = 2,                         // Number of interfaces in this instance
        .funcDriverIndex = 0,                            // Function Driver instance index is 0
        .funcDriverInit = &cdcInit0,                      // Function Driver initialization data structure
        .driver = USB_DEVICE_CDC_FUNCTION_DRIVER         // Pointer to Function Driver - Device Layer interface
    },
    /* This is the second instance of the CDC Function Driver */
    {
        .speed = USB_SPEED_FULL|USB_SPEED_HIGH,           // Supported speed
        .configurationValue = 1,                          // To be initialized for Configuration 1
        .interfaceNumber = 2,                            // Starting interface number.
        .numberOfInterfaces = 2,                         // Number of interfaces in this instance
    }
};
```

```

    .funcDriverIndex = 1,                                // Function Driver instance index is 1
    .funcDriverInit = &cdcInit1,                         // Function Driver initialization data structure
    .driver = USB_DEVICE_CDC_FUNCTION_DRIVER           // Pointer to Function Driver - Device Layer interface
}
};


```

Event Handling

Describes CDC Function Driver event handler registration and event handling.

Description

Registering a CDC Function Driver Event Handler

While creating USB CDC Device-based application, an event handler must be registered with the Device Layer (the Device Layer Event Handler) and every CDC Function Driver instance (CDC Function Driver Event Handler). The CDC Function Driver event handler receives CDC and CDC ACM events. This event handler should be registered before the USB device layer acknowledges the SET CONFIGURATION request from the USB Host. To ensure this, the event handler should be set in the USB_DEVICE_EVENT_CONFIGURED event that is generated by the device layer. While registering the CDC Function Driver event handler, the CDC Function Driver allows the application to also pass a data object in the event handler register function. This data object gets associated with the instance of the CDC Function Driver and is returned by the CDC Function Driver when a CDC Function Driver event occurs. The following code shows an example of how this can be done.

```

/* This is a sample Application Device Layer Event Handler
 * Note how the CDC Function Driver event handler APP_USBDeviceCDCEventHandler()
 * is registered in the USB_DEVICE_EVENT_CONFIGURED event. The appData
 * object that is passed in the USB_DEVICE_CDC_EventHandlerSet()
 * function will be returned as the userData parameter in the
 * when the APP_USBDeviceCDCEventHandler() function is invoked */

/* Application states */
typedef enum
{
    /* Application's state machine's initial state. */
    APP_STATE_INIT=0,
    APP_STATE_SERVICE_TASKS,
    APP_STATE_WAIT_FOR_CONFIGURATION,
} APP_STATES;

USB_DEVICE_HANDLE usbDeviceHandle;
APP_STATES appState;

/* This is the application device layer event handler function. */

USB_DEVICE_EVENT_RESPONSE APP_USBDeviceEventHandler
(
    USB_DEVICE_EVENT event,
    void * eventData,
    uintptr_t context
)
{
    uint8_t activeConfiguration;
    USB_SETUP_PACKET * setupPacket;
    switch(event)
    {
        case USB_DEVICE_EVENT_POWER_DETECTED:
            /* This event is generated when VBUS is detected. Attach the device */
            USB_DEVICE_Attach(usbDeviceHandle);
            break;

        case USB_DEVICE_EVENT_POWER_REMOVED:
            /* This event is generated when VBUS is removed. Detach the device */
            USB_DEVICE_Detach (usbDeviceHandle);
            break;

        case USB_DEVICE_EVENT_CONFIGURED:
            /* This event indicates that Host has set Configuration in the Device.*/
            /* Check the configuration */
            activeConfiguration = ((USB_DEVICE_EVENT_DATA_CONFIGURED *)pData)->configurationValue;
            if ( activeConfiguration == 1)


```

```

    {
        /* Register the CDC Device application event handler here.
         * Note how the appData object pointer is passed as the
         * user data */
        USB_DEVICE_CDC_EventHandlerSet(USB_DEVICE_CDC_INDEX_0, APP_USBDeviceCDCEventHandler,
                                        (uintptr_t)&appData);
        /* Mark that set configuration is complete */
        appData.isConfigured = true;
    }
    break;

    case USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST:
        /* This event indicates a Control transfer setup stage has been completed. */
        setupPacket = (USB_SETUP_PACKET *)eventData;

        /* Parse the setup packet and respond with a USB_DEVICE_ControlSend(),
         * USB_DEVICE_ControlReceive or USB_DEVICE_ControlStatus(). */

        break;

    case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT:
        /* This event indicates that a Control transfer Data has been sent to Host. */
        break;

    case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA RECEIVED:
        /* This event indicates that a Control transfer Data has been received from Host. */
        break;

    case USB_DEVICE_EVENT_CONTROL_TRANSFER_ABORTED:
        /* This event indicates a control transfer was aborted. */
        break;

    case USB_DEVICE_EVENT_SUSPENDED:
        break;

    case USB_DEVICE_EVENT_RESUMED:
        break;

    case USB_DEVICE_EVENT_ERROR:
        break;

    case USB_DEVICE_EVENT_RESET:
        break;

    case USB_DEVICE_EVENT_SOF:
        /* This event indicates an SOF is detected on the bus. The USB_DEVICE_SOF_EVENT_ENABLE
         * macro should be defined to get this event. */
        break;
    default:
        break;
    }
}

```

The CDC Function Driver event handler executes in an interrupt context when the device stack is configured for Interrupt mode. In Polled mode, the event handler is invoked in the context of the SYS_Tasks function. The application should not call computationally intensive functions, blocking functions, functions that are not interrupt safe, or functions that poll on hardware conditions from the event handler. Doing so will affect the ability of the USB device stack to respond to USB events and could potentially make the USB device non-compliant.

CDC Function Driver Events

The CDC Function Driver generates events to which the application must respond. Some of these events are management requests communicated through control transfers. Therefore, the application must use the Device Layer Control Transfer routines to complete the control transfer. Based on the generated event, the application may be required to:

- Respond with a [USB_DEVICE_ControlSend](#) function, which completes the data stage of a Control Read Transfer
- Respond with a [USB_DEVICE_ControlReceive](#) function, which provisions the data stage of a Control Write Transfer
- Respond with a [USB_DEVICE_ControlStatus](#) function, which completes the handshake stage of the Control Transfer. The application can either STALL or Acknowledge the handshake stage via the [USB_DEVICE_ControlStatus](#) function. The following table shows the CDC Function Driver Control Transfer related events and the required application control transfer actions.

CDC Function Driver Control Transfer Event	Required Application Action
USB_DEVICE_CDC_EVENT_SET_LINE_CODING	Call USB_DEVICE_ControlReceive function with a buffer to receive the USB_CDC_LINE_CODING type data.
USB_DEVICE_CDC_EVENT_SET_LINE_CODING	Call USB_DEVICE_ControlSend function with a buffer that contains the current USB_CDC_LINE_CODING type data.
USB_DEVICE_CDC_EVENT_SET_CONTROL_LINE_STATE	Acknowledge or stall using the USB_DEVICE_ControlStatus function.
USB_DEVICE_CDC_EVENT_SET_CONTROL_LINE_STATE	Acknowledge or stall using the USB_DEVICE_ControlStatus function.
USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_SENT	Action not required.
USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA RECEIVED	Acknowledge or stall using the USB_DEVICE_ControlStatus function.

Based on the type of event, the application should analyze the pData member of the event handler. This data member should be type cast to an event specific data type. The following table shows the event and the data type to use while type casting. Note that the pData member is not required for all events

CDC Function Driver Event	Related pData type
USB_DEVICE_CDC_EVENT_SET_LINE_CODING	NULL
USB_DEVICE_CDC_EVENT_GET_LINE_CODING	NULL
USB_DEVICE_CDC_EVENT_SET_CONTROL_LINE_STATE	USB_CDC_CONTROL_LINE_STATE *
USB_DEVICE_CDC_EVENT_SEND_BREAK	USB_DEVICE_CDC_EVENT_DATA_SEND_BREAK *
USB_DEVICE_CDC_EVENT_WRITE_COMPLETE	USB_DEVICE_CDC_EVENT_DATA_WRITE_COMPLETE *
USB_DEVICE_CDC_EVENT_READ_COMPLETE	USB_DEVICE_CDC_EVENT_DATA_READ_COMPLETE *
USB_DEVICE_CDC_EVENT_SERIAL_STATE_NOTIFICATION_COMPLETE	USB_DEVICE_CDC_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE *
USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_SENT	NULL
USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA RECEIVED	NULL

The possible CDC Function Driver events are described here with the required application response, event specific data, and likely follow-up function driver event:

[USB_DEVICE_CDC_EVENT_SET_LINE_CODING](#)

Application Response: This event occurs when the host issues a SET LINE CODING command. The application must provide a USB_CDC_LINE_CODING data structure to the device layer to receive the line coding data that the host will provide. The application must provide the buffer by calling the [USB_DEVICE_CDC_ControlReceive](#) function either in the event handler or in the application after returning from the event handler. The application can use the [USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_SENT](#) event to track completion of the command.

Event Specific Data (pData): The pData parameter will be NULL.

Likely Follow-up event: This event will likely be followed by the [USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA RECEIVED](#) event. This indicates that the data was received successfully. The application must either acknowledge or stall the handshake stage of the control transfer by calling the [USB_DEVICE_ControlStatus](#) function with the [USB_DEVICE_CONTROL_STATUS_OK](#) or [USB_DEVICE_CONTROL_STATUS_ERROR](#) flag, respectively.

[USB_DEVICE_CDC_EVENT_GET_LINE_CODING](#)

Application Response: This event occurs when the host issues a GET LINE CODING command. The application must provide a USB_CDC_LINE_CODING data structure to the device layer that contains the line coding data to be provided to the Host. The application must provide the buffer by calling the [USB_DEVICE_ControlSend](#) function either in the event handler or in the application after returning from the event handler. The size of the buffer is indicated by the length parameter. The application can use the [USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_SENT](#) event to track completion of the command.

Event Specific Data (pData): The pData parameter will be NULL.

Likely Follow-up event: This event will likely be followed by the [USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_SENT](#) event. This indicates that the data was sent to the Host successfully.

[USB_DEVICE_CDC_EVENT_SET_CONTROL_LINE_STATE](#)

Application Response: This event occurs when the host issues a SET CONTROL LINE STATE command. The application can then use the [USB_DEVICE_ControlStatus](#) function to indicate acceptance of rejection of the command. The [USB_DEVICE_ControlStatus](#) function can be

called from the event handler or in the application after returning from the event handler.

Event Specific Data (pData): The application must interpret the pData parameter as a pointer to a USB_CDC_CONTROL_LINE_STATE data type that contains the control line state data.

Likely Follow-up event: None.

USB_DEVICE_CDC_EVENT_SEND_BREAK

Application Response: This event occurs when the Host issues a SEND BREAK command. The application can then use the [USB_DEVICE_ControlStatus](#) function to indicate acceptance or rejection of the command. The [USB_DEVICE_ControlStatus](#) function can be called from the event handler or in the application after returning from the event handler.

Event Specific Data (pData): The application must interpret the pData parameter as a pointer to a uint16_t data type that contains the break duration data.

Likely Follow-up event: None.

USB_DEVICE_CDC_EVENT_WRITE_COMPLETE

Application Response: This event occurs when a write operation scheduled by calling the [USB_DEVICE_CDC_Write](#) function has completed. This event does not require the application to respond with any function calls.

Event Specific Data (pData): The pData member in the event handler will point to the [USB_DEVICE_CDC_EVENT_DATA_WRITE_COMPLETE](#) data type.

Likely Follow-up event: None.

USB_DEVICE_CDC_EVENT_READ_COMPLETE

Application Response: This event occurs when a read operation scheduled by calling the [USB_DEVICE_CDC_Read](#) function has completed. This event does not require the application to respond with any function calls.

Event Specific Data (pData): The pData member in the event handler will point to the [USB_DEVICE_CDC_EVENT_DATA_READ_COMPLETE](#) type.

Likely Follow-up event: None.

USB_DEVICE_CDC_EVENT_SERIAL_STATE_NOTIFICATION_COMPLETE

Application Response: This event occurs when a serial state notification send scheduled by calling the [USB_DEVICE_CDC_SerialStateNotificationSend](#) function has completed. This event does not require the application to respond with any function calls.

Event Specific Data (pData): The pData member in the event handler will point to the [USB_DEVICE_CDC_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE](#) data type.

Likely Follow-up event: None.

USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_SENT

Application Response: This event occurs when the data stage of a control read transfer has completed in response to the [USB_DEVICE_ControlSend](#) function (in the [USB_DEVICE_CDC_EVENT_GET_LINE_CODING](#) event). The application must acknowledge the handshake stage of the control transfer by calling the [USB_DEVICE_ControlStatus](#) function with the [USB_DEVICE_CONTROL_STATUS_OK](#) flag.

Event Specific Data (pData): The pData parameter will be NULL.

Likely Follow-up event: None.

USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA RECEIVED

Application Response: This event occurs when the data stage of a control write transfer has completed in response to the [USB_DEVICE_ControlReceive](#) function (in the [USB_DEVICE_CDC_EVENT_SET_LINE_CODING](#) event).

Event Specific Data (pData): The pData parameter will be NULL.

Likely Follow-up event: None.

CDC Function Driver Event Handling

The following code shows an event handling scheme example. The application always returns from the event handler with a [USB_DEVICE_CDC_EVENT_RESPONSE_NONE](#) value.

```
// This code example shows all CDC Function Driver possible events
// and a possible scheme for handling these events. In this case
// event responses are not deferred.

uint16_t * breakData;
USB_DEVICE_HANDLE usbDeviceHandle;
USB_CDC_LINE_CODING lineCoding;
USB_CDC_CONTROL_LINE_STATE * controlLineStateData
```

```
USB_DEVICE_CDC_EVENT_RESPONSE USBDeviceCDCEventHandler
(
    USB_DEVICE_CDC_INDEX instanceIndex,
    USB_DEVICE_CDC_EVENT event,
    void * data,
    uintptr_t userData
)
{
    switch(event)
    {
        case USB_DEVICE_CDC_EVENT_SET_LINE_CODING:

            // In this case, the application should read the line coding
            // data that is sent by the host.

            USB_DEVICE_ControlReceive(usbDeviceHandle, &lineCoding,
                                      sizeof(USB_CDC_LINE_CODING));
            break;

        case USB_DEVICE_CDC_EVENT_GET_LINE_CODING:

            // In this case, the application should send the line coding
            // data to the host.

            USB_DEVICE_ControlSend(usbDeviceHandle, &lineCoding,
                                   sizeof(USB_CDC_LINE_CODING));
            break;

        case USB_DEVICE_CDC_EVENT_SET_CONTROL_LINE_STATE:

            // In this case, pData should be interpreted as a
            // USB_CDC_CONTROL_LINE_STATE pointer type. The application
            // acknowledges the parameters by calling the
            // USB_DEVICE_ControlStatus() function with the
            // USB_DEVICE_CONTROL_STATUS_OK option.

            controlLineStateData = (USB_CDC_CONTROL_LINE_STATE *)pData;
            USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);

            break;

        case USB_DEVICE_CDC_EVENT_SEND_BREAK:

            // In this case, pData should be interpreted as a uint16_t
            // pointer type to the break duration. The application
            // acknowledges the parameters by calling the
            // USB_DEVICE_ControlStatus() function with the
            // USB_DEVICE_CONTROL_STATUS_OK option.

            breakDuration = (USB_DEVICE_CDC_EVENT_DATA_SEND_BREAK *)pData;
            USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);

            break;

        case USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_SENT:

            // This event indicates the data send request associated with
            // the latest USB_DEVICE_ControlSend() function was
            // completed. The application could use this event to track
            // the completion of the USB_DEVICE_CDC_EVENT_GET_LINE_CODING
            // request.

            break;
        case USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_RECEIVED:

            // This means that the data stage is complete. The data in
            // setLineCodingData is valid or data in getLineCodingData was
            // sent to the host. The application can now decide whether it
            // supports this data. It is not mandatory to do this in the
    }
}
```

```

// event handler.

USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);

case USB_DEVICE_CDC_EVENT_WRITE_COMPLETE:

    // This means USB_DEVICE_CDC_Write() operation completed.
    // The pData member will point to a
    // USB_DEVICE_CDC_EVENT_DATA_WRITE_COMPLETE type of data.

    break;
case USB_DEVICE_CDC_EVENT_READ_COMPLETE:

    // This means USB_DEVICE_CDC_Read() operation completed.
    // The pData member will point to a
    // USB_DEVICE_CDC_EVENT_DATA_READ_COMPLETE type of data.

    break;

case USB_DEVICE_CDC_EVENT_SERIAL_STATE_NOTIFICATION_COMPLETE:

    // This means USB_DEVICE_CDC_SerialStateNotification() operation
    // completed. The pData member will point to a
    // USB_DEVICE_CDC_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE type of data.

    break;

default:
    break;
}

return (USB_DEVICE_CDC_EVENT_RESPONSE_NONE);
}

```

Refer to the [USB_DEVICE_CDC_EVENT](#) enumeration for more details on each event.

Sending Data

Describes how to send data to the CDC Host.

Description

The application may need to send data or serial state notification to the USB CDC Host. This is done by using the [USB_DEVICE_CDC_Write](#) and [USB_DEVICE_CDC_SerialStateNotificationSend](#) functions, respectively.

Sending Data to the USB Host

The application can send data to the Host by using the [USB_DEVICE_CDC_Write](#) function. This function returns a transfer handle that allows the application to track the write request. The request is completed when the Host has requested the data. The completion of the write transfer is indicated by a [USB_DEVICE_CDC_EVENT_WRITE_COMPLETE](#) event. A write request could fail if the function driver instance transfer queue is full.

The [USB_DEVICE_CDC_Write](#) function also allows the application to send data to the host without ending the transfer. This is done by specifying the [USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_PENDING](#) flag. The application can use this option when the data to be sent is not readily available or when the application is memory constrained. The combination of the transfer flag and the transfer size affects how the function driver sends the data to the host:

- If size is a multiple of maxPacketSize (the IN endpoint size), and the flag is set as [USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE](#), the write function will append a Zero Length Packet (ZLP) to complete the transfer
- If size is a multiple of maxPacketSize, and the flag is set as [USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING](#), the write function will not append a ZLP and therefore, will not complete the transfer
- If size is greater than but not a multiple of maxPacketSize, and the flag is set as [USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE](#), the write function schedules (length/maxPacketSize) packets and one packet for the residual data
- If size is greater than but not a multiple of maxPacketSize, and the flag is set as [USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING](#), the write function returns an error code and sets the transferHandle parameter to [USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID](#)
- If size is less than maxPacketSize, and the flag is set as [USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE](#), the write function schedules one packet
- If size is less than maxPacketSize, and the flag is set as [USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING](#), the write

function returns an error code and sets the transferHandle parameter to `USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID`.
The following code shows a set of examples of various conditions attempting to send data with the `USB_DEVICE_CDC_Write` command.

Example 1

```
// This example assume that the maxPacketSize is 64.
USB_DEVICE_CDC_TRANSFER_HANDLE transferHandle;
USB_DEVICE_CDC_INDEX instance;
USB_DEVICE_CDC_RESULT writeRequestResult;
uint8_t data[34];

// In this example we want to send 34 bytes only.
writeRequestResult = USB_DEVICE_CDC_Write(instance,&transferHandle, data, 34,
                                         USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE);
if(USB_DEVICE_CDC_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}
```

Example 2

```
-----
// In this example we want to send 64 bytes only.
// This will cause a ZLP to be sent.

USB_DEVICE_CDC_TRANSFER_HANDLE transferHandle;
USB_DEVICE_CDC_INDEX instance;
USB_DEVICE_CDC_RESULT writeRequestResult;
uint8_t data[64];

writeRequestResult = USB_DEVICE_CDC_Write(instance,&transferHandle, data, 64,
                                         USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE);
if(USB_DEVICE_CDC_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}
```

Example 3

```
-----
// This example will return an error because size is less
// than maxPacketSize and the flag indicates that more
// data is pending.

USB_DEVICE_CDC_TRANSFER_HANDLE transferHandle;
USB_DEVICE_CDC_INDEX instance;
USB_DEVICE_CDC_RESULT writeRequestResult;
uint8_t data[64];

writeRequestResult = USB_DEVICE_CDC_Write(instance,&transferHandle, data, 32,
                                         USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING);
```

Example 4

```
-----
// In this example we want to place a request for a 70 byte transfer.
// The 70 bytes will be sent out in a 64 byte transaction and a 6 byte
// transaction completing the transfer.
```

```
USB_DEVICE_CDC_TRANSFER_HANDLE transferHandle;
USB_DEVICE_CDC_INDEX instance;
USB_DEVICE_CDC_RESULT writeRequestResult;
uint8_t data[70];

writeRequestResult = USB_DEVICE_CDC_Write(instance,&transferHandle, data, 70,
                                         USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE);

if(USB_DEVICE_CDC_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}
```

Example 5

```
-----
// In this example we want to place a request for a 70 bytes to be sent
```

```

// but that we don't end the transfer as more data is coming. 64 bytes
// of the 70 will be sent out and the USB_DEVICE_CDC_EVENT_WRITE_COMPLETE
// with 64 bytes. This indicates that the extra 6 bytes weren't
// sent because it would cause the end of the transfer. Thus the
// user needs to add these 6 bytes back to the buffer for the next group
// of data that needs to be sent out.

USB_DEVICE_CDC_TRANSFER_HANDLE transferHandle;
USB_DEVICE_CDC_INDEX instance;
USB_DEVICE_CDC_RESULT writeRequestResult;
uint8_t data[70];

writeRequestResult = USB_DEVICE_CDC_Write(instance,&transferHandle, data, 70,
                                         USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING);

if(USB_DEVICE_CDC_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}
// The completion of the write request will be indicated by the
// USB_DEVICE_CDC_EVENT_WRITE_COMPLETE event.

```

Sending a Serial State Notification

The application can send a Serial State Notification by using the `USB_DEVICE_CDC_SerialStateSend` function. This function returns a transfer handle that allows the application to track the read request. The request is completed when the Host has requested the data. The completion of the transfer is indicated by a `USB_DEVICE_CDC_EVENT_SERIAL_STATE_NOTIFICATION_COMPLETE` event. The transfer request could fail if the function driver transfer queue is full. The following code shows an example of how this can be done.

```

USB_DEVICE_CDC_INDEX instanceIndex;
USB_DEVICE_CDC_TRANSFER_HANDLE transferHandle;
USB_DEVICE_CDC_SERIAL_STATE_NOTIFICATION_DATA notificationData;

// This application function could possibly update the notificationData
// data structure.

APP_UpdateNotificationData(&notificationData);

// Now send the updated notification data to the host.

result = USB_DEVICE_CDC_SerialStateDataSend(instanceIndex, &transferHandle,
                                             &notificationData);

if(USB_DEVICE_CDC_RESULT_OK != result)
{
    // Error handling here
}

```

Receiving Data

Describes how the CDC device can read data from the Host.

Description

The application can receive data from the host by using the `USB_DEVICE_CDC_Read` function. This function returns a transfer handle that allows the application to track the read request. The request is completed when the Host sends the required amount or less than required amount of data. The application must make sure that it allocates a buffer size that is at least the size or a multiple of the receive endpoint size. The return value of the function indicates the success of the request. A read request could fail if the function driver transfer queue is full. The completion of the read transfer is indicated by the `USB_DEVICE_CDC_EVENT_READ_COMPLETE` event. The request completes based on the amount of the data that was requested and size of the transaction initiated by the Host:

- If the size parameter is not a multiple of maxPacketSize or is '0', the function returns `USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID` in transferHandle and returns `USB_DEVICE_CDC_RESULT_ERROR_TRANSFER_SIZE_INVALID` as a return value
- If the size parameter is a multiple of maxPacketSize and the Host sends less than maxPacketSize data in any transaction, the transfer completes and the function driver will issue a `USB_DEVICE_CDC_EVENT_READ_COMPLETE` event along with the `USB_DEVICE_CDC_EVENT_READ_COMPLETE_DATA` data structure
- If the size parameter is a multiple of maxPacketSize and the Host sends maxPacketSize amount of data, and total data received does not exceed size, the function driver will wait for the next packet

The following code shows an example of the `USB_DEVICE_CDC_Read` function:

```

// Shows an example of how to read. This assumes that
// driver was opened successfully.

```

```

USB_DEVICE_CDC_TRANSFER_HANDLE transferHandle;
USB_DEVICE_CDC_RESULT readRequestResult;
USB_DEVICE_CDC_HANDLE instanceHandle;

readRequestResult = USB_DEVICE_CDC_Read(instanceHandle,
                                       &transferHandle, data, 128);

if(USB_DEVICE_CDC_RESULT_OK != readRequestResult)
{
    //Do Error handling here
}

// The completion of the read request will be indicated by the
// USB_DEVICE_CDC_EVENT_READ_COMPLETE event.

```

Configuring the Library

Describes how to configure the CDC Function Driver.

Macros

	Name	Description
	USB_DEVICE_CDC_INSTANCES_NUMBER	Specifies the number of CDC instances.
	USB_DEVICE_CDC_QUEUE_DEPTH_COMBINED	Specifies the combined queue size of all CDC instances.

Description

The application designer must specify the following configuration parameters while using the CDC Function Driver. The configuration macros that implement these parameters must be located in the `system_config.h` file in the application project and a compiler include path (to point to the folder that contains this file) should be specified.

USB_DEVICE_CDC_INSTANCES_NUMBER Macro

Specifies the number of CDC instances.

File

[usb_device_cdc_config_template.h](#)

C

```
#define USB_DEVICE_CDC_INSTANCES_NUMBER
```

Description

USB device CDC Maximum Number of instances

This macro defines the number of instances of the CDC Function Driver. For example, if the application needs to implement two instances of the CDC Function Driver (to create two COM ports) on one USB Device, the macro should be set to 2. Note that implementing a USB Device that features multiple CDC interfaces requires appropriate USB configuration descriptors.

Remarks

None.

USB_DEVICE_CDC_QUEUE_DEPTH_COMBINED Macro

Specifies the combined queue size of all CDC instances.

File

[usb_device_cdc_config_template.h](#)

C

```
#define USB_DEVICE_CDC_QUEUE_DEPTH_COMBINED
```

Description

USB device CDC Combined Queue Size

This macro defines the number of entries in all queues in all instances of the CDC function driver. This value can be obtained by adding up the read and write queue sizes of each CDC Function driver instance. In a simple single instance USB CDC device application, that does not require

buffer queuing and serial state notification, the `USB_DEVICE_CDC_QUEUE_DEPTH_COMBINED` macro can be set to 2. Consider a case with two CDC function driver instances, CDC 1 has a read queue size of 2 and write queue size of 3, CDC 2 has a read queue size of 4 and write queue size of 1, this macro should be set to 10 (2 + 3 + 4 + 1).

Remarks

None.

Building the Library

Describes the files to be included in the project while using the CDC Function Driver.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/usb`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
usb_device_cdc.h	This header file should be included in any .c file that accesses the USB Device CDC Function Driver API.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
<code>/src/dynamic/usb_device_cdc.c</code>	This file implements the CDC Data Interface and Communications interface and should be included in the project if the CDC Device function is desired.
<code>/src/dynamic/usb_device_cdc_acm.c</code>	This file implements the CDC-ACM layer and should be included in the project if the CDC Device function is desired.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB CDC Device Library depends on the following modules:

- [USB Device Layer Library](#)

Library Interface

a) Functions

	Name	Description
≡	USB_DEVICE_CDC_EventHandlerSet	This function registers a event handler for the specified CDC function driver instance.
≡	USB_DEVICE_CDC_Read	This function requests a data read from the USB Device CDC Function Driver Layer.
≡	USB_DEVICE_CDC_Write	This function requests a data write to the USB Device CDC Function Driver Layer.
≡	USB_DEVICE_CDC_SerialStateNotificationSend	This function schedules a request to send serial state notification to the host.

b) Data Types and Constants

	Name	Description
	USB_DEVICE_CDC_EVENT	USB Device CDC Function Driver Events

	USB_DEVICE_CDC_EVENT_DATA_READ_COMPLETE	USB Device CDC Function Driver Read and Write Complete Event Data.
	USB_DEVICE_CDC_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE	USB Device CDC Function Driver Read and Write Complete Event Data.
	USB_DEVICE_CDC_EVENT_DATA_WRITE_COMPLETE	USB Device CDC Function Driver Read and Write Complete Event Data.
	USB_DEVICE_CDC_EVENT_HANDLER	USB Device CDC Event Handler Function Pointer Type.
	USB_DEVICE_CDC_EVENT_RESPONSE	USB Device CDC Function Driver Event Callback Response Type
	USB_DEVICE_CDC_EVENT_DATA_SEND_BREAK	USB Device CDC Function Driver Send Break Event Data
	USB_DEVICE_CDC_INDEX	USB Device CDC Function Driver Index
	USB_DEVICE_CDC_INIT	USB Device CDC Function Driver Initialization Data Structure
	USB_DEVICE_CDC_RESULT	USB Device CDC Function Driver USB Device CDC Result enumeration.
	USB_DEVICE_CDC_TRANSFER_FLAGS	USB Device CDC Function Driver Transfer Flags
	USB_DEVICE_CDC_TRANSFER_HANDLE	USB Device CDC Function Driver Transfer Handle Definition.
	USB_DEVICE_CDC_EVENT_RESPONSE_NONE	USB Device CDC Function Driver Event Handler Response Type None.
	USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID	USB Device CDC Function Driver Invalid Transfer Handle Definition.
	USB_DEVICE_CDC_FUNCTION_DRIVER	USB Device CDC Function Driver Function pointer
	USB_DEVICE_CDC_INDEX_0	Use this to specify CDC Function Driver Instance 0
	USB_DEVICE_CDC_INDEX_1	Use this to specify CDC Function Driver Instance 1
	USB_DEVICE_CDC_INDEX_2	Use this to specify CDC Function Driver Instance 2
	USB_DEVICE_CDC_INDEX_3	Use this to specify CDC Function Driver Instance 3
	USB_DEVICE_CDC_INDEX_4	Use this to specify CDC Function Driver Instance 4
	USB_DEVICE_CDC_INDEX_5	Use this to specify CDC Function Driver Instance 5
	USB_DEVICE_CDC_INDEX_6	Use this to specify CDC Function Driver Instance 6
	USB_DEVICE_CDC_INDEX_7	Use this to specify CDC Function Driver Instance 7

Description

This section describes the Application Programming Interface (API) functions of the USB CDC Device Library.

Refer to each section for a detailed description.

a) Functions

[USB_DEVICE_CDC_EventHandlerSet Function](#)

This function registers a event handler for the specified CDC function driver instance.

File

[usb_device_cdc.h](#)

C

```
USB_DEVICE_CDC_RESULT USB_DEVICE_CDC_EventHandlerSet(USB_DEVICE_CDC_INDEX instanceIndex,
USB_DEVICE_CDC_EVENT_HANDLER eventHandler, uintptr_t context);
```

Returns

USB_DEVICE_CDC_RESULT_OK - The operation was successful
 USB_DEVICE_CDC_RESULT_ERROR_INSTANCE_INVALID - The specified instance does not exist
 USB_DEVICE_CDC_RESULT_ERROR_PARAMETER_INVALID - The eventHandler parameter is NULL

Description

This function registers a event handler for the specified CDC function driver instance. This function should be called by the client when it receives a SET CONFIGURATION event from the device layer. A event handler must be registered for function driver to respond to function driver specific commands. If the event handler is not registered, the device layer will stall function driver specific commands and the USB device may not function.

Remarks

None.

Preconditions

This function should be called when the function driver has been initialized as a result of a set configuration.

Example

```
// This code snippet shows an example registering an event handler. Here
// the application specifies the context parameter as a pointer to an
// application object (appObject) that should be associated with this
// instance of the CDC function driver.

// Application states
typedef enum
{
    //Application's state machine's initial state.
    APP_STATE_INIT=0,
    APP_STATE_SERVICE_TASKS,
    APP_STATE_WAIT_FOR_CONFIGURATION,
} APP_STATES;

USB_DEVICE_HANDLE usbDeviceHandle;

APP_STATES gameState;

// Get Line Coding Data
USB_CDC_LINE_CODING getLineCodingData;

// Control Line State
USB_CDC_CONTROL_LINE_STATE controlLineStateData;

// Set Line Coding Data
USB_CDC_LINE_CODING setLineCodingData;

USB_DEVICE_CDC_RESULT result;

USB_DEVICE_CDC_EVENT_RESPONSE APP_USBDeviceCDCEventHandler
(
    USB_DEVICE_CDC_INDEX instanceIndex ,
    USB_DEVICE_CDC_EVENT event ,
    void* pData,
    uintptr_t context
)
{
    // Event Handling comes here

    switch(event)
    {
        case USB_DEVICE_CDC_EVENT_GET_LINE_CODING:
            // This means the host wants to know the current line
            // coding. This is a control transfer request. Use the
            // USB_DEVICE_ControlSend() function to send the data to
            // host.

            USB_DEVICE_ControlSend(usbDeviceHandle,
                &getLineCodingData, sizeof(USB_CDC_LINE_CODING));
    }
}
```

```
break;

case USB_DEVICE_CDC_EVENT_SET_LINE_CODING:
    // This means the host wants to set the line coding.
    // This is a control transfer request. Use the
    // USB_DEVICE_ControlReceive() function to receive the
    // data from the host

    USB_DEVICE_ControlReceive(usbDeviceHandle,
        &setLineCodingData, sizeof(USB_CDC_LINE_CODING));

break;

case USB_DEVICE_CDC_EVENT_SET_CONTROL_LINE_STATE:
    // This means the host is setting the control line state.
    // Read the control line state. We will accept this request
    // for now.
    controlLineStateData.dtr = ((USB_CDC_CONTROL_LINE_STATE *)pData)->dtr;
    controlLineStateData.carrier = ((USB_CDC_CONTROL_LINE_STATE *)pData)->carrier;
    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);

break;

case USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA RECEIVED:
    // The data stage of the last control transfer is
    // complete. For now we accept all the data

    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);

break;

case USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_SENT:
    // This means the GET LINE CODING function data is valid. We dont
    // do much with this data in this demo.
break;

case USB_DEVICE_CDC_EVENT_SEND_BREAK:
    // This means that the host is requesting that a break of the
    // specified duration be sent.
    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);

break;

case USB_DEVICE_CDC_EVENT_READ_COMPLETE:
    // This means that the host has sent some data
    break;

case USB_DEVICE_CDC_EVENT_WRITE_COMPLETE:
    // This means that the host has sent some data
    break;

default:
    break;
}

return USB_DEVICE_CDC_EVENT_RESPONSE_NONE;
}

// This is the application device layer event handler function.

USB_DEVICE_EVENT_RESPONSE APP_USBDeviceEventHandler(
    USB_DEVICE_EVENT event,
    void * pData,
```

```
    uintptr_t context
)
{
    USB_SETUP_PACKET * setupPacket;
    switch(event)
    {
        case USB_DEVICE_EVENT_POWER_DETECTED:
            // This event is generated when VBUS is detected. Attach the device
            USB_DEVICE_Attach(usbDeviceHandle);
            break;

        case USB_DEVICE_EVENT_POWER_REMOVED:
            // This event is generated when VBUS is removed. Detach the device
            USB_DEVICE_Detach (usbDeviceHandle);
            break;

        case USB_DEVICE_EVENT_CONFIGURED:
            // This event indicates that Host has set Configuration in the Device.
            // Register CDC Function driver Event Handler.
            USB_DEVICE_CDC_EventHandlerSet(USB_DEVICE_CDC_INDEX_0, APP_USBDeviceCDCEventHandler,
(uintptr_t)0);
            break;

        case USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST:
            // This event indicates a Control transfer setup stage has been completed.
            setupPacket = (USB_SETUP_PACKET *)pData;

            // Parse the setup packet and respond with a USB_DEVICE_ControlSend(),
            // USB_DEVICE_ControlReceive or USB_DEVICE_ControlStatus().

            break;

        case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT:
            // This event indicates that a Control transfer Data has been sent to Host.
            break;

        case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA RECEIVED:
            // This event indicates that a Control transfer Data has been received from Host.
            break;

        case USB_DEVICE_EVENT_CONTROL_TRANSFER_ABORTED:
            // This event indicates a control transfer was aborted.
            break;

        case USB_DEVICE_EVENT_SUSPENDED:
            break;

        case USB_DEVICE_EVENT_RESUMED:
            break;

        case USB_DEVICE_EVENT_ERROR:
            break;

        case USB_DEVICE_EVENT_RESET:
            break;

        case USB_DEVICE_EVENT_SOF:
            // This event indicates an SOF is detected on the bus. The USB_DEVICE_SOF_EVENT_ENABLE
            // macro should be defined to get this event.
            break;
        default:
            break;
    }
}

void APP_Tasks ( void )
{
    // Check the application's current state.
```

```

switch ( AppState )
{
    // Application's initial state.
    case APP_STATE_INIT:
        // Open the device layer
        usbDeviceHandle = USB_DEVICE_Open( USB_DEVICE_INDEX_0,
                                         DRV_IO_INTENT_READWRITE );

        if(usbDeviceHandle != USB_DEVICE_HANDLE_INVALID)
        {
            // Register a callback with device layer to get event notification
            USB_DEVICE_EventHandlerSet(usbDeviceHandle,
                                         APP_USBDeviceEventHandler, 0);
            AppState = APP_STATE_WAIT_FOR_CONFIGURATION;
        }
        else
        {
            // The Device Layer is not ready to be opened. We should try
            // gain later.
        }
        break;

    case APP_STATE_SERVICE_TASKS:
        break;

        // The default state should never be executed.
    default:
        break;
}
}

```

Parameters

Parameters	Description
instance	Instance of the CDC Function Driver.
eventHandler	A pointer to event handler function.
context	Application specific context that is returned in the event handler.

Function

```

USB_DEVICE_CDC_RESULT USB_DEVICE_CDC_EventHandlerSet
(
    USB_DEVICE_CDC_INDEX instance
    USB_DEVICE_CDC_EVENT_HANDLER eventHandler
    uintptr_t context
);

```

USB_DEVICE_CDC_Read Function

This function requests a data read from the USB Device CDC Function Driver Layer.

File

[usb_device_cdc.h](#)

C

```

USB_DEVICE_CDC_RESULT USB_DEVICE_CDC_Read(USB_DEVICE_CDC_INDEX instanceIndex,
                                         USB_DEVICE_CDC_TRANSFER_HANDLE * transferHandle, void * data, size_t size);

```

Returns

USB_DEVICE_CDC_RESULT_OK - The read request was successful. transferHandle contains a valid transfer handle.
 USB_DEVICE_CDC_RESULT_ERROR_TRANSFER_QUEUE_FULL - internal request queue is full. The write request could not be added.
 USB_DEVICE_CDC_RESULT_ERROR_TRANSFER_SIZE_INVALID - The specified transfer size was not a multiple of endpoint size or is 0.
 USB_DEVICE_CDC_RESULT_ERROR_INSTANCE_NOT_CONFIGURED - The specified instance is not configured yet.
 USB_DEVICE_CDC_RESULT_ERROR_INSTANCE_INVALID - The specified instance was not provisioned in the application and is invalid.

Description

This function requests a data read from the USB Device CDC Function Driver Layer. The function places a request with driver, the request will get serviced as data is made available by the USB Host. A handle to the request is returned in the transferHandle parameter. The termination of the request is indicated by the `USB_DEVICE_CDC_EVENT_READ_COMPLETE` event. The amount of data read and the transfer handle associated with the request is returned along with the event in the pData parameter of the event handler. The transfer handle expires when event handler for the `USB_DEVICE_CDC_EVENT_READ_COMPLETE` exits. If the read request could not be accepted, the function returns an error code and transferHandle will contain the value `USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID`.

If the size parameter is not a multiple of maxPacketSize or is 0, the function returns `USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID` in transferHandle and returns an error code as a return value. If the size parameter is a multiple of maxPacketSize and the host send less than maxPacketSize data in any transaction, the transfer completes and the function driver will issue a `USB_DEVICE_CDC_EVENT_READ_COMPLETE` event along with the `USB_DEVICE_CDC_EVENT_READ_COMPLETE_DATA` data structure. If the size parameter is a multiple of maxPacketSize and the host sends maxPacketSize amount of data, and total data received does not exceed size, then the function driver will wait for the next packet.

Remarks

While the using the CDC Function Driver with the PIC32MZ USB module, the receive buffer provided to the `USB_DEVICE_CDC_Read` function should be placed in coherent memory and aligned at a 16 byte boundary. This can be done by declaring the buffer using the `_attribute_((coherent, aligned(16)))` attribute. An example is shown here

```
uint8_t data[256] __attribute__((coherent, aligned(16)));
```

Preconditions

The function driver should have been configured.

Example

```
// Shows an example of how to read. This assumes that
// driver was opened successfully.

USB_DEVICE_CDC_TRANSFER_HANDLE transferHandle;
USB_DEVICE_CDC_RESULT readRequestResult;
USB_DEVICE_CDC_HANDLE instanceHandle;

readRequestResult = USB_DEVICE_CDC_Read(instanceHandle,
                                         &transferHandle, data, 128);

if(USB_DEVICE_CDC_RESULT_OK != readRequestResult)
{
    //Do Error handling here
}

// The completion of the read request will be indicated by the
// USB_DEVICE_CDC_EVENT_READ_COMPLETE event.
```

Parameters

Parameters	Description
instance	USB Device CDC Function Driver instance.
transferHandle	Pointer to a <code>USB_DEVICE_CDC_TRANSFER_HANDLE</code> type of variable. This variable will contain the transfer handle in case the read request was successful.
data	pointer to the data buffer where read data will be stored.
size	Size of the data buffer. Refer to the description section for more details on how the size affects the transfer.

Function

```
USB_DEVICE_CDC_RESULT USB_DEVICE_CDC_Read
(
    USB_DEVICE_CDC_INDEX instance,
    USB_CDC_DEVICE_TRANSFER_HANDLE * transferHandle,
    void * data,
    size_t size
);
```

USB_DEVICE_CDC_Write Function

This function requests a data write to the USB Device CDC Function Driver Layer.

File

[usb_device_cdc.h](#)

C

```
USB_DEVICE_CDC_RESULT USB_DEVICE_CDC_Write(USB_DEVICE_CDC_INDEX instanceIndex,
USB_DEVICE_CDC_TRANSFER_HANDLE * transferHandle, const void * data, size_t size,
USB_DEVICE_CDC_TRANSFER_FLAGS flags);
```

Returns

USB_DEVICE_CDC_RESULT_OK - The write request was successful. transferHandle contains a valid transfer handle.

USB_DEVICE_CDC_RESULT_ERROR_TRANSFER_QUEUE_FULL - internal request queue is full. The write request could not be added.

USB_DEVICE_CDC_RESULT_ERROR_TRANSFER_SIZE_INVALID - The specified transfer size and flag parameter are invalid.

USB_DEVICE_CDC_RESULT_ERROR_INSTANCE_NOT_CONFIGURED - The specified instance is not configured yet.

USB_DEVICE_CDC_RESULT_ERROR_INSTANCE_INVALID - The specified instance was not provisioned in the application and is invalid.

Description

This function requests a data write to the USB Device CDC Function Driver Layer. The function places a request with the driver, the request will get serviced as data is requested by the USB Host. A handle to the request is returned in the transferHandle parameter. The termination of the request is indicated by the USB_DEVICE_CDC_EVENT_WRITE_COMPLETE event. The amount of data written and the transfer handle associated with the request is returned along with the event in writeCompleteData member of the pData parameter in the event handler. The transfer handle expires when the event handler for the USB_DEVICE_CDC_EVENT_WRITE_COMPLETE exits. If the read request could not be accepted, the function returns an error code and transferHandle will contain the value [USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID](#).

The behavior of the write request depends on the flags and size parameter. If the application intends to send more data in a request, then it should use the USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING flag. If there is no more data to be sent in the request, the application must use the USB_DEVICE_CDC_EVENT_WRITE_COMPLETE flag. This is explained in more detail here:

- If size is a multiple of maxPacketSize and flag is set as

[USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE](#), the write function will append a Zero Length Packet (ZLP) to complete the transfer.

- If size is a multiple of maxPacketSize and flag is set as

[USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING](#), the write function will not append a ZLP and hence will not complete the transfer.

- If size is greater than but not a multiple of maxPacketSize and flags is

set as [USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE](#), the write function returns an error code and sets the transferHandle parameter to [USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID](#).

- If size is greater than but not a multiple of maxPacketSize and flags is

set as [USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING](#), the write function fails and return an error code and sets the transferHandle parameter to [USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID](#).

- If size is less than maxPacketSize and flag is set as

[USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE](#), the write function schedules one packet.

- If size is less than maxPacketSize and flag is set as

[USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING](#), the write function returns an error code and sets the transferHandle parameter to [USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID](#).

- If size is 0 and the flag is set

[USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE](#), the function driver will schedule a Zero Length Packet.

Completion of the write transfer is indicated by the [USB_DEVICE_CDC_EVENT_WRITE_COMPLETE](#) event. The amount of data written along with the transfer handle is returned along with the event.

Remarks

While using the CDC Function Driver with the PIC32MZ USB module, the transmit buffer provided to the USB_DEVICE_CDC_Write function should be placed in coherent memory and aligned at a 16 byte boundary. This can be done by declaring the buffer using the __attribute__((coherent, aligned(16))) attribute. An example is shown here

```
uint8_t data[256] __attribute__((coherent, aligned(16)));
```

Preconditions

The function driver should have been configured.

Example

```
// Below is a set of examples showing various conditions trying to
// send data with the Write command.
//
// This assumes that driver was opened successfully.
// Assume maxPacketSize is 64.

USB_DEVICE_CDC_TRANSFER_HANDLE transferHandle;
USB_DEVICE_CDC_RESULT writeRequestHandle;
USB_DEVICE_CDC_INDEX instance;

//-----
// In this example we want to send 34 bytes only.

writeRequestResult = USB_DEVICE_CDC_Write(instance,
                                         &transferHandle, data, 34,
                                         USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE);

if(USB_DEVICE_CDC_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}

//-----
// In this example we want to send 64 bytes only.
// This will cause a ZLP to be sent.

writeRequestResult = USB_DEVICE_CDC_Write(instance,
                                         &transferHandle, data, 64,
                                         USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE);

if(USB_DEVICE_CDC_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}

//-----
// This example will return an error because size is less
// than maxPacketSize and the flag indicates that more
// data is pending.

writeRequestResult = USB_DEVICE_CDC_Write(instanceHandle,
                                         &transferHandle, data, 32,
                                         USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING);

//-----
// In this example we want to place a request for a 70 byte transfer.
// The 70 bytes will be sent out in a 64 byte transaction and a 6 byte
// transaction completing the transfer.

writeRequestResult = USB_DEVICE_CDC_Write(instanceHandle,
                                         &transferHandle, data, 70,
                                         USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE);

if(USB_DEVICE_CDC_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}
```

```

//-----
// In this example we want to place a request for a 70 bytes and the flag
// is set to data pending. This will result in an error. The size of data
// when the data pending flag is specified should be a multiple of the
// endpoint size.

writeRequestResult = USB_DEVICE_CDC_Write(instanceHandle,
                                         &transferHandle, data, 70,
                                         USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING);

if(USB_DEVICE_CDC_RESULT_OK != writeRequestResult)
{
    //Do Error handling here
}

// The completion of the write request will be indicated by the
// USB_DEVICE_CDC_EVENT_WRITE_COMPLETE event.

```

Parameters

Parameters	Description
instance	USB Device CDC Function Driver instance.
transferHandle	Pointer to a USB_DEVICE_CDC_TRANSFER_HANDLE type of variable. This variable will contain the transfer handle in case the write request was successful.
data	pointer to the data buffer that contains the data to written.
size	Size of the data buffer. Refer to the description section for more details on how the size affects the transfer.
flags	Flags that indicate whether the transfer should continue or end. Refer to the description for more details.

Function

```

USB\_DEVICE\_CDC\_RESULT USB_DEVICE_CDC_Write
(
    USB\_DEVICE\_CDC\_INDEX instance,
    USB\_CDC\_DEVICE\_TRANSFER\_HANDLE * transferHandle,
    const void * data,
    size_t size,
    USB\_DEVICE\_CDC\_TRANSFER\_FLAGS flags
);

```

[USB_DEVICE_CDC_SerialStateNotificationSend](#) Function

This function schedules a request to send serial state notification to the host.

File

[usb_device_cdc.h](#)

C

```

USB\_DEVICE\_CDC\_RESULT USB\_DEVICE\_CDC\_SerialStateNotificationSend(USB\_DEVICE\_CDC\_INDEX instanceIndex,
USB\_DEVICE\_CDC\_TRANSFER\_HANDLE * transferHandle, USB\_CDC\_SERIAL\_STATE * notificationData);

```

Returns

[USB_DEVICE_CDC_RESULT_OK](#) - The request was successful. transferHandle contains a valid transfer handle.
[USB_DEVICE_CDC_RESULT_ERROR_TRANSFER_QUEUE_FULL](#) - Internal request queue is full. The request could not be added.
[USB_DEVICE_CDC_RESULT_ERROR_INSTANCE_NOT_CONFIGURED](#) - The specified instance is not configured yet.
[USB_DEVICE_CDC_RESULT_ERROR_INSTANCE_INVALID](#) - The specified instance was not provisioned in the application and is invalid.

Description

This function places a request to send serial state notification data to the host. The function will place the request with the driver, the request will get serviced when the data is requested by the USB host. A handle to the request is returned in the transferHandle parameter. The termination of the request is indicated by the [USB_DEVICE_CDC_EVENT_SERIAL_STATE_NOTIFICATION_COMPLETE](#) event. The amount of data transmitted and the transfer handle associated with the request is returned along with the event in the serialStateNotificationCompleteData

member of pData parameter of the event handler. The transfer handle expires when the event handler for the USB_DEVICE_CDC_EVENT_SERIAL_STATE_NOTIFICATION_COMPLETE event exits. If the send request could not be accepted, the function returns an error code and transferHandle will contain the value [USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID](#).

Remarks

While the using the CDC Function Driver with the PIC32MZ USB module, the notification data buffer provided to the `USB_DEVICE_CDC_SerialStateNotificationSend` function should be placed in coherent memory and aligned at a 16 byte boundary. This can be done by declaring the buffer using the `__attribute__((coherent, aligned(16)))` attribute. An example is shown here

```
uint8_t data[256] __attribute__((coherent, aligned(16)));
```

Preconditions

The function driver should have been configured

Example

```
USB_CDC_SERIAL_STATE notificationData;

// This application function could possibly update the notificationData
// data structure.

APP_UpdateNotificationData(&notificationData);

// Now send the updated notification data to the host.

result = USB_DEVICE_CDC_SerialStateNotificationSend
    (instanceIndex, &transferHandle, &notificationData);

if(USB_DEVICE_CDC_RESULT_OK != result)
{
    // Error handling here. The transferHandle will contain
    // USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID in this case.
}
```

Parameters

Parameters	Description
instance	USB Device CDC Function Driver instance.
transferHandle	Pointer to a output only variable that will contain transfer handle.
notificationData	USB_DEVICE_CDC_SERIAL_STATE_NOTIFICATION type of notification data to be sent to the host.

Function

```
USB_DEVICE_CDC_RESULT USB_DEVICE_CDC_SerialStateNotificationSend
(
    USB_DEVICE_CDC_INDEX instanceIndex,
    USB_DEVICE_CDC_TRANSFER_HANDLE * transferHandle,
    USB_CDC_SERIAL_STATE * notificationData
);
```

b) Data Types and Constants

USB_DEVICE_CDC_EVENT Enumeration

USB Device CDC Function Driver Events

File

[usb_device_cdc.h](#)

C

```
typedef enum {
    USB_DEVICE_CDC_EVENT_SET_LINE_CODING,
    USB_DEVICE_CDC_EVENT_GET_LINE_CODING,
    USB_DEVICE_CDC_EVENT_SET_CONTROL_LINE_STATE,
```

```

USB_DEVICE_CDC_EVENT_SEND_BREAK,
USB_DEVICE_CDC_EVENT_WRITE_COMPLETE,
USB_DEVICE_CDC_EVENT_READ_COMPLETE,
USB_DEVICE_CDC_EVENT_SERIAL_STATE_NOTIFICATION_COMPLETE,
USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_SENT,
USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_RECEIVED,
USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_ABORTED
} USB_DEVICE_CDC_EVENT;

```

Members

Members	Description
USB_DEVICE_CDC_EVENT_SET_LINE_CODING	This event occurs when the host issues a SET LINE CODING command. The application must provide a USB_CDC_LINE_CODING data structure to the device layer to receive the line coding data that the host will provide. The application must provide the buffer by calling the USB_DEVICE_ControlReceive function either in the event handler or in the application, after returning from the event handler function. The pData parameter will be NULL. The application can use the USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA RECEIVED event to track completion of the command.
USB_DEVICE_CDC_EVENT_GET_LINE_CODING	This event occurs when the host issues a GET LINE CODING command. The application must provide a USB_CDC_LINE_CODING data structure to the device layer that contains the line coding data to be provided to the host. The application must provide the buffer by calling the USB_DEVICE_ControlSend function either in the event handler or in the application, after returning from the event handler function. The application can use the USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA SENT event to track completion of the command.
USB_DEVICE_CDC_EVENT_SET_CONTROL_LINE_STATE	This event occurs when the host issues a SET CONTROL LINE STATE command. The application must interpret the pData parameter as USB_CDC_CONTROL_LINE_STATE pointer type. This data structure contains the control line state data. The application can then use the USB_DEVICE_ControlStatus function to indicate acceptance or rejection of the command. The USB_DEVICE_ControlStatus function can be called from the event handler or in the application, after returning from the event handler.
USB_DEVICE_CDC_EVENT_SEND_BREAK	This event occurs when the host issues a SEND BREAK command. The application must interpret the pData parameter as a USB_DEVICE_EVENT_DATA_SEND_BREAK pointer type. This data structure contains the break duration data. The application can then use the USB_DEVICE_ControlStatus function to indicate acceptance or rejection of the command. The USB_DEVICE_ControlStatus function can be called from the event handler or in the application, after returning from the event handler.
USB_DEVICE_CDC_EVENT_WRITE_COMPLETE	This event occurs when a write operation scheduled by calling the USB_DEVICE_CDC_Write function has completed. The pData parameter should be interpreted as a USB_DEVICE_EVENT_DATA_WRITE_COMPLETE pointer type. This will contain the transfer handle associated with the completed write transfer and the amount of data written.
USB_DEVICE_CDC_EVENT_READ_COMPLETE	This event occurs when a read operation scheduled by calling the USB_DEVICE_CDC_Read function has completed. The pData parameter should be interpreted as a USB_DEVICE_EVENT_DATA_READ_COMPLETE pointer type. This will contain the transfer handle associated with the completed read transfer and the amount of data read.
USB_DEVICE_CDC_EVENT_SERIAL_STATE_NOTIFICATION_COMPLETE	This event occurs when a serial state notification scheduled using the USB_DEVICE_CDC_SerialStateNotificationSend function, was sent to the host. The pData parameter should be interpreted as a USB_DEVICE_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE pointer type and will contain the transfer handle associated with the completed send transfer and the amount of data sent.

USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_SENT	This event occurs when the data stage of a control read transfer has completed. This event would occur after the application uses the USB_DEVICE_ControlSend function to respond to the USB_DEVICE_CDC_EVENT_GET_LINE_CODING event.
USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_RECEIVED	This event occurs when the data stage of a control write transfer has completed. This would occur after the application would respond with a USB_DEVICE_ControlReceive function to the USB_DEVICE_CDC_EVENT_SET_LINE_CODING_EVENT and the data has been received. The application should respond to this event by calling the USB_DEVICE_ControlStatus function with the USB_DEVICE_CONTROL_STATUS_OK flag to acknowledge the received data or the USB_DEVICE_CONTROL_STATUS_ERROR flag to reject it and stall the control transfer
USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_ABORTED	This event occurs when a control transfer that this instance of CDC function driver responded to was aborted by the host. The application can use this event to reset its CDC function driver related control transfer state machine

Description

USB Device CDC Function Driver Events

These events are specific to the USB Device CDC Function Driver instance. Each event description contains details about the parameters passed with event. The contents of pData depends on the generated event.

Events associated with the CDC Function Driver Specific Control Transfers require application response. The application should respond to these events by using the [USB_DEVICE_ControlReceive](#), [USB_DEVICE_ControlSend](#) and [USB_DEVICE_ControlStatus](#) functions.

Calling the [USB_DEVICE_ControlStatus](#) function with a [USB_DEVICE_CONTROL_STATUS_ERROR](#) will stall the control transfer request. The application would do this if the control transfer request is not supported. Calling the [USB_DEVICE_ControlStatus](#) function with a [USB_DEVICE_CONTROL_STATUS_OK](#) will complete the status stage of the control transfer request. The application would do this if the control transfer request is supported

The following code snippet shows an example of a possible event handling scheme.

```
// This code example shows all CDC Function Driver events
// and a possible scheme for handling these events. In this example
// event responses are not deferred. usbDeviceHandle is obtained while
// opening the USB Device Layer through the USB_DEVICE_Open function.
```

```
uint16_t * breakData;
USB_DEVICE_HANDLE usbDeviceHandle;
USB_CDC_LINE_CODING lineCoding;
USB_CDC_CONTROL_LINE_STATE * controlLineStateData;

USB_DEVICE_EVENT_RESPONSE USBDeviceCDCEventHandler
(
    USB_DEVICE_CDC_INDEX instanceIndex,
    USB_DEVICE_CDC_EVENT event,
    void * pData,
    uintptr_t userData
)
{
    switch(event)
    {
        case USB_DEVICE_CDC_EVENT_SET_LINE_CODING:

            // In this case, the application should read the line coding
            // data that is sent by the host. The application must use the
            // USB_DEVICE_ControlReceive function to receive the
            // USB_CDC_LINE_CODING type of data.

            USB_DEVICE_ControlReceive(usbDeviceHandle, &lineCoding, sizeof(USB_CDC_LINE_CODING));
            break;

        case USB_DEVICE_CDC_EVENT_GET_LINE_CODING:

            // In this case, the application should send the line coding
            // data to the host. The application must send the
            // USB_DEVICE_ControlSend function to send the data.

            USB_DEVICE_ControlSend(usbDeviceHandle, &lineCoding, sizeof(USB_CDC_LINE_CODING));
            break;
    }
}
```

```
case USB_DEVICE_CDC_EVENT_SET_CONTROL_LINE_STATE:  
  
    // In this case, pData should be interpreted as a  
    // USB_CDC_CONTROL_LINE_STATE pointer type. The application  
    // acknowledges the parameters by calling the  
    // USB_DEVICE_ControlStatus function with the  
    // USB_DEVICE_CONTROL_STATUS_OK option.  
  
    controlLineStateData = (USB_CDC_CONTROL_LINE_STATE *)pData;  
    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);  
    break;  
  
case USB_DEVICE_CDC_EVENT_SEND_BREAK:  
  
    // In this case, pData should be interpreted as a uint16_t  
    // pointer type to the break duration. The application  
    // acknowledges the parameters by calling the  
    // USB_DEVICE_ControlStatus() function with the  
    // USB_DEVICE_CONTROL_STATUS_OK option.  
  
    breakDuration = (uint16_t *)pData;  
    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);  
    break;  
  
case USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_SENT:  
  
    // This event indicates the data send request associated with  
    // the latest USB_DEVICE_ControlSend function was  
    // completed. The application could use this event to track  
    // the completion of the USB_DEVICE_CDC_EVENT_GET_LINE_CODING  
    // request.  
  
    break;  
  
case USB_DEVICE_CDC_EVENT_CONTROL_TRANSFER_DATA_RECEIVED:  
  
    // This event indicates the data that was requested using the  
    // USB_DEVICE_ControlReceive function is available for the  
    // application to peruse. The application could use this event  
    // to track the completion of the  
    // USB_DEVICE_CDC_EVENT_SET_LINE_CODING_EVENT event. The  
    // application can then either accept the line coding data (as  
    // shown here) or decline it by using the  
    // USB_DEVICE_CONTROL_STATUS_ERROR flag in the  
    // USB_DEVICE_ControlStatus function.  
  
    USB_DEVICE_ControlStatus(usbDeviceHandle, USB_DEVICE_CONTROL_STATUS_OK);  
    break;  
  
case USB_DEVICE_CDC_EVENT_WRITE_COMPLETE:  
  
    // This event indicates that a CDC Write Transfer request has  
    // completed. pData should be interpreted as a  
    // USB_DEVICE_CDC_EVENT_DATA_WRITE_COMPLETE pointer type. This  
    // contains the transfer handle of the write transfer that  
    // completed and amount of data that was written.  
  
    break;  
  
case USB_DEVICE_CDC_EVENT_READ_COMPLETE:  
  
    // This event indicates that a CDC Read Transfer request has  
    // completed. pData should be interpreted as a  
    // USB_DEVICE_CDC_EVENT_DATA_READ_COMPLETE pointer type. This  
    // contains the transfer handle of the read transfer that  
    // completed and amount of data that was written.  
  
    break;
```

```

case USB_DEVICE_CDC_EVENT_SERIAL_STATE_NOTIFICATION_COMPLETE:

    // This event indicates that a CDC Serial State Notification
    // Send request has completed. pData should be interpreted as a
    // USB_DEVICE_CDC_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE
    // pointer type. This will contain the transfer handle
    // associated with the send request and the amount of data that
    // was sent.

    break

default:
    break;
}

return(USB_DEVICE_CDC_EVENT_RESPONSE_NONE);
}

```

Remarks

The USB Device CDC control transfer related events allow the application to defer responses. This allows the application some time to obtain the response data rather than having to respond to the event immediately. Note that a USB host will typically wait for event response for a finite time duration before timing out and canceling the event and associated transactions. Even when deferring response, the application must respond promptly if such time outs have to be avoided.

USB_DEVICE_CDC_EVENT_DATA_READ_COMPLETE Structure

USB Device CDC Function Driver Read and Write Complete Event Data.

File

[usb_device_cdc.h](#)

C

```

typedef struct {
    USB_DEVICE_CDC_TRANSFER_HANDLE handle;
    size_t length;
    USB_DEVICE_CDC_RESULT status;
} USB_DEVICE_CDC_EVENT_DATA_WRITE_COMPLETE, USB_DEVICE_CDC_EVENT_DATA_READ_COMPLETE,
USB_DEVICE_CDC_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE;

```

Members

Members	Description
USB_DEVICE_CDC_TRANSFER_HANDLE handle;	Transfer handle associated with this <ul style="list-style-type: none"> • read or write request
size_t length;	Indicates the amount of data (in bytes) that was <ul style="list-style-type: none"> • read or written
USB_DEVICE_CDC_RESULT status;	Completion status of the transfer

Description

USB Device CDC Function Driver Read and Write Complete Event Data.

This data type defines the data structure returned by the driver along with USB_DEVICE_CDC_EVENT_READ_COMPLETE and USB_DEVICE_CDC_EVENT_WRITE_COMPLETE events.

Remarks

None.

USB_DEVICE_CDC_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE Structure

USB Device CDC Function Driver Read and Write Complete Event Data.

File

[usb_device_cdc.h](#)

C

```
typedef struct {
    USB_DEVICE_CDC_TRANSFER_HANDLE handle;
    size_t length;
    USB_DEVICE_CDC_RESULT status;
} USB_DEVICE_CDC_EVENT_DATA_WRITE_COMPLETE, USB_DEVICE_CDC_EVENT_DATA_READ_COMPLETE,
USB_DEVICE_CDC_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE;
```

Members

Members	Description
USB_DEVICE_CDC_TRANSFER_HANDLE handle;	Transfer handle associated with this <ul style="list-style-type: none"> • read or write request
size_t length;	Indicates the amount of data (in bytes) that was <ul style="list-style-type: none"> • read or written
USB_DEVICE_CDC_RESULT status;	Completion status of the transfer

Description

USB Device CDC Function Driver Read and Write Complete Event Data.

This data type defines the data structure returned by the driver along with USB_DEVICE_CDC_EVENT_READ_COMPLETE and USB_DEVICE_CDC_EVENT_WRITE_COMPLETE events.

Remarks

None.

USB_DEVICE_CDC_EVENT_DATA_WRITE_COMPLETE Structure

USB Device CDC Function Driver Read and Write Complete Event Data.

File

[usb_device_cdc.h](#)

C

```
typedef struct {
    USB_DEVICE_CDC_TRANSFER_HANDLE handle;
    size_t length;
    USB_DEVICE_CDC_RESULT status;
} USB_DEVICE_CDC_EVENT_DATA_WRITE_COMPLETE, USB_DEVICE_CDC_EVENT_DATA_READ_COMPLETE,
USB_DEVICE_CDC_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE;
```

Members

Members	Description
USB_DEVICE_CDC_TRANSFER_HANDLE handle;	Transfer handle associated with this <ul style="list-style-type: none"> • read or write request
size_t length;	Indicates the amount of data (in bytes) that was <ul style="list-style-type: none"> • read or written
USB_DEVICE_CDC_RESULT status;	Completion status of the transfer

Description

USB Device CDC Function Driver Read and Write Complete Event Data.

This data type defines the data structure returned by the driver along with USB_DEVICE_CDC_EVENT_READ_COMPLETE and USB_DEVICE_CDC_EVENT_WRITE_COMPLETE events.

Remarks

None.

USB_DEVICE_CDC_EVENT_HANDLER Type

USB Device CDC Event Handler Function Pointer Type.

File

[usb_device_cdc.h](#)

C

```
typedef USB_DEVICE_CDC_EVENT_RESPONSE (* USB_DEVICE_CDC_EVENT_HANDLER)(USB_DEVICE_CDC_INDEX instanceIndex,  
USB_DEVICE_CDC_EVENT event, void * pData, uintptr_t context);
```

Description

USB Device CDC Event Handler Function Pointer Type.

This data type defines the required function signature of the USB Device CDC Function Driver event handling callback function. The application must register a pointer to a CDC Function Driver events handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event call backs from the CDC Function Driver. The function driver will invoke this function with event relevant parameters. The description of the event handler function parameters is given here.

instanceIndex - Instance index of the CDC Function Driver that generated the event.

event - Type of event generated.

pData - This parameter should be type cast to an event specific pointer type based on the event that has occurred. Refer to the [USB_DEVICE_CDC_EVENT](#) enumeration description for more details.

context - Value identifying the context of the application that was registered along with the event handling function.

Remarks

The event handler function executes in the USB interrupt context when the USB Device Stack is configured for interrupt based operation. It is not advisable to call blocking functions or computationally intensive functions in the event handler. Where the response to a control transfer related event requires extended processing, the response to the control transfer should be deferred and the event handler should be allowed to complete execution.

USB_DEVICE_CDC_EVENT_RESPONSE Type

USB Device CDC Function Driver Event Callback Response Type

File

[usb_device_cdc.h](#)

C

```
typedef void USB_DEVICE_CDC_EVENT_RESPONSE;
```

Description

USB Device CDC Function Driver Event Handler Response Type

This is the return type of the CDC Function Driver event handler.

Remarks

None.

USB_DEVICE_CDC_EVENT_DATA_SEND_BREAK Structure

USB Device CDC Function Driver Send Break Event Data

File

[usb_device_cdc.h](#)

C

```
typedef struct {  
    uint16_t breakDuration;  
} USB_DEVICE_CDC_EVENT_DATA_SEND_BREAK;
```

Members

Members	Description
uint16_t breakDuration;	Duration of break signal

Description

USB Device CDC Function Driver Send Break Event Data

This data type defines the data structure returned by the driver along with USB_DEVICE_CDC_EVENT_SEND_BREAK event.

Remarks

None.

USB_DEVICE_CDC_INDEX Type

USB Device CDC Function Driver Index

File

[usb_device_cdc.h](#)

C

```
typedef uintptr_t USB_DEVICE_CDC_INDEX;
```

Description

USB Device CDC Function Driver Index

This uniquely identifies a CDC Function Driver instance.

Remarks

None.

USB_DEVICE_CDC_INIT Structure

USB Device CDC Function Driver Initialization Data Structure

File

[usb_device_cdc.h](#)

C

```
typedef struct {
    size_t queueSizeRead;
    size_t queueSizeWrite;
    size_t queueSizeSerialStateNotification;
} USB_DEVICE_CDC_INIT;
```

Members

Members	Description
size_t queueSizeRead;	Size of the read queue for this instance <ul style="list-style-type: none"> of the CDC function driver
size_t queueSizeWrite;	Size of the write queue for this instance <ul style="list-style-type: none"> of the CDC function driver
size_t queueSizeSerialStateNotification;	Size of the serial state notification <ul style="list-style-type: none"> queue size

Description

USB Device CDC Function Driver Initialization Data Structure

This data structure must be defined for every instance of the CDC function driver. It is passed to the CDC function driver, by the Device Layer, at the time of initialization. The funcDriverInit member of the Device Layer Function Driver registration table entry must point to this data structure for an instance of the CDC function driver.

Remarks

The queue sizes that are specified in this data structure are also affected by the [USB_DEVICE_CDC_QUEUE_DEPTH_COMBINED](#) configuration macro.

USB_DEVICE_CDC_RESULT Enumeration

USB Device CDC Function Driver USB Device CDC Result enumeration.

File

[usb_device_cdc.h](#)

C

```
typedef enum {
    USB_DEVICE_CDC_RESULT_OK,
    USB_DEVICE_CDC_RESULT_ERROR_TRANSFER_SIZE_INVALID,
    USB_DEVICE_CDC_RESULT_ERROR_TRANSFER_QUEUE_FULL,
```

```

USB_DEVICE_CDC_RESULT_ERROR_INSTANCE_INVALID,
USB_DEVICE_CDC_RESULT_ERROR_INSTANCE_NOT_CONFIGURED,
USB_DEVICE_CDC_RESULT_ERROR_PARAMETER_INVALID,
USB_DEVICE_CDC_RESULT_ERROR_ENDPOINT_HALTED,
USB_DEVICE_CDC_RESULT_ERROR_TERMINATED_BY_HOST,
USB_DEVICE_CDC_RESULT_ERROR
} USB_DEVICE_CDC_RESULT;

```

Members

Members	Description
USB_DEVICE_CDC_RESULT_OK	The operation was successful
USB_DEVICE_CDC_RESULT_ERROR_TRANSFER_SIZE_INVALID	The transfer size is invalid. Refer to the description <ul style="list-style-type: none"> • of the read or write function for more details
USB_DEVICE_CDC_RESULT_ERROR_TRANSFER_QUEUE_FULL	The transfer queue is full and no new transfers can be <ul style="list-style-type: none"> • scheduled
USB_DEVICE_CDC_RESULT_ERROR_INSTANCE_INVALID	The specified instance is not provisioned in the system
USB_DEVICE_CDC_RESULT_ERROR_INSTANCE_NOT_CONFIGURED	The specified instance is not configured yet
USB_DEVICE_CDC_RESULT_ERROR_PARAMETER_INVALID	The event handler provided is NULL
USB_DEVICE_CDC_RESULT_ERROR_ENDPOINT_HALTED	Transfer terminated because host halted the endpoint
USB_DEVICE_CDC_RESULT_ERROR_TERMINATED_BY_HOST	Transfer terminated by host because of a stall clear
USB_DEVICE_CDC_RESULT_ERROR	General CDC Function driver error

Description

USB Device CDC Function Driver USB Device CDC Result enumeration.

This enumeration lists the possible USB Device CDC Function Driver operation results. These values are returned by USB Device CDC Library functions.

Remarks

None.

USB_DEVICE_CDC_TRANSFER_FLAGS Enumeration

USB Device CDC Function Driver Transfer Flags

File

[usb_device_cdc.h](#)

C

```

typedef enum {
    USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE,
    USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING
} USB_DEVICE_CDC_TRANSFER_FLAGS;

```

Members

Members	Description
USB_DEVICE_CDC_TRANSFER_FLAGS_DATA_COMPLETE	This flag indicates there is no further data to be sent in this transfer and that the transfer should end. If the size of the transfer is a multiple of the maximum packet size for related endpoint configuration, the function driver will send a zero length packet to indicate end of the transfer to the host.
USB_DEVICE_CDC_TRANSFER_FLAGS_MORE_DATA_PENDING	This flag indicates there is more data to be sent in this transfer. If the size of the transfer is a multiple of the maximum packet size for the related endpoint configuration, the function driver will not send a zero length packet. If the size of the transfer is greater than (but not a multiple of) the maximum packet size, the function driver will only send maximum packet size amount of data. If the size of the transfer is greater than endpoint size but not an exact multiple of endpoint size, only the closest endpoint size multiple bytes of data will be sent. This flag should not be specified if the size of the transfer is less than maximum packet size.

Description

USB Device CDC Transfer Flags

These flags are used to indicate status of the pending data while sending data to the host by using the [USB_DEVICE_CDC_Write](#) function.

Remarks

The relevance of the specified flag depends on the size of the buffer. Refer to the individual flag descriptions for more details.

USB_DEVICE_CDC_TRANSFER_HANDLE Type

USB Device CDC Function Driver Transfer Handle Definition.

File

[usb_device_cdc.h](#)

C

```
typedef uintptr_t USB_DEVICE_CDC_TRANSFER_HANDLE;
```

Description

USB Device CDC Function Driver Transfer Handle Definition

This definition defines a USB Device CDC Function Driver Transfer Handle. A Transfer Handle is owned by the application but its value is modified by the [USB_DEVICE_CDC_Write](#), [USB_DEVICE_CDC_Read](#) and the [USB_DEVICE_CDC_SerialStateNotificationSend](#) functions. The transfer handle is valid for the life time of the transfer and expires when the transfer related event had occurred.

Remarks

None.

USB_DEVICE_CDC_EVENT_RESPONSE_NONE Macro

USB Device CDC Function Driver Event Handler Response Type None.

File

[usb_device_cdc.h](#)

C

```
#define USB_DEVICE_CDC_EVENT_RESPONSE_NONE
```

Description

USB Device CDC Function Driver Event Handler Response None

This is the definition of the CDC Function Driver Event Handler Response Type none.

Remarks

Intentionally defined to be empty.

USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID Macro

USB Device CDC Function Driver Invalid Transfer Handle Definition.

File

[usb_device_cdc.h](#)

C

```
#define USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID ((USB_DEVICE_CDC_TRANSFER_HANDLE)(-1))
```

Description

USB Device CDC Function Driver Invalid Transfer Handle Definition

This definition defines a USB Device CDC Function Driver Invalid Transfer Handle. A Invalid Transfer Handle is returned by the [USB_DEVICE_CDC_Write](#), [USB_DEVICE_CDC_Read](#) and the [USB_DEVICE_CDC_SerialStateNotificationSend](#) functions when the request was not successful.

Remarks

None.

USB_DEVICE_CDC_FUNCTION_DRIVER Macro

USB Device CDC Function Driver Function pointer

File

[usb_device_cdc.h](#)

C

```
#define USB_DEVICE_CDC_FUNCTION_DRIVER
```

Description

USB Device CDC Function Driver Function Pointer

This is the USB Device CDC Function Driver Function pointer. This should registered with the device layer in the function driver registration table.

Remarks

None.

USB_DEVICE_CDC_INDEX_0 Macro**File**

[usb_device_cdc.h](#)

C

```
#define USB_DEVICE_CDC_INDEX_0 0
```

Description

Use this to specify CDC Function Driver Instance 0

USB_DEVICE_CDC_INDEX_1 Macro**File**

[usb_device_cdc.h](#)

C

```
#define USB_DEVICE_CDC_INDEX_1 1
```

Description

Use this to specify CDC Function Driver Instance 1

USB_DEVICE_CDC_INDEX_2 Macro**File**

[usb_device_cdc.h](#)

C

```
#define USB_DEVICE_CDC_INDEX_2 2
```

Description

Use this to specify CDC Function Driver Instance 2

USB_DEVICE_CDC_INDEX_3 Macro**File**

[usb_device_cdc.h](#)

C

```
#define USB_DEVICE_CDC_INDEX_3 3
```

Description

Use this to specify CDC Function Driver Instance 3

USB_DEVICE_CDC_INDEX_4 Macro

File

[usb_device_cdc.h](#)

C

```
#define USB_DEVICE_CDC_INDEX_4 4
```

Description

Use this to specify CDC Function Driver Instance 4

USB_DEVICE_CDC_INDEX_5 Macro

File

[usb_device_cdc.h](#)

C

```
#define USB_DEVICE_CDC_INDEX_5 5
```

Description

Use this to specify CDC Function Driver Instance 5

USB_DEVICE_CDC_INDEX_6 Macro

File

[usb_device_cdc.h](#)

C

```
#define USB_DEVICE_CDC_INDEX_6 6
```

Description

Use this to specify CDC Function Driver Instance 6

USB_DEVICE_CDC_INDEX_7 Macro

File

[usb_device_cdc.h](#)

C

```
#define USB_DEVICE_CDC_INDEX_7 7
```

Description

Use this to specify CDC Function Driver Instance 7

Files

Files

Name	Description
usb_device_cdc.h	USB Device CDC Function Driver Interface
usb_device_cdc_config_template.h	USB device CDC Class configuration definitions template

Description

This section lists the source and header files used by the library.

usb_device_cdc.h

USB Device CDC Function Driver Interface

Enumerations

	Name	Description
	USB_DEVICE_CDC_EVENT	USB Device CDC Function Driver Events
	USB_DEVICE_CDC_RESULT	USB Device CDC Function Driver USB Device CDC Result enumeration.
	USB_DEVICE_CDC_TRANSFER_FLAGS	USB Device CDC Function Driver Transfer Flags

Functions

	Name	Description
≡	USB_DEVICE_CDC_EventHandlerSet	This function registers a event handler for the specified CDC function driver instance.
≡	USB_DEVICE_CDC_Read	This function requests a data read from the USB Device CDC Function Driver Layer.
≡	USB_DEVICE_CDC_SerialStateNotificationSend	This function schedules a request to send serial state notification to the host.
≡	USB_DEVICE_CDC_Write	This function requests a data write to the USB Device CDC Function Driver Layer.

Macros

	Name	Description
	USB_DEVICE_CDC_EVENT_RESPONSE_NONE	USB Device CDC Function Driver Event Handler Response Type None.
	USB_DEVICE_CDC_FUNCTION_DRIVER	USB Device CDC Function Driver Function pointer
	USB_DEVICE_CDC_INDEX_0	Use this to specify CDC Function Driver Instance 0
	USB_DEVICE_CDC_INDEX_1	Use this to specify CDC Function Driver Instance 1
	USB_DEVICE_CDC_INDEX_2	Use this to specify CDC Function Driver Instance 2
	USB_DEVICE_CDC_INDEX_3	Use this to specify CDC Function Driver Instance 3
	USB_DEVICE_CDC_INDEX_4	Use this to specify CDC Function Driver Instance 4
	USB_DEVICE_CDC_INDEX_5	Use this to specify CDC Function Driver Instance 5
	USB_DEVICE_CDC_INDEX_6	Use this to specify CDC Function Driver Instance 6
	USB_DEVICE_CDC_INDEX_7	Use this to specify CDC Function Driver Instance 7
	USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID	USB Device CDC Function Driver Invalid Transfer Handle Definition.

Structures

	Name	Description
	USB_DEVICE_CDC_EVENT_DATA_READ_COMPLETE	USB Device CDC Function Driver Read and Write Complete Event Data.
	USB_DEVICE_CDC_EVENT_DATA_SEND_BREAK	USB Device CDC Function Driver Send Break Event Data
	USB_DEVICE_CDC_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE	USB Device CDC Function Driver Read and Write Complete Event Data.
	USB_DEVICE_CDC_EVENT_DATA_WRITE_COMPLETE	USB Device CDC Function Driver Read and Write Complete Event Data.
	USB_DEVICE_CDC_INIT	USB Device CDC Function Driver Initialization Data Structure

Types

	Name	Description
	USB_DEVICE_CDC_EVENT_HANDLER	USB Device CDC Event Handler Function Pointer Type.
	USB_DEVICE_CDC_EVENT_RESPONSE	USB Device CDC Function Driver Event Callback Response Type
	USB_DEVICE_CDC_INDEX	USB Device CDC Function Driver Index
	USB_DEVICE_CDC_TRANSFER_HANDLE	USB Device CDC Function Driver Transfer Handle Definition.

Description

USB Device CDC Function Driver Interface

This file describes the USB Device CDC Function Driver interface. The application should include this file if it needs to use the CDC Function Driver API.

File Name

usb_device_cdc.h

Company

Microchip Technology Inc.

usb_device_cdc_config_template.h

USB device CDC Class configuration definitions template

Macros

	Name	Description
	USB_DEVICE_CDC_INSTANCES_NUMBER	Specifies the number of CDC instances.
	USB_DEVICE_CDC_QUEUE_DEPTH_COMBINED	Specifies the combined queue size of all CDC instances.

Description

USB Device CDC Class Configuration Definitions

This file contains configurations macros needed to configure the CDC Function Driver. This file is a template file only. It should not be included by the application. The configuration macros defined in the file should be defined in the configuration specific system_config.h.

File Name

usb_device_cdc_config_template.h

Company

Microchip Technology Inc.

USB HID Device Library

This section describes the USB HID Device Library.

Introduction

Introduces the MPLAB Harmony USB Human Interface Device (HID) Device Library.

Description

The MPLAB Harmony USB Human Interface Device (HID) Device Library (also referred to as the HID Function Driver or Library) provides a high-level abstraction of the Human Interface Device (HID) class under the Universal Serial Bus (USB) communication with a convenient C language interface. This library supports revision 1.11 of the USB HID specification released by the USB Implementers forum. This library is part of the MPLAB Harmony USB Device stack.

The USB HID Device Class supports devices that are used by humans to control the operation of computer systems. The HID class of devices include a wide variety of human interface, data indicator, and data feedback devices with various types of output directed to the end user. Some common examples of HID class devices include:

- Keyboards
- Pointing devices such as a standard mouse, joysticks, and trackballs
- Front-panel controls like knobs, switches, buttons, and sliders
- Controls found on telephony, gaming or simulation devices such as steering wheels, rudder pedals, and dial pads
- Data devices such as bar-code scanners, thermometers, analyzers

The USB HID Device Library offers services to the application to interact and respond to the host requests. Additional information about the HID class can be obtained from the HID specification available from the USB Implementers Forum at: www.usbif.org.

Using the Library

This topic describes the basic architecture of the HID Function Driver and provides information and examples on its use.

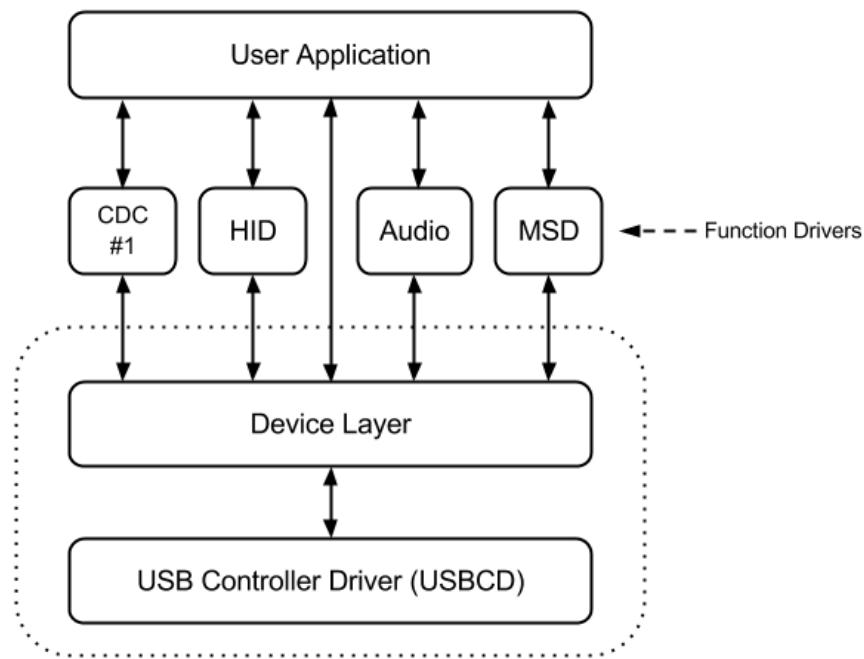
Abstraction Model

Provides an architectural overview of the USB HID Function Driver.

Description

The HID Function Driver offers services to a USB HID device to communicate with the host by abstracting the HID specification details. It must be used along with the USB Device Layer and USB Controller Driver to communicate with the USB Host. Figure 1 shows a block diagram of the MPLAB Harmony USB Architecture and where the HID Function Driver is placed.

Figure 1: HID Function Driver



The HID Function Driver together with USB Device Layer and the USB Controller Driver forms the basic library entity through which a HID device can communicate with the USB Host. The USB Controller Driver takes the responsibility of managing the USB peripheral on the device. The USB Device Layer handles the device enumeration, etc. The USB Device Layer forwards all HID-specific control transfers to the HID Function Driver. The HID Function Driver interprets the control transfers and requests application's intervention through event handlers and a well-defined set of API functions. The application must register a event handler with the HID Function Driver in the Device Layer Set Configuration Event. While the application must respond to the HID Function Driver events, it can do this either in the HID Function Driver event handler or after the event handler routine has returned. The application interacts with HID Function Driver routines to send and receive HID reports over the USB.

Figure 2 shows the architecture of the HID Function Driver. The HID Function Driver maintains the state of each instance. It receives HID class-specific control transfers from the USB Device Layer. Class-specific control transfers that require application response are forwarded to the application as function driver events. The application responds to these class specific control transfer event by directly calling Device Layer control transfer routines. Depending on the type of device, the HID Function Driver can use the control endpoint and/or interrupt endpoints for data transfers. The USB HID Device Driver exchanges data with the Host through data objects called reports. The report data format is described by the HID report descriptor, which is provided to the Host when requested. Refer to the HID specification available from www.usb.org for more details on the USB HID Device class and how report descriptors can be created. The HID Function Driver allows report descriptors to be specified for every instance. This allow the application to implement a composite HID device.

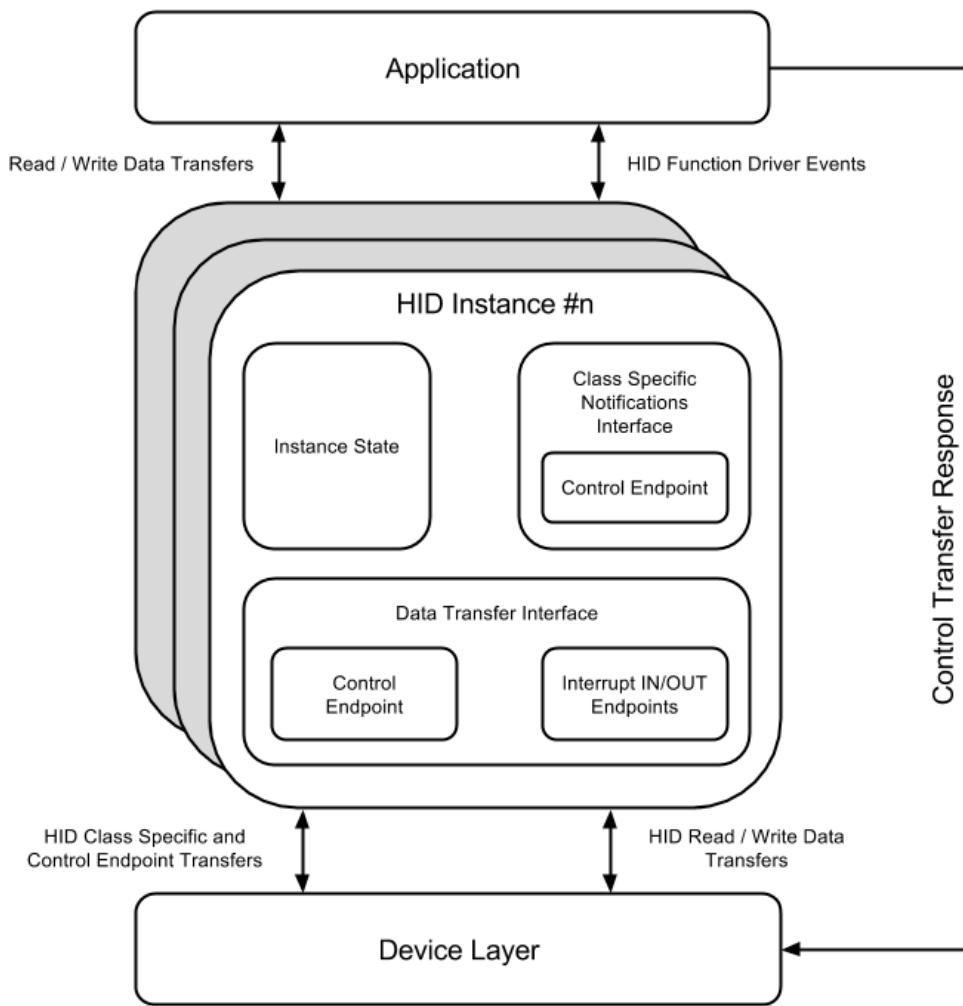


Figure 2: Architecture of the HID Function Driver

Library Overview

The USB HID Device Library mainly interacts with the system, its clients and function drivers, as shown in the [Abstraction Model](#).

The library interface routines are divided into sub-sections, which address one of the blocks or the overall operation of the USB HID Device Library.

Library Interface Section	Description
Functions	Provides event handler, report send/receive, and transfer cancellation functions.

How the Library Works

Library Initialization

Describes how the HID Function Driver is initialized.

Description

The HID Function Driver instance for a USB device configuration is initialized by the Device Layer when the configuration is set by the Host. This process does not require application intervention. Each instance of the HID Function be registered with the Device Layer through the [Device Layer Function Driver Registration Table](#). The HID function driver requires a initialization data structure that contains details about the report descriptor and the reports send/receive queue size associated with the specific instance of the HID Function Driver. The funcDriver member of the registration entry must be set to [USB_DEVICE_HID_FUNCTION_DRIVER](#). This object is a global object provided by the HID Function Driver and points to the HID Function Driver - Device Layer interface functions, which are required by the Device Layer. The following code shows an

example of how a HID Function Driver instance (implementing a USB HID Mouse) can be registered with the Device Layer.

```
/* This code shows an example of registering a HID function driver
 * with the Device Layer. While registering the function driver, an initialization
 * data structure must be specified. In this example, hidInit is the HID function
 * driver initialization data structure. */

/* This hid_rpt01 report descriptor describes a 3 button 2
 * axis mouse pointing device */
const uint8_t hid_rpt01[]=
{
    0x06, 0x00, 0xFF,      // Usage Page = 0xFF00 (Vendor Defined Page 1)
    0x09, 0x01,            // Usage (Vendor Usage 1)
    0xA1, 0x01,            // Collection (Application)
    0x19, 0x01,            // Usage Minimum
    0x29, 0x40,            // Usage Maximum      //64 input usages total (0x01 to 0x40)
    0x15, 0x01,            // Logical Minimum (data bytes in the report may have minimum value = 0x00)
    0x25, 0x40,            // Logical Maximum (data bytes in the report may have maximum value = 0x00FF =
unsigned 255)
    0x75, 0x08,            // Report Size: 8-bit field size
    0x95, 0x40,            // Report Count: Make sixty-four 8-bit fields (the next time the parser hits an
"Input", "Output",
               // or "Feature" item)
               // Input (Data, Array, Abs): Instantiates input packet fields based on the
previous report size,
               // count, logical min/max, and usage.
    0x19, 0x01,            // Usage Minimum
    0x29, 0x40,            // Usage Maximum      //64 output usages total (0x01 to 0x40)
    0x91, 0x00,            // Output (Data, Array, Abs): Instantiates output packet fields. Uses same report
size and
               // count as "Input" fields, since nothing new or different was specified to the
parser since
               // the "Input" item.
    0xC0                  // End Collection
};

/* HID Function Driver Initialization data structure. This
 * contains the size of the report descriptor and a pointer
 * to the report descriptor. If there are multiple HID instances
 * each with different report descriptors, multiple such data
 * structures may be needed */

USB_DEVICE_HID_INITIALIZATION hidInit =
{
    sizeof(hid_rpt01), // Size of the report
    (uint8_t *)&hid_rpt01 // Pointer to the report
    1, // Send queue size is 1. We will not queue up reports.
    0 // Receive queue size 0. We will not receive reports.
};

/* The HID function driver instance is now registered with
 * device layer through the function driver registration
 * table. */

const USB_DEVICE_FUNCTION_REGISTRATION_TABLE funcRegistrationTable[1] =
{
    {
        .speed = USB_SPEED_FULL|USB_SPEED_HIGH,      // Supported speed
        .configurationValue = 1,                      // To be initialized for Configuration 1
        .interfaceNumber = 0,                         // Starting interface number
        .numberOfInterfaces = 1,                      // Number of Interfaces
        .funcDriverIndex = 0,                         // Function Driver instance index is 0
        .funcDriverInit = &hidInit,                   // Function Driver Initialization
        .driver = USB_DEVICE_HID_FUNCTION_DRIVER     // Pointer to the function driver - Device Layer
Interface functions
    }
};
}
```

Event Handling

Describes HID Function Driver event handler registration and event handling.

Description

Registering a HID Function Driver Event Handler

While creating a USB HID Device-based application, an event handler must be registered with the Device Layer (the Device Layer Event Handler) and every HID Function Driver instance (HID Function Driver Event Handler). The HID Function Driver event handler receives HID events. This event handler should be registered before the USB Device Layer acknowledges the SET CONFIGURATION request from the USB Host. To ensure this, the event handler should be set in the USB_DEVICE_EVENT_CONFIGURED event that is generated by the device layer. While registering the HID Function Driver event handler, the HID Function Driver allows the application to also pass a data object in the event handler register function. This data object gets associated with the instance of the HID Function Driver and is returned by the driver when a HID Function Driver event occurs. The following code shows an example of how this can be done.

```
/* This is a sample Application Device Layer Event Handler
 * Note how the HID Function Driver event handler APP_USBDeviceHIDEEventHandler()
 * is registered in the USB_DEVICE_EVENT_CONFIGURED event. The appData
 * object that is passed in the USB_DEVICE_HID_EventHandlerSet()
 * function will be returned as the userData parameter in the
 * when the APP_USBDeviceHIDEEventHandler() function is invoked */

/* Application states */
typedef enum
{
    /* Application's state machine's initial state. */
    APP_STATE_INIT=0,
    APP_STATE_SERVICE_TASKS,
    APP_STATE_WAIT_FOR_CONFIGURATION,
} APP_STATES;

USB_DEVICE_HANDLE usbDeviceHandle;
APP_STATES appState;

/* This is the application device layer event handler function. */

USB_DEVICE_EVENT_RESPONSE APP_USBDeviceEventHandler
(
    USB_DEVICE_EVENT event,
    void * eventData,
    uintptr_t context
)
{
    uint8_t activeConfiguration;
    USB_SETUP_PACKET * setupPacket;
    switch(event)
    {
        case USB_DEVICE_EVENT_POWER_DETECTED:
            /* This event is generated when VBUS is detected. Attach the device */
            USB_DEVICE_Attach(usbDeviceHandle);
            break;

        case USB_DEVICE_EVENT_POWER_REMOVED:
            /* This event is generated when VBUS is removed. Detach the device */
            USB_DEVICE_Detach (usbDeviceHandle);
            break;

        case USB_DEVICE_EVENT_CONFIGURED:
            /* This event indicates that Host has set Configuration in the Device.*/
            /* Check the configuration */
            activeConfiguration = ((USB_DEVICE_EVENT_DATA_CONFIGURED *)pData)->configurationValue;
            if ( activeConfiguration == 1 )
            {
                /* Register the HID Device application event handler here.
                 * Note how the appData object pointer is passed as the
                 * user data */
                USB_DEVICE_HID_EventHandlerSet(USB_DEVICE_HID_INDEX_0, APP_USBDeviceHIDEEventHandler,
                (uintptr_t)&appData);
            }
    }
}
```

```

        /* Mark that set configuration is complete */
        appData.isConfigured = true;
    }
    break;

    case USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST:
        /* This event indicates a Control transfer setup stage has been completed. */
        setupPacket = (USB_SETUP_PACKET *)eventData;

        /* Parse the setup packet and respond with a USB_DEVICE_ControlSend(),
           USB_DEVICE_ControlReceive or USB_DEVICE_ControlStatus(). */

        break;

    case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT:
        /* This event indicates that a Control transfer Data has been sent to Host. */
        break;

    case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA RECEIVED:
        /* This event indicates that a Control transfer Data has been received from Host. */
        break;

    case USB_DEVICE_EVENT_CONTROL_TRANSFER_ABORTED:
        /* This event indicates a control transfer was aborted. */
        break;

    case USB_DEVICE_EVENT_SUSPENDED:
        break;

    case USB_DEVICE_EVENT_RESUMED:
        break;

    case USB_DEVICE_EVENT_ERROR:
        break;

    case USB_DEVICE_EVENT_RESET:
        break;

    case USB_DEVICE_EVENT_SOF:
        /* This event indicates an SOF is detected on the bus. The USB_DEVICE_SOF_EVENT_ENABLE
           macro should be defined to get this event. */
        break;
    default:
        break;
}
}

```

The HID Function Driver event handler executes in an interrupt context when the device stack is configured for Interrupt mode.

In Polled mode, the event handler is invoked in the context of the SYS_Tasks function. The application should not call computationally intensive functions, blocking functions, functions that are not interrupt safe, or functions that poll on hardware conditions from the event handler. Doing so will affect the ability of the USB device stack to respond to USB events and could potentially make the USB device non-compliant.

HID Function Driver Events:

The HID Function Driver generates events to which the application must respond. Some of these events are control requests communicated through control transfers. The application must therefore complete the control transfer. Based on the generated event, the application may be required to:

- Respond with a [USB_DEVICE_ControlSend](#) function, which completes the data stage of a Control Read Transfer
- Respond with a [USB_DEVICE_ControlReceive](#) function, which provisions the data stage of a Control Write Transfer
- Respond with a [USB_DEVICE_ControlStatus](#) function which completes the handshake stage of the Control Transfer. The application can either STALL or Acknowledge the handshake stage via the [USB_DEVICE_HID_ControlStatus](#) function.

The following table shows the HID Function Driver control transfer related events and the required application control transfer action.

HID Function Driver Control Transfer Event	Required Application Action
USB_DEVICE_HID_EVENT_GET_REPORT	Call USB_DEVICE_ControlSend function with a buffer containing the requested report.
USB_DEVICE_HID_EVENT_SET_REPORT	Call USB_DEVICE_ControlReceive function with a buffer to receive the report.

USB_DEVICE_HID_EVENT_SET_REPORT	Call the USB_DEVICE_ControlSend function with the pointer to the current USB_HID_PROTOCOL_CODE type data.
USB_DEVICE_HID_EVENT_SET_PROTOCOL	Acknowledge or stall using the USB_DEVICE_ControlStatus function.
USB_DEVICE_HID_EVENT_SET_IDLE	Acknowledge or stall using the USB_DEVICE_ControlStatus function.
USB_DEVICE_HID_EVENT_GET_IDLE	Call the USB_DEVICE_ControlSend function to send the current idle rate.
USB_DEVICE_HID_SET_DESCRIPTOR	Call the USB_DEVICE_ControlReceive function with a buffer to receive the report.
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT	No action required.
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA RECEIVED	Acknowledge or stall using the USB_DEVICE_ControlStatus function.
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_ABORTED	No action required.

The application can respond to HID Function Driver control transfer-related events in the function driver event handler. In a case where the data required for the response is not immediately available, the application can respond to the control transfer events after returning from the event handler. This defers the response to the control transfer event. However, please note that a USB host will typically wait for control transfer response for a finite time duration before timing out and canceling the transfer and associated transactions. Even when deferring response, the application must respond promptly if such timeouts have to be avoided.

The application should analyze the pData member of the event handler and check for event specific data. The following table shows the pData parameter data type for each HID function driver event.

Event Type	pData Parameter Data Type
USB_DEVICE_HID_EVENT_GET_REPORT	USB_DEVICE_HID_EVENT_DATA_GET_REPORT *
USB_DEVICE_HID_EVENT_SET_REPORT	USB_DEVICE_HID_EVENT_DATA_SET_REPORT *
USB_DEVICE_HID_EVENT_GET_IDLE	uint8_t*
USB_DEVICE_HID_EVENT_SET_IDLE	USB_DEVICE_HID_EVENT_DATA_SET_IDLE *
USB_DEVICE_HID_EVENT_SET_PROTOCOL	USB_HID_PROTOCOL_CODE*
USB_DEVICE_HID_EVENT_GET_PROTOCOL	NULL
USB_DEVICE_HID_EVENT_SET_DESCRIPTOR	USB_DEVICE_HID_EVENT_DATA_SET_DESCRIPTOR *
USB_DEVICE_HID_EVENT_REPORT_SENT	USB_DEVICE_HID_EVENT_DATA_REPORT_SENT *
USB_DEVICE_HID_EVENT_REPORT_RECEIVED	USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED *
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT	NULL
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA RECEIVED	NULL
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_ABORTED	NULL

The possible HID Function Driver events are described here along with the required application response, event specific data and likely follow up function driver event:

USB_DEVICE_HID_EVENT_GET_REPORT

Application Response: This event is generated when the USB HID Host is requesting a report over the control interface. The application must provide the report by calling the [USB_DEVICE_HID_ControlSend](#) function, either in the event handler, or in the application (after event handler function has exited). The application can use the [USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT](#) event to track completion of the command.

Event Specific Data (eventData): The application must interpret the pData parameter as a pointer to a

[USB_DEVICE_HID_EVENT_DATA_GET_REPORT](#) data type, which contains details about the requested report.

Likely Follow-up event: This event will likely be followed by the [USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT](#) event. This indicates that the data was sent to the Host successfully. The application must acknowledge the handshake stage of the control transfer by calling the [USB_DEVICE_HID_ControlStatus](#) function with the [USB_DEVICE_HID_CONTROL_STATUS_OK](#) flag.

USB_DEVICE_HID_EVENT_SET_REPORT

Application Response: This event is generated when the USB HID Host wants to send a report over the control interface. The application must provide a buffer to receive the report by calling the [USB_DEVICE_HID_ControlReceive](#) function either in the event handler or in the application (after the event handler function has exited). The application can use the

[USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA RECEIVED](#) event to track completion of the command.

Event Specific Data (eventData): The application must interpret the pData parameter as a pointer to a

[USB_DEVICE_HID_EVENT_DATA_SET_REPORT](#) data type, which contains details about the report that the Host intends to send.

Likely Follow-up event: This event will likely be followed by the

USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_RECEIVED event. This indicates that the data was received successfully. The application must either acknowledge or stall the handshake stage of the control transfer by calling the USB_DEVICE_HID_ControlStatus function with the USB_DEVICE_HID_CONTROL_STATUS_OK or USB_DEVICE_HID_CONTROL_STATUS_ERROR flag, respectively.

USB_DEVICE_HID_EVENT_GET_IDLE

Application Response: This event is generated when the USB HID Host wants to read the current idle rate for the specified report. The application must provide the idle rate through the USB_DEVICE_HID_ControlSend function, either in the event handler, or in the application (after the event handler function has exited). The application must use the controlTransferHandle parameter provided in the event while calling the USB_DEVICE_HID_ControlSend function. The application can use the USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT event to track completion of the command.

Event Specific Data (eventData): The application must interpret the pData parameter as a pointer to a uint8_t data type, which contains a report ID of the report for which the idle rate is requested.

Likely Follow-up event: This event will likely be followed by the

USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT event. This indicates that the data was sent to the Host successfully. The application must acknowledge the handshake stage of the control transfer by calling the USB_DEVICE_HID_ControlStatus function with the USB_DEVICE_HID_CONTROL_STATUS_OK flag.

USB_DEVICE_HID_EVENT_SET_IDLE

Application Response: This event is generated when the USB HID Host sends a Set Idle request to the device. The application must inspect the eventData and determine if the idle rate is to be supported. The application must either acknowledge (if the idle rate is supported) or stall the handshake stage of the control transfer (if the idle rate is not supported) by calling the USB_DEVICE_HID_ControlStatus function with the USB_DEVICE_HID_CONTROL_STATUS_OK or USB_DEVICE_HID_CONTROL_STATUS_ERROR flag, respectively.

Event Specific Data (eventData): The application must interpret the pData parameter as a pointer to a

USB_DEVICE_HID_EVENT_DATA_SET_IDLE data type that contains details about the report ID and the idle duration.

Likely Follow-up event: None.

USB_DEVICE_HID_EVENT_SET_PROTOCOL

Application Response: This event is generated when the USB HID Host sends a Set Protocol request to the device . The application must inspect the eventData and determine if the protocol is to be supported. The application must either acknowledge (if the protocol is supported) or stall the handshake stage of the control transfer (if the protocol is not supported) by calling USB_DEVICE_HID_ControlStatus function with SB_DEVICE_HID_CONTROL_STATUS_OK or USB_DEVICE_HID_CONTROL_STATUS_ERROR flag, respectively.

Event Specific Data (eventData): The application must interpret the pData parameter as a pointer to a USB_HID_PROTOCOL_CODE data type that contains details about the protocol to be set.

Likely Follow-up event: None.

USB_DEVICE_HID_EVENT_GET_PROTOCOL

Application Response: This event is generated when the USB HID Host issues a Get Protocol Request. The application must provide the current protocol through the USB_DEVICE_HID_ControlSend function either in the event handler or in the application (after the event handler has exited). The application can use the USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT event to track completion of the command.

Event Specific Data (eventData): None.

Likely Follow-up event: This event will likely be followed by the USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT event. This indicates that the data was sent to the host successfully. The application must acknowledge the handshake stage of the control transfer by calling the USB_DEVICE_HID_ControlStatus function with the USB_DEVICE_HID_CONTROL_STATUS_OK flag.

USB_DEVICE_HID_EVENT_SET_DESCRIPTOR

Application Response: This event is generated when the HID Host issues a Set Descriptor request. The application must provide a buffer to receive the descriptor through the USB_DEVICE_HID_ControlReceive function, either in the event handler, or in the application (after the event handler has exited). The application can use the USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_RECEIVED event to track completion of the command.

Event Specific Data: None

Likely Follow-up event: This event will likely be followed by the USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_RECEIVED event. This indicates that the data was received successfully. The application must either acknowledge or stall the handshake stage of the control transfer by calling USB_DEVICE_HID_ControlStatus function with USB_DEVICE_HID_CONTROL_STATUS_OK or the USB_DEVICE_HID_CONTROL_STATUS_ERROR flag, respectively.

USB_DEVICE_HID_EVENT_REPORT_SENT

Application Response: This event occurs when a report send operation scheduled by calling the [USB_DEVICE_HID_ReportSend](#) function has completed. This event does not require the application to respond with any function calls.

Event Specific Data (pData): The application must interpret the pData parameter as a pointer to a

[USB_DEVICE_HID_EVENT_DATA_REPORT_SENT](#) data type that contains details about the report that was sent.

Likely Follow-up event: None.

USB_DEVICE_HID_EVENT_REPORT_RECEIVED

Application Response: This event occurs when a report receive operation scheduled by calling the [USB_DEVICE_HID_ReportReceive](#) function has completed. This event does not require the application to respond with any function calls.

Event Specific Data (pData): The application must interpret the pData parameter as a pointer to a [USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED](#) data type that contains details about the report that was received.

Likely Follow-up event: None

USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT

Application Response: This event occurs when the data stage of a control read transfer has completed in response to the [USB_DEVICE_HID_ControlSend](#) function. The application must acknowledge the handshake stage of the control transfer by calling the [USB_DEVICE_HID_ControlStatus](#) function with the [USB_DEVICE_HID_CONTROL_STATUS_OK](#) flag.

Event Specific Data (pData): None.

Likely Follow-up event: None.

USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA RECEIVED

Application Response: This event occurs when the data stage of a control write transfer has completed in response to the [USB_DEVICE_HID_ControlReceive](#) function. The application must either acknowledge or stall the handshake stage of the control transfer by calling [USB_DEVICE_HID_ControlStatus](#) function with the [USB_DEVICE_HID_CONTROL_STATUS_OK](#) or [USB_DEVICE_HID_CONTROL_STATUS_ERROR](#) flag, respectively.

Event Specific Data (pData): None

Likely Follow-up event: None.

USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_ABORTED

Application Response: This event occurs when the a control transfer request is aborted by the Host. The application can use this event to update its HID class-specific control transfer state machine.

Event Specific Data (pData): None

Likely Follow-up event: None. The following code shows an example HID Function Driver event handling scheme.

The following code shows an example HID Function Driver event handling scheme.

```
// This code example shows all USB HID Driver events and a possible
// scheme for handling these events. In this example event responses are not
// deferred.
```

```
USB_DEVICE_HID_EVENT_RESPONSE USB_AppHIDEEventHandler
(
    USB_DEVICE_HID_INDEX instanceIndex,
    USB_DEVICE_HID_EVENT event,
    void * pData,
    uintptr_t userData
)
{
    uint8_t currentIdleRate;
    uint8_t someHIDReport[128];
    uint8_t someHIDDescriptor[128];
    USB_DEVICE_HANDLE          usbDeviceHandle;
    USB_HID_PROTOCOL_CODE * currentProtocol;
    USB_DEVICE_HID_EVENT_DATA_GET_REPORT      * getReportEventData;
    USB_DEVICE_HID_EVENT_DATA_SET_IDLE        * setIdleEventData;
    USB_DEVICE_HID_EVENT_DATA_SET_DESCRIPTOR   * setDescriptorEventData;
    USB_DEVICE_HID_EVENT_DATA_SET_REPORT       * setReportEventData;

    switch(event)
    {
        case USB_DEVICE_HID_EVENT_GET_REPORT:
            // In this case, pData should be interpreted as a
            // USB_DEVICE_HID_EVENT_DATA_GET_REPORT pointer. The application
            // must send the requested report by using the
            // USB_DEVICE_ControlSend() function.
            getReportEventData = (USB_DEVICE_HID_EVENT_DATA_GET_REPORT *)pData;
            USB_DEVICE_ControlSend(usbDeviceHandle, someHIDReport, getReportEventData->reportLength);
    }
}
```

```
break;

case USB_DEVICE_HID_EVENT_GET_PROTOCOL:

    // In this case, pData will be NULL. The application
    // must send the current protocol to the host by using
    // the USB_DEVICE_ControlSend() function.
    USB_DEVICE_ControlSend(deviceHandle, &currentProtocol, sizeof(USB_HID_PROTOCOL_CODE));

    break;
case USB_DEVICE_HID_EVENT_GET_IDLE:

    // In this case, pData will be a uint8_t pointer type to the
    // report ID for which the idle rate is being requested. The
    // application must send the current idle rate to the host by
    // using the USB_DEVICE_ControlSend() function.
    USB_DEVICE_ControlSend(deviceHandle, &currentIdleRate, 1);

    break;
case USB_DEVICE_HID_EVENT_SET_REPORT:

    // In this case, pData should be interpreted as a
    // USB_DEVICE_HID_EVENT_DATA_SET_REPORT type pointer. The
    // application can analyze the request and then obtain the
    // report by using the USB_DEVICE_ControlReceive() function.
    setReportEventData = (USB_DEVICE_HID_EVENT_DATA_SET_REPORT *)pData;
    USB_DEVICE_ControlReceive(deviceHandle, someHIDReport, setReportEventData->reportLength);

    break;
case USB_DEVICE_HID_EVENT_SET_PROTOCOL:

    // In this case, pData should be interpreted as a
    // USB_HID_PROTOCOL_CODE type pointer. The application can
    // analyze the data and decide to stall or accept the setting.
    // This shows an example of accepting the protocol setting.
    USB_DEVICE_ControlStatus(deviceHandle, USB_DEVICE_CONTROL_STATUS_OK);

    break;
case USB_DEVICE_HID_EVENT_SET_IDLE:

    // In this case, pData should be interpreted as a
    // USB_DEVICE_HID_EVENT_DATA_SET_IDLE type pointer. The
    // application can analyze the data and decide to stall
    // or accept the setting. This shows an example of accepting
    // the protocol setting.
    setIdleEventData = (USB_DEVICE_HID_EVENT_DATA_SET_IDLE *)pData;
    USB_DEVICE_ControlStatus(deviceHandle, USB_DEVICE_CONTROL_STATUS_OK);

    break;
case USB_DEVICE_HID_EVENT_SET_DESCRIPTOR:

    // In this case, the pData should be interpreted as a
    // USB_DEVICE_HID_EVENT_DATA_SET_DESCRIPTOR type pointer. The
    // application can analyze the request and then obtain the
    // descriptor by using the USB_DEVICE_ControlReceive() function.
    setDescriptorEventData = (USB_DEVICE_HID_EVENT_DATA_SET_DESCRIPTOR *)pData;
    USB_DEVICE_ControlReceive(deviceHandle, someHIDReport, setReportEventData->reportLength);

    break;
case USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_RECEIVED:

    // In this case, control transfer data was received. The
```

```

    // application can inspect that data and then stall the
    // handshake stage of the control transfer or accept it
    // (as shown here).
    USB_DEVICE_ControlStatus(deviceHandle, USB_DEVICE_CONTROL_STATUS_OK);

    break;

case USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT:

    // This means that control transfer data was sent. The
    // application would typically acknowledge the handshake
    // stage of the control transfer.

    USB_DEVICE_HID_ControlStatus(instanceIndex, controlTransferHandle,
        USB_DEVICE_HID_CONTROL_STATUS_OK);

    break;

case USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_ABORTED:

    // This means that control transfer data was sent. The
    // application would typically acknowledge the handshake
    // stage of the control transfer.

    break;

case USB_DEVICE_HID_EVENT_REPORT_RECEIVED:

    // This means a HID report receive request has completed.
    // The pData member should be interpreted as a
    // USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED pointer type.

    break;

case USB_DEVICE_HID_EVENT_REPORT_SENT:

    // This means a HID report send request has completed.
    // The pData member should be interpreted as a
    // USB_DEVICE_HID_EVENT_DATA_REPORT_SENT pointer type.

    break;
}

return(USB_DEVICE_HID_EVENT_RESPONSE_NONE);
}

```

Sending a Report

Describes how to send a report.

Description

The USB HID Device sends data to the USB HID Host as reports. The USB HID Device application should use the `USB_DEVICE_HID_ReportSend` function to send the report. This function returns a transfer handle that allows the application to track the read request. The request is completed when the Host has requested the data. A report send request could fail if the driver instance transfer queue is full. The completion of the write transfer is indicated by a `USB_DEVICE_HID_EVENT_REPORT_SENT` event. The transfer handle and the amount of data sent is returned in the `reportSent` member of the `eventData` data structure along with the event.

The following code shows an example of how a USB HID Mouse application sends a report to the host.

```

/* In this code example, the application uses the
 * USB_HID_MOUSE_ReportCreate to create the mouse report
 * and then uses the USB_DEVICE_HID_ReportSend() function
 * to send the report */

USB_HID_MOUSE_ReportCreate(appData.xCoordinate, appData.yCoordinate,
    appData.mouseButton, &appData.mouseReport);

/* Send the mouse report. */
USB_DEVICE_HID_ReportSend(appData.hidInstance,
    &appData.reportTransferHandle, (uint8_t*)&appData.mouseReport,
    sizeof(USB_HID_MOUSE_REPORT));

```

Receiving a Report

Describes how to receive a report.

Description

The application can receive a report from the Host by using the [USB_DEVICE_HID_ReportReceive](#) function. This function returns a transfer handler that allows the application to track the read request. The request is completed when the Host sends the report. The application must make sure that it allocates a buffer size that is at least the size of the report. The return value of the function indicates the success of the request. A read request could fail if the driver transfer queue is full. The completion of the read transfer is indicated by a [USB_DEVICE_HID_EVENT_REPORT_RECEIVED](#) event. The [reportReceived](#) member of the [eventData](#) data structure contains details about the received report. The following code shows an example of how a USB HID Keyboard can schedule a receive report operation to get the keyboard LED status.

```
/* The following code shows how the
 * USB HID Keyboard application schedules a
 * receive report operation to receive the
 * keyboard output report from the host. This
 * report contains the keyboard LED status. The
 * size of the report is 1 byte */

result = USB_DEVICE_HID_ReportReceive(appData.hidInstance,
                                      &appData.receiveTransferHandle,
                                      (uint8_t *)&appData.keyboardOutputReport,1);

if(USB_DEVICE_HID_RESULT_OK != result)
{
    /* Do error handling here */
}
```

Configuring the Library

Describes how to configure the HID Function Driver.

Macros

	Name	Description
	USB_DEVICE_HID_INSTANCES_NUMBER	Specifies the number of HID instances.
	USB_DEVICE_HID_QUEUE_DEPTH_COMINED	DOM-IGONORE-BEGIN

Description

The following configuration parameters must be defined while using the HID Function Driver. The configuration macros that implement these parameters must be located in the `system_config.h` file in the application project and a compiler include path (to point to the folder that contains this file) should be specified.

USB_DEVICE_HID_INSTANCES_NUMBER Macro

Specifies the number of HID instances.

File

[usb_device_hid_config_template.h](#)

C

```
#define USB_DEVICE_HID_INSTANCES_NUMBER
```

Description

USB Device HID Maximum Number of Instances

This macro defines the number of instances of the HID Function Driver. For example, if the application needs to implement two instances of the HID Function Driver (to create composite device) on one USB Device, the macro should be set to 2. Note that implementing a USB Device that features multiple HID interfaces requires appropriate USB configuration descriptors.

Remarks

None.

USB_DEVICE_HID_QUEUE_DEPTH_COMINED Macro**File**

[usb_device_hid_config_template.h](#)

C

```
#define USB_DEVICE_HID_QUEUE_DEPTH_COMINED
```

Description

DOM-IGNORE-BEGIN

Building the Library

Describes the files to be included in the project while using the HID Function Driver.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/usb.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
usb_device_hid.h	This header file should be included in any .c file that accesses the USB Device HID Function Driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/usb_device_hid.c	This file implements the HID Function driver interface and should be included in the project if the HID Device function is desired.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB HID Device Library depends on the following modules:

- [USB Device Layer Library](#)

Library Interface**a) Functions**

	Name	Description
≡	USB_DEVICE_HID_EventHandlerSet	This function registers a event handler for the specified HID function driver instance.
≡	USB_DEVICE_HID_ReportReceive	This function submits the buffer to HID function driver library to receive a report from host to device.
≡	USB_DEVICE_HID_ReportSend	This function submits the buffer to HID function driver library to send a report from device to host.
≡	USB_DEVICE_HID_TransferCancel	This function cancels a scheduled HID Device data transfer.

b) Data Types and Constants

	Name	Description
	USB_DEVICE_HID_EVENT	USB Device HID Function Driver Events
	USB_DEVICE_HID_EVENT_DATA_GET_REPORT	USB Device HID Get Report Event Data Type.
	USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED	USB Device HID Report Received Event Data Type.
	USB_DEVICE_HID_EVENT_DATA_REPORT_SENT	USB Device HID Report Sent Event Data Type.
	USB_DEVICE_HID_EVENT_DATA_SET_IDLE	USB Device HID Set Idle Event Data Type.
	USB_DEVICE_HID_EVENT_DATA_SET_REPORT	USB Device HID Set Report Event Data Type.
	USB_DEVICE_HID_INDEX	USB device HID Function Driver Index.
	USB_DEVICE_HID_EVENT_DATA_GET_IDLE	USB Device HID Get Idle Event Data Type.
	USB_DEVICE_HID_TRANSFER_HANDLE	USB Device HID Function Driver Transfer Handle Definition.
	USB_DEVICE_HID_EVENT_DATA_SET_PROTOCOL	USB Device HID Set Protocol Event Data Type.
	USB_DEVICE_HID_EVENT_HANDLER	USB Device HID Event Handler Function Pointer Type.
	USB_DEVICE_HID_EVENT_RESPONSE	USB Device HID Function Driver Event Callback Response Type
	USB_DEVICE_HID_INIT	USB Device HID Function Driver Initialization Data Structure
	USB_DEVICE_HID_RESULT	USB Device HID Function Driver USB Device HID Result enumeration.
	USB_DEVICE_HID_EVENT_RESPONSE_NONE	USB Device HID Function Driver Event Handler Response Type None.
	USB_DEVICE_HID_TRANSFER_HANDLE_INVALID	USB Device HID Function Driver Invalid Transfer Handle Definition.
	USB_DEVICE_HID_FUNCTION_DRIVER	This is a pointer to a group of HID Function Driver callback function pointers.
	USB_DEVICE_HID_INDEX_0	USB Device HID Function Driver Index Constants
	USB_DEVICE_HID_INDEX_1	This is macro USB_DEVICE_HID_INDEX_1 .
	USB_DEVICE_HID_INDEX_2	This is macro USB_DEVICE_HID_INDEX_2 .
	USB_DEVICE_HID_INDEX_3	This is macro USB_DEVICE_HID_INDEX_3 .
	USB_DEVICE_HID_INDEX_4	This is macro USB_DEVICE_HID_INDEX_4 .
	USB_DEVICE_HID_INDEX_5	This is macro USB_DEVICE_HID_INDEX_5 .
	USB_DEVICE_HID_INDEX_6	This is macro USB_DEVICE_HID_INDEX_6 .
	USB_DEVICE_HID_INDEX_7	This is macro USB_DEVICE_HID_INDEX_7 .

Description

This section describes the Application Programming Interface (API) functions of the USB Device HID library.

Refer to each section for a detailed description.

a) Functions

USB_DEVICE_HID_EventHandlerSet Function

This function registers a event handler for the specified HID function driver instance.

File

[usb_device_hid.h](#)

C

```
USB_DEVICE_HID_RESULT USB_DEVICE_HID_EventHandlerSet(USB_DEVICE_HID_INDEX instanceIndex,
USB_DEVICE_HID_EVENT_HANDLER eventHandler, uintptr_t context);
```

Returns

USB_DEVICE_HID_RESULT_OK - The operation was successful

USB_DEVICE_HID_RESULT_ERROR_INSTANCE_INVALID - The specified instance does not exist.

USB_DEVICE_HID_RESULT_ERROR_PARAMETER_INVALID - The eventHandler parameter is NULL

Description

This function registers a event handler for the specified HID function driver instance. This function should be called by the client when it receives a SET CONFIGURATION event from the device layer. A event handler must be registered for function driver to respond to function driver specific commands. If the event handler is not registered, the device layer will stall function driver specific commands and the USB device may not function.

Remarks

None.

Preconditions

This function should be called when the function driver has been initialized as a result of a set configuration.

Example

```
// This code snippet shows an example registering an event handler. Here
// the application specifies the context parameter as a pointer to an
// application object (appObject) that should be associated with this
// instance of the HID function driver.

USB_DEVICE_HID_RESULT result;

USB_DEVICE_HID_EVENT_RESPONSE APP_USBDeviceHIDEEventHandler
(
    USB_DEVICE_HID_INDEX instanceIndex,
    USB_DEVICE_HID_EVENT event,
    void * pData,
    uintptr_t context
)
{
    // Event Handling comes here

    switch(event)
    {
        ...
    }

    return(USB_DEVICE_HID_EVENT_RESPONSE_NONE);
}

result = USB_DEVICE_HID_EventHandlerSet (0, &APP_EventHandler, (uintptr_t) &appObject);

if(USB_DEVICE_HID_RESULT_OK != result)
{
    SYS_ASSERT ( false , "Error while registering event handler" );
}
```

Parameters

Parameters	Description
instance	Instance of the HID Function Driver.
eventHandler	A pointer to event handler function.
context	Application specific context that is returned in the event handler.

Function

```
USB_DEVICE_HID_RESULT USB_DEVICE_HID_EventHandlerSet
(
    USB_DEVICE_HID_INDEX instance
    USB_DEVICE_HID_EVENT_HANDLER eventHandler
    uintptr_t context
);
```

USB_DEVICE_HID_ReportReceive Function

This function submits the buffer to HID function driver library to receive a report from host to device.

File

[usb_device_hid.h](#)

C

```
USB_DEVICE_HID_RESULT USB_DEVICE_HID_ReportReceive(USB_DEVICE_HID_INDEX instanceIndex,
USB_DEVICE_HID_TRANSFER_HANDLE * handle, void * buffer, size_t size);
```

Returns

USB_DEVICE_HID_RESULT_OK - The receive request was successful. transferHandle contains a valid transfer handle.
 USB_DEVICE_HID_RESULT_ERROR_TRANSFER_QUEUE_FULL - internal request queue is full. The receive request could not be added.
 USB_DEVICE_HID_RESULT_ERROR_INSTANCE_NOT_CONFIGURED - The specified instance is not configured yet.
 USB_DEVICE_HID_RESULT_ERROR_INSTANCE_INVALID - The specified instance was not provisioned in the application and is invalid.

Description

This function submits the buffer to HID function driver library to receive a report from host to device. On completion of the transfer the library generates USB_DEVICE_HID_EVENT_REPORT_RECEIVED event to the application. A handle to the request is passed in the transferHandle parameter. The transfer handle expires when event handler for the USB_DEVICE_HID_EVENT_REPORT_RECEIVED exits. If the receive request could not be accepted, the function returns an error code and transferHandle will contain the value

[USB_DEVICE_HID_TRANSFER_HANDLE_INVALID](#).

Remarks

While the using the HID Function Driver with the PIC32MZ USB module, the report data buffer provided to the USB_DEVICE_HID_ReportReceive function should be placed in coherent memory and aligned at a 16 byte boundary. This can be done by declaring the buffer using the

`_attribute__((coherent, aligned(16)))` attribute. An example is shown here

```
uint8_t data[256] __attribute__((coherent, aligned(16)));
```

Preconditions

USB device layer must be initialized.

Example

```
USB_DEVICE_HID_TRANSFER_HANDLE hidTransferHandle;
USB_DEVICE_HID_RESULT result;

// Register APP_HIDEEventHandler function
USB_DEVICE_HID_EventHandlerSet( USB_DEVICE_HID_INDEX_0 ,
                                APP_HIDEEventHandler );

// Prepare report and request HID to send the report.
result = USB_DEVICE_HID_ReportReceive( USB_DEVICE_HID_INDEX_0 ,
                                       &hidTransferHandle ,
                                       &appReport[0], sizeof(appReport));

if( result != USB_DEVICE_HID_RESULT_OK)
{
    //Handle error.

}

//Implementation of APP_HIDEEventHandler

USB_DEVICE_HIDE_EVENT_RESPONSE APP_HIDEEventHandler
{
    USB_DEVICE_HID_INDEX instanceIndex,
    USB_DEVICE_HID_EVENT event,
    void * pData,
    uintptr_t context
}
{
    USB_DEVICE_HID_EVENT_DATA_REPORT RECEIVED reportReceivedEventData;
    // Handle HID events here.
    switch (event)
    {
        case USB_DEVICE_HID_EVENT_REPORT RECEIVED:
            if( (reportReceivedEventData->reportSize == sizeof(appReport)
                 && reportReceivedEventData->report == &appReport[0])
            {
                // Previous transfer was complete.
            }
            break;
        ....
    }
}
```

Parameters

Parameters	Description
instanceIndex	HID instance index.
transferHandle	HID transfer handle.
buffer	Pointer to buffer where the received report has to be received stored.
size	Buffer size.

Function

```
USB_DEVICE_HID_RESULT USB_DEVICE_HID_ReportReceive
(
    USB_DEVICE_HID_INDEX instanceIndex,
    USB_DEVICE_HID_TRANSFER_HANDLE * transferHandle,
    void * buffer,
    size_t size
);
```

USB_DEVICE_HID_ReportSend Function

This function submits the buffer to HID function driver library to send a report from device to host.

File

[usb_device_hid.h](#)

C

```
USB_DEVICE_HID_RESULT USB_DEVICE_HID_ReportSend(USB_DEVICE_HID_INDEX instanceIndex,
USB_DEVICE_HID_TRANSFER_HANDLE * handle, void * buffer, size_t size);
```

Returns

USB_DEVICE_HID_RESULT_OK - The send request was successful. transferHandle contains a valid transfer handle.
 USB_DEVICE_HID_RESULT_ERROR_TRANSFER_QUEUE_FULL - Internal request queue is full. The send request could not be added.
 USB_DEVICE_HID_RESULT_ERROR_INSTANCE_NOT_CONFIGURED - The specified instance is not configured yet.
 USB_DEVICE_HID_RESULT_ERROR_INSTANCE_INVALID - The specified instance was not provisioned in the application and is invalid.

Description

This function places a request to send a HID report with the USB Device HID Function Driver Layer. The function places a requests with driver, the request will get serviced when report is requested by the USB Host. A handle to the request is returned in the transferHandle parameter. The termination of the request is indicated by the USB_DEVICE_HID_EVENT_REPORT_SENT event. The amount of data sent, a pointer to the report and the transfer handle associated with the request is returned along with the event in the pData parameter of the event handler. The transfer handle expires when event handler for the USB_DEVICE_HID_EVENT_REPORT_SENT exits. If the send request could not be accepted, the function returns an error code and transferHandle will contain the value [USB_DEVICE_HID_TRANSFER_HANDLE_INVALID](#).

Remarks

While the using the HID Function Driver with the PIC32MZ USB module, the report data buffer provided to the USB_DEVICE_HID_ReportSend function should be placed in coherent memory and aligned at a 16 byte boundary. This can be done by declaring the buffer using the `__attribute__((coherent, aligned(16)))` attribute. An example is shown here

```
uint8_t data[256] __attribute__((coherent, aligned(16)));
```

Preconditions

USB device layer must be initialized.

Example

```
USB_DEVICE_HID_TRANSFER_HANDLE hidTransferHandle;
USB_DEVICE_HID_RESULT result;

// Register APP_HIDEEventHandler function
USB_DEVICE_HID_EventHandlerSet( USB_DEVICE_HID_INDEX_0 ,
                                APP_HIDEEventHandler );

// Prepare report and request HID to send the report.
result = USB_DEVICE_HID_ReportSend( USB_DEVICE_HID_INDEX_0 ,
                                    &hidTransferHandle ,
```

```

        &appReport[0], sizeof(appReport));

if( result != USB_DEVICE_HID_RESULT_OK)
{
    //Handle error.

}

//Implementation of APP_HIDEEventHandler

USB_DEVICE_HIDE_EVENT_RESPONSE APP_HIDEEventHandler
{
    USB_DEVICE_HID_INDEX instanceIndex,
    USB_DEVICE_HID_EVENT event,
    void * pData,
    uintptr_t context
}
{
    USB_DEVICE_HID_EVENT_DATA_REPORT_SENT * reportSentEventData;

    // Handle HID events here.
    switch (event)
    {
        case USB_DEVICE_HID_EVENT_REPORT_SENT:

            reportSentEventData = (USB_DEVICE_HID_EVENT_REPORT_SENT *)pData;
            if(reportSentEventData->reportSize == sizeof(appReport))
            {
                // The report was sent completely.
            }
            break;

            ....
    }
    return(USB_DEVICE_HID_EVENT_RESPONSE_NONE);
}

```

Parameters

Parameters	Description
instance	USB Device HID Function Driver instance.
transferHandle	Pointer to a USB_DEVICE_HID_TRANSFER_HANDLE type of variable. This variable will contain the transfer handle in case the send request was successful.
data	pointer to the data buffer containing the report to be sent.
size	Size (in bytes) of the report to be sent.

Function

```

USB_DEVICE_HID_RESULT USB_DEVICE_HID_ReportSend
(
    USB_DEVICE_HID_INDEX instanceIndex,
    USB_DEVICE_HID_TRANSFER_HANDLE * transferHandle,
    void * buffer,
    size_t size
)

```

USB_DEVICE_HID_TransferCancel Function

This function cancels a scheduled HID Device data transfer.

File

[usb_device_hid.h](#)

C

```

USB_DEVICE_HID_RESULT USB_DEVICE_HID_TransferCancel(USB_DEVICE_HID_INDEX usbDeviceHandle,

```

```
USB_DEVICE_HID_TRANSFER_HANDLE transferHandle);
```

Returns

USB_DEVICE_HID_RESULT_OK - The transfer will be canceled completely or partially.
 USB_DEVICE_HID_RESULT_ERROR_PARAMETER_INVALID - Invalid transfer handle
 USB_DEVICE_HID_RESULT_ERROR_INSTANCE_INVALID - Invalid HID instance index
 USB_DEVICE_HID_RESULT_ERROR - The transfer could not be canceled because it has either completed, the transfer handle is invalid or the last transaction is in progress.

Description

This function cancels a scheduled HID Device data transfer. The transfer could have been scheduled using the [USB_DEVICE_HID_ReportReceive](#), [USB_DEVICE_HID_ReportSend](#) function. If a transfer is still in the queue and its processing has not started, then the transfer is canceled completely. A transfer that is in progress may or may not get canceled depending on the transaction that is presently in progress. If the last transaction of the transfer is in progress, then the transfer will not be canceled. If it is not the last transaction in progress, the in-progress will be allowed to complete. Pending transactions will be canceled. The first transaction of an in progress transfer cannot be canceled.

Remarks

The buffer specific to the transfer handle should not be released unless the transfer abort event is notified through callback.

Preconditions

The USB Device should be in a configured state.

Example

```
// The following code snippet cancels a HID transfer.

USB_DEVICE_HID_TRANSFER_HANDLE transferHandle;
USB_DEVICE_HID_RESULT result;

result = USB_DEVICE_HID_TransferCancel(instanceIndex, transferHandle);

if(USB_DEVICE_HID_RESULT_OK == result)
{
    // The transfer cancellation was either completely or
    // partially successful.
}
```

Parameters

Parameters	Description
instanceIndex	HID Function Driver instance index.
transferHandle	Transfer handle of the transfer to be canceled.

Function

```
USB_DEVICE_HID_RESULT USB_DEVICE_HID_TransferCancel
(
    USB_DEVICE_HID_INDEX instanceIndex,
    USB_DEVICE_HID_TRANSFER_HANDLE transferHandle
);
```

b) Data Types and Constants

USB_DEVICE_HID_EVENT Enumeration

USB Device HID Function Driver Events

File

[usb_device_hid.h](#)

C

```
typedef enum {
    USB_DEVICE_HID_EVENT_GET_REPORT,
```

```

USB_DEVICE_HID_EVENT_GET_IDLE,
USB_DEVICE_HID_EVENT_GET_PROTOCOL,
USB_DEVICE_HID_EVENT_SET_REPORT,
USB_DEVICE_HID_EVENT_SET_IDLE,
USB_DEVICE_HID_EVENT_SET_PROTOCOL,
USB_DEVICE_HID_EVENT_REPORT_SENT,
USB_DEVICE_HID_EVENT_REPORT_RECEIVED,
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_RECEIVED,
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT,
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_ABORTED
} USB_DEVICE_HID_EVENT;

```

Members

Members	Description
USB_DEVICE_HID_EVENT_GET_REPORT	<p>This event occurs when the host issues a GET REPORT command. This is a HID class specific control transfer related event. The application must interpret the pData parameter as USB_DEVICE_HID_EVENT_DATA_GET_REPORT pointer type. If the report request is supported, the application must send the report to the host by using the USB_DEVICE_ControlSend function either in the event handler or after the event handler routine has returned. The application can track the completion of the request by using the USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT event. If the report request is not supported, the application must stall the request by calling the USB_DEVICE_ControlStatus function with a USB_DEVICE_CONTROL_STATUS_ERROR flag.</p>
USB_DEVICE_HID_EVENT_GET_IDLE	<p>This event occurs when the host issues a GET IDLE command. This is a HID class specific control transfer related event. The pData parameter will be a USB_DEVICE_HID_EVENT_DATA_GET_IDLE pointer type containing the ID of the report for which the idle parameter is requested. If the request is supported, the application must send the idle rate to the host by calling the USB_DEVICE_ControlSend function. This function can be called either in the event handler or after the event handler routine has returned. The application can track the completion of the request by using the USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT event. If the request is not supported, the application must stall the request by calling the USB_DEVICE_ControlStatus function with a USB_DEVICE_CONTROL_STATUS_ERROR flag.</p>
USB_DEVICE_HID_EVENT_GET_PROTOCOL	<p>This event occurs when the host issues a GET PROTOCOL command. This is a HID class specific control transfer related event. The pData parameter will be NULL. If the request is supported, the application must send a USB_HID_PROTOCOL_CODE data type object, containing the current protocol, to the host by calling the USB_DEVICE_ControlSend function. This function can be called either in the event handler or after the event handler routine has returned. The application can track the completion of the request by using the USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT event. If the request is not supported, the application must stall the request by calling the USB_DEVICE_ControlStatus function with a USB_DEVICE_CONTROL_STATUS_ERROR flag.</p>
USB_DEVICE_HID_EVENT_SET_REPORT	<p>This event occurs when the host issues a SET REPORT command. This is a HID class specific control transfer related event. The application must interpret the pData parameter as a USB_DEVICE_HID_EVENT_DATA_SET_REPORT pointer type. If the report request is supported, the application must provide a buffer, to receive the report, to the host by calling the USB_DEVICE_ControlReceive function either in the event handler or after the event handler routine has returned. The application can track the completion of the request by using the USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_RECEIVED event. If the report request is not supported, the application must stall the request by calling the USB_DEVICE_ControlStatus function with a USB_DEVICE_CONTROL_STATUS_ERROR flag.</p>

USB_DEVICE_HID_EVENT_SET_IDLE	This event occurs when the host issues a SET IDLE command. This is a HID class specific control transfer related event. The pData parameter will be USB_DEVICE_HID_EVENT_DATA_SET_IDLE pointer type. The application can analyze the idle duration and acknowledge or reject the setting by calling the USB_DEVICE_ControlStatus function. This function can be called in the event handler or after the event handler exits. If application can reject the request by calling the USB_DEVICE_ControlStatus function with a USB_DEVICE_CONTROL_STATUS_ERROR flag. It can accept the request by calling this function with USB_DEVICE_CONTROL_STATUS_OK flag.
USB_DEVICE_HID_EVENT_SET_PROTOCOL	This event occurs when the host issues a SET PROTOCOL command. This is a HID class specific control transfer related event. The pData parameter will be a pointer to a USB_DEVICE_HID_EVENT_DATA_SET_PROTOCOL data type. If the request is supported, the application must acknowledge the request by calling the USB_DEVICE_ControlStatus function with a USB_DEVICE_CONTROL_STATUS_OK flag. If the request is not supported, the application must stall the request by calling the USB_DEVICE_ControlStatus function with a USB_DEVICE_CONTROL_STATUS_ERROR flag.
USB_DEVICE_HID_EVENT_REPORT_SENT	This event indicates that USB_DEVICE_HID_ReportSend function completed a report transfer on interrupt endpoint from host to device. The pData parameter will be a USB_DEVICE_HID_EVENT_DATA_REPORT_SENT type.
USB_DEVICE_HID_EVENT_REPORT RECEIVED	This event indicates that USB_DEVICE_HID_ReportReceive function completed a report transfer on interrupt endpoint from device to host. The pData parameter will be a USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED type
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA RECEIVED	This event occurs when the data stage of a control write transfer has completed. This happens after the application uses the USB_DEVICE_ControlReceive function to respond to a HID Function Driver Control Transfer Event that requires data to be received from the host. The pData parameter will be NULL. The application should call the USB_DEVICE_ControlStatus function with the USB_DEVICE_CONTROL_STATUS_OK flag if the received data is acceptable or should call this function with USB_DEVICE_CONTROL_STATUS_ERROR flag if the received data needs to be rejected.
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA SENT	This event occurs when the data stage of a control read transfer has completed. This happens after the application uses the USB_DEVICE_ControlSend function to respond to a HID Function Driver Control Transfer Event that requires data to be sent to the host. The pData parameter will be NULL.
USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_ABORTED	This event occurs when an ongoing control transfer was aborted. The application must stop any pending control transfer related activities.

Description

USB Device HID Function Driver Events

These events are specific to the USB Device HID Function Driver instance. Each event description contains details about the parameters passed with event. The contents of pData depends on the generated event.

Events that are associated with the HID Function Driver Specific Control Transfers require application response. The application should respond to these events by using the [USB_DEVICE_ControlReceive](#), [USB_DEVICE_ControlSend](#) and [USB_DEVICE_ControlStatus](#) functions.

Calling the [USB_DEVICE_ControlStatus](#) function with a [USB_DEVICE_CONTROL_STATUS_ERROR](#) will stall the control transfer request. The application would do this if the control transfer request is not supported. Calling the [USB_DEVICE_ControlStatus](#) function with a [USB_DEVICE_CONTROL_STATUS_OK](#) will complete the status stage of the control transfer request. The application would do this if the control transfer request is supported

The following code snippet shows an example of a possible event handling scheme.

```
// This code example shows all HID Function Driver events and a possible
// scheme for handling these events. In this example event responses are not
// deferred.
```

```
USB_DEVICE_HID_EVENT_RESPONSE USB_AppHIDEEventHandler
(
    USB_DEVICE_HID_INDEX instanceIndex,
    USB_DEVICE_HID_EVENT event,
    void * pData,
    uintptr_t userData
)
```

```
{  
    uint8_t currentIdleRate;  
    uint8_t someHIDReport[128];  
    uint8_t someHIDDescriptor[128];  
    USB_DEVICE_HANDLE         usbDeviceHandle;  
    USB_HID_PROTOCOL_CODE    currentProtocol;  
    USB_DEVICE_HID_EVENT_DATA_GET_REPORT      * getReportEventData;  
    USB_DEVICE_HID_EVENT_DATA_SET_IDLE        * setIdleEventData;  
    USB_DEVICE_HID_EVENT_DATA_SET_DESCRIPTOR  * setDescriptorEventData;  
    USB_DEVICE_HID_EVENT_DATA_SET_REPORT      * setReportEventData;  
  
    switch(event)  
    {  
        case USB_DEVICE_HID_EVENT_GET_REPORT:  
  
            // In this case, pData should be interpreted as a  
            // USB_DEVICE_HID_EVENT_DATA_GET_REPORT pointer. The application  
            // must send the requested report by using the  
            // USB_DEVICE_ControlSend() function.  
  
            getReportEventData = (USB_DEVICE_HID_EVENT_DATA_GET_REPORT *)pData;  
            USB_DEVICE_ControlSend(usbDeviceHandle, someHIDReport, getReportEventData->reportLength);  
  
            break;  
  
        case USB_DEVICE_HID_EVENT_GET_PROTOCOL:  
  
            // In this case, pData will be NULL. The application  
            // must send the current protocol to the host by using  
            // the USB_DEVICE_ControlSend() function.  
  
            USB_DEVICE_ControlSend(usbDeviceHandle, &currentProtocol, sizeof(USB_HID_PROTOCOL_CODE));  
            break;  
  
        case USB_DEVICE_HID_EVENT_GET_IDLE:  
  
            // In this case, pData will be a  
            // USB_DEVICE_HID_EVENT_DATA_GET_IDLE pointer type containing the  
            // ID of the report for which the idle rate is being requested.  
            // The application must send the current idle rate to the host  
            // by using the USB_DEVICE_ControlSend() function.  
  
            USB_DEVICE_ControlSend(usbDeviceHandle, &currentIdleRate, 1);  
            break;  
  
        case USB_DEVICE_HID_EVENT_SET_REPORT:  
  
            // In this case, pData should be interpreted as a  
            // USB_DEVICE_HID_EVENT_DATA_SET_REPORT type pointer. The  
            // application can analyze the request and then obtain the  
            // report by using the USB_DEVICE_ControlReceive() function.  
  
            setReportEventData = (USB_DEVICE_HID_EVENT_DATA_SET_REPORT *)pData;  
            USB_DEVICE_ControlReceive(deviceHandle, someHIDReport, setReportEventData->reportLength);  
            break;  
  
        case USB_DEVICE_HID_EVENT_SET_PROTOCOL:  
  
            // In this case, pData should be interpreted as a  
            // USB_DEVICE_HID_EVENT_DATA_SET_PROTOCOL type pointer. The application can  
            // analyze the data and decide to stall or accept the setting.  
            // This shows an example of accepting the protocol setting.  
  
            USB_DEVICE_ControlStatus(deviceHandle, USB_DEVICE_CONTROL_STATUS_OK);  
            break;  
  
        case USB_DEVICE_HID_EVENT_SET_IDLE:  
  
            // In this case, pData should be interpreted as a
```

```
// USB_DEVICE_HID_EVENT_DATA_SET_IDLE type pointer. The
// application can analyze the data and decide to stall
// or accept the setting. This shows an example of accepting
// the protocol setting.

setIdleEventData = (USB_DEVICE_HID_EVENT_DATA_SET_IDLE *)pData;
USB_DEVICE_ControlStatus(deviceHandle, USB_DEVICE_CONTROL_STATUS_OK);
break;

case USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_RECEIVED:

    // In this case, control transfer data was received. The
    // application can inspect that data and then stall the
    // handshake stage of the control transfer or accept it
    // (as shown here).

    USB_DEVICE_ControlStatus(deviceHandle, USB_DEVICE_CONTROL_STATUS_OK);
    break;

case USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_DATA_SENT:

    // This means that control transfer data was sent. The
    // application would typically acknowledge the handshake
    // stage of the control transfer.

    break;

case USB_DEVICE_HID_EVENT_CONTROL_TRANSFER_ABORTED:

    // This is an indication only event. The application must
    // reset any HID control transfer related tasks when it receives
    // this event.

    break;

case USB_DEVICE_HID_EVENT_REPORT RECEIVED:

    // This means a HID report receive request has completed.
    // The pData member should be interpreted as a
    // USB_DEVICE_HID_EVENT_DATA_REPORT RECEIVED pointer type.

    break;

case USB_DEVICE_HID_EVENT_REPORT_SENT:

    // This means a HID report send request has completed.
    // The pData member should be interpreted as a
    // USB_DEVICE_HID_EVENT_DATA_REPORT_SENT pointer type.

    break;
}

return(USB_DEVICE_HID_EVENT_RESPONSE_NONE);
}
```

Remarks

Some of the events allow the application to defer responses. This allows the application some time to obtain the response data rather than having to respond to the event immediately. Note that a USB host will typically wait for event response for a finite time duration before timing out and canceling the event and associated transactions. Even when deferring response, the application must respond promptly if such timeouts have to be avoided.

USB_DEVICE_HID_EVENT_DATA_GET_REPORT Structure

USB Device HID Get Report Event Data Type.

File

[usb_device_hid.h](#)

C

```
typedef struct {
    uint8_t reportType;
    uint8_t reportID;
    uint16_t reportLength;
} USB_DEVICE_HID_EVENT_DATA_GET_REPORT;
```

Members

Members	Description
uint8_t reportType;	Report type
uint8_t reportID;	Report ID
uint16_t reportLength;	Report Length

Description

USB Device HID Get Report Event Data Type.

This defines the data type of the data generated to the HID event handler on a USB_DEVICE_HID_EVENT_GET_REPORT event.

Remarks

None.

USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED Structure

USB Device HID Report Received Event Data Type.

File

[usb_device_hid.h](#)

C

```
typedef struct {
    USB_DEVICE_HID_TRANSFER_HANDLE handle;
    size_t length;
    USB_DEVICE_HID_RESULT status;
} USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED;
```

Members

Members	Description
USB_DEVICE_HID_TRANSFER_HANDLE handle;	Transfer handle
size_t length;	Report size received
USB_DEVICE_HID_RESULT status;	Completion status of the transfer

Description

USB Device HID Report Received Event Data Type.

This defines the data type of the data generated to the HID event handler on a USB_DEVICE_HID_EVENT_REPORT_RECEIVED event.

Remarks

None.

USB_DEVICE_HID_EVENT_DATA_REPORT_SENT Structure

USB Device HID Report Sent Event Data Type.

File

[usb_device_hid.h](#)

C

```
typedef struct {
    USB_DEVICE_HID_TRANSFER_HANDLE handle;
    size_t length;
    USB_DEVICE_HID_RESULT status;
} USB_DEVICE_HID_EVENT_DATA_REPORT_SENT;
```

Members

Members	Description
USB_DEVICE_HID_TRANSFER_HANDLE handle;	Transfer handle
size_t length;	Report size transmitted
USB_DEVICE_HID_RESULT status;	Completion status of the transfer

Description

USB Device HID Report Sent Event Data Type.

This defines the data type of the data generated to the HID event handler on a USB_DEVICE_HID_EVENT_REPORT_SENT event.

Remarks

None.

USB_DEVICE_HID_EVENT_DATA_SET_IDLE Structure

USB Device HID Set Idle Event Data Type.

File

[usb_device_hid.h](#)

C

```
typedef struct {
    uint8_t duration;
    uint8_t reportID;
} USB_DEVICE_HID_EVENT_DATA_SET_IDLE;
```

Members

Members	Description
uint8_t duration;	Idle duration
uint8_t reportID;	Report ID

Description

USB Device HID Set Idle Event Data Type.

This defines the data type of the data generated to the HID event handler on a USB_DEVICE_HID_EVENT_SET_IDLE event.

Remarks

None.

USB_DEVICE_HID_EVENT_DATA_SET_REPORT Structure

USB Device HID Set Report Event Data Type.

File

[usb_device_hid.h](#)

C

```
typedef struct {
    uint8_t reportType;
    uint8_t reportID;
    uint16_t reportLength;
} USB_DEVICE_HID_EVENT_DATA_SET_REPORT;
```

Members

Members	Description
uint8_t reportType;	Report type
uint8_t reportID;	Report ID
uint16_t reportLength;	Report Length

Description

USB Device HID Set Report Event Data Type.

This defines the data type of the data generated to the HID event handler on a USB_DEVICE_HID_EVENT_SET_REPORT event.

Remarks

None.

USB_DEVICE_HID_INDEX Type

USB device HID Function Driver Index.

File

[usb_device_hid.h](#)

C

```
typedef uintptr_t USB_DEVICE_HID_INDEX;
```

Description

USB Device HID Driver Index Numbers

This uniquely identifies a HID Function Driver instance.

Remarks

None.

USB_DEVICE_HID_EVENT_DATA_GET_IDLE Structure

USB Device HID Get Idle Event Data Type.

File

[usb_device_hid.h](#)

C

```
typedef struct {
    uint8_t reportID;
} USB_DEVICE_HID_EVENT_DATA_GET_IDLE;
```

Members

Members	Description
uint8_t reportID;	The protocol code

Description

USB Device HID Get Idle Event Data

This defines the data type of the data generated to the HID event handler on a USB_DEVICE_HID_EVENT_GET_IDLE event.

Remarks

None.

USB_DEVICE_HID_TRANSFER_HANDLE Type

USB Device HID Function Driver Transfer Handle Definition.

File

[usb_device_hid.h](#)

C

```
typedef uintptr_t USB_DEVICE_HID_TRANSFER_HANDLE;
```

Description

USB Device HID Function Driver Transfer Handle Definition

This definition defines a USB Device HID Function Driver Transfer Handle. A Transfer Handle is owned by the application but its value is modified

by the [USB_DEVICE_HID_ReportSend](#) and [USB_DEVICE_HID_ReportReceive](#) functions. The transfer handle is valid for the life time of the transfer and expires when the transfer related event has occurred.

Remarks

None.

USB_DEVICE_HID_EVENT_DATA_SET_PROTOCOL Structure

USB Device HID Set Protocol Event Data Type.

File

[usb_device_hid.h](#)

C

```
typedef struct {
    USB_HID_PROTOCOL_CODE protocolCode;
} USB_DEVICE_HID_EVENT_DATA_SET_PROTOCOL;
```

Members

Members	Description
USB_HID_PROTOCOL_CODE protocolCode;	The protocol code

Description

USB Device HID Set Protocol Event Data

This defines the data type of the data generated to the HID event handler on a [USB_DEVICE_HID_EVENT_SET_PROTOCOL](#) event.

Remarks

None.

USB_DEVICE_HID_EVENT_HANDLER Type

USB Device HID Event Handler Function Pointer Type.

File

[usb_device_hid.h](#)

C

```
typedef USB_DEVICE_HID_EVENT_RESPONSE (* USB_DEVICE_HID_EVENT_HANDLER)(USB_DEVICE_HID_INDEX instanceIndex,
USB_DEVICE_HID_EVENT event, void * pData, uintptr_t context);
```

Description

USB Device HID Event Handler Function Pointer Type.

This data type defines the required function signature of the USB Device HID Function Driver event handling callback function. The application must register a pointer to a HID Function Driver events handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event call backs from the HID Function Driver. The function driver will invoke this function with event relevant parameters. The description of the event handler function parameters is given here.

instanceIndex - Instance index of the HID Function Driver that generated the event.

event - Type of event generated.

pData - This parameter should be type casted to a event specific pointer type based on the event that has occurred. Refer to the [USB_DEVICE_HID_EVENT](#) enumeration description for more details.

context - Value identifying the context of the application that registered the event handling function.

Remarks

None.

USB_DEVICE_HID_EVENT_RESPONSE Type

USB Device HID Function Driver Event Callback Response Type

File

[usb_device_hid.h](#)

C

```
typedef void USB_DEVICE_HID_EVENT_RESPONSE;
```

Description

USB Device HID Function Driver Event Handler Response Type

This is the return type of the HID Function Driver event handler.

Remarks

None.

USB_DEVICE_HID_INIT Structure

USB Device HID Function Driver Initialization Data Structure

File

[usb_device_hid.h](#)

C

```
typedef struct {
    size_t hidReportDescriptorSize;
    void * hidReportDescriptor;
    size_t queueSizeReportSend;
    size_t queueSizeReportReceive;
} USB_DEVICE_HID_INIT;
```

Members

Members	Description
size_t hidReportDescriptorSize;	Size of the HID report descriptor
void * hidReportDescriptor;	Pointer to HID report descriptor
size_t queueSizeReportSend;	Report send queue size
size_t queueSizeReportReceive;	Report receive queue size

Description

USB Device HID Function Driver Initialization Data Structure

This data structure must be defined for every instance of the HID function driver. It is passed to the HID function driver, by the Device Layer, at the time of initialization. The funcDriverInit member of the Device Layer Function Driver registration table entry must point to this data structure for an instance of the HID function driver.

Remarks

None.

USB_DEVICE_HID_RESULT Enumeration

USB Device HID Function Driver USB Device HID Result enumeration.

File

[usb_device_hid.h](#)

C

```
typedef enum {
    USB_DEVICE_HID_RESULT_OK,
    USB_DEVICE_HID_RESULT_ERROR_TRANSFER_QUEUE_FULL,
    USB_DEVICE_HID_RESULT_ERROR_INSTANCE_NOT_CONFIGURED,
    USB_DEVICE_HID_RESULT_ERROR_INSTANCE_INVALID,
    USB_DEVICE_HID_RESULT_ERROR_TERMINATED_BY_HOST,
    USB_DEVICE_HID_RESULT_ERROR
} USB_DEVICE_HID_RESULT;
```

Members

Members	Description
USB_DEVICE_HID_RESULT_OK	The operation was successful

USB_DEVICE_HID_RESULT_ERROR_TRANSFER_QUEUE_FULL	The transfer queue is full. No new transfers can be <ul style="list-style-type: none">• scheduled
USB_DEVICE_HID_RESULT_ERROR_INSTANCE_NOT_CONFIGURED	The specified instance is not configured yet
USB_DEVICE_HID_RESULT_ERROR_INSTANCE_INVALID	The specified instance is not provisioned in the system
USB_DEVICE_HID_RESULT_ERROR_TERMINATED_BY_HOST	Transfer terminated by host because of a stall clear
USB_DEVICE_HID_RESULT_ERROR	General Error

Description

USB Device HID Function Driver USB Device HID Result enumeration.

This enumeration lists the possible USB Device HID Function Driver operation results. These values USB Device HID Library functions.

Remarks

None.

USB_DEVICE_HID_EVENT_RESPONSE_NONE Macro

USB Device HID Function Driver Event Handler Response Type None.

File

[usb_device_hid.h](#)

C

```
#define USB_DEVICE_HID_EVENT_RESPONSE_NONE
```

Description

USB Device HID Function Driver Event Handler Response None

This is the definition of the HID Function Driver Event Handler Response Type none.

Remarks

Intentionally defined to be empty.

USB_DEVICE_HID_TRANSFER_HANDLE_INVALID Macro

USB Device HID Function Driver Invalid Transfer Handle Definition.

File

[usb_device_hid.h](#)

C

```
#define USB_DEVICE_HID_TRANSFER_HANDLE_INVALID
```

Description

USB Device HID Function Driver Invalid Transfer Handle Definition

This definition defines a USB Device HID Function Driver Invalid Transfer Handle. A Invalid Transfer Handle is returned by the [USB_DEVICE_HID_ReportReceive](#) and [USB_DEVICE_HID_ReportSend](#) functions when the request was not successful.

Remarks

None.

USB_DEVICE_HID_FUNCTION_DRIVER Macro

This is a pointer to a group of HID Function Driver callback function pointers.

File

[usb_device_hid.h](#)

C

```
#define USB_DEVICE_HID_FUNCTION_DRIVER
```

Description

USB Device HID Function Driver Device Layer callback function pointer group

This is a pointer to a group of HID Function Driver callback function pointers. The application must use this pointer while registering an instance of the HID function driver with the Device Layer via the function driver registration table i.e. the driver member of the function driver registration object in the device layer function driver registration table should be set to this value.

Remarks

None.

USB_DEVICE_HID_INDEX_0 Macro

USB Device HID Function Driver Index Constants

File

[usb_device_hid.h](#)

C

```
#define USB_DEVICE_HID_INDEX_0 0
```

Description

USB Device HID Function Driver Index Constants

This constants can be used by the application to specify HID function driver instance indexes.

Remarks

None.

USB_DEVICE_HID_INDEX_1 Macro

File

[usb_device_hid.h](#)

C

```
#define USB_DEVICE_HID_INDEX_1 1
```

Description

This is macro USB_DEVICE_HID_INDEX_1.

USB_DEVICE_HID_INDEX_2 Macro

File

[usb_device_hid.h](#)

C

```
#define USB_DEVICE_HID_INDEX_2 2
```

Description

This is macro USB_DEVICE_HID_INDEX_2.

USB_DEVICE_HID_INDEX_3 Macro

File

[usb_device_hid.h](#)

C

```
#define USB_DEVICE_HID_INDEX_3 3
```

Description

This is macro USB_DEVICE_HID_INDEX_3.

USB_DEVICE_HID_INDEX_4 Macro

File

[usb_device_hid.h](#)

C

```
#define USB_DEVICE_HID_INDEX_4 4
```

Description

This is macro USB_DEVICE_HID_INDEX_4.

USB_DEVICE_HID_INDEX_5 Macro

File

[usb_device_hid.h](#)

C

```
#define USB_DEVICE_HID_INDEX_5 5
```

Description

This is macro USB_DEVICE_HID_INDEX_5.

USB_DEVICE_HID_INDEX_6 Macro

File

[usb_device_hid.h](#)

C

```
#define USB_DEVICE_HID_INDEX_6 6
```

Description

This is macro USB_DEVICE_HID_INDEX_6.

USB_DEVICE_HID_INDEX_7 Macro

File

[usb_device_hid.h](#)

C

```
#define USB_DEVICE_HID_INDEX_7 7
```

Description

This is macro USB_DEVICE_HID_INDEX_7.

Files

Files

Name	Description
usb_device_hid.h	USB HID Function Driver
usb_device_hid_config_template.h	USB device HID class configuration definitions template,

Description

This section lists the source and header files used by the library.

usb_device_hid.h

USB HID Function Driver

Enumerations

	Name	Description
	USB_DEVICE_HID_EVENT	USB Device HID Function Driver Events
	USB_DEVICE_HID_RESULT	USB Device HID Function Driver USB Device HID Result enumeration.

Functions

	Name	Description
≡	USB_DEVICE_HID_EventHandlerSet	This function registers a event handler for the specified HID function driver instance.
≡	USB_DEVICE_HID_ReportReceive	This function submits the buffer to HID function driver library to receive a report from host to device.
≡	USB_DEVICE_HID_ReportSend	This function submits the buffer to HID function driver library to send a report from device to host.
≡	USB_DEVICE_HID_TransferCancel	This function cancels a scheduled HID Device data transfer.

Macros

	Name	Description
	USB_DEVICE_HID_EVENT_RESPONSE_NONE	USB Device HID Function Driver Event Handler Response Type None.
	USB_DEVICE_HID_FUNCTION_DRIVER	This is a pointer to a group of HID Function Driver callback function pointers.
	USB_DEVICE_HID_INDEX_0	USB Device HID Function Driver Index Constants
	USB_DEVICE_HID_INDEX_1	This is macro USB_DEVICE_HID_INDEX_1 .
	USB_DEVICE_HID_INDEX_2	This is macro USB_DEVICE_HID_INDEX_2 .
	USB_DEVICE_HID_INDEX_3	This is macro USB_DEVICE_HID_INDEX_3 .
	USB_DEVICE_HID_INDEX_4	This is macro USB_DEVICE_HID_INDEX_4 .
	USB_DEVICE_HID_INDEX_5	This is macro USB_DEVICE_HID_INDEX_5 .
	USB_DEVICE_HID_INDEX_6	This is macro USB_DEVICE_HID_INDEX_6 .
	USB_DEVICE_HID_INDEX_7	This is macro USB_DEVICE_HID_INDEX_7 .
	USB_DEVICE_HID_TRANSFER_HANDLE_INVALID	USB Device HID Function Driver Invalid Transfer Handle Definition.

Structures

	Name	Description
	USB_DEVICE_HID_EVENT_DATA_GET_IDLE	USB Device HID Get Idle Event Data Type.
	USB_DEVICE_HID_EVENT_DATA_GET_REPORT	USB Device HID Get Report Event Data Type.
	USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED	USB Device HID Report Received Event Data Type.
	USB_DEVICE_HID_EVENT_DATA_REPORT_SENT	USB Device HID Report Sent Event Data Type.
	USB_DEVICE_HID_EVENT_DATA_SET_IDLE	USB Device HID Set Idle Event Data Type.
	USB_DEVICE_HID_EVENT_DATA_SET_PROTOCOL	USB Device HID Set Protocol Event Data Type.
	USB_DEVICE_HID_EVENT_DATA_SET_REPORT	USB Device HID Set Report Event Data Type.
	USB_DEVICE_HID_INIT	USB Device HID Function Driver Initialization Data Structure

Types

	Name	Description
	USB_DEVICE_HID_EVENT_HANDLER	USB Device HID Event Handler Function Pointer Type.
	USB_DEVICE_HID_EVENT_RESPONSE	USB Device HID Function Driver Event Callback Response Type
	USB_DEVICE_HID_INDEX	USB device HID Function Driver Index.
	USB_DEVICE_HID_TRANSFER_HANDLE	USB Device HID Function Driver Transfer Handle Definition.

Description

USB HID Function Driver

This file contains the API definitions for the USB Device HID Function Driver. The application should include this file if it needs to use the HID Function Driver API.

File Name

usb_hid_function_driver.h

Company

Microchip Technology Inc.

usb_device_hid_config_template.h

USB device HID class configuration definitions template,

Macros

	Name	Description
	USB_DEVICE_HID_INSTANCES_NUMBER	Specifies the number of HID instances.
	USB_DEVICE_HID_QUEUE_DEPTH_COMINED	DOM-IGONORE-BEGIN

Description

USB Device HID Class Configuration Definitions

This file contains configurations macros needed to configure the HID Function Driver. This file is a template file only. It should not be included by the application. The configuration macros defined in the file should be defined in the configuration specific system_config.h.

File Name

usb_device_hid_config_template.h

Company

Microchip Technology Inc.

USB MSD Device Library

This section describes the USB MSD Device Library.

Introduction

Introduces the MPLAB Harmony USB Mass Storage Device (MSD) Library.

Description

The USB Mass Storage Device Library (also referred to as the MSD Function Driver) allows applications to create USB Mass Storage device such as USB Pen Drives or USB-based SD Card readers. Applications can also leverage the ready support for Mass Storage Devices by popular Host personal computer operating systems by using the MSD Function Driver interfaces as a means to access the device functionality. The MSD Function Driver also features the following:

- Supports Bulk Only Transport (BOT) protocol
- Allows implementation of multiple Logical Unit Number (LUN) storage devices
- Uses the MPLAB Harmony Block Driver interface to connect to storage media drivers

Using the Library

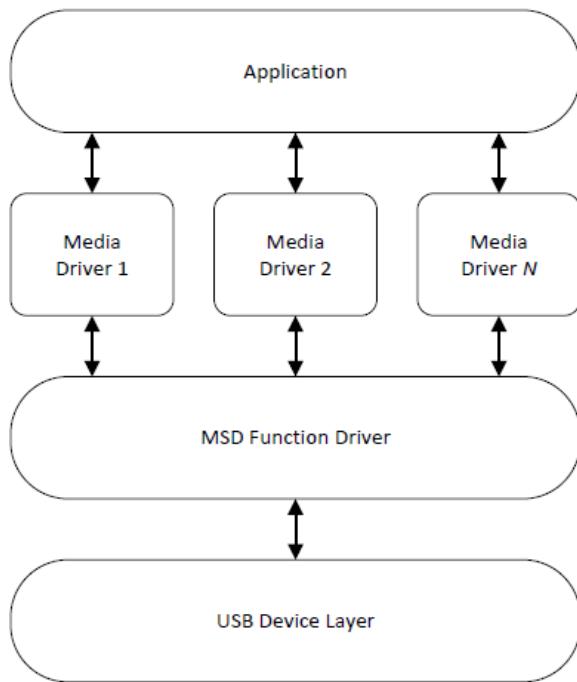
This topic describes the basic architecture of the USB MSD Device Library and provides information and examples on its use.

Abstraction Model

Provides an architectural overview of the USB MSD Device Library driver.

Description

The following diagram illustrates the functional interaction between the application, the MSD Function Driver, the media drivers, and the USB Device Layer.



As seen in the previous figure, the application does not have to interact with MSD function driver. Also, the MSD Function Driver does not have application functions that can be called. The media drivers control the storage media. The application interacts with the media drivers to update or access the information on the storage media. The MSD Function Driver interacts with the media drivers to process data read and write requests that it receives from the Host. This data is always accessed in blocks.

The MPLAB Harmony System module initializes the Device Layer and media drivers. A media driver is plugged into the MSD Function Driver by providing a media driver entry point in the MSD Function Driver initialization data structure. In the case of a multi-LUN storage, multiple media drivers can be plugged into the MSD Function Driver, with each one being capable of accessing different storage media types. The Device Layer initializes the MSD Function Driver when the Host sets the configuration that contains the Mass Storage interfaces. The MSD Function Driver Tasks routine is invoked in the context of the Device Layer Tasks routine. The MSD Function Driver interfaces should be registered in the USB Device Layer Function Driver Registration Table.

Library Overview

The USB MSD Device Library mainly interacts with the system, its clients and function drivers, as shown in the [Abstraction Model](#).

The library interface routines are divided into sub-sections, which address one of the blocks or the overall operation of the USB MSD Device Library.

Library Interface Section	Description
System Configuration Functions	Provides event handler, report send/receive, and transfer cancellation functions.

How the Library Works

This section explains how the MSD Function Driver should be added to the USB Device application and how a media driver should be plugged into it. Considerations while creating new media drivers to operate with the MSD function driver are also discussed.

Library Initialization

Describes how to initialize the MSD Function Driver.

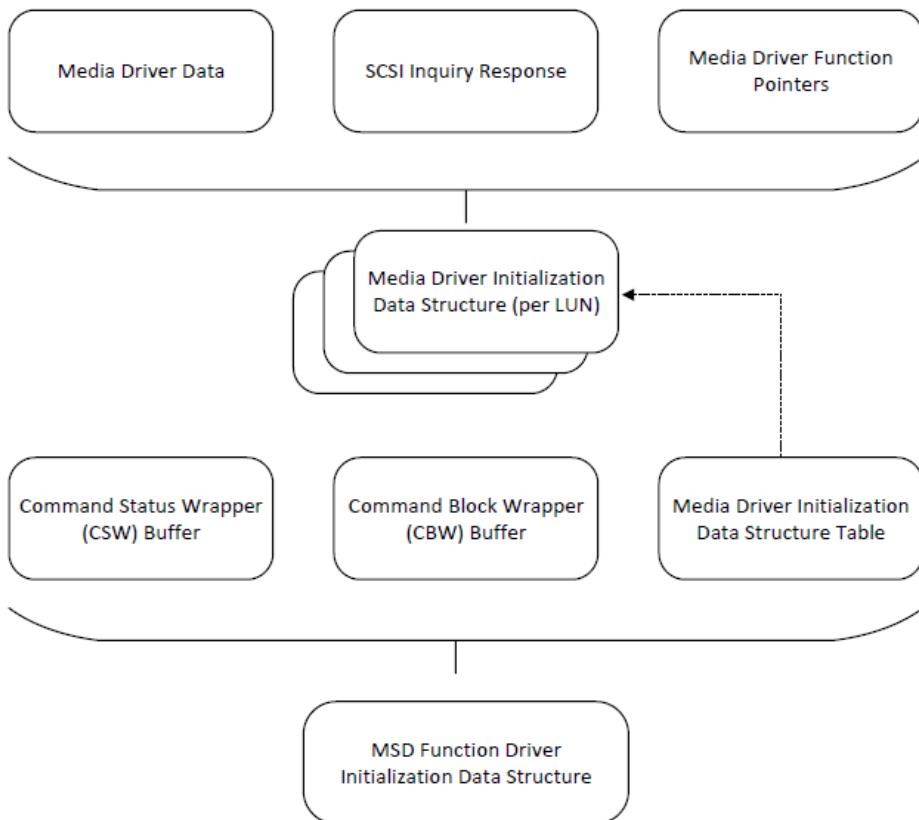
Description

The MSD Function Driver instance for a USB Device configuration is initialized by the USB Device Layer when the Host sets that configuration. This process does not require application intervention. Each instance of the MSD Function Driver should be registered with the USB Device Layer through the [Device Layer Function Driver Registration Table](#). While registering the MSD Function Driver, the driver member of the Function Driver Registration Table entry should be set to [USB_DEVICE_MSD_FUNCTION_DRIVER](#). This is an opaque function driver entry point provided by the MSD Function Driver for the Device Layer to use.

The MSD Function Driver requires an initialization data structure to be defined for each instance of the function driver. This initialization data structure should be of the type [USB_DEVICE_MSD_INIT](#). This initialization data structure contains the following:

- The number of Logical Unit Numbers (LUNs) in this MSD Function Driver instance
- A pointer to the USB_MSD_CBW type data structure. This pointer is used by the MSD Function Driver to receive the Command Block Wrapper (CBW) from the Host. For a PIC32MZ device, this array should be placed in coherent memory and should be aligned on a 4-byte boundary.
- A pointer to the USB_MSD_CSW type data structure. This pointer is used by the MSD Function Driver to send the Command Status Wrapper (CSW) to the Host. For a PIC32MZ device, this array should be placed in coherent memory and should be aligned on a 4-byte boundary.
- A pointer to the array of media driver initialization data structure. There should be one structure for every LUN. This is a [USB_DEVICE_MSD_MEDIA_INIT_DATA](#) type of data structure. There exists a one-to-one mapping between the LUN and the media driver initialization data structure.

The following figure shows a pictorial representation of the MSD Function Driver initialization data structure.



The [USB_DEVICE_MSD_MEDIA_INIT_DATA](#) data structure allows a media driver to be plugged into the MSD Function Driver. Any media driver that needs to be plugged into the MSD Function Driver needs to implement the interface (function pointer signatures) specified by the [USB_DEVICE_MSD_MEDIA_FUNCTIONS](#) type. For every LUN, a SCSI Inquiry Response data structure needs to be made available.

Use the following guidelines while implementing the media driver:

- Read functions should be non-blocking
- Write functions should be non-blocking
- The media driver should provide an event to indicate when a block transfer has complete. It should allow the event handler to be registered.
- Where required, the write function should erase and write to the storage area in one operation. The MSD Function Driver does not explicitly call the erase operation.
- The media driver should provide a media geometry object when required. This media geometry object allows the MSD Function Driver to understand the media characteristics. This object is of the type, SYS_FS_MEDIA_GEOMETRY.

The following code shows an example of plugging the MPLAB Harmony NVM Driver into the MSD Function Driver. The coherency and alignment attributes that are applied to the sectorBuffer, msdCBW, and msdCSW data objects is needed for operation on PIC32MZ devices.

```

*****
 * Sector buffer needed by for the MSD LUN.
 *****
uint8_t sectorBuffer[512] __attribute__((coherent)) __attribute__((aligned(4)));

*****
 * CBW and CSW structure needed by the MSD
 * function driver instance.
 *****
USB_MSD_CBW msdCBW __attribute__((coherent)) __attribute__((aligned(4)));
USB_MSD_CSW msdCSW __attribute__((coherent)) __attribute__((aligned(4)));

*****

```

```

* Because the PIC32MZ flash row size if 2048
* and the media sector size if 512 bytes, we
* have to allocate a buffer of size 2048
* to backup the row. A pointer to this row
* is passed in the media initialization data
* structure.
*****/*****/*****
uint8_t flashRowBackupBuffer [DRV_NVM_ROW_SIZE];
*****/*****/*****
* MSD Function Driver initialization
*****/*****/*****

USB_DEVICE_MSD_MEDIA_INIT_DATA msdMediaInit[1] =
{
{
    {
        DRV_NVM_INDEX_0,
        512,
        sectorBuffer,
        flashRowBackupBuffer,
        (void *)diskImage,
        {
            0x00,      // peripheral device is connected, direct access block device
            0x80,      // removable
            0x04,      // version = 00=> does not conform to any standard, 4=> SPC-2
            0x02,      // response is in format specified by SPC-2
            0x20,      // n-4 = 36-4=32= 0x20
            0x00,      // sccs etc.
            0x00,      // bque=1 and cmdque=0, indicates simple queuing 00 is obsolete,
                       // but as in case of other device, we are just using 00
            0x00,      // 00 obsolete, 0x80 for basic task queuing
            {
                'M','i','c','r','o','c','h','i'
            },
            {
                'M','a','s','s',' ', 'S','t','o','r','a','g','e',' '
            },
            {
                '0','0','0','1'
            }
        },
        {
            DRV_NVM_IsAttached,
            DRV_NVM_BLOCK_Open,
            DRV_NVM_BLOCK_Close,
            DRV_NVM_GeometryGet,
            DRV_NVM_BlockRead,
            DRV_NVM_BlockEraseWrite,
            DRV_NVM_IsWriteProtected,
            DRV_NVM_BLOCK_EventHandlerSet,
            DRV_NVM_BlockStartAddressSet
        }
    }
};

/* MSD Function Driver initialization
*****/*****/*****/

USB_DEVICE_MSD_INIT msdInit =
{
    /* Number of LUNS */
    1,
    /* Pointer to a CBW structure */
    &msdCBW,
    /* Pointer to a CSW structure */
    &msdCSW,
    /* Pointer to a table of Media Initialization data structures */
    &msdMediaInit[0]
};

/* USB Device Function Registration Table
*****/*****/*****

```

```
*****
const USB_DEVICE_FUNCTION_REGISTRATION_TABLE funcRegistrationTable[1] =
{
{
    .speed = USB_SPEED_FULL | USB_SPEED_HIGH,      // Device Speed
    .configurationValue = 1,                        // Configuration value
    .interfaceNumber = 0,                          // Start interface number
    .numberOfInterfaces = 1,                       // Number of interfaces owned
    .funcDriverIndex = 0,                          // Function driver index
    .funcDriverInit = (void*)&msdInit,            // Pointer to initialization data structure
    .driver = USB_DEVICE_MSD_FUNCTION_DRIVER      // Pointer to function driver
}
};
```

Data Transfer

Describes how the MSD Function Driver accesses the media.

Description

The MSD Function Driver opens the media drivers for read/write operations when the function driver is initialized by the Device Layer. This happens when the Host sets a configuration containing MSD interfaces. The Open operation is complete in the MSD Function Driver Tasks routines (called by the Device Layer).

The MSD Function Driver registers its own block operation event handler with the media drivers. Media Read and Write functions are called when the function driver receives a Sector Read or Sector Write request from the Host. The request will be tracked in the function driver Task routine. While the function driver waits for the media to complete the block operation, the function driver will NAK the data stage of the MSD data transfer request.

The MSD Function Driver does not provide any events to the application. It is possible that the application may also open the media driver while they are already opened by the MSD Function Driver. If the application and the MSD Function Driver try to write to the same media driver, the result could be unpredictable. It is recommended that the application restrict write access to the media driver while the USB device is plugged into the Host.

The application does not have to intervene in the functioning of the MSD Function Driver. Basically, the MSD Function Driver does provide any application callable functions.

Configuring the Library

Describes how to configure the MSD Function Driver.

Macros

	Name	Description
	USB_DEVICE_MSD_INSTANCES_NUMBER	Number of MSD function Driver instances required in the USB Device.
	USB_DEVICE_MSD_LUNS_NUMBER	Defines the number of LUNs per MSD function driver instance.

Description

The following configuration parameters must be defined while using the MSD Function Driver. The configuration macros that implement these parameters must be located in the `system_config.h` file in the application project and a compiler include path (to point to the folder that contains this file) should be specified.

USB_DEVICE_MSD_INSTANCES_NUMBER Macro

Number of MSD function Driver instances required in the USB Device.

File

[usb_device_msd_config_template.h](#)

C

```
#define USB_DEVICE_MSD_INSTANCES_NUMBER 1
```

Description

MSD Function Driver Instances Number

This configuration constant defines the number of MSD Function Driver instances in the USB Device. This value should be atleast 1 if the MSD function is required. In case where multiple MSD function drivers are required, it should be noted the MSD function driver supports multiple LUNs. This allows one MSD function driver instance to manage multiple media. Using multiple LUNs can be considered as an alternative to using multiple MSD function driver instances.

Remarks

None.

USB_DEVICE_MSD_LUNS_NUMBER Macro

Defines the number of LUNs per MSD function driver instance.

File

[usb_device_msd_config_template.h](#)

C

```
#define USB_DEVICE_MSD_LUNS_NUMBER 1
```

Description

Number of LUNs

This constant sets maximum possible number of Logical Unit (LUN) an instance of MSD can support. This value should be atleast 1. In cases where multiple MSD Function Driver instances are required, this constant should be set to the maximum number of LUNs required by any MSD Function Driver instance. The following figure shows a pictorial representation of the MSD function driver initialization data structure.

Remarks

None.

Building the Library

Describes the files to be included in the project while using the MSD Function Driver.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/usb.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
usb_device_msd.h	This header file should be included in any .c file that accesses the USB Device MSD Function Driver API.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/usb_device_msd.c	This file implements the MSD Function driver interface and should be included in the project if the MSD Device function is desired.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB CDC Device Library depends on the following modules:

- [USB Device Layer Library](#)

Based on application needs, the library may depend on the related storage media libraries, such as:

- Secure Digital (SD) Card Driver Library
- NVM Driver Library

Library Interface

Data Types and Constants

	Name	Description
	USB_DEVICE_MSD_MEDIA_FUNCTIONS	Pointer to the media driver functions for media instances to be used with the MSD function driver.
	USB_DEVICE_MSD_INIT	This structure contains required parameters for MSD function driver initialization.
	USB_DEVICE_MSD_MEDIA_INIT_DATA	This structure holds media related data of a particular logical unit.
	USB_DEVICE_MSD_FUNCTION_DRIVER	USB Device MSD Function Driver Function pointer

Description

This section describes the Application Programming Interface (API) functions of the USB MSD Device Library.

Refer to each section for a detailed description.

a) System Configuration Functions

Data Types and Constants

[USB_DEVICE_MSD_MEDIA_FUNCTIONS](#) Structure

Pointer to the media driver functions for media instances to be used with the MSD function driver.

File

[usb_device_msd.h](#)

C

```
struct USB_DEVICE_MSD_MEDIA_FUNCTIONS {
    bool (* isAttached)(const DRV_HANDLE handle);
    DRV_HANDLE (* open)(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);
    void (* close)(DRV_HANDLE hClient);
    SYS_FS_MEDIA_GEOMETRY * (* geometryGet)(DRV_HANDLE hClient);
    void (* blockRead)(DRV_HANDLE handle, uintptr_t * blockOperationHandle, void * data, uint32_t blockStart,
    uint32_t nBlocks);
    void (* blockWrite)(DRV_HANDLE handle, uintptr_t * blockOperationHandle, void * data, uint32_t
    blockStart, uint32_t nBlocks);
    bool (* isWriteProtected)(DRV_HANDLE drvHandle);
    void (* blockEventHandlerSet)(const DRV_HANDLE drvHandle, const void * eventHandler, const uintptr_t
    context);
    void (* blockStartAddressSet)(const DRV_HANDLE drvHandle, const void * addressOfStartBlock);
};
```

Members

Members	Description
bool (* isAttached)(const DRV_HANDLE handle);	In case of pluggable media, such as SD Card, this function returns true when the media is inserted, initialized and ready to be used. In case of non-pluggable media, such as Internal Flash memory, this function can return true when the media is ready to be used. The MSD host may not detect the media until this function returns true. This function pointer cannot be NULL
DRV_HANDLE (* open)(const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent);	The MSD Function Driver calls this function to obtain a handle and gain access to functionality of the specified instance of the media driver. The MSD function driver will attempt to open the driver with DRV_IO_INTENT_READWRITE and DRV_IO_INTENT_NONBLOCKING. The MSD host may not detect the media until the MSD function driver obtains a valid driver handle. The function driver will use this handle in all other functions to communicate with the media driver. This function pointer cannot be NULL
void (* close)(DRV_HANDLE hClient);	The MSD function driver calls this function when the function driver gets deinitialized as result of device detach or a change in configuration. The MSD function driver will open the media driver again to obtain a fresh driver handle when it gets initialized again. This function pointer cannot be NULL.

SYS_FS_MEDIA_GEOMETRY * (* geometryGet)(DRV_HANDLE hClient);	The MSD function driver calls this function when the function driver needs to know the storage capacity of the media. The MSD function driver uses the size of read region and number of read blocks to report the media capacity to the MSD host. This function pointer cannot be NULL.
void (* blockRead)(DRV_HANDLE handle, uintptr_t * blockOperationHandle, void * data, uint32_t blockStart, uint32_t nBlocks);	The MSD function driver calls this function when it needs to read a block of data. This function pointer cannot be NULL.
void (* blockWrite)(DRV_HANDLE handle, uintptr_t * blockOperationHandle, void * data, uint32_t blockStart, uint32_t nBlocks);	The MSD function driver calls this function when it needs to write a block of data. This function pointer can be NULL if the media is write protected.
bool (* isWriteProtected)(DRV_HANDLE drvHandle);	The MSD function driver calls this function to find out if the media is write-protected. This function pointer cannot be NULL.
void (* blockEventHandlerSet)(const DRV_HANDLE drvHandle, const void * eventHandler, const uintptr_t context);	The MSD function driver calls this function to register an block event call back function with the media driver. This event call back will be called when a block related operation has completed. This function pointer should not be NULL.
void (* blockStartAddressSet)(const DRV_HANDLE drvHandle, const void * addressOfStartBlock);	If not NULL and if the blockStartAddress parameter in the USB_DEVICE_MSD_MEDIA_INIT_DATA data structure for this media is not 0, then the MSD function driver calls this function immediately after opening the media driver. For media such a NVM, where the storage media is a part of the program memory flash, this function sets the start of the storage area on the media. This function is not required for media such as SD Card.

Description

Media Driver Function Pointer Data Structure

This structure contains function pointers, pointing to the media driver functions. The MSD function driver calls these functions at run time to access the media. This data structure should be specified during compilation and is a part of the MSD function driver initialization data structure. It is processed by the function driver when the function driver is initialized by the Device Layer.

Remarks

None.

USB_DEVICE_MSD_INIT Structure

This structure contains required parameters for MSD function driver initialization.

File

[usb_device_msd.h](#)

C

```
typedef struct {
    uint8_t numberOfLogicalUnits;
    USB_MSD_CBW * msdCBW;
    USB_MSD_CSW * msdCSW;
    USB_DEVICE_MSD_MEDIA_INIT_DATA * mediaInit;
} USB_DEVICE_MSD_INIT;
```

Members

Members	Description
uint8_t numberOfLogicalUnits;	Number of logical units supported.
USB_MSD_CBW * msdCBW;	Pointer to a Command Block Wrapper structure allocated to this instance <ul style="list-style-type: none"> • of the MSD function driver. In case of PIC32MZ device, this should be • placed in non cacheable section of RAM and should be aligned at a 4 byte boundary.
USB_MSD_CSW * msdCSW;	Pointer to a Command Status Wrapper structure allocated to this instance <ul style="list-style-type: none"> • of the MSD function driver. In case of PIC32MZ device, this should be • placed in non cacheable section of RAM and should be aligned at a 4 byte boundary.
USB_DEVICE_MSD_MEDIA_INIT_DATA * mediaInit;	Pointer to a table of media initialization data. This should contain an entry for every logical unit.

Description

USB MSD init structure.

This structure contains interface number, bulk-IN and bulk-OUT endpoint addresses, endpointSize, number of logical units supported and pointer to array of structure that contains media initialization.

Remarks

This structure must be configured by the user at compile time.

USB_DEVICE_MSD_MEDIA_INIT_DATA Structure

This structure holds media related data of a particular logical unit.

File

[usb_device_msd.h](#)

C

```
typedef struct {
    SYS_MODULE_INDEX instanceIndex;
    uint32_t sectorSize;
    uint8_t * sectorBuffer;
    uint8_t * blockBuffer;
    void * block0StartAddress;
    SCSI_INQUIRY_RESPONSE inquiryResponse;
    USB_DEVICE_MSD_MEDIA_FUNCTIONS mediaFunctions;
} USB_DEVICE_MSD_MEDIA_INIT_DATA;
```

Members

Members	Description
SYS_MODULE_INDEX instanceIndex;	Instance index of the media driver to opened for this LUN
uint32_t sectorSize;	Sector size for this LUN. If 0, means that sector size will be available from media geometry.
uint8_t * sectorBuffer;	Pointer to a byte buffer whose size is the size of the sector on this <ul style="list-style-type: none"> • media. In case of a PIC32MZ device, this buffer should be coherent and • should be aligned on a 16 byte boundary
uint8_t * blockBuffer;	In a case where the sector size of this media is less than the size of <ul style="list-style-type: none"> • the write block, a byte buffer of write block size should be provided to • the function driver. For example, the PIC32MZ NVM flash driver has a • flash program memory row size of 4096 bytes which is more than the • standard 512 byte sector. In such a case the application should set this • pointer to 4096 byte buffer
void * block0StartAddress;	Block 0 Start Address on this media. If non zero, then this address will be passed to blockStartAddressSet function. This should be set to start of the storage address on the media.
SCSI_INQUIRY_RESPONSE inquiryResponse;	Pointer to SCSI inquiry response for this LUN
USB_DEVICE_MSD_MEDIA_FUNCTIONS mediaFunctions;	Function pointers to the media driver functions

Description

USB Device MSD Media Initialization Data Member

It holds pointer to inquiry response, instance index and pointer to a structure that contains all media callback functions.

Remarks

An object of this structure must be configured by the user at compile time.

USB_DEVICE_MSD_FUNCTION_DRIVER Macro

USB Device MSD Function Driver Function pointer

File

[usb_device_msd.h](#)

C

```
#define USB_DEVICE_MSD_FUNCTION_DRIVER
```

Description

USB Device MSD Function Driver Function Pointer

This is the USB Device MSD Function Driver Function pointer. This should be registered with the device layer in the function driver registration table.

Remarks

None.

Files

Files

Name	Description
usb_device_msd.h	USB device MSD function driver interface header
usb_device_msd_config_template.h	USB Device MSD configuration template header file

Description

This section lists the source and header files used by the library.

[usb_device_msd.h](#)

USB device MSD function driver interface header

Macros

	Name	Description
	USB_DEVICE_MSD_FUNCTION_DRIVER	USB Device MSD Function Driver Function pointer

Structures

	Name	Description
	USB_DEVICE_MSD_MEDIA_FUNCTIONS	Pointer to the media driver functions for media instances to be used with the MSD function driver.
	USB_DEVICE_MSD_INIT	This structure contains required parameters for MSD function driver initialization.
	USB_DEVICE_MSD_MEDIA_INIT_DATA	This structure holds media related data of a particular logical unit.

Description

USB MSD function driver interface header

USB device MSD function driver interface header. This file should be included in the application if USB MSD functionality is required.

File Name

usb_device_msd.h

Company

Microchip Technology Inc.

[usb_device_msd_config_template.h](#)

USB Device MSD configuration template header file

Macros

	Name	Description
	USB_DEVICE_MSD_INSTANCES_NUMBER	Number of MSD function Driver instances required in the USB Device.
	USB_DEVICE_MSD_LUNS_NUMBER	Defines the number of LUNs per MSD function driver instance.

Description

USB Device MSD function driver compile time options

This file contains USB device MSD function driver compile time options(macros) that has to be configured by the user. This file is a template file and must be used as an example only. This file must not be directly included in the project.

File Name

usb_device_msd_config_template.h

Company

Microchip Technology Inc.

Generic USB Device Library

This section describes the Generic USB Device Library.

Introduction

Introduces the MPLAB Harmony Generic USB Device Library.

Description

A USB Device that does not follow any of the standard USB device class specifications is referred to as Generic (or a Vendor) USB Device. Such a device may be needed in cases where a standard USB device class does not meet application requirements with respect to transfer type, throughput or available interfaces. Generic USB Devices also typically require custom USB Host drivers.

The MPLAB Harmony USB Device Layer API features Endpoint API and events that facilitate development of a Generic USB Device. These API and events allow the application to do the following:

- Configure, enable, and disable endpoints
- Schedule Bulk, Interrupt and, Isochronous transfers
- Respond to control transfers
- Receive control and other transfer type related events

Using the Library

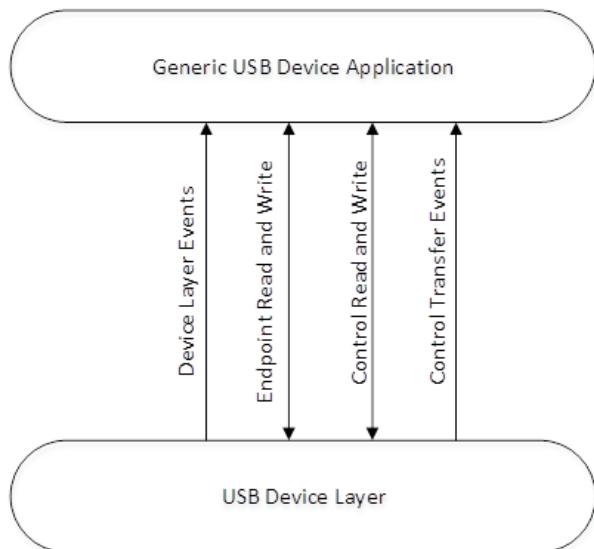
This topic describes the basic architecture of the Generic USB Device Library and provides information and examples on its use.

Abstraction Model

Provides an architectural overview of the Generic USB Device Driver.

Description

The Generic USB Device Library consists of USB Device Layer Endpoint API and events. The API allows the application to configure, enable, and disable endpoints. Endpoints can be configured for bulk, isochronous, and interrupt transfers. The events allow the application to track the completion of transfers and respond to control transfer events. It should be noted that the Generic USB Device Library in the MPLAB Harmony USB Device Stack does not have its own implementation, but rather, uses a subset of the Device Layer API to access the USB, as shown in the following diagram.



As seen in the figure, the application must implement the logic to implement the Generic USB Device behavior. It must respond to interface, class, and other control transfers. It must configure endpoints when the Host sets the configuration. Thus, the application implements the function driver for the Generic USB Function Driver.

The Generic USB Device Endpoint function and events provided by the Device Layer API abstract the details of configuring the USB peripheral. The Device Layer responds to standard USB requests as a part of the device enumeration process. The Device Layer control transfer functions and events allow the application to complete control transfers that are targeted to an endpoint, interface or others. The Device Layer endpoint read and write API provide a USB transaction or transfer level interface. Transactions or transfers can be queued.

Library Overview

Provides an overview of the Generic USB Device Driver.

Description

The Generic Function Driver features API to set application event handlers and transfer data over non-zero endpoints. The function driver is initialized by the Device Layer when a Set Configuration request is received by the device. This process does not require application intervention. As a part of this initialization process, all the endpoints belonging to the Generic Function Driver Interfaces will be enabled and configured. When the application receives the `USB_DEVICE_EVENT_CONFIGURED`, these endpoints are ready for data transfers.

The application design must ensure that the Generic Function Driver is registered in the Device Layer Function Driver Registration Table.

How the Library Works

This topic describes the basic architecture of the Generic USB Device Library and provides information and examples on its use.

Library Initialization

Describes how the Generic USB Device Library is initialized.

Description

Unlike the standard USB function drivers in the MPLAB Harmony USB Device Stack, in the case of a Generic USB Device, the USB Device Layer does not automatically enable or disable endpoints that belong to the Generic interface. This must be done by the application when the device is configured by the Host.

A USB Device can have multiple Generic interfaces. Each of these interfaces must have corresponding entries in the USB Device Layer function driver registration table. For Generic interfaces, the driver and funcDriverInit member of the function driver registration table entry should be set to NULL. The following code shows an example of how this is done.

```
/* This code shows an example function driver registration table entry
 * for a Generic USB Device Interface. Note that the function driver entry point
 * member is NULL. This instructs the Device Layer to pass all interface related
 * control transfers to the application. */
const USB_DEVICE_FUNCTION_REGISTRATION_TABLE funcRegistrationTable[1] =
{
    {
        .configurationValue = 1,           // Configuration descriptor index
        .driver = NULL,                  // No APIs exposed to the device layer
        .funcDriverIndex = 0,             // Zero Instance index
        .funcDriverInit = NULL,           // No init data
        .interfaceNumber = 0,             // Start interface number of this instance
        .numberOfInterfaces = 1,          // Total number of interfaces contained in this instance
        .speed = USB_SPEED_FULL|USB_SPEED_HIGH // USB Speed
    }
};
```

The endpoint read and endpoint write queue sizes are specified by the queueSizeEndpointRead and queueSizeEndpointWrite members of the `USB_DEVICE_INIT` device layer initialization data structure. These read and write queue sizes define the size of the read and write buffer object pools. Objects from these pools are then queued up at each read and write endpoint, when an endpoint read or write is requested. The total number of buffer objects is specified by the `USB_DEVICE_ENDPOINT_QUEUE_DEPTH_COMBINED` configuration constant.

Event Handling

This topic explains how the application should handle Generic USB Device events.

Description

The USB Device Layer generates two different types of events for a Generic USB Device.

- Control transfer events
- Endpoint data transfer events

While handing Device Layer events, it is recommended that computationally intensive operations or hardware access should not be performed with in the event handler. Doing so may affect the capability of the Device Stack to respond to changes on the USB and could cause the Device to become non-compliant.

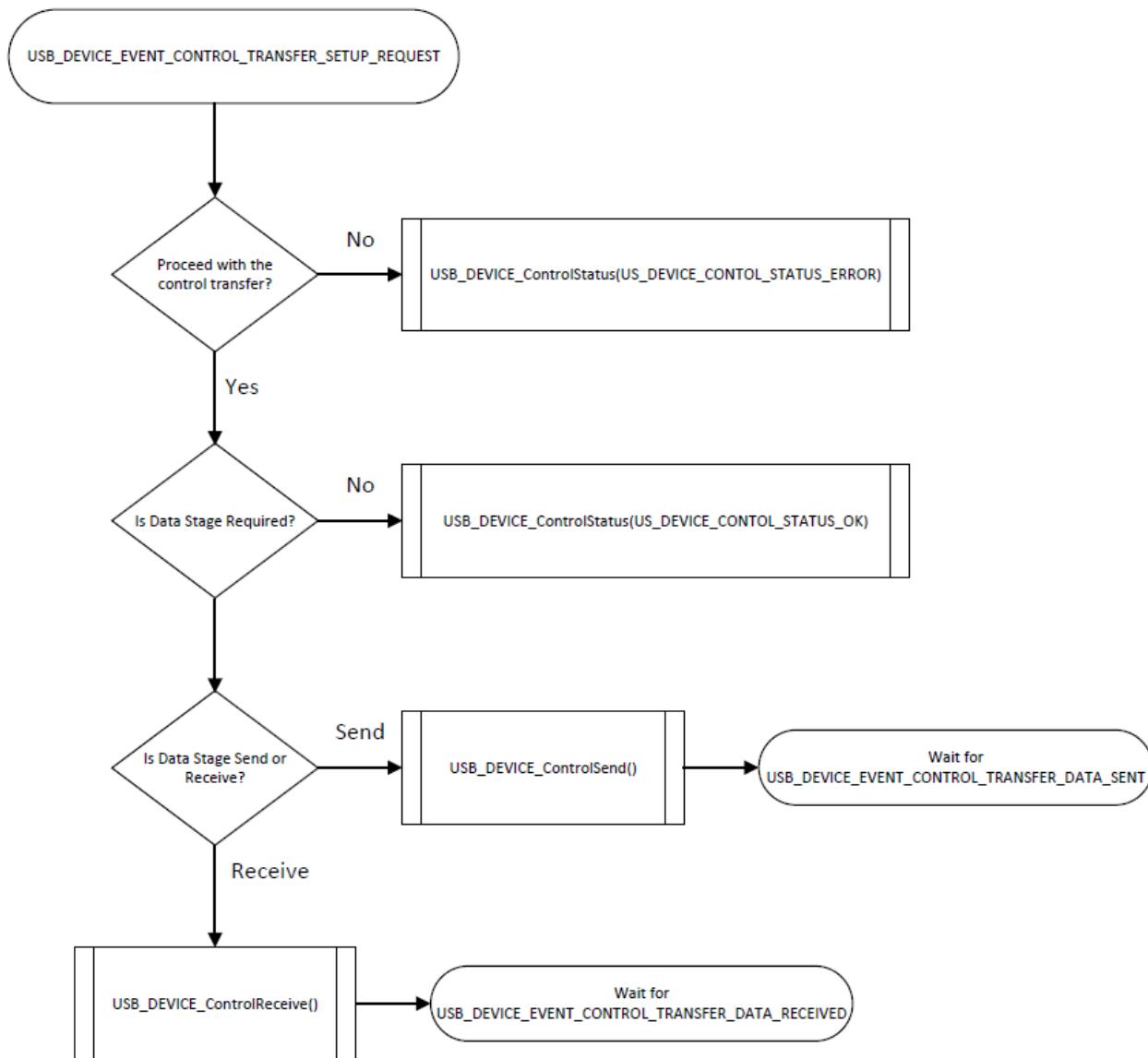
A Generic USB Device application must handle the above events along with the other Device Layer events.

Control Transfer Events

Describes control transfer events and provides a code example.

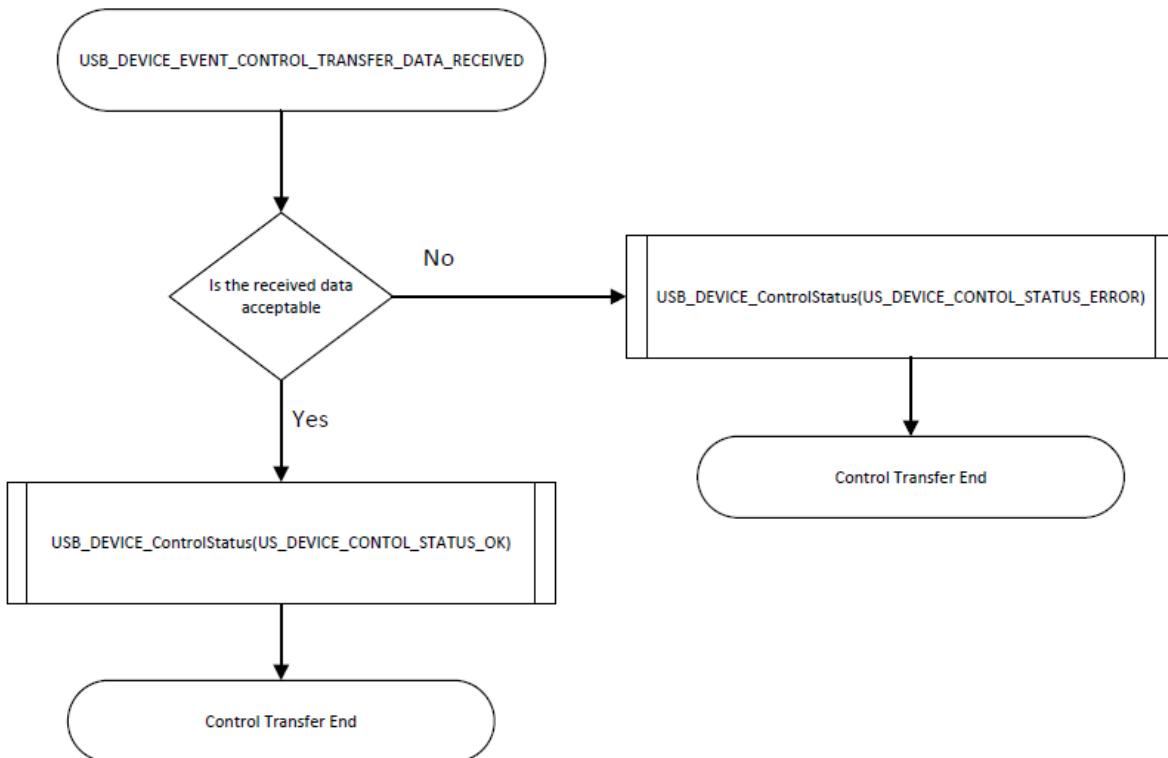
Description

These events occur when the Device Layer has received a control transfer that is targeted to an interface or an endpoint which is managed by the Generic USB Device Application. The `USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST` event is generated when the Setup stage of the control transfer has been received. The application must investigate the 8-byte setup command that accompanies this event. The following flowchart explains the interaction.



The application can then either choose to continue the control transfer or stall it. The control transfer is stalled by calling the `USB_DEVICE_ControlStatus` function with the `USB_DEVICE_CONTROL_STATUS_ERROR` flag. In case of zero data stage control transfers, the application can complete the control transfer by calling the `USB_DEVICE_ControlStatus` function with the `USB_DEVICE_CONTROL_STATUS_OK` flag. In case of control transfers that contain a data stage, the application must use the `USB_DEVICE_ControlSend` or the `USB_DEVICE_ControlReceive` function to send and receive data from the Host, respectively.

In a case where data is to be received from the host, the device layer generates `USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_RECEIVED` event when the data stage has completed. The application can analyze the received data and can then either choose to acknowledge or stall the control transfer by the calling the `USB_DEVICE_ControlStatus` function with the `USB_DEVICE_CONTROL_STATUS_ERROR` flag. This is shown in the following flow chart.



The following code shows an example of handling control transfer in a Generic USB Device. Note that the control transfer events are generated by the Device Layer.

```

/* This code shows an example of how the control transfer events
 * can be handled in a Generic USB Device. The example device will accept the
 * Set Interface Control Request and replies to the Get Interface Control Request
 * with the current alternate setting. */
case USB_DEVICE_EVENT_CONTROL_TRANSFER_SETUP_REQUEST:
    /* This means we have received a setup packet */
    setupPacket = (USB_SETUP_PACKET *)eventData;
    if(setupPacket->bRequest == USB_REQUEST_SET_INTERFACE)
    {
        /* If we have got the SET_INTERFACE request, we just acknowledge
         * for now. In this example, there is one alternate setting which
         * is already active. */
        USB_DEVICE_ControlStatus(appData.usbDevHandle,USB_DEVICE_CONTROL_STATUS_OK);
    }
    else if(setupPacket->bRequest == USB_REQUEST_GET_INTERFACE)
    {
        /* We have only one alternate setting and this setting 0. So
         * we send this information to the host. */
        USB_DEVICE_ControlSend(appData.usbDevHandle, &appData.altSetting, 1);
    }
    else
    {
        /* We have received a request that we cannot handle. Stall it*/
        USB_DEVICE_ControlStatus(appData.usbDevHandle, USB_DEVICE_CONTROL_STATUS_ERROR);
    }
    break;
case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_SENT:
    /* This is a notification event which the application can use to free
     * buffer that was used in a USB_DEVICE_ControlSend() function. */
    break;
case USB_DEVICE_EVENT_CONTROL_TRANSFER_DATA_RECEIVED:
    /* This event means that data has been received in the control transfer
     * and the application must either stall or acknowledge the data stage
     * by calling the USB_DEVICE_ControlStatus() function. Here we simply
     * acknowledge the received data. This is an example only. */
    USB_DEVICE_ControlStatus(appData.usbDevHandle, USB_DEVICE_CONTROL_STATUS_OK);
    break;
  
```

Endpoint Data Transfer Events

Describes endpoint data transfer events and provides a code example.

Description

The USB Device Layer provides notification events to indicate completion of transfers. These events are generated by the Device Layer and are made available in the Device Layer event handler. The `USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE` event occurs when a transfer scheduled by the `USB_DEVICE_EndpointRead` function has completed. The `USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE` event occurs when a transfer scheduled by the `USB_DEVICE_EndpointWrite` function has completed. The event data accompanying these events contains the transfer handle and number of bytes that were transferred.

The following code shows an example of handling these events.

```
/* The following code shows an example handling of the
 * endpoint transfer events. Here the code updates a transfer
 * pending flag indicating to the application that transfers have
 * completed. */
case USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE:
    /* Endpoint read is complete */
    appData.epDataReadPending = false;
    break;
case USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE:
    /* Endpoint write is complete */
    appData.epDataWritePending = false;
    break;
```

Endpoint Management

Describes how the application can enable and disable endpoints.

Description

Unlike standard USB function drivers, such as CDC, MSD, Audio, and HID, the Device Layer does not automatically manage endpoints for a Generic USB Device interface. This means that the application must maintain all endpoint that belong to a Generic USB Device Interface. Maintaining the endpoint involves the following:

- Enabling the endpoints for the desired transfer type when the host sets the configuration,
- Disabling the endpoint when the device receives a USB reset or when the Host changes the configuration
- Enabling and clearing endpoint stall conditions



The application should never access Endpoint 0 directly. Doing so may cause the Device Stack to malfunction, which could cause the USB device to be non-compliant.

Warning

Endpoints can be enabled or disabled with the `USB_DEVICE_EndpointEnable` and `USB_DEVICE_EndpointDisable` functions. The `USB_DEVICE_EndpointIsEnabled` function can be used to check if an endpoint is enabled. The application should enable the endpoint when host sets the configuration which contains interfaces that use the endpoint. The endpoints should otherwise be disabled. The endpoint function should not be called in the Device Layer event handler. Instead, they should be called in the application task routine. The following code shows an example of how an endpoint is enabled.

```
/* The following code shows an example of how the endpoint enable functions
 * are called to enable a Receive and Transmit Bulk endpoints. Note that the size
 * of the endpoint must be specified and this size should match the endpoint size
 * mentioned in the endpoint descriptor */
if (USB_DEVICE_EndpointIsEnabled(appData.usbDevHandle, appData.endpointRx) == false )
{
    /* Enable Read Endpoint */
    USB_DEVICE_EndpointEnable(appData.usbDevHandle, 0, appData.endpointRx,
                            USB_TRANSFER_TYPE_BULK, sizeof(receivedDataBuffer));
}
if (USB_DEVICE_EndpointIsEnabled(appData.usbDevHandle, appData.endpointTx) == false )
{
    /* Enable Write Endpoint */
    USB_DEVICE_EndpointEnable(appData.usbDevHandle, 0, appData.endpointTx,
                            USB_TRANSFER_TYPE_BULK, sizeof(transmitDataBuffer));
}
```

An endpoint should be disabled when the host has changed the device configuration and the new configuration does not contain any interfaces that use this endpoint. The endpoint can also be disabled when the application receives `USB_DEVICE_EVENT_RESET` or when the `USB_DEVICE_EVENT_DECONFIGURED` event has occurred. The following code shows an example of disabling the endpoint.

```
/* In this example, the endpoints are disabled when
```

```

* when the device has not been configured. This can happen
* if the configuration set is 0 or if the device is reset. */
if(!appData.deviceIsConfigured)
{
    /* This means the device got deconfigured. Change the
     * application state back to waiting for configuration. */
    appData.state = APP_STATE_WAIT_FOR_CONFIGURATION;

    /* Disable the endpoint*/
    USB_DEVICE_EndpointDisable(appData.usbDevHandle, appData.endpointRx);
    USB_DEVICE_EndpointDisable(appData.usbDevHandle, appData.endpointTx);
    appData.epDataReadPending = false;
    appData.epDataWritePending = false;
}

```

The application can use the [USB_DEVICE_EndpointStall](#) and [USB_DEVICE_EndpointStallClear](#) functions to enable stall and clear the stall on endpoints. The [USB_DEVICE_EndpointIsStalled](#) function can be called to check stall status of the endpoint.

Endpoint Data Transfer

Describes how the application can transfer data over endpoints.

Description

The application should call the [USB_DEVICE_EndpointRead](#) and [USB_DEVICE_EndpointWrite](#) functions to transfer data over an enabled endpoint. Calling this function causes a USB transfer to be scheduled on the endpoint. The transfer is added to the endpoint queue and is serviced as the host schedules the transaction on the bus. The [USB_DEVICE_EndpointRead](#) and [USB_DEVICE_EndpointWrite](#) functions return a unique transfer handle which can be tracked by the application. These transfer handles are returned along with the [USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE](#) (when an endpoint read transfer is complete) and [USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE](#) (when an endpoint write is complete) events.

The following code shows an example of sending data over an endpoint.

```

/* This code shows an example of using the USB_DEVICE_EndpointWrite
 * function to send data over the endpoint. The completion of the write is
 * indicated by the USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE event. The
 * transfer handle is returned in appData.writeTransferHandle */
USB_DEVICE_EndpointWrite ( appData.usbDevHandle, &appData.writeTransferHandle,
                           appData.endpointTx, &transmitDataBuffer[0], sizeof(transmitDataBuffer),
                           USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE );

void APP_USBDeviceEventHandler(USB_DEVICE_EVENT event, void * eventData, uintptr_t context)
{
    /* This is the Device Layer event handler */
    case USB_DEVICE_EVENT_ENDPOINT_WRITE_COMPLETE:
        /* Endpoint write is complete */
        appData.epDataWritePending = false;
        break;
}

```

The [USB_DEVICE_EndpointWrite](#) function allows the application to send data to the host without ending the transfer. This is done by specifying [USB_DEVICE_TRANSFER_FLAGS_DATA_PENDING](#) as the transfer flag in the call to the [USB_DEVICE_EndpointWrite](#) function. The application can use this option when the data to be sent is not readily available or when the application is memory constrained. The combination of the transfer flag and the transfer size affects how the data is sent to the host:

- If size is a multiple of maxPacketSize (the IN endpoint size) and flag is set as [USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE](#), the write function will append a Zero Length Packet (ZLP) to complete the transfer
- If size is a multiple of maxPacketSize and flag is set as [USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING](#), the write function will not append a ZLP and therefore will not complete the transfer
- If size is greater than but not a multiple of maxPacketSize and flags is set as [USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE](#), the write function schedules (length/maxPacketSize) packets and one packet for the residual data
- If size is greater than but not a multiple of maxPacketSize and flags is set as [USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING](#), the write function returns an error code and sets the transferHandle parameter to [USB_DEVICE_TRANSFER_HANDLE_INVALID](#)
- If size is less than maxPacketSize and flag is set [USB_DEVICE_TRANSFER_FLAGS_DATA_COMPLETE](#), the write function schedules one packet
- If size is less than maxPacketSize and flag is set as [USB_DEVICE_TRANSFER_FLAGS_MORE_DATA_PENDING](#), the write function returns an error code and sets the transferHandle parameter to [USB_DEVICE_TRANSFER_HANDLE_INVALID](#)

Refer to [USB_DEVICE_EndpointWrite](#) function API description for more details and code examples.

The application should use the [USB_DEVICE_EndpointRead](#) function to read data from an endpoint. The size of the buffer that is specified in this function should always be a multiple of the endpoint size. The following code shows an example of using the [USB_DEVICE_EndpointRead](#) function.

```

/* This code shows to use the USB_DEVICE_EndpointRead function

```

```

* to read from an endpoint. The transfer handle is returned in
* appData.readTransferHandle. The size of receivedDataBuffer should
* be a multiple of the receive endpoint size. */

USB_DEVICE_EndpointRead(appData.usbDevHandle, &appData.readTransferHandle,
    appData.endpointRx, &receivedDataBuffer[0], sizeof(receivedDataBuffer) );
void APP_USBDeviceEventHandler(USB_DEVICE_EVENT event, void * eventData, uintptr_t context)
{
    /* This is the Device Layer event handler */
    case USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE:
        /* Endpoint write is complete */
        appData.epDataReadPending = false;
        break;
}

```

In a case where a transfer is in progress, the [USB_DEVICE_EndpointRead](#) and [USB_DEVICE_EndpointWrite](#) functions can queue up transfers. The maximum number of read transfers that can queued (on any receive endpoint) is specified by the endpointQueueSizeRead member of the [USB_DEVICE_INIT](#) data structure. The maximum number of write transfers that can queued (on any transmit endpoint) is specified by the endpointQueueSizeWrite member of the [USB_DEVICE_INIT](#) data structure. The [USB_DEVICE_ENDPOINT_QUEUE_DEPTH_COMBINED](#) configuration macro should be set to total of read and write transfers that need to be queued.

For example, consider a Generic USB Device that contains two OUT (read) endpoint (EP1 and EP2) and one IN write endpoint (EP1). The application will queue a maximum of three read transfers on EP1, a maximum of five read transfers on EP2 and a maximum of four write transfers on EP1. Therefore, the total read transfer that will be queued in eight (3 + 5) and total write transfers that will be queued is four. The endpointQueueSizeRead member of the [USB_DEVICE_INIT](#) data structure should be set to eight. The endpointQueueSizeWrite member of the [USB_DEVICE_INIT](#) data structure should be set to four. The [USB_DEVICE_ENDPOINT_QUEUE_DEPTH_COMBINED](#) configuration macro should be set to 12 (8 + 4).

Configuring the Library

Describes how to configure the Generic USB Device Library.

Description

The application designer must specify the following configuration parameters while implementing the Generic USB Device. The configuration macros that implement these parameters must be located in the `system_config.h` file in the application project and a compiler include path (to point to the folder that contains this file) should be specified.

Configuration Macro Name	Description	Comments
USB_DEVICE_ENDPOINT_QUEUE_DEPTH_COMBINED	Size of buffer object pool for Endpoint Read and Endpoint Write functions.	This macro defines the total number of transfers that can be queued across all Generic USB Device endpoints. The number of read transfers that can be queued is specified by the endpointQueueSizeRead member of the USB_DEVICE_INIT data structure. The number of write transfers that can be queued is specified by the endpointQueueSizeWrite member of the USB_DEVICE_INIT data structure.

Building the Library

This section lists the files to be included in the project to implement a Generic USB Device Library.

Description

The Generic USB Device library does not have its own implementation. It is implemented using Device Layer API which are implemented in the Device Layer Files.

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/usb`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
usb_device.h	This header file should be included in any .c file that accesses the Device Layer API needed to implement the Generic USB Device.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/usb_device.c	This file contains the Device Layer API implementation.
/src/dynamic/usb_device_endpoint_functions.c	This file contains the endpoint transfer and management routines that are needed to implement the Generic USB Device.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The Generic USB Device Library depends on the following modules:

- [USB Device Layer Library](#)

Library Interface

The API for implementing the Generic USB Device is contained the USB Device Library. Please refer to the [Library Interface](#) section in the USB Device Layer Library for more details.

USB Host Library

This section provides information on the USB Host libraries that are available in MPLAB Harmony.

USB Host Library - Getting Started

This section provides information for getting started with the USB Host Library.

Introduction

Provides an introduction to the MPLAB Harmony USB Host Library

Description

The MPLAB Harmony USB Host Library (referred to as the USB Host Library) provides embedded application developers with a framework to design and develop USB Host Support for a wide variety of USB Device Classes. Low-Speed and Full-Speed USB Devices can be supported with PIC32MX microcontrollers. High-Speed devices can be supported with PIC32MZ microcontrollers. The USB Host Library facilitates support of standard USB devices through client drivers that implement standard the USB Device class specification. The library is modular, thus allowing application developers to readily support composite USB devices.

The USB Host Library is a part of the MPLAB Harmony installation and is accompanied by demonstration applications that highlight library usage. These demonstration applications can also be modified or updated to build custom applications. The USB Host Library also features the following:

- Class Driver Support (CDC, Audio, HID, and MSD)
- Designed to support USB devices with multiple configurations at different speeds
- Supports low-speed, full-speed and high-speed operation
- Supports multiple USB peripherals (allows multiple host stacks)
- Modular and Layered architecture
- Completely non-blocking
- Supports both polled and interrupt operation
- Works readily in an RTOS environment
- Designed to readily integrate with other Harmony Middleware

This document serves as a getting started guide and provides information on the following:

- USB Host Stack Architecture
- USB Host Library - Application Interaction

 **Note:** It is assumed that the reader is familiar with the USB 2.0 specification (available at www.usbif.org). While certain topics in USB may be discussed in this document, it is recommended that the reader refer to the specification documentation for a complete description.

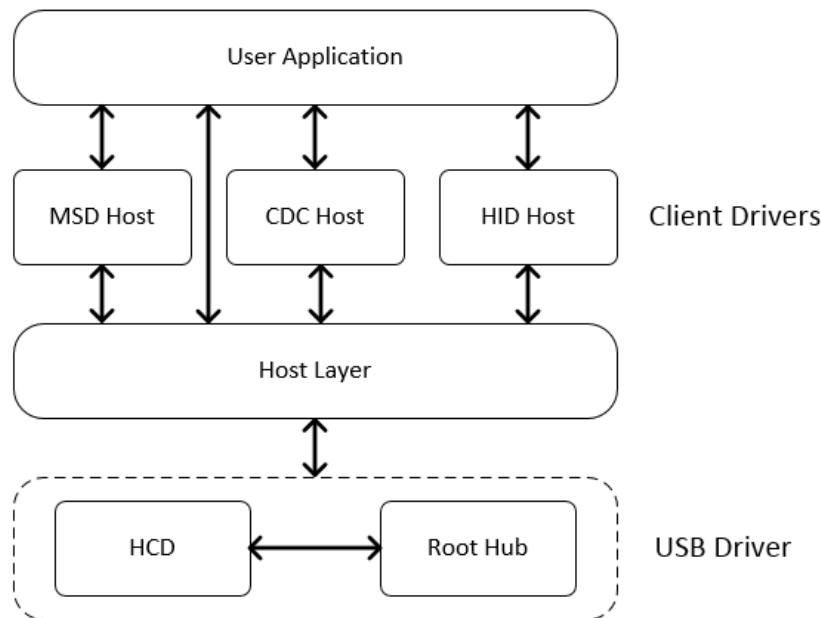
USB Host Library Architecture

Describes the USB Host Library Architecture.

Description

The USB Host Library Architecture features a modular and layered architecture as illustrated in the following figure.

USB Host Library Architecture



As seen in the figure, the USB Device Library consists of the following three major components.

Host Controller Driver (HCD)

The HCD manages the state of the USB peripheral and provides the Host Layer with structured methods to access data on the USB. The HCD is a MPLAB Harmony driver and uses the MPLAB Harmony framework components (USB Peripheral Library and the Interrupt System Service) of its operation. The HCD is initialized in the system initialization routine and its tasks routine is invoked in the system tasks routine. It is accessed exclusively by the Host layer. The HCD provides the following services to the host layer:

- Establish and manage communication pipes between the host layer and the attached devices
- Manage USB transfers

Root Hub Driver

The Root Hub Driver models the USB peripheral as a Hub. It then allows the Host Layer to perform the same actions on the Root Hub port that would be performed on an external Hub's port. The Root Hub Driver thus leads to an optimized implementation of Hub support in the Host Layer. The Root Hub Driver is hardware specific and is implemented as a part of the HCD. It provides the following services to the Host Layer

- Provides device attach and detach events
- Allows the Host to suspend, resume, and reset the port

The Root Hub Driver works in tandem with the HCD to provides the Host Layer with required USB protocol related means and methods to manage the attached USB device.

Host Layer

The Host Layer receives attach and detach events from the Root Hub Driver. It enumerates attached devices based on information contained in the Target Peripheral List (TPL). It allows client drivers to access the attached device through Host Layer methods. This includes allowing the client driver to set the device configuration. Where the client driver does not set the device configuration, the Host Layer will set the device configuration.

The Host layer opens the HCD, instantiates the Root Hub Driver, then controls and communicates with the attached device. The user application can call the Host Layer API to get information on attached devices. It can also register a Host Layer Event handler to get device related events. The user application can additionally suspend or resume a device. The Host Layer also provides bus level control where the application can suspend or resume all devices connected to a USB.

Client Driver

The USB Host Stack Client Drivers implement the support for different device classes as per the class specifications. Along with Host Layer, the client drivers are designed to support multiple device of the same type (where multiple devices are connected to the host through a hub or is a single device with multiple interfaces). A client driver abstracts intricate details of the class specification and provides a high level command and data interface to the application. Completion of requests is indicated by events. The application must register an event handler to receive these events.

The Client Driver may manage devices whose functionality is specified by USB VID and PID. In such cases, the client driver can set the device configuration. The client driver may manage a device whose functionality is defined by an interface class, subclass and protocol. In such a case, the configuration is set by the Host layer. The client driver can also manage devices whose functionality is defined by a combination of VID PID and class, sub-class and protocol.

USB Host Library - Application Interaction

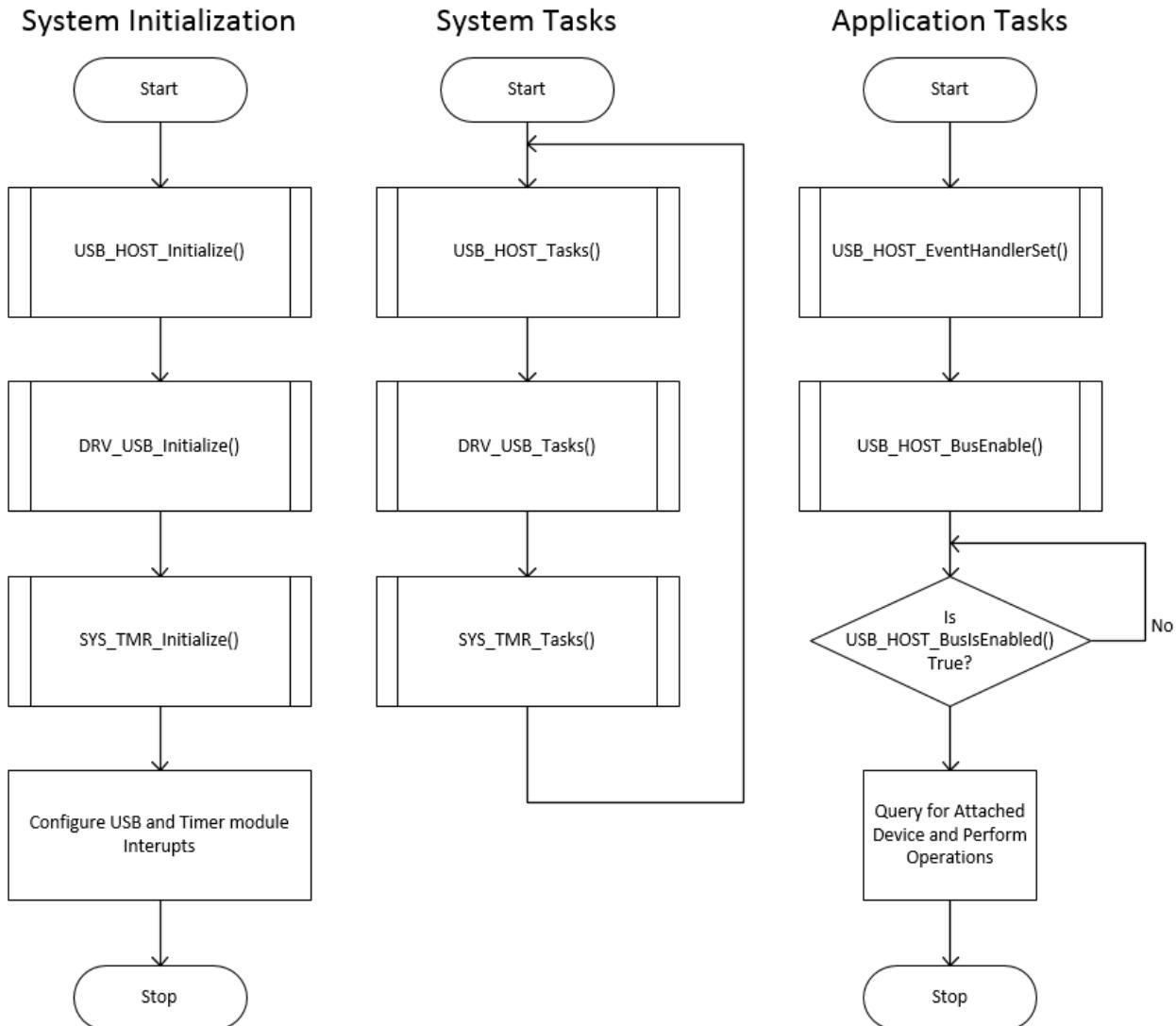
Describes how the application must interact with the USB Host Stack.

Description

 **Note:** Additional information on the tests conducted on Flash devices (i.e., Pen Drives) and a list of USB application configurations is available in the USB Demonstrations section.

The following figure highlights the steps that the application must follow to use the USB Host Library.

Application Interaction with Host Layer



The USB Host stack is initialized in the MPLAB Harmony System Initialization function. The Host Stack requires the Timer System Service and USB Driver. So these must be initialized as well. Note that the figure refers to a general USB Driver. The application may use the USBFS Driver (DRV_USBFS) for PIC32MX microcontroller or use the USBHS Driver (DRV_USBHS) for PIC32MZ microcontroller. The Timer and USB module interrupt priorities must be configured.

The USB Host layer, the USB Driver and the Timer System Service tasks must be called in the MPLAB Harmony System Tasks Routine. This ensures that the state machines of these module stays updated. If the USB Driver and the Timer driver have been configured for interrupt operation, then their corresponding interrupt tasks routines should be called in the corresponding module interrupt service routines.

The application state machine must first set the Host Layer event handler and then enable the bus. Enabling the bus will enable device detection and the Host Layer will enumerate attached devices. The application can query for attached devices and perform operations on attached devices.

USB Host Library Migration Guide

This section provides information on migrating from MPLAB Harmony v1.03.01 and earlier to the MPLAB Harmony USB Host Stack in MPLAB

Harmony v1.04 and later.

Introduction

Provides an introduction to migrating from older versions to v1.04 and later versions of MPLAB Harmony.

Description

The USB Host Stack API in MPLAB Harmony v1.04 has changed from that of previous versions of MPLAB Harmony.

USB CDC and MSD Host applications that were developed using the MPLAB Harmony USB Host Stack v1.03.01 and earlier, will not build unless the application calls to the USB Host Stack API is updated. While the MHC utility provides an option to continue creating USB Host applications using the v1.03.01 and earlier USB Host Stack API, it is recommended that existing USB Host application migrate to the latest USB Host API to take advantage of the latest features in the USB Host Stack. The following sections describe the API changes and other considerations while updating the application for changes in the USB Host Stack.

 **Note:** All USB Host Stack Demonstration Applications and USB Host Stack related documentation have been updated to the latest (new) USB Host API. The following sections do not discuss changes in the USB Host Stack configuration related code. This is updated automatically when the project is re-generated using MHC utility. Only the application related API changes are discussed.

USB Host Layer

Describes differences in the USB Host Library. Describes differences from earlier versions of MPLAB Harmony and the USB Host Layer Library in v1.04 of MPLAB Harmony.

Description

In MPLAB Harmony v1.03.01, the application was required to open the Host Layer to obtain a handle by calling the `USB_HOST_Open` function. This handle was then used along with other Host Layer API. Once opened, the application must enable Host Layer operation by calling the `USB_HOST_OperationEnable` function. The application must check the status of the enable operation function by calling the `USB_HOST_OperationIsEnabled` function. This is shown in the following code example.

Example:

```
/* This code shows an example application tasks that enabled Host
 * operation and waits for the enable operation to complete */

void APP_Tasks ( void )
{
    /* Check the application's current state. */
    USB_HOST_CDC_RESULT result;
    uint8_t temp;

    switch (appData.state)
    {
        case APP_STATE_OPEN_HOST_LAYER:

            /* Open the host layer and then enable Host layer operation */
            appData.hostHandle = USB_HOST_Open(USB_HOST_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);

            if (appData.hostHandle != USB_HOST_HANDLE_INVALID)
            {
                /* Host layer was opened successfully. Enable operation
                 * and then wait for operation to be enabled */

                USB_HOST_OperationEnable(appData.hostHandle );
                appData.state = APP_STATE_WAIT_FOR_HOST_ENABLE;

            }
            break;

        case APP_STATE_WAIT_FOR_HOST_ENABLE:

            /* Check if the host operation has been enabled */
            if(USB_HOST_OperationIsEnabled(appData.hostHandle))
            {
                /* This means host operation is enabled. We can
                 * move on to the next state */
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            }
    }
}
```

```

        break;

    default:
        break;
}
}

In MPLAB Harmony v1.04, the application is not required to open the Host Layer. This is because the Host Layer by itself does not support multi-client operation. The application must first set the Host Layer event handler using the USB\_HOST\_EventHandlerSet function and then enable the desired bus using the USB\_HOST\_BusEnable function. The completion of the Bus Enable function can be checked by calling the USB\_HOST\_BusIsEnabled function. Once enabled, the application can perform operations on the bus, look for attached devices, and perform operations on the attached device. The following code shows an example of enabling the bus.

/* This code shows an example of how the bus is enabled in the MPLAB Harmony v1.04
* USB Host Stack. The application state machine then waits for the bus enable
* operation to complete */

void APP_Tasks ( void )
{
    switch (appData.state)
    {
        case APP_STATE_BUS_ENABLE:

            /* In this state the application enables the USB Host Bus. Note
 * how the Host event handler are registered before the bus
 * is enabled. */

            USB_HOST_EventHandlerSet(APP_USBHostEventHandler, (uintptr_t)0);
            USB_HOST_BusEnable(0);
            appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:

            /* In this state we wait for the Bus enable to complete */
            if(USB_HOST_BusIsEnabled(0))
            {
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            }
            break;

        default:
            break;
    }
}
}

```

API Changes

The following table shows the legacy Host API and corresponding v1.04 MPLAB Harmony Host Stack API.

USB Host Layer API

MPLAB Harmony v1.03.01 and Earlier	MPLAB Harmony v1.04 and Later
USB_HOST_Open	N/A The USB Host Layer does not have to be opened.
USB_HOST_Close	N/A The USB Host Layer does not have to be closed.
USB_HOST_EventCallBackSet	USB_HOST_EventHandlerSet Events have changed.
USB_HOST_OperationEnable	USB_HOST_BusEnable Multiple busses can be managed.
USB_HOST_OperationDisable	N/A A bus disable function is not currently available and will be added in a future release of MPLAB Harmony.
USB_HOST_OperationIsEnabled	USB_HOST_BusIsEnabled

USB_HOST_DeviceSuspend	USB_HOST_DeviceSuspend Function parameters have changed.
USB_HOST_DeviceResume	USB_HOST_DeviceResume Function parameters have changed.
N/A	USB_HOST_BusSuspend Provides multiple bus support.
N/A	USB_HOST_BusResume Provides multiple bus support.
N/A	USB_HOST_DeviceGetFirst Provides multiple bus support.
N/A	USB_HOST_DeviceGetNext Provides multiple bus support.
N/A	USB_HOST_DeviceSpeedGet
N/A	USB_HOST_DeviceIsSuspended
N/A	USB_HOST_DeviceStringDescriptorGet

Event Changes

The type of events that the Host Layer generates in earlier versions of MPLAB Harmony and v1.04 of the MPLAB Harmony USB Host Stack have changed. The following table shows a comparison.

USB Host Layer Events

MPLAB Harmony v1.03.01 and Earlier	MPLAB Harmony v1.04 and Later
N/A	USB_HOST_EVENT_DEVICE_REJECTED_INSUFFICIENT_POWER
USB_HOST_EVENT_UNSUPPORTED_DEVICE	USB_HOST_EVENT_DEVICE_UNSUPPORTED
USB_HOST_EVENT_CANNOT_ENUMERATE	This event is not needed and is the same as an unsupported device.
USB_HOST_EVENT_CONFIGURATION_FAILED	This event is not needed and is the same as an unsupported device.
USB_HOST_EVENT_DEVICE_SUSPENDED	This event is not needed. A polling based function is available.
USB_HOST_EVENT_DEVICE_RESUMED	This event is not needed. A polling based function is available.
N/A	USB_HOST_EVENT_HUB_TIER_LEVEL_EXCEEDED
N/A	USB_HOST_EVENT_PORT_OVERCURRENT_DETECTED

USB MSD Host Client Driver and SCSI Block Storage Driver

Provides migration information for the MSD Host Client Driver and the SCSI Block Storage Driver.

Description

The application would use the MSD Host Client Driver and SCSI Block Storage Driver for accessing USB Storage Devices, such as USB Pen Drives. The key difference between the MSD Host Client Drivers in previous versions of the MPLAB Harmony USB Host Stack and the v1.04 MPLAB Harmony USB Host Stack MSD Host Client Drivers is the way the storage device attach and detach events are handled.

In previous versions (i.e., v1.03.01 or earlier) of USB Host Stack MSD Host Client Driver an application would have to register an event handler using the [USB_HOST_MSDEventHandlerSet](#) function after Host operation has been enabled. When the application would receive an event, [USB_HOST_MSDEVENT_ATTACH](#) would be detected, and the application would then try to mount the drive in this event. This is shown in the following code example.

Example:

```
/* In a v1.03.01 or earlier MSD Host Client Driver implementation, the driver will send an
 * attach/detach event to the application. The application must use this to mount or unmount
 * the drive in its main state machine */
```

```
bool APP_USBHostMSDEventHandler
(
    USB_HOST_MSDEVENT index,
    USB_HOST_MSDEVENT event,
    void* pData
)
```

```
switch ( event )
{
    case USB_HOST_MSD_EVENT_ATTACH:

        /* This means a USB storage device was plugged in. Update the
         * application state */
        appData.state = APP_STATE_DEVICE_CONNECTED;

        break;

    case USB_HOST_MSD_EVENT_DETACH:

        /* This means the USB storage was unplugged. Update the application
         * state */
        appData.state = APP_STATE_UNMOUNT_DISK;
        break;

    default:
        break;
}

return 0;
}

/* In the main application task routine, the host layer is opened and the host
 * operation is enabled. A MSD Host Client Driver event handler is registered
 * and disk is mounted when an attach event has been detected */

void APP_Tasks ( void )
{
    /* The application task state machine */

    switch(appData.state)
    {
        case APP_STATE_OPEN_HOST_LAYER:

            /* Open the host layer and then enable Host layer operation */
            appData.hostHandle = USB_HOST_Open(USB_HOST_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);

            if (appData.hostHandle != USB_HOST_HANDLE_INVALID)
            {
                /* Host layer was opened successfully. Enable operation
                 * and then wait for operation to be enabled */

                USB_HOST_OperationEnable(appData.hostHandle );
                appData.state = APP_STATE_WAIT_FOR_HOST_ENABLE;

            }
            break;

        case APP_STATE_WAIT_FOR_HOST_ENABLE:

            /* Check if the host operation has been enabled */
            if(USB_HOST_OperationIsEnabled(appData.hostHandle))
            {
                /* This means host operation is enabled. We can
                 * move on to the next state */

                USB_HOST_EventCallBackSet(appData.hostHandle,APP_USBHostEventHandler , 0 );
                USB_HOST_MSD_EventHandlerSet (APP_USBHostMSDEventHandler );
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            }

            break;

        case APP_STATE_WAIT_FOR_DEVICE_ATTACH:

            /* Wait for device attach. The state machine will move
```

```

        * to the next state when the attach event
        * is received. */

break;

case APP_STATE_DEVICE_CONNECTED:

    /* Device was connected. We can try mounting the disk */
    appData.state = APP_STATE_MOUNT_DISK;
    break;

case APP_STATE_MOUNT_DISK:

    /* The application gets into this state when the drive is attached
     * */

    if(SYS_FS_Mount("/dev/sdal", "/mnt/myDrive", FAT, 0, NULL) != 0)
    {
        /* The disk could not be mounted. Try
         * mounting again until success. */

        appData.state = APP_STATE_MOUNT_DISK;
    }
    else
    {
        /* Mount was successful. Try opening the file */
        appData.state = APP_STATE_OPEN_FILE;
    }
    break;

case APP_STATE_UNMOUNT_DISK:

    /* The application gets into this state when the drive is detached
     * */

    if(SYS_FS_Unmount("/mnt/myDrive") != 0)
    {
        /* The disk could not be unmounted. Try
         * unmounting again until success. */

        appData.state = APP_STATE_UNMOUNT_DISK;
    }
    else
    {
        /* Unmount was successful. Wait for device attach */
        appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
    }
    break;

default:
    break;
}
}

```

In the v1.04 USB Host Stack MSD Host Client Driver, the application must use the auto mount feature of the MPLAB Harmony File System. This feature should be enabled when the project is generated via MHC. The application then uses the MPLAB Harmony File System event handler to know when the File System has mounted the attached USB Storage Device. The application does not have to explicitly mount or unmount the drive. This is shown in the following code example.

Example:

```

/* In v1.04 USB MSD Host Client Driver, the driver uses the auto mount feature
 * of the MPLAB Harmony File System. The application must make sure that this
 * feature is enabled when the project is generated using MHC. Note that unlike
 * previous implement, here the USB Storage device attach/detach event appears
 * as a File System Event. */

void APP_SYSFSEventHandler(SYS_FS_EVENT event, void * eventData, uintptr_t context)
{
    switch(event)

```

```
{  
    case SYS_FS_EVENT_MOUNT:  
  
        /* The file system generates this event when the drive is mounted.  
         * This happens when the USB storage device is connected */  
        appData.deviceIsConnected = true;  
        break;  
  
    case SYS_FS_EVENT_UNMOUNT:  
  
        /* The file system generates this event when the drive is unmounted.  
         * This happens when the USB storage device is disconnected */  
        appData.deviceIsConnected = false;  
  
        break;  
  
    default:  
        break;  
}  
}  
  
/* This is the application state machine. Note how the application register the  
 * event handler with the File System before enabling the bus */  
  
void APP_Tasks ( void )  
{  
    switch(appData.state)  
    {  
        case APP_STATE_BUS_ENABLE:  
  
            /* Set the event handler and enable the bus */  
            SYS_FS_EventHandlerSet(APP_SYSFSEventHandler, (uintptr_t)NULL);  
            USB_HOST_EventHandlerSet(APP_USBHostEventHandler, 0);  
            USB_HOST_BusEnable(0);  
            appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;  
            break;  
  
        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:  
            if(USB_HOST_BusEnabled(0))  
            {  
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;  
            }  
            break;  
  
        case APP_STATE_WAIT_FOR_DEVICE_ATTACH:  
  
            /* Wait for device attach. The state machine will move  
             * to the next state when the attach event  
             * is received. */  
            if(appData.deviceIsConnected)  
            {  
                appData.state = APP_STATE_DEVICE_CONNECTED;  
            }  
  
            break;  
  
        case APP_STATE_DEVICE_CONNECTED:  
  
            /* Device was connected. We can try mounting the disk */  
            appData.state = APP_STATE_OPEN_FILE;  
            break;  
  
        case APP_STATE_IDLE:  
  
            /* The application reaches here after completing operation and waits  
             * for device detach and if detached, wait for attach. */  
            if(appData.deviceIsConnected == false)  
            {  
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;  
            }  
    }  
}
```

```

    }

    break;

default:
    break;
}

}

```

USB CDC Host Client Driver

Describes differences from earlier versions of MPLAB Harmony and the USB CDC Host Client Driver API in v1.04 of MPLAB Harmony.

Description

The key differences between the USB CDC Host Client Driver API in earlier versions of MPLAB Harmony and MPLAB Harmony v1.04 are:

- In the v1.04 release, the Device Attach event handler must be registered before enabling the bus and the General CDC Host Client Driver Event Handler is registered after the attached CDC device has been opened. In v1.03.01 and earlier, there was only one event handler which received all events and this event handler was to be registered after the Host Operation was enabled.
- In the v1.04 release, the application must open the attached CDC device. In v1.03.01 and earlier, the application does not have to open the attached CDC Device.

Registering Events

In v1.03.01 and earlier versions of the USB CDC Host Client Driver, the application registers the CDC Event Handler after the Host Operation was enabled. Only one event handler could be registered for all attached CDC Devices. The CDC Host Client Driver would specify the instance of the CDC Device generating the event when the event handler was invoked. The following code shows an example of this.

```
/* This code shows how a single CDC event handler is registered for all
 * instances of the CDC Host Client Driver. The instance of the client driver
 * generating this event is available in the index parameter of the event
 * handler. This code example is applicable to pre-v1.04 releases of the client
 * driver */
```

```
USB_HOST_CDC_EVENT_RESPONSE APP_USBHostCDCEventHandler
(
    USB_HOST_CDC_INDEX index,
    USB_HOST_CDC_EVENT event,
    void * eventData,
    uintptr_t context
)
{
    /* Get the application context */
    uint8_t deviceAddress;

    switch(event)
    {
        case USB_HOST_CDC_EVENT_ATTACH:
            /* The event data in this case is the address of the
             * attached device. */

            appData.state = APP_STATE_DEVICE_CONNECTED;
            deviceAddress = *((uint8_t *)eventData);
            break;

        case USB_HOST_CDC_EVENT_DETACH:
            /* This means the device was detached. There is no event data
             * associated with this event.*/

            appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            break;

        /* Other events not shown here for the sake of brevity */

        default:
            break;
    }

    return USB_HOST_CDC_EVENT_RESPONSE_NONE;
}
```

```

}

/* This is a pre v1.04 release application tasks for a CDC host client driver.
 * Note that application does not open the client driver and hence when
 * registering an event handler, is registering it all CDC Host Client Driver
 * instances. */

void APP_Tasks ( void )
{
    /* Check the application's current state. */
    USB_HOST_CDC_RESULT result;
    uint8_t temp;

    switch (appData.state)
    {
        case APP_STATE_OPEN_HOST_LAYER:

            /* Open the host layer and then enable Host layer operation */
            appData.hostHandle = USB_HOST_Open(USB_HOST_INDEX_0, DRV_IO_INTENT_EXCLUSIVE);

            if (appData.hostHandle != USB_HOST_HANDLE_INVALID)
            {
                /* Host layer was opened successfully. Enable operation
                 * and then wait for operation to be enabled */

                USB_HOST_OperationEnable(appData.hostHandle );
                appData.state = APP_STATE_WAIT_FOR_HOST_ENABLE;

            }
            break;

        case APP_STATE_WAIT_FOR_HOST_ENABLE:

            /* Check if the host operation has been enabled */
            if(USB_HOST_OperationIsEnabled(appData.hostHandle))
            {
                /* This means host operation is enabled. We can
                 * move on to the next state */
                USB_HOST_EventCallBackSet(appData.hostHandle,APP_USBHostEventHandler , 0 );
                USB_HOST_CDC_EventHandlerSet (APP_USBHostCDCEventHandler );
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            }

            break;

        case APP_STATE_WAIT_FOR_DEVICE_ATTACH:

            /* Wait for device attach. The state machine will move
             * to the next state when the USB_HOST_CDC_EVENT_ATTACH
             * is received. The application state is update in the
             * CDC Host event handler */

            break;

        default:
            break;
    }
}

```

In the v1.04 released version of the USB CDC Host Client Driver, the application must register an Attach Event Handler before enabling the bus operation. This one Attach Event handler will be invoked when even a CDC Device is attached. The application must use the CDC object returned in the attach event handler to open the device. The application can then register an event handler using the handle returned by the open function. This is shown in the code snippet here.

```

/* This code how event handling is performed in the v1.04 release of the
 * CDC Host Client Driver. The application must register a listener function
 * that check if the CDC Device is attached. When the attached device is opened
 * the application must then register event handler to register other CDC Host
 * Client Driver events */

```

```
void APP_USBHostCDCAttachEventListener(USB_HOST_CDC_OBJ cdcObj, uintptr_t context)
```

```
{  
    /* This function gets called when the CDC device is attached. Update the  
     * application data structure to let the application know that this device  
     * is attached */  
  
    appData.deviceIsAttached = true;  
    appData.cdcObj = cdcObj;  
}  
  
USB_HOST_CDC_EVENT_RESPONSE APP_USBHostCDCEventHandler  
(  
    USB_HOST_CDC_HANDLE cdcHandle,  
    USB_HOST_CDC_EVENT event,  
    void * eventData,  
    uintptr_t context  
)  
{  
    /* This function is called when a CDC Host event has occurred. A pointer to  
     * this function is registered after opening the device. See the call to  
     * USB_HOST_CDC_EventHandlerSet() function. */  
  
    switch(event)  
    {  
        case USB_HOST_CDC_EVENT_ACN_SET_LINE_CODING_COMPLETE:  
  
            /* This means the application requested Set Line Coding request is  
             * complete. */  
            break;  
  
        case USB_HOST_CDC_EVENT_ACN_SET_CONTROL_LINE_STATE_COMPLETE:  
  
            /* This means the application requested Set Control Line State  
             * request has completed. */  
            break;  
  
        case USB_HOST_CDC_EVENT_WRITE_COMPLETE:  
  
            /* This means an application requested write has completed */  
            break;  
  
        case USB_HOST_CDC_EVENT_READ_COMPLETE:  
  
            /* This means an application requested write has completed */  
            break;  
  
        case USB_HOST_CDC_EVENT_DEVICE_DETACHED:  
  
            /* The device was detached */  
            appData.deviceWasDetached = true;  
            break;  
  
        default:  
            break;  
    }  
  
    return(USB_HOST_CDC_EVENT_RESPONSE_NONE);  
}  
  
/* This is the example application task routine. The application sets the CDC  
 * attach listener function and then enables bus. When the CDC device is  
 * attached, the state machine opens the CDC Host Client Driver suing the object  
 * that was returned in the attach function. An event handler is then set with  
 * using the handle returned by the CDC Host Client Driver Open Function */  
  
void APP_Tasks ( void )  
{  
    /* Check the application's current state. */  
    USB_HOST_CDC_RESULT result;
```

```

if(appData.deviceWasDetached)
{
    /* This means the device is not attached. Reset the application state */

    appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
}

switch (appData.state)
{
    case APP_STATE_BUS_ENABLE:

        /* In this state the application enables the USB Host Bus. Note
         * how the CDC Attach event handler are registered before the bus
         * is enabled. */

        USB_HOST_EventHandlerSet(APP_USBHostEventHandler, (uintptr_t)0);
        USB_HOST_CDC_AttachEventHandlerSet(APP_USBHostCDCAttachEventListener, (uintptr_t) 0);
        USB_HOST_BusEnable(0);
        appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
        break;

    case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:

        /* In this state we wait for the Bus enable to complete */
        if(USB_HOST_BusIsEnabled(0))
        {
            appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
        }
        break;

    case APP_STATE_WAIT_FOR_DEVICE_ATTACH:

        /* In this state the application is waiting for the device to be
         * attached */
        if(appData.deviceIsAttached)
        {
            /* A device is attached. We can open this device */
            appData.state = APP_STATE_OPEN_DEVICE;
            appData.deviceIsAttached = false;
        }
        break;

    case APP_STATE_OPEN_DEVICE:

        /* In this state the application opens the attached device */
        appData.cdcHostHandle = USB_HOST_CDC_Open(appData.cdcObj);
        if(appData.cdcHostHandle != USB_HOST_CDC_HANDLE_INVALID)
        {
            /* The driver was opened successfully. Set the event handler
             * and then go to the next state. */
            USB_HOST_CDC_EventHandlerSet(appData.cdcHostHandle, APP_USBHostCDCEventHandler,
(uintptr_t)0);
            appData.state = APP_STATE_SET_LINE_CODING;
        }
        break;

    default:
        break;
}
}

```

API Differences

The following table shows the difference in API names between the pre v1.04 release and the v1.04 release versions of the CDC Host Client Driver.

MPLAB Harmony v1.03.01 and Earlier	MPLAB Harmony v1.04 and Later
USB_HOST_CDC_EventHandlerSet	USB_HOST_CDC_EventHandlerSet

USB_HOST_CDC_Read	USB_HOST_CDC_Read
USB_HOST_CDC_Write	USB_HOST_CDC_Write
USB_HOST_CDC_SerialStateNotificationGet	USB_HOST_CDC_SerialStateNotificationGet
USB_HOST_CDC_LineCodingGet	USB_HOST_CDC_ACM_LineCodingGet
USB_HOST_CDC_ControlLineStateSet	USB_HOST_CDC_ACM_ControlLineStateSet
USB_HOST_CDC_BreakSend	USB_HOST_CDC_ACM_BreakSend
Not needed.	USB_HOST_CDC_EventHandlerSet The attach event requires a separate event handler in v1.04.
Not available.	USB_HOST_CDC_DeviceObjHandleGet
Not available.	USB_HOST_CDC_Open The driver needs to be opened in v1.04.
Not available.	USB_HOST_CDC_Close

Event Name Differences

The following table shows the difference in event names between earlier versions and the MPLAB Harmony v1.04 release of the CDC Host Client Driver.

MPLAB Harmony v1.03.01 and Earlier	MPLAB Harmony v1.04 and Later
USB_HOST_CDC_EVENT_READ_COMPLETE	USB_HOST_CDC_EVENT_READ_COMPLETE
USB_HOST_CDC_EVENT_WRITE_COMPLETE	USB_HOST_CDC_EVENT_WRITE_COMPLETE
USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION_RECEIVED	USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION_RECEIVED
USB_HOST_CDC_EVENT_GET_LINE_CODING_COMPLETE	USB_HOST_CDC_EVENT_ACN_GET_LINE_CODING_COMPLETE
USB_HOST_CDC_EVENT_SET_LINE_CODING_COMPLETE	USB_HOST_CDC_EVENT_ACN_SET_LINE_CODING_COMPLETE
USB_HOST_CDC_EVENT_SET_CONTROL_LINE_STATE_COMPLETE	USB_HOST_CDC_EVENT_ACN_SET_CONTROL_LINE_STATE_COMPLETE
USB_HOST_CDC_EVENT_SEND_BREAK_COMPLETE	USB_HOST_CDC_EVENT_ACN_SEND_BREAK_COMPLETE
USB_HOST_CDC_EVENT_DETACH	USB_HOST_CDC_EVENT_DEVICE_DETACHED
USB_HOST_CDC_EVENT_ATTACH	Not needed. Refer to the USB_HOST_AttachEventHandlerSet function.

USB Host Layer Library

This section describes the USB Host Layer Library.

Introduction

Introduces the MPLAB Harmony USB Host Layer Library.

Description

The USB Host Layer in the MPLAB Harmony USB Host Stack performs the tasks of enumerating an attached device and interfacing the HCD. The following are the key features of the MPLAB Harmony USB Host Layer:

- Supports multi-configuration and composite USB Devices
- Supports VID PID and class, subclass, and protocol devices
- Can manage multiple USB devices through the Root Hub
- Concise API simplifies application development
- Modular architecture allows support for multiple (and different) USB controller in one application. Can operate multiple USB segments.
- Supports Low-Speed, Full-Speed, and Hi-Speed USB devices

Using the Library

This topic describes the basic architecture of the USB Host Layer and provides information and examples on its use.

Abstraction Model

Describes the abstraction model of the USB Host Layer.

Description

The USB Host Layer abstracts USB HCD hardware interaction details and presents an easy-to-use interface to the application and the client drivers. The Host Layer provides the application with a device object handle, which the application can use to suspend or resume the device. The Host Layer provides client drivers with device client handles and interface handles. These handles allow the client drivers to interact with the device and its interfaces. The Host Layer allows the client drivers to

- Open control pipes and schedule control transfers
- Open bulk, isochronous, and interrupt pipes
- Perform data transfers
- Claim and release ownership of the device and device interfaces
- Perform standard device operations.

The Host Layer has exclusive access to the HCD and the Root Hub. It opens the HCD and presents an abstracted interface to the application and client drivers.

Library Overview

The USB Host layer API is grouped functionally, as shown in the following table.

Library Interface Section	Description
System Interface Functions	These functions make the USB Host Layer compatible with MPLAB Harmony.
Bus Control Functions	These functions allow the application to enable, disable, suspend and resume the USB.
Device Related Functions	These functions allow the application to suspend and resume the USB. Attached devices can be queried and their string descriptors can be obtained.
Event Handling	Allows the application to register an event handler.
Client Driver Routines	These functions are exclusive to the client drivers and should not be accessed by the application.

How the Library Works

Describes how the Library works and how it should be used.

Description

The Host Layer in the MPLAB Harmony USB Host Stack plays the key role of enumerating an attached device and facilitating the communication between the USB Host Client Driver and the attached devices. The following sections describe the steps and methods that the user application must follow to use the Host Layer (and the USB Host stack). The following topics are discussed:

- Host Layer Initialization
- Operating the Host layer
- Host Layer Application Events

Host Layer Initialization

This topic describes how to initialize the Host Layer and includes code examples.

Description

The Host Layer must be initialized with relevant data to enable correct operation. This initialization must be performed in the SYS_Initialize function of the MPLAB Harmony application. The Host Layer will require the USB Controller Peripheral driver to be initialized for host mode operation (and hence operate as a HCD). This initialization must be performed in the SYS_Initialize function. The order in which the Host Layer and the USB Peripheral Driver are initialized does not affect the Host Layer operation. The Host Layer could be initialized before or after the USB Controller Peripheral Driver initialization.

The Host Layer requires the following information for initialization:

- The HCD interface for each bus

- The Target Peripheral List (TPL)

The Host Layer is capable of operating more than one USB device. This is possible on PIC32 microcontrollers that feature multiple USB Controller Peripherals. The one instance of the Host Layer manages multiple HCDs. The interface to each to every instance of the HCD that the Host Layer must operate must be specified in the Host Layer initialization. The total number of USB devices the Host Layer should manage is defined statically by the `USB_HOST_CONTROLLERS_NUMBER` configuration macro in the `system_config.h` file. The following code shows an example initialization of a PIC32MX USB HCD.

Example: PIC32MZ USB HCD Initialization

```
/* This code shows an example of how to initialize the PIC32MX USB
 * Driver for host mode operation. For more details on the PIC32MX Full-Speed
 * USB Driver, please refer to the Driver Libraries documentation. */

/* Include the full-speed USB driver header file */
#include "driver/usb/usbfs/drv_usbfs.h"

/* Create a driver initialization data structure */
DRV_USBFS_INIT drvUSBFSInit;

/* The PIC32MX Full-Speed USB Driver when operating in host mode requires an
 * endpoint table (a byte array) whose size should be 32 bytes. This table should
 * be aligned at 512 byte address boundary */
uint8_t __attribute__((aligned(512))) endpointTable[32];

/* Configure the driver initialization data structure */
DRV_USBFS_INIT drvUSBFSInit =
{
    /* This parameter should be set to SYS_MODULE_POWER_RUN_FULL. */
    .moduleInit = {SYS_MODULE_POWER_RUN_FULL},

    /* Driver operates in Host mode */
    .operationMode = USB_OPMODE_HOST,

    /* USB module interrupt source */
    .interruptSource = INT_SOURCE_USB_1,

    /* Continue operation when CPU is in Idle mode */
    .stopInIdle = false,

    /* Do not suspend operation when CPU enters Sleep mode */
    .suspendInSleep = false,

    /* The USB module index */
    .usbID = USB_ID_1,

    /* The maximum current that the VBUS supply can provide */
    .rootHubAvailableCurrent = 500,

    /* Pointer to the endpoint table */
    .endpointTable = endpointTable,

    /* Pointer to the Port Power Enable function. Driver will cause this
     * function when the port power must be enabled */
    .portPowerEnable = PortPowerEnable,

    /* Pointer to the Port Over Current Detect function. Driver will cause this
     * function periodically to check if the port current has exceeded limit */
    .portOverCurrentDetect = PortOverCurrentDetect,

    /* Pointer to the Port LED indication function. The driver will call this
     * function to update the Port LED status */
    .portIndication = PortIndication
};

/* USB Driver system module object */
SYS_MODULE_OBJ drvUSBObj = SYS_MODULE_OBJ_INVALID;

void SYS_Initialize(void * data)
{
```

```

/* Initialize the driver */
drvUSBObj = DRV_USBFS_Initialize(DRV_USBFS_INDEX_0, (SYS_MODULE_INIT *)(&drvUSBFSInit));
}

void SYS_Tasks(void)
{
    /* Call the driver tasks routine in SYS_Tasks() function */
    DRV_USBFS_Tasks(drvUSBObj);
}

void __ISR(_USB_1_VECTOR, ipl4AUTO) _IntHandlerUSBInstance0(void)
{
    /* Call the driver interrupt tasks routine in the USB module ISR */
    DRV_USBFS_Tasks_ISR(sysObj.drvUSBModuleObj);
}

```

The Host Layer Initialization requires a **USB_HOST_HCD** data structure. This data structure specifies the HCD module index and the HCD Host Layer Interface for each bus. The following code shows the **USB_HOST_HCD** data structure is initialized for a single USB Controller Peripheral PIC32MX microcontroller device.

Example: Data Structure Initialized for a Single USB Controller Peripheral PIC32MX MCU

```

/* This code shows an example of setting up the USB_HOST_HCD data
 * structure for the PIC32MX USB controller */

USB_HOST_HCD usbHostHCD =
{
    /* This is the driver instance index that the USB Host Layer will use */
    .drvIndex = DRV_USBFS_INDEX_0,

    /* This is the interface to the PIC32MX USB HCD. The
     * DRV_USBHS_HOST_INTERFACE pointer is exported by the PIC32MX Host Mode USB
     * Driver. */
    .hcdInterface = DRV_USBHS_HOST_INTERFACE
};

```

The other important component required for USB Host Layer initialization is the Target Peripheral List (TPL). Embedded USB Hosts unlike standard USB Host are not expected to support all USB Device Types. The device types to be supported are specified in the TPL. The TPL contains an entry for every device type that the Embedded USB host must support. If the attached device matches the criteria specified in the TPL entry , the Host Layer attaches the driver corresponding to that entry to the manage device. A device may match multiple entries in the TPL. This happens in the case of composite devices.

An entry in the TPL contains the following information:

- Device Type: This specifies whether the Host must inspect the VID, PID field or Class, Subclass and Protocol fields while matching the attached device to the entry
- Flags: These flags provide the system designer with various options while matching the attached device to a driver. For example, a flag can be specified to ignore the device PID and only consider the VID while matching VID PID device.
- PID Mask: This is a PID mask that can be applied to the PID before matching the PID to the attached device PID
- Driver: This is the pointer to the interface of the client driver that should manage the device if the matching criteria is met

The following code shows an example TPL table.

Example: TPL Table

```

/* This code shows some examples of configuring the USB Host Layer
 * TPL Table. In this example, the USB Host layer is configured to support
 * three different types of devices. */

USB_HOST_TARGET_PERIPHERAL_LIST usbHostTPL[4] =
{

    /* Catch every device with the exact Vendor ID = 0x04D9 and Product ID = 0x0001.
     * Every other device will not load this driver. */
    TPL_DEVICE_VID_PID( 0x04D9, 0x0001, &driverInitData, &DEVICE_DRIVER_EXAMPLE1_Driver ),

    /* This driver will catch any device with the Vendor ID of 0x04D9 and any
     * product ID = 0x0000 or 0x0002-0x00FF. The entry in the TPL before this
     * caught the Product ID = 0x0001 case so that is why it is not caught by
     * this entry. Those devices have already been caught. */
    TPL_DEVICE_VID_PID_MASKED( 0x04D9, 0x0002, 0xFF00, &driverInitData, &DEVICE_DRIVER_EXAMPLE2_Driver
),

    /* This entry will catch all other devices. */
    TPL_DEVICE_ANY( &driverInitData, &DEVICE_DRIVER_EXAMPLE3_Driver ),
}

```

```

/* This entry will catch only a HID boot keyboard. All other devices,
 * including other HID keyboards that are non-boot, will be skipped by this
 * entry. This driver will handle only this specific case. */
TPL_INTERFACE_CLASS_SUBCLASS_PROTOCOL( USB_HID_CLASS_CODE, USB_HID_SUBCLASS_CODE_BOOT_INTERFACE,
                                       USB_HID_PROTOCOL_CODE_KEYBOARD, &hidDriverInitData,
                                       USB_HOST_HID_BOOT_KEYBOARD_DRIVER ),

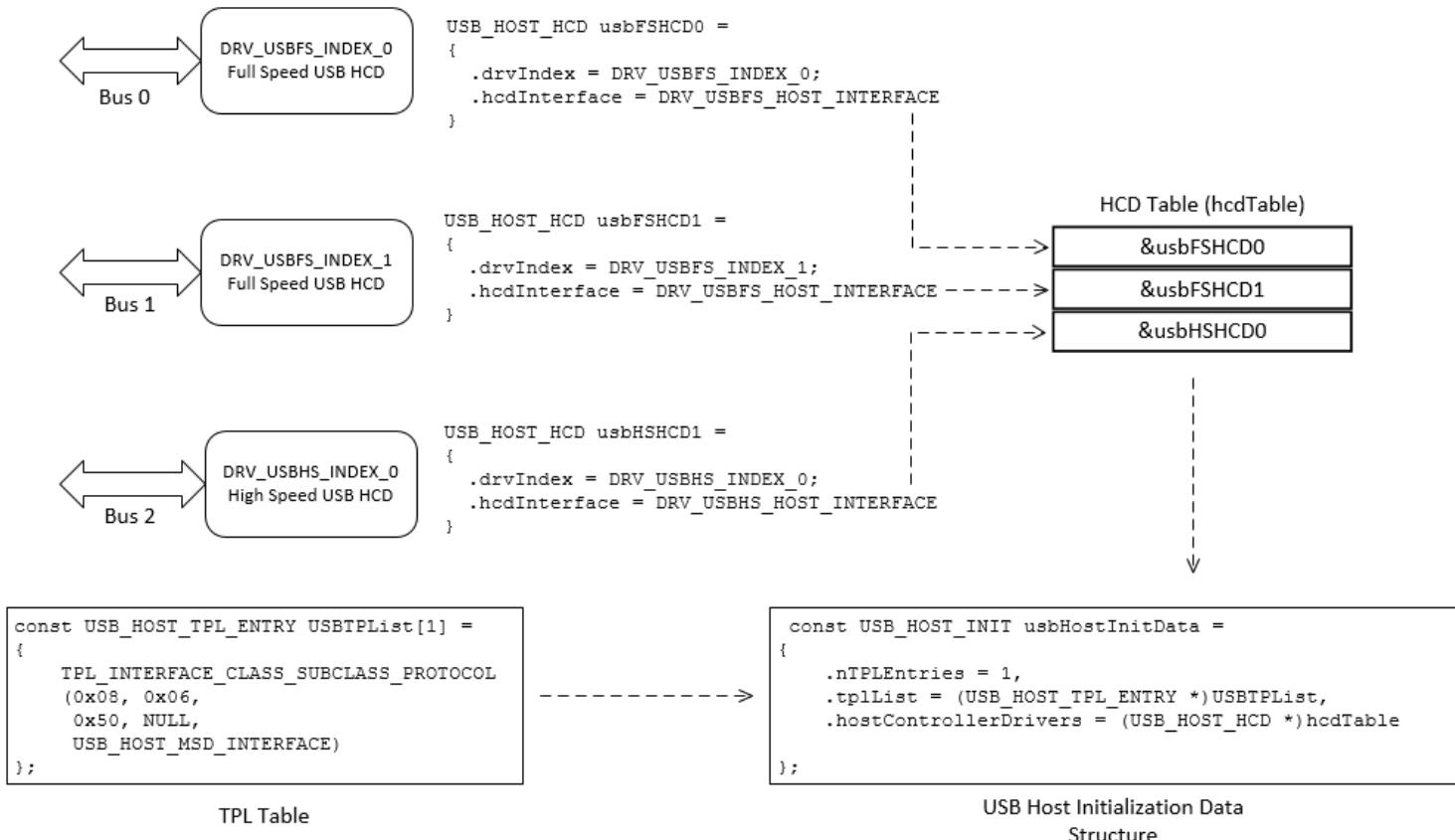
/* This entry will catch all CDC-ACM devices. It filters on the class and
 * subclass but ignores the protocol since the driver will handle all
 * possible protocol options. */
TPL_INTERFACE_CLASS_SUBCLASS( USB_CDC_CLASS_CODE, USB_CDC_SUBCLASS_CODE_ABSTRACT_CONTROL_MODEL,
                             &cdcDriverInitData, USB_HOST_CDC_ACN_DRIVER ),

/* This will catch all instances of the MSD class regardless subclass or
 * protocol. In this case the driver will sort out if it supports the
 * device or not. */
TPL_INTERFACE_CLASS( USB_MSD_CLASS_CODE, &msdDriverInitData, USB_HOST_MSD_DRIVER ),

/* Any unclaimed interfaces can be sent to a particular driver if desired.
 * This can be used to create a similar mechanism that libUSB or WinUSB
 * provides on a PC where any unused interface can be opened and utilized by
 * these drivers. */
TPL_INTERFACE_ANY( &driverInitData, USB_HOST_VENDOR_DRIVER )
}

```

The Host Layer can now be initialized. The following code shows how the [USB_HOST_HCD](#) and the TPL table are specified in the [USB_HOST_INIT](#) (the Host Layer Initialization) data structure. In addition, the following figure illustrates the various initialization inputs needed by the Host Layer.



The [USB_HOST_Initialize](#) function is called to initialize the Host Layer. The initialization process may not complete when the [USB_HOST_Initialization](#) function exits. This will complete in subsequent calls to the [USB_HOST_Tasks](#) function.

Example: Specifying the TPL Table

```

/* This code shows an example of the USB Host Layer Initialization data
 * structure. In this case the number of TPL entries is one and there is only
 * one HCD (and hence only one USB bus) in the application */

const USB_HOST_TPL_ENTRY USBTPLList[1] =
{

```

```

/* This is the TPL */
TPL_INTERFACE_CLASS_SUBCLASS_PROTOCOL(0x08, 0x06, 0x50, NULL, USB_HOST_MSD_INTERFACE)

};

const USB_HOST_HCD hcdTable =
{
    /* The HCD table only contains one entry */
    .drvIndex = DRV_USBFS_INDEX_0,
    .hcdInterface = DRV_USBFS_HOST_INTERFACE
};

const USB_HOST_INIT usbHostInitData =
{
    /* This is the Host Layer Initialization data structure */
    .nTPLEntries = 1,
    .tplList = (USB_HOST_TPL_ENTRY *)USBTPLList,
    .hostControllerDrivers = (USB_HOST_HCD *)&hcdTable
};

}

```

Host Layer - Application Interaction

This topic describes application interaction with the USB Host Layer.

Description

The Host Layer in the MPLAB Harmony USB Host stack provides the user application with API methods to operate the USB Host. The following sections discuss these API methods.

Registering the Event Handler

The application must register an event handler to receive device related USB Host events. The application sets the events handler by using the [USB_HOST_EventHandlerSet](#) function. An application defined context can also be provided. This context is returned along with the event handler and helps the application to identify the context in case of a dynamic application use cases. The host layer provides events when a connected device requires more current than can be provided or when a unsupported device was attached. The following code shows an example of registering the event handler.

```

/* This code shows an example of registering an event handler with the
 * Host Layer */

USB_HOST_EVENT_RESPONSE APP_USBHostEventHandler
(
    USB_HOST_EVENT event,
    void * eventData,
    uintptr_t context
)
{
    /* This is the event handler implementation */
    switch (event)
    {
        case USB_HOST_EVENT_DEVICE_UNSUPPORTED:
            break;
        case USB_HOST_EVENT_DEVICE_REJECTED_INSUFFICIENT_POWER:
            break;
        case USB_HOST_EVENT_HUB_TIER_LEVEL_EXCEEDED:
            break;
        case USB_HOST_EVENT_PORT_OVERCURRENT_DETECTED:
            break;
        default:
            break;
    }

    return(USB_HOST_EVENT_RESPONSE_NONE);
}

void APP_Tasks(void)
{
    /* This shows an example app state machine implementation in which the event
     * handler is set and the bus is then enabled. */
}

```

```

switch(appData.state)
{
    case APP_STATE_BUS_ENABLE:

        /* Set the event handler and enable the bus */
        USB_HOST_EventHandlerSet(APP_USBHostEventHandler, 0);
        USB_HOST_BusEnable(0);
        appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
        break;

    default:
        break;
}
}

```

Enabling the Bus

The user application must call the [USB_HOST_BusEnable](#) function to enable the bus. This function enables the 5V VBUS supply to root hub port thus powering up the bus powered device that are attached to the bus. The attached devices will then indicate attach. The root hub will provide these attach events to the Host layer which in turn starts the enumeration process. The application can call other Host Layer functions only after the bus has been enabled. The [USB_HOST_BusIsEnabled](#) function must be called to check if the enable process has completed. The following code shows an example application state machine that enables the bus.

```

void APP_Tasks ( void )
{
    /* The application shows an example of how the USB bus is enabled and how the
     * application must wait for the bus to enabled */

    switch(appData.state)
    {
        case APP_STATE_BUS_ENABLE:

            /* Set the event handler and enable the bus */
            USB_HOST_EventHandlerSet(APP_USBHostEventHandler, 0);
            USB_HOST_BusEnable(0);
            appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:
            /* Check if the bus is enabled */
            if(USB_HOST_BusIsEnabled(0))
            {
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            }
            break;

        default:
            break;
    }
}

```

Attached Device Information

The application can use the [USB_HOST_DeviceFirstGet](#) and the [USB_HOST_DeviceNextGet](#) function to query for attached devices. The [USB_HOST_DeviceFirstGet](#) function will provide information on the first device that was attached to the bus. Information is returned in application specified [USB_HOST_DEVICE_INFO](#) object. The [USB_HOST_DeviceFirstGet](#) function will return the following information in the [USB_HOST_DEVICE_INFO](#) object:

- A Device Object Handle of the type [USB_HOST_DEVICE_OBJ_HANDLE](#). The application can use this device object handle to perform operations on the device.
- The address of the device on the USB
- The bus to which this device belongs

The application can access the contents of the [USB_HOST_DEVICE_INFO](#) object but should not alter it contents. The same object is passed to the [USB_HOST_DeviceNextGet](#) function to get the information about the next device attached on the bus. Each call to this function defines the point at which the [USB_HOST_DeviceNextGet](#) function will start searching. If the device that is represented by the [USB_HOST_DEVICE_INFO](#) object has been disconnected, calling the [USB_HOST_DeviceNextGet](#) function will return an error. The search must be reset by calling the [USB_HOST_DeviceFirstGet](#) function. The application can define multiple [USB_HOST_DEVICE_INFO](#) objects to search on different busses or maintain different search points.

```

void APP_Tasks(void)
{
}

```

```

USB_HOST_DEVICE_INFO deviceInfo;
USB_HOST_RESULT result;

/* Get information about the first device on Bus 0 */
result = USB_HOST_DeviceGetFirst(0, &deviceInfo);

while(result != USB_HOST_RESULT_END_OF_DEVICE_LIST)
{
    /* deviceInfo.address has the address of the bus */
    /* deviceInfo.deviceObjHandle will have the device object handle */

    /* Now we can get the information about the next device on the bus. */
    result = USB_HOST_DeviceGetNext(&deviceInfo);
}

}

```

Suspend and Resume

The USB Host Layer allows the application to suspend and resume a device. The [USB_HOST_DeviceSuspend](#) and the [USB_HOST_DeviceResume](#) function are provided for this purpose. The application must use the device object handles, obtained from the [USB_HOST_DeviceFirstGet](#) or [USB_HOST_DeviceNextGet](#) function, to specify the device to suspend or resume when calling [USB_HOST_DeviceSuspend](#) and the [USB_HOST_DeviceResume\(\)](#) function. The [USB_HOST_DeviceIsSuspended](#) function can be called to check the suspend status of the device.

In a case where the entire bus (and hence all device connected on the bus) need to be suspended or resumed, the application must call [USB_HOST_BusSuspend](#) and [USB_HOST_BusResume](#) functions to suspend or resume the entire bus. The [USB_HOST_BusIsSuspended](#) function can be called to check the suspend status of the bus.

Device String Descriptors

The application may want to obtain the string descriptors of a device. String descriptors are optionally provided by the USB device manufacturer and provide device information. The [USB_HOST_DeviceStringDescriptorGet](#) function is available to read the string descriptors. Calling this function will cause the Host Layer to invoke a control transfer request to read the string descriptor. The string descriptor will be available when the control transfer completes. The host layer calls the [USB_HOST_STRING_REQUEST_COMPLETE_CALLBACK](#) type callback function, that is provided in the [USB_HOST_DeviceStringDescriptorGet](#) function, when the control transfer has completed. The completion status of the request and the size of the string descriptor are available in the callback.

The function allows the application to obtain the supported string language IDs. The language ID of the string can be specified or a default can be used.

```

typedef struct
{
    /* This is an application specific data structure */
    char string[APP_STRING_SIZE];
    USB_HOST_REQUEST_HANDLE requestHandle;
    uintptr_t context;
} APP_DATA;

APP_DATA appData;

void APP_USBHostStringDescriptorGetCallBack
(
    USB_HOST_REQUEST_HANDLE requestHandle,
    size_t size,
    uintptr_t context
)
{
    /* This function is called when the string descriptor get function has
     * completed. */

    if(size != 0)
    {
        /* This means the function executed successfully and we have a string.
         * An application function prints the string to the console. */
        APP_PrintStringToConsole(appData.string, size);
    }
}

void APP_Tasks(void)
{
    USB_HOST_DEVICE_INFO deviceInfo;

```

```

USB_HOST_RESULT result;

/* Get information about the first device on Bus 0 */
result = USB_HOST_DeviceGetFirst(0, &deviceInfo);

if(result != USB_HOST_RESULT_END_OF_DEVICE_LIST)
{
    /* deviceInfo.deviceObjHandle will have the device object handle. Use
     * this device object handle along with the
     * USB_HOST_DeviceStringDescriptorGet() function to read the product
     * string ID using the default Language ID. */

    USB_HOST_DeviceStringDescriptorGet(deviceInfo.deviceObjHandle, USB_HOST_DEVICE_STRING_PRODUCT,
                                       USB_HOST_DEVICE_STRING_LANG_ID_DEFAULT, appData.string, APP_STRING_SIZE,
                                       &appData.requestHandle, APP_USBHostSrингDescriptorGetCallBack, appData.context );
}
}
}

```

Event Handling

This topic describes event handling.

Description

The USB Host Layer provides general device related events to the application. The application must register an event handling function by using the [USB_HOST_EventHandlerSet](#) function. A context specified at the time of calling this function, is returned in the event handler. The event handler must be registered before the bus is enabled. Refer to the description of [USB_HOST_DEVICE_EVENT](#) events for details on the available events.

Configuring the Library

Describes how to configure the USB Host Layer.

Macros

	Name	Description
	USB_HOST_CONTROLLERS_NUMBER	Defines the number of USB Host Controllers that this Host Layer must manage.
	USB_HOST_DEVICE_INTERFACES_NUMBER	Defines the maximum number of interface that the attached device can contain in order for the USB Host Layer to process the device.
	USB_HOST_DEVICES_NUMBER	Defines the maximum number of devices to support.
	USB_HOST_HUB_SUPPORT_ENABLE	Defines if this USB Host application must support a Hub.
	USB_HOST_HUB_TIER_LEVEL	Defines the maximum tier of connected hubs to be supported.
	USB_HOST_PIPES_NUMBER	Defines the maximum number of pipes that the application will need.
	USB_HOST_TRANSFERS_NUMBER	Defines the maximum number of transfers that host layer should handle.

Description

The following configuration parameters must be defined while using the USB Host Layer. The configuration macros that implement these parameters must be located in the `system_config.h` file in the application project and a compiler include path (to point to the folder that contains this file) should be specified.

USB_HOST_CONTROLLERS_NUMBER Macro

Defines the number of USB Host Controllers that this Host Layer must manage.

File

[usb_host_config_template.h](#)

C

```
#define USB_HOST_CONTROLLERS_NUMBER
```

Description

USB Host Layer Controller Numbers

This constant defines the number of USB Host Controllers that this Host Layer must manage. The value of this constant should be atleast 1. Typical embedded applicatons contains only 1 USB host controller and hence only 1 USB. A microcontroller that features multiple USB modules can support multiple USB Host controllers and multipel USBs. USB controllers can also be interfaced to the microcontroller through common

communication peripherals such as SPI.

This constant also defines the number of entries in the Host Controller Driver interface table, a pointer to which is passed in the hostControllerDrivers member of the [USB_HOST_INIT](#) data structure.

Remarks

None.

USB_HOST_DEVICE_INTERFACES_NUMBER Macro

Defines the maximum number of interface that the attached device can contain in order for the USB Host Layer to process the device.

File

[usb_host_config_template.h](#)

C

```
#define USB_HOST_DEVICE_INTERFACES_NUMBER
```

Description

USB Host Device Interface Numbers

This constant defines the maximum number of interface that the attached device can contain in order for the USB Host Layer to process the device. The device will be processed if it only contains less interfaces than the value of this constant.

Remarks

Supporting more interface per device required more processing time and data memory.

Example

An attached device contains a configuration that contains 10 interfaces, but the USB_HOST_DEVICE_INTERFACES_NUMBER is set to 5. The device will not be processed by the Host Layer. A dual CDC device needs to be supported. This device will have 4 interfaces. The USB_HOST_DEVICE_INTERFACES_NUMBER constant should be atleast 4.

USB_HOST_DEVICES_NUMBER Macro

Defines the maximum number of devices to support.

File

[usb_host_config_template.h](#)

C

```
#define USB_HOST_DEVICES_NUMBER
```

Description

USB Host Layer Devices Number

This configuration constant defines the maximum number of devices that this USB Host application must support. The value of this constant should be atleast 1. Multiple devices can be supported if Hub support is enabled. See [USB_HOST_HUB_SUPPORT_ENABLE](#). The Hub itself will be treated as a device.

Remarks

Supporting multiple devices requires more data memory and processing time.

Example

If the USB Host application must support one USB Pen Drive and one USB Serial COM port (CDC Device), then this constant should be set to 3 (one additional device will be the Hub).

USB_HOST_HUB_SUPPORT_ENABLE Macro

Defines if this USB Host application must support a Hub.

File

[usb_host_config_template.h](#)

C

```
#define USB_HOST_HUB_SUPPORT_ENABLE
```

Description

USB Host Layer Hub Support

Specifying this macro will enable Hub Support. The HUB tier level to be supported is then specified by [USB_HOST_HUB_TIER_LEVEL](#) constant. If this macro is specified, the file `usb_host_hub.c` must be included in the application.

Remarks

Not specifying this macro will disable Hub support.

USB_HOST_HUB_TIER_LEVEL Macro

Defines the maximum tier of connected hubs to be supported.

File

[usb_host_config_template.h](#)

C

```
#define USB_HOST_HUB_TIER_LEVEL
```

Description

USB Host Hub Tier Level

This constant defines the maximum hub tiers to be supported by the USB Host application. This constant is considered only if the [USB_HOST_HUB_SUPPORT_ENABLE](#) option is specified. If specified, the value should be atleast 1. This means that one hub should be supported. In a case where another hub will be connected to the hub which is connected to the USB Host, the value should be 2. As per the USB specification the maximum number of non-root hub tiers can be 5. Hence the value of this configuration constant should not exceed 5.

Remarks

None.

USB_HOST_PIPES_NUMBER Macro

Defines the maximum number of pipes that the application will need.

File

[usb_host_config_template.h](#)

C

```
#define USB_HOST_PIPES_NUMBER
```

Description

USB Host Layer Pipes Number

This configuration constant defines the maximum number of device communication pipes that Host Layer would need in the application. Every attached device requires atleast one pipe. This pipe is the control transfer pipe. Additional pipes are needed based on the type of device. For example, a standard Mass Storage Class device will need 2 pipes, a Communication Class Device will need 3 pipes, a HID device will need atleast 1 pipe. Vendor device will need communication pipe based on the device implementation. The number of pipes must also take into account the number of devices to support.

Remarks

This number should match the number of pipes configured in the HCD that this application will use, that is `DRV_USBHS_HOST_PIPES_NUMBER` or `DRV_USBFS_HOST_PIPES_NUMBER`.

Example

If a USB Host application must support 2 devices, either 2 USB pen driver or 2 CDC devices or a mix of both, then the pipes number should be set to

8. This is because the 2 CDC devices connected to the host will pose larger pipe requirement. Two such devices will require 2 control pipes, 2 interrupt pipes, 2 Bulk IN and 2 Bulk OUT pipes.

USB_HOST_TRANSFERS_NUMBER Macro

Defines the maximum number of transfers that host layer should handle.

File

[usb_host_config_template.h](#)

C

```
#define USB_HOST_TRANSFERS_NUMBER
```

Description

USB Host Layer Transfers Number

This constant defines the maximum number of transfers that the host layer should handle. The choice of this constant depends on the nature of devices that the USB Host application must support. Atleast two transfers are needed per pipe in the system. If the number of transfers provisioned in the system are insufficient, the USB Host will decline transfer request citing a busy status. This will affect the speed performance of the system.

Remarks

None.

Example

A USB Host application will support an MSD device and a CDC Device. The total number of pipes needed in the system are 7. The USB_HOST_TRANSFERS_NUMBER constant should be atleast 14 (2 per pipe). Specifying a larger number will enable more transfers to be queued but will also require more data memory.

Building the Library

Describes the files to be included in the project while using the USB Host Layer Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/usb.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
usb_host.h	This header file should be included in any .c file that accesses the USB Host Layer API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/usb_host.c	This file implements the USB Host Layer interface and should be included in the project if USB Host mode operation is desired.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB Host Layer Library depends on the following modules:

- USB Driver Library (Host mode files)
- Timer System Service Library

Library Interface**a) Functions**

	Name	Description
≡	USB_HOST_Deinitialize	Deinitializes the specified instance of the USB Host Layer.
≡	USB_HOST_Status	Gets the current status of the USB Host Layer.
≡	USB_HOST_Tasks	Maintains the USB Host Layer state machine.
≡	USB_HOST_BusEnable	Starts host operations.
≡	USB_HOST_BusIsEnabled	Checks if the bus is enabled.
≡	USB_HOST_BusIsSuspended	Returns the suspend status of the bus.
≡	USB_HOST_BusResume	Resumes the bus.
≡	USB_HOST_BusSuspend	Suspends the bus.
≡	USB_HOST_DeviceGetFirst	Returns information about the first attached device on the bus.
≡	USB_HOST_DeviceGetNext	Returns information about the next device on the bus.
≡	USB_HOST_DeviceIsSuspended	Returns the suspend state of the device is suspended.
≡	USB_HOST_DeviceResume	Resumes the selected device
≡	USB_HOST_DeviceSpeedGet	Returns the speed at which this device is operating.
≡	USB_HOST_DeviceStringDescriptorGet	Retrieves specified string descriptor from the device
≡	USB_HOST_DeviceSuspend	Suspends the specified device.
≡	USB_HOST_EventHandlerSet	USB Host Layer Event Handler Callback Function set function.
≡	USB_HOST_Initialize	Initializes the USB Host layer instance specified by the index.

b) Data Types and Constants

	Name	Description
	USB_HOST_INIT	Defines the data required to initialize a USB Host Layer instance.
	USB_HOST_EVENT_RESPONSE	Host Layer Events Handler Function Response Type.
	\.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL)	USB Host Layer TPL Table Entry Matching Criteria flag
	\.tplFlags.driverType = (TPL_FLAG_VID_PID)	USB Host Layer TPL Table Entry Matching Criteria flag
	0	USB Host Layer TPL Table Entry Matching Criteria flag
	0x0000	USB Host Layer TPL Table Entry Matching Criteria flag
	0xFF	USB Host Layer TPL Table Entry Matching Criteria flag
	0xFF }	USB Host Layer TPL Table Entry Matching Criteria flag
	0xFFFF	USB Host Layer TPL Table Entry Matching Criteria flag
	0xFFFF }	USB Host Layer TPL Table Entry Matching Criteria flag
	1	USB Host Layer TPL Table Entry Matching Criteria flag
	classCode	USB Host Layer TPL Table Entry Matching Criteria flag
	false	USB Host Layer TPL Table Entry Matching Criteria flag
	initData	USB Host Layer TPL Table Entry Matching Criteria flag
	mask	USB Host Layer TPL Table Entry Matching Criteria flag
	pid	USB Host Layer TPL Table Entry Matching Criteria flag
	pid }	USB Host Layer TPL Table Entry Matching Criteria flag
	subClassCode	USB Host Layer TPL Table Entry Matching Criteria flag
	true	USB Host Layer TPL Table Entry Matching Criteria flag
	USB_HOST_BUS	Defines a USB Bus Data Type.
	USB_HOST_DEVICE_INFO	Defines the data type that is used by the USB_HOST_DeviceGetFirst() and USB_HOST_DeviceGetNext() functions.
	USB_HOST_DEVICE_OBJ_HANDLE	Handle to an attached USB Device.
	USB_HOST_DEVICE_STRING	Defines a defines types of strings that can be request through the USB_HOST_DeviceStringDescriptorGet() function.
	USB_HOST_EVENT	Defines the different events that the USB Host Layer can generate.
	USB_HOST_EVENT_HANDLER	USB Host Layer Event Handler Function Pointer Type
	USB_HOST_HCD	Defines the USB Host HCD Information object that is provided to the host layer.

USB_HOST_REQUEST_HANDLE	USB Host Request Handle Type
USB_HOST_RESULT	USB Host Results.
USB_HOST_STRING_REQUEST_COMPLETE_CALLBACK	USB Host Device String Descriptor Request Complete Callback Function Type
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY	USB Host Layer TPL Table Entry Matching Criteria flag
USB_HOST_TPL_ENTRY	USB Host Layer TPL Table Entry Matching Criteria flag
vid	USB Host Layer TPL Table Entry Matching Criteria flag
USB_HOST_BUS_ALL	USB Host Bus All
USB_HOST_DEVICE_OBJ_HANDLE_INVALID	Defines an invalid USB Device Object Handle.
USB_HOST_DEVICE_STRING_LANG_ID_DEFAULT	Defines the default Lang ID to be used while obtaining the string.
USB_HOST_REQUEST_HANDLE_INVALID	USB Host Request Invalid Handle
USB_HOST_RESULT_MIN	USB Host Result Minimum Constant.

Description

This section describes the Application Programming Interface (API) functions of the USB Host Layer Library.

Refer to each section for a detailed description.

a) Functions

[USB_HOST_Deinitialize Function](#)

Deinitializes the specified instance of the USB Host Layer.

File

[usb_host.h](#)

C

```
void USB_HOST_Deinitialize(SYS_MODULE_OBJ hostLayerObject);
```

Returns

None.

Description

Deinitializes the USB Host Layer. All internal data structures will be reset.

Remarks

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again. This routine will NEVER block waiting for hardware.

Preconditions

Function [USB_HOST_Initialize](#) should have been called before calling this function.

Example

```
SYS_MODULE_OBJ object;      // Returned from USB_HOST_Initialize
SYS_STATUS status;

USB_HOST_Deinitialize(object);

status = USB_HOST_Status(object);
if (SYS_MODULE_DEINITIALIZED != status)
{
    // Can check again later if you need to know
    // when the driver is deinitialized.
}
```

Parameters

Parameters	Description
object	USB Host layer object handle, returned from the USB_HOST_Initialize routine

Function

```
void USB_HOST_Deinitialize( SYS_MODULE_OBJ object )
```

USB_HOST_Status Function

Gets the current status of the USB Host Layer.

File

[usb_host.h](#)

C

```
SYS_STATUS USB_HOST_Status( SYS_MODULE_OBJ hostLayerObject );
```

Returns

SYS_STATUS_READY - Indicates that the USB Host layer is ready for operations.

SYS_STATUS_BUSY - The initialization is in progress.

SYS_STATUS_DEINITIALIZED - Indicates that the driver has been deinitialized

Description

This routine provides the current status of the USB Host Layer.

Remarks

This function is typically called by the MPLAB Harmony System to check the system status of the USB Host Layer. This function is not intended to be called directly by the application tasks routine.

Preconditions

Function [USB_HOST_Initialize](#) should have been called before calling this function.

Example

```
SYS_MODULE_OBJ          object;      // Returned from USB_HOST_Initialize
SYS_STATUS              status;

status = USB_HOST_Status(object);
if (SYS_STATUS_READY == status)
{
    // The USB Host system is ready and is running.
}
```

Parameters

Parameters	Description
object	USB Host Layer object handle, returned from the USB_HOST_Initialize routine

Function

```
SYS_STATUS USB_HOST_Status( SYS_MODULE_OBJ object )
```

USB_HOST_Tasks Function

Maintains the USB Host Layer state machine.

File

[usb_host.h](#)

C

```
void USB_HOST_Tasks( SYS_MODULE_OBJ hostLayerObject );
```

Returns

None.

Description

This routine maintains the USB Host layer's state machine. It must be called frequently to ensure proper operation of the USB. This function should be called from the [SYS_Tasks](#) function.

Remarks

This routine is not intended to be called directly by an application. It is called by the MPLAB Harmony System Tasks function.

Preconditions

The [USB_HOST_Initialize](#) routine must have been called for the specified USB Host Layer instance.

Example

```
SYS_MODULE_OBJ object;      // Returned from USB_HOST_Initialize

void SYS_Tasks(void)
{
    USB_HOST_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the specified driver instance (returned from USB_HOST_Initialize)

Function

```
void USB_HOST_Tasks (SYS_MODULE_OBJ object);
```

USB_HOST_BusEnable Function

Starts host operations.

File

[usb_host.h](#)

C

```
USB_HOST_RESULT USB_HOST_BusEnable(USB_HOST_BUS bus);
```

Returns

USB_HOST_RESULT_SUCCESS if the request was accepted. USB_HOST_RESULT_BUS_UNKNOWN if the specified bus is invalid (it does not exist in the system). USB_HOST_RESULT_FAILURE if an unknown failure occurred.

Description

The function starts the operation of the USB Host Bus. It enables the root hub associated with specified bus and starts the process of detecting attached devices and enumerating them. The [USB_HOST_EventHandlerSet\(\)](#) function should have been called to register an application host layer event handler before the bus is enabled (before the [USB_HOST_BusEnable\(\)](#) function is called). This will ensure that the application does not miss any events.

Remarks

The Host Layer may generate events after the [USB_HOST_BusEnable\(\)](#) function is called. The application should have registered an event handler using the [USB_HOST_EventHandlerSet\(\)](#) function to handle these events. The [USB_HOST_EventHandlerSet\(\)](#) function should have been called before the [USB_HOST_BusEnable\(\)](#) function is called.

Preconditions

The [USB_HOST_Initialize\(\)](#) function should have been called before calling this function.

Example

TBD.

Parameters

Parameters	Description
bus	the bus to be enabled. If this is set to USB_HOST_BUS_ALL , all buses will be enabled.

Function

```
USB_HOST_RESULT USB_HOST_BusEnable(USB_HOST_BUS bus)
```

USB_HOST_BusIsEnabled Function

Checks if the bus is enabled.

File

[usb_host.h](#)

C

```
USB_HOST_RESULT USB_HOST_BusIsEnabled(USB_HOST_BUS bus);
```

Returns

USB_HOST_RESULT_TRUE if the bus is enabled. USB_HOST_RESULT_FALSE if the bus is not enabled..
USB_HOST_RESULT_BUS_UNKNOWN if the specified bus is invalid (it does not exist in the system). USB_HOST_RESULT_FAILURE if an unknown failure occurred.

Description

The function returns the enable status of the bus. It can be called after the [USB_HOST_BusEnable\(\)](#) function is called, to check if the bus has been enabled yet. If the bus parameter is set to [USB_HOST_BUS_ALL](#), then the function will check the enable status of all the busses and will return true only if all the busses are enabled.

Remarks

None.

Preconditions

The [USB_HOST_Initialize\(\)](#) function should have been called before calling this function.

Example

TBD.

Parameters

Parameters	Description
bus	the bus that needs to checked for enable status. If this is set to USB_HOST_BUS_ALL , all buses will be checked.

Function

```
USB_HOST_RESULT USB_HOST_BusIsEnabled(USB_HOST_BUS bus)
```

USB_HOST_BusIsSuspended Function

Returns the suspend status of the bus.

File

[usb_host.h](#)

C

```
USB_HOST_RESULT USB_HOST_BusIsSuspended(USB_HOST_BUS bus);
```

Returns

USB_HOST_RESULT_TRUE - if the bus is suspended. USB_HOST_RESULT_FALSE - if the bus is not suspended.
USB_HOST_RESULT_BUS_NOT_ENABLED - if the bus was not enabled. USB_HOST_RESULT_BUS_UNKNOWN - if the specified bus does not exist in the system. USB_HOST_RESULT_FAILURE - an unknown error occurred.

Description

This function returns suspend status of the specified USB bus. This function can be used to check the completion of the Suspend operation started by using the [USB_HOST_BusSuspend\(\)](#) function. The function would return USB_HOST_RESULT_FALSE if the bus is not suspended. Calling the [USB_HOST_BusIsSuspended\(\)](#) with bus specified as [USB_HOST_BUS_ALL](#) returns the suspend status of the all USB segments that are managed by the host layer. The function would return USB_HOST_RESULT_TRUE only if all the bus are in a suspended state.

Remarks

None.

Preconditions

The [USB_HOST_BusEnable\(\)](#) function should have been called to enable the bus.

Example

TBD.

Parameters

Parameters	Description
bus	the bus whose suspend status is to be queried.

Function

[USB_HOST_RESULT USB_HOST_BusIsSuspended \(USB_HOST_BUS bus\)](#)

USB_HOST_BusResume Function

Resumes the bus.

File

[usb_host.h](#)

C

```
USB_HOST_RESULT USB_HOST_BusResume(USB_HOST_BUS bus);
```

Returns

[USB_HOST_RESULT_SUCCESS](#) - if the request was successful or if the bus was already resumed. [USB_HOST_RESULT_BUS_UNKNOWN](#) - the request failed because the bus does not exist in the system. [USB_HOST_RESULT_BUS_NOT_ENABLED](#) - the bus was not enabled. [USB_HOST_RESULT_FAILURE](#) - An unknown error occurred.

Description

The function resumes the bus. All devices on the bus will be receive resume signaling. If bus is specified as [USB_HOST_BUS_ALL](#), all the buses managed by this host will be resumed.

Remarks

None.

Preconditions

The [USB_HOST_BusEnable\(\)](#) function should have been called to enable the bus.

Example

```
// Resume bus 0
USB_HOST_BusResume(0);

// Resume all buses
USB_HOST_BusSuspend(USB_HOST_BUS_ALL);
```

Parameters

Parameters	Description
bus	The bus to be resume or USB_HOST_BUS_ALL to resume all buses.

Function

[USB_HOST_RESULT USB_HOST_BusResume \(USB_HOST_BUS bus\);](#)

USB_HOST_BusSuspend Function

Suspends the bus.

File

[usb_host.h](#)

C

```
USB_HOST_RESULT USB_HOST_BusSuspend(USB_HOST_BUS bus);
```

Returns

`USB_HOST_RESULT_SUCCESS` - if the request was successful. `USB_HOST_RESULT_BUS_NOT_ENABLED` - if the bus was not enabled.
`USB_HOST_RESULT_FAILURE` - An unknown error has occurred. `USB_HOST_RESULT_BUS_UNKNOWN` - if the specified bus does not exist in the system.

Description

The function suspends the bus. All devices on the bus will be suspended. If bus is specified as `USB_HOST_BUS_ALL`, all the buses managed by this host will be suspended.

Remarks

None.

Preconditions

The `USB_HOST_BusEnable()` function should have been called to enable the bus.

Example

```
// Suspend the bus 0
USB_HOST_BusSuspend(0);

// Suspend all buses
USB_HOST_BusSuspend(USB_HOST_BUS_ALL);
```

Parameters

Parameters	Description
bus	The bus to be suspended or <code>USB_HOST_BUS_ALL</code> to suspend all buses.

Function

```
USB_HOST_RESULT USB_HOST_BusSuspend (USB_HOST_BUS bus);
```

USB_HOST_DeviceGetFirst Function

Returns information about the first attached device on the bus.

File

`usb_host.h`

C

```
USB_HOST_RESULT USB_HOST_DeviceGetFirst(USB_HOST_BUS bus, USB_HOST_DEVICE_INFO * deviceInfo);
```

Returns

`USB_HOST_RESULT_SUCCESS` - The function executed successfully. `USB_HOST_RESULT_END_OF_DEVICE_LIST` - There are no attached devices on the bus. `USB_HOST_RESULT_BUS_UNKNOWN` - The specified bus does not exist in the system.
`USB_HOST_RESULT_BUS_NOT_ENABLED` - The specified bus is not enabled. `USB_HOST_RESULT_PARAMETER_INVALID` - the deviceInfo parameter is NULL. `USB_HOST_RESULT_FAILURE` - an unknown failure occurred.

Description

This function returns information about the first attached device on the specified bus. The `USB_HOST_DeviceGetNext()` function can be used to get the reference to the next attached device on the bus. The `USB_HOST_DEVICE_INFO` object is provided by the application. The device information will be populated into this object. If there are no devices attached on the bus, the function will set the deviceObjHandle parameter, in the `USB_HOST_DEVICE_INFO` object, to `USB_HOST_DEVICE_OBJ_HANDLE_INVALID`.

Remarks

None.

Preconditions

The `USB_HOST_BusEnable` function should have been called to enable detection of attached devices.

Example

TBD.

Parameters

Parameters	Description
bus	the bus to be queried for attached devices.
deviceInfo	output parameter. Will contain device information when the function returns. If the deviceObjHandle member of the structure contains USB_HOST_DEVICE_OBJ_HANDLE_INVALID , then there are no attached devices on the bus and the deviceAddress and the bus member of the info object will contain indeterminate values.

Function

```
USB_HOST_RESULT USB_HOST_DeviceGetFirst
(
    USB_HOST_BUS bus,
    USB_HOST_DEVICE_INFO * deviceInfo
);
```

USB_HOST_DeviceGetNext Function

Returns information about the next device on the bus.

File

[usb_host.h](#)

C

```
USB_HOST_RESULT USB_HOST_DeviceGetNext(USB_HOST_DEVICE_INFO * deviceInfo);
```

Returns

USB_HOST_RESULT_SUCCESS - The function executed successfully.
 USB_HOST_RESULT_END_OF_DEVICE_LIST - There are no attached devices on the bus.
 USB_HOST_RESULT_PARAMETER_INVALID - the deviceInfo parameter is NULL.
 USB_HOST_RESULT_DEVICE_UNKNOWN - the device specified in deviceInfo does not exist in the system. The search should be restarted.
 USB_HOST_RESULT_FAILURE - an unknown failure occurred. Application can restart the search by calling the [USB_HOST_DeviceGetFirst\(\)](#) function.

Description

This function returns information of the next device attached on the bus. The [USB_HOST_DeviceGetFirst\(\)](#) function should have been called at least once on the deviceInfo object. Then calling this function repeatedly on the deviceInfo object will return information about the next attached device on the bus. When there are no more attached devices to report, the function returns [USB_HOST_RESULT_END_OF_DEVICE_LIST](#).

Calling the [USB_HOST_DeviceGetFirst\(\)](#) function on the deviceInfo object after the [USB_HOST_DeviceGetNext\(\)](#) function has been called will cause the host to reset the deviceInfo object to point to the first attached device.

Remarks

None.

Preconditions

The [USB_HOST_DeviceGetFirst\(\)](#) function must have been called before calling this function.

Example

TBD.

Parameters

Parameters	Description
deviceInfo	pointer to the USB_HOST_DEVICE_INFO object.

Function

```
USB_HOST_RESULT USB_HOST_DeviceGetNext (USB_HOST_DEVICE_INFO * deviceInfo);
```

USB_HOST_DeviceIsSuspended Function

Returns the suspend state of the device is suspended.

File

[usb_host.h](#)

C

```
USB_HOST_RESULT USB_HOST_DeviceIsSuspended(USB_HOST_DEVICE_OBJ_HANDLE deviceObjHandle);
```

Returns

USB_HOST_RESULT_TRUE - if the device is suspended. USB_HOST_RESULT_FALSE - if the device is not suspended.
 USB_HOST_RESULT_DEVICE_UNKNOWN - the specified device does not exist in the system. USB_HOST_RESULT_FAILURE - An unknown failure occurred.

Description

This function returns the suspend state of the specified USB device. This function can be used to check the completion of the Resume operation started by using the `USB_HOST_Resume()` function. If the Resume signaling has completed, the `USB_HOST_IsSuspended()` function would return `USB_HOST_RESULT_TRUE`.

Remarks

None.

Preconditions

The `USB_HOST_BusEnable()` function should have been called.

Example

TBD.

Parameters

Parameters	Description
deviceObjHandle	handle to the device that needs to be checked for suspend status.

Function

```
USB_HOST_RESULT USB_HOST_DeviceIsSuspended
(
  USB_HOST_DEVICE_OBJ_HANDLE deviceObjHandle
);
```

USB_HOST_DeviceResume Function

Resumes the selected device

File

`usb_host.h`

C

```
USB_HOST_RESULT USB_HOST_DeviceResume(USB_HOST_DEVICE_OBJ_HANDLE deviceObjHandle);
```

Returns

USB_HOST_RESULT_SUCCESS - The request was accepted and the device will be resumed or the device was already resumed.
 USB_HOST_RESULT_DEVICE_UNKNOWN - The request failed. The device may have been detached. USB_HOST_RESULT_FAILURE - An unknown failure occurred.

Description

The function resumes the selected device. A device can be resumed only if it was suspended.

Remarks

None.

Preconditions

None.

Example

TBD.

Parameters

Parameters	Description
deviceObjHandle	handle to the device to be resumed.

Function

```
USB_HOST_RESULT USB_HOST_DeviceResume
(
    USB_HOST_DEVICE_OBJ_HANDLE deviceObjHandle
);
```

USB_HOST_DeviceSpeedGet Function

Returns the speed at which this device is operating.

File

[usb_host.h](#)

C

```
USB_HOST_RESULT USB_HOST_DeviceSpeedGet(USB_HOST_DEVICE_OBJ_HANDLE deviceHandle, USB_SPEED * speed);
```

Returns

USB_HOST_RESULT_SUCCESS - The function was successful. speed will contain the speed of the device.

USB_HOST_RESULT_DEVICE_UNKNOWN - The device does not exist in the system. speed will contain USB_SPEED_ERROR.

USB_HOST_RESULT_FAILURE - an unknown error occurred.

Description

This function returns the speed at which this device is operating.

Remarks

None.

Preconditions

The [USB_HOST_Initialize\(\)](#) function should have been called.

Example

Parameters

Parameters	Description
deviceObjHandle	handle to the device whose speed is required.
speed	output parameter. Will contain the speed of the device if the function was successful.

Function

```
USB_HOST_RESULT USB_HOST_DeviceSpeedGet
(
    USB_HOST_DEVICE_OBJ_HANDLE deviceObjHandle,
    USB_SPEED * speed
)
```

USB_HOST_DeviceStringDescriptorGet Function

Retrieves specified string descriptor from the device

File

[usb_host.h](#)

C

```
USB_HOST_RESULT USB_HOST_DeviceStringDescriptorGet(USB_HOST_DEVICE_OBJ_HANDLE deviceObjHandle,
USB_HOST_DEVICE_STRING stringType, uint16_t languageID, void * stringDescriptor, size_t length,
USB_HOST_REQUEST_HANDLE * requestHandle, USB_HOST_STRING_REQUEST_COMPLETE_CALLBACK callback, uintptr_t context);
```

Returns

`USB_HOST_RESULT_SUCCESS` - The request was scheduled successfully. `requestHandle` will contain a valid request handle.
`USB_HOST_RESULT_DEVICE_UNKNOWN` - The request failed because the device was detached. `USB_HOST_RESULT_FAILURE` - An unknown error occurred. `USB_HOST_RESULT_REQUEST_BUSY` - The host layer cannot take more requests at this point. The application should try later. `USB_HOST_RESULT_STRING_DESCRIPTOR_UNSUPPORTED` - The device does not support the specified string descriptor type.

Description

This function retrieves the specified string descriptor from the device. This function will cause the host layer to issue a control transfer to the device. When the string descriptor is available, the host layer will call the callback function to let the application know that the request has completed.

The function will return a valid request handle in `requestHandle`, if the request was successful. This request handle will be returned in the callback function. The size of the `stringDescriptor` buffer is specified by the `length` parameter. Only length number of bytes will be retrieved. The type of device string descriptor to be retrieved is specified by the `stringType` parameter. The supported language IDs, manufacturer, product and serial number strings can be obtained. While obtaining the supported language IDs, the `languageID` parameter will be ignored.

Remarks

None.

Preconditions

The `USB_HOST_BusEnable()` function should have been called.

Example

Parameters

Parameters	Description
<code>deviceObjHandle</code>	handle to the device whose string descriptor is to be retrieved.
<code>stringType</code>	type of string descriptor to be retrieved
<code>languageID</code>	the language ID of the string descriptor
<code>stringDescriptor</code>	output buffer for the descriptor
<code>length</code>	size of the specified output buffer
<code>requestHandle</code>	This is an output parameter. It will contain a valid request handle if the request was successful. It will contain <code>USB_HOST_REQUEST_HANDLE_INVALID</code> if the request was not successful.
<code>callback</code>	Function that will be called when this request completes. If this is NULL, then the application will not receive indication of completion.
<code>context</code>	Calling application context to be returned in the callback function.

Function

```
USB_HOST_RESULT USB_HOST_DeviceStringDescriptorGet
(
    USB_HOST_DEVICE_OBJ_HANDLE deviceObjHandle,
    USB_HOST_DEVICE_STRING stringType,
    uint16_t languageID,
    void * stringDescriptor,
    size_t length,
    USB_HOST_REQUEST_HANDLE * requestHandle,
    USB_HOST_STRING_REQUEST_COMPLETE_CALLBACK callback,
    uintptr_t context
);
```

USB_HOST_DeviceSuspend Function

Suspends the specified device.

File

`usb_host.h`

C

```
USB_HOST_RESULT USB_HOST_DeviceSuspend(USB_HOST_DEVICE_OBJ_HANDLE deviceObjHandle);
```

Returns

USB_HOST_RESULT_SUCCESS - The request was accepted and the device will be suspended. USB_HOST_RESULT_DEVICE_UNKNOWN - The request failed. The device may have been detached. USB_HOST_RESULT_FAILURE - An unknown failure occurred.

Description

The function suspends the specified device.

Remarks

None.

Preconditions

The [USB_HOST_BusEnable\(\)](#) function should have been called.

Example

TBD.

Parameters

Parameters	Description
deviceObjHandle	handle to the device to suspend.

Function

```
USB_HOST_RESULT USB_HOST_DeviceSuspend
(
    USB_HOST_DEVICE_OBJ_HANDLE deviceObjHandle
);
```

USB_HOST_EventHandlerSet Function

USB Host Layer Event Handler Callback Function set function.

File

[usb_host.h](#)

C

```
USB_HOST_RESULT USB_HOST_EventHandlerSet(USB_HOST_EVENT_HANDLER eventHandler, uintptr_t context);
```

Returns

USB_HOST_RESULT_SUCCESS - The function was successful. USB_HOST_RESULT_FAILURE - An unknown failure occurred.

Description

This is the USB Host Layer Event Handler Callback Set function. An application can receive USB Host Layer events by using this function to register and event handler callback function. The application can additionally specify a specific context which will be returned with the event handler callback function. The event handler must be set (this function must be called) before any of the USB buses are enabled.

Remarks

None.

Preconditions

The host layer should have been initialized.

Example

TBD.

Parameters

Parameters	Description
eventHandler	Pointer to the call back function. The host layer notifies the application about host layer events by calling this function. If this is NULL, then events will not be generated.
context	application specific context.

Function

```
USB_HOST_RESULT USB_HOST_EventHandlerSet
(
    USB_HOST_EVENT_HANDLER * eventHandler,
    uintptr_t context
)
```

USB_HOST_Initialize Function

Initializes the USB Host layer instance specified by the index.

File

[usb_host.h](#)

C

```
SYS_MODULE_OBJ USB_HOST_Initialize(const SYS_MODULE_INIT * init);
```

Returns

Return a SYS_MODULE_OBJ_INVALID if the initialization failed.

Description

This routine initializes the USB Host Layer. This function must be called before any other Host layer function can be called. The initialization data is specified by the init parameter. This function is typically called in the SYS_Initialize() function. The initialization completion may require the [USB_HOST_Tasks\(\)](#) routine to execute. The initialization function does not start the operation of the Host on the USB. This must be done explicitly via the [USB_HOST_BusEnable\(\)](#) function. This function will initialize all client drivers listed in the TPL.

Remarks

This routine must be called before any other USB Host routine is called. This routine should only be called once during system initialization unless [USB_HOST_Deinitialize](#) is called to deinitialize the Host Layer instance. This routine will NEVER block for hardware access. The [USB_HOST_Tasks\(\)](#) function should be called to complete the initialization.

Preconditions

The USB Host Controller driver initialization should be called somewhere in the SYS_Initialize() function.

Example

TBD.

Parameters

Parameters	Description
init	Pointer to a USB_HOST_INIT data structure containing data necessary to initialize the driver.

Function

```
SYS_MODULE_OBJ USB_HOST_Initialize
(
    const SYS_MODULE_INIT * const init
)
```

b) Data Types and Constants

USB_HOST_INIT Structure

Defines the data required to initialize a USB Host Layer instance.

File

[usb_host.h](#)

C

```
typedef struct {
    size_t nTPLEntries;
```

```
USB_HOST_TPL_ENTRY * tplList;
USB_HOST_HCD * hostControllerDrivers;
} USB_HOST_INIT;
```

Members

Members	Description
size_t nTPEEntries;	Size of the TPL table
USB_HOST_TPL_ENTRY * tplList;	Pointer to the TPL table for this host layer implementation.
USB_HOST_HCD * hostControllerDrivers;	This is a pointer to a table of host controller drivers that the host layer will operate on. The number of entries in this table is specified via the USB_HOST_CONTROLLERS_NUMBER configuration macro in system_config.h

Description

USB Host Initialization Data Structure

This data type defines the data required to initialize the host layer. A pointer to a structure of this type is required by the [USB_HOST_Initialize\(\)](#) function.

Remarks

This data structure is specific to the PIC32MX implementation of the USB Host layer.

USB_HOST_EVENT_RESPONSE Enumeration

Host Layer Events Handler Function Response Type.

File

[usb_host.h](#)

C

```
typedef enum {
    USB_HOST_EVENT_RESPONSE_NONE = 0
} USB_HOST_EVENT_RESPONSE;
```

Members

Members	Description
USB_HOST_EVENT_RESPONSE_NONE = 0	Returning this value indicates no application response to the host event

Description

Host Layer Events Handler Function Response Type.

This is the definition of the Host Layer Event Handler Response Type.

Remarks

None.

\.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL) Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
```

```
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData,
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \{\ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData,
driver)\ \{\ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver)\ \{\ .id.vid_pid = {
vid, pid }, driver)\ \{\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \{
.id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \{\ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t value;
struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; \
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

1.`tplFlags.driverType = (TPL_FLAG_VID_PID)` Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
  initData,
  classCode,
  subClassCode,
  protocolCode
} driver\ \{\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \{\ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \{\ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \{\ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData =
initData, driver)\ \{\ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \{\ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData,
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \{\ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, driver)\ \{\ .id.vid_pid = {
vid, pid, \ .hostClientDriverInitData = initData, driver)\ \{\ .id.vid_pid = { vid, pid, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData =
initData, driver)\ \{\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData,
driver)\ \{\ .id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID),
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData,
.hostClientDriverInitData = initData, driver\ } typedef struct { union { uint32_t value;
struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
```

```
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

0 Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
.driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ \
.id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t \
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned \
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

0x0000 Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```

typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t value; \
struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned \
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;

```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

0xFF Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```

typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \

```

```

.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t \
value; struct { uint16_t vid; uint16_t pid; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned \
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;

```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

0xFF } Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

`usb_host.h`

C

```

typedef enum {
  initData,
  classCode,
  subClassCode,
  protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \

```

```
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode;
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

0xFFFF Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
driver\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver\ \ .id.vid_pid = { \
vid, pid }, driver\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver\ \
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver\ \ .id.vid_pid = { \
vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode;
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined

with the Class, Subclass, Protocol flags. For example, the TPL_FLAG_VID_PID flag cannot be combined with TPL_FLAG_IGNORE_SUBCLASS.

Remarks

None.

0xFFFF } Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.vid_pid = { vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = \
(TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t \
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned \
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; } \
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

1 Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
    initData,
```

```

classCode,
subClassCode,
protocolCode
} driver)\{\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\{\ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\{\ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\\
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\{\ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\{\ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\{\ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\{\ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, driver)\{\ .id.vid_pid = { \
vid, pid, \ .hostClientDriverInitData = initData, driver)\{\ .id.vid_pid = { vid, pid }, driver)\{\ .id.vid_pid = { \
0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\\
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\{\ \
.id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t \
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned \
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; \
} USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;

```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

classCode Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```

typedef enum {
  initData,
  classCode,
  subClassCode,
  protocolCode
} driver)\{\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\{\ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\{\ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\\
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\{\ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\{\ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\{\ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\{\ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, driver)\{\ .id.vid_pid = { \
vid, pid, \ .hostClientDriverInitData = initData, driver)\{\ .id.vid_pid = { vid, pid }, driver)\{\ .id.vid_pid = { \
0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \

```

```
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
{\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver}\ \} \
.id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \{ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .hostClientDriver = driver\ } \
typedef struct { union { uint32_t value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned \
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

false Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver\ \{ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \} .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \} .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver\ \
{\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \} .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \} .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver\ \} .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver\ \} .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, driver\ \} \ .id.vid_pid = { vid, \
pid, \ .hostClientDriverInitData = initData, driver)\ \} .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = \
initData, driver)\ \} .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver\ \
{\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \} \
.id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \} .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } \
typedef struct { union { uint32_t value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned \
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to

the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the TPL_FLAG_IGNORE_SUBCLASS flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the TPL_FLAG_VID_PID flag cannot be combined with TPL_FLAG_IGNORE_SUBCLASS.

Remarks

None.

initData Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { vid, pid }, driver)\ \ .id.vid_pid = { \
0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ \
.id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t \
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned \
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; } \
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the TPL_FLAG_IGNORE_SUBCLASS flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the TPL_FLAG_VID_PID flag cannot be combined with TPL_FLAG_IGNORE_SUBCLASS.

Remarks

None.

mask Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```

typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFFFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.vid_pid = { vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode;
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; \
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;

```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

pid Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```

typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData,

```

```
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData,
driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = {
vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ \
.id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode;
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; \
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

`pid` Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData =
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData,
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData =
initData, driver)\ \ .id.vid_pid = { vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType =
(TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ \
.id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode;
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; \
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

`subClassCode` Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
.driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ \
.id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t \
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned \
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

true Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```

typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t \
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned \
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;

```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The tplFlag member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the TPL_FLAG_IGNORE_SUBCLASS flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the TPL_FLAG_VID_PID flag cannot be combined with TPL_FLAG_IGNORE_SUBCLASS.

Remarks

None.

USB_HOST_BUS Type

Defines a USB Bus Data Type.

File

[usb_host.h](#)

C

```
typedef uint8_t USB_HOST_BUS;
```

Description

USB Bus Data Type

This data type defines a USB Bus. In microcontroller devices, that may have multiple USB peripherals, this type identifies the USB bus associated

with each peripheral. Bus numbers start from 0 and counts up to include all the busses in the system. The total number of busses and the mapping between a bus and the USB controller is specified in the Host Layer initialization data structure.

Remarks

None.

USB_HOST_DEVICE_INFO Structure

Defines the data type that is used by the [USB_HOST_DeviceGetFirst\(\)](#) and [USB_HOST_DeviceGetNext\(\)](#) functions.

File

[usb_host.h](#)

C

```
typedef struct {
    USB_HOST_DEVICE_OBJ_HANDLE deviceObjHandle;
    uint8_t deviceAddress;
    USB_HOST_BUS bus;
} USB_HOST_DEVICE_INFO;
```

Members

Members	Description
USB_HOST_DEVICE_OBJ_HANDLE deviceObjHandle;	USB Host Device Object Handle
uint8_t deviceAddress;	Address of the device on the USB
USB_HOST_BUS bus;	The bus to which this device is connected

Description

USB Host Device Info Type

This data type defines the type of data that is used by the [USB_HOST_DeviceGetFirst\(\)](#) and [USB_HOST_DeviceGetNext\(\)](#) functions. The application must provide an object of this type to these functions to obtain information about the devices attached on the USB.

Remarks

The application must only instantiate this data structure and should not modify its contents. Multiple objects can be instantiated and used.

USB_HOST_DEVICE_OBJ_HANDLE Type

Handle to an attached USB Device.

File

[usb_host.h](#)

C

```
typedef uintptr_t USB_HOST_DEVICE_OBJ_HANDLE;
```

Description

USB Host Device Object Handle

This data type defines the type of handle to an attached USB Device. This handle uniquely identifies the attached device. A handle of this type is returned in the deviceObjHandle member of the [USB_HOST_DEVICE_INFO](#) structure when the [USB_HOST_DeviceGetFirst\(\)](#) and the [USB_HOST_DeviceGetNext\(\)](#) functions are called.

Remarks

None.

USB_HOST_DEVICE_STRING Enumeration

Defines a defines types of strings that can be request through the [USB_HOST_DeviceStringDescriptorGet\(\)](#) function.

File

[usb_host.h](#)

C

```
typedef enum {
    USB_HOST_DEVICE_STRING_LANG_ID = 0,
    USB_HOST_DEVICE_STRING_MANUFACTURER,
    USB_HOST_DEVICE_STRING_PRODUCT,
    USB_HOST_DEVICE_STRING_SERIAL_NUMBER
} USB_HOST_DEVICE_STRING;
```

Members

Members	Description
USB_HOST_DEVICE_STRING_LANG_ID = 0	Specifies the language ID string
USB_HOST_DEVICE_STRING_MANUFACTURER	Specifies the manufacturer string
USB_HOST_DEVICE_STRING_PRODUCT	Specifies the product string
USB_HOST_DEVICE_STRING_SERIAL_NUMBER	Specifies the serial number string

Description

USB Host Device String Type

This type defines the types of strings that can be request through the [USB_HOST_DeviceStringDescriptorGet\(\)](#) function. The stringType parameter in the function call can be set any one of these types.

Remarks

None.

USB_HOST_EVENT Enumeration

Defines the different events that the USB Host Layer can generate.

File

[usb_host.h](#)

C

```
typedef enum {
    USB_HOST_EVENT_DEVICE_REJECTED_INSUFFICIENT_POWER,
    USB_HOST_EVENT_DEVICE_UNSUPPORTED,
    USB_HOST_EVENT_HUB_TIER_LEVEL_EXCEEDED,
    USB_HOST_EVENT_PORT_OVERCURRENT_DETECTED
} USB_HOST_EVENT;
```

Members

Members	Description
USB_HOST_EVENT_DEVICE_REJECTED_INSUFFICIENT_POWER	This event occurs when device needs more current than what the host can supply.
USB_HOST_EVENT_DEVICE_UNSUPPORTED	This event occurs when a host layer could not attach any drivers to the attached device or when an error has occurred. There is no event data associated with this event.
USB_HOST_EVENT_HUB_TIER_LEVEL_EXCEEDED	This event occurs when the number of hubs connected to the host exceeds the configured maximum number of hubs USB_HOST_HUB_TIER_LEVEL . There is no event data associated with this event.
USB_HOST_EVENT_PORT_OVERCURRENT_DETECTED	This event occurs when an over-current condition is detected at the root <ul style="list-style-type: none"> • hub or an external hub port.

Description

USB Host Events

This data type defines the different events that USB Host Layer can generate. The application is intended recipient of these events. Some events return event related data. The application must register an event handler with the host layer (via the [USB_HOST_EventHandlerSet\(\)](#) function) before enabling any of the buses.

Remarks

None.

USB_HOST_EVENT_HANDLER Type

USB Host Layer Event Handler Function Pointer Type

File

[usb_host.h](#)

C

```
typedef USB_HOST_EVENT_RESPONSE (* USB_HOST_EVENT_HANDLER)(USB_HOST_EVENT event, void * eventData,  
uintptr_t context);
```

Description

USB Host Layer Event Handler Function Pointer Type

This data type defines the required function signature of the USB Host Layer Event handling callback function. The application must register a pointer to a Host Layer Event handling function who's function signature (parameter and return value types) match the types specified by this function pointer in order to receive event call backs from the Host Layer. The Host Layer will invoke this function with event relevant parameters. The description of the event handler function parameters is given here.

event - Type of event generated.

pData - This parameter should be type cast to an event specific pointer type based on the event that has occurred. Refer to the [USB_HOST_EVENT](#) enumeration description for more details.

context - Value identifying the context of the application that was registered along with the event handling function.

Remarks

None.

USB_HOST_HCD Structure

Defines the USB Host HCD Information object that is provided to the host layer.

File

[usb_host.h](#)

C

```
typedef struct {  
    SYS_MODULE_INDEX drvIndex;  
    void * hcdInterface;  
} USB_HOST_HCD;
```

Members

Members	Description
SYS_MODULE_INDEX drvIndex;	Index of the USB Host Controller driver that the host layer should open and use.
void * hcdInterface;	USB Host Controller Driver function pointers

Description

USB Host Controller Driver Information

This data type defines the data required to connect a Host Controller Driver to the host layer. The USB Host layer used the HCD routines to access the root hub and the USB.

Remarks

This data structure is specific to the PIC32 implementation of the USB Host layer.

USB_HOST_REQUEST_HANDLE Type

USB Host Request Handle Type

File

[usb_host.h](#)

C

```
typedef uintptr_t USB_HOST_REQUEST_HANDLE;
```

Description

USB Host Request Handle Type

This type defines the USB Host Request Handle. This type of handle is returned by the [USB_HOST_DeviceStringDescriptorGet\(\)](#) function. Each request will generate a unique handle. This handle will be returned in the event associated with the completion of the string descriptor request.

Remarks

None.

USB_HOST_RESULT Enumeration

USB Host Results.

File

[usb_host.h](#)

C

```
typedef enum {
    USB_HOST_RESULT_REQUEST_BUSY = USB_HOST_RESULT_MIN,
    USB_HOST_RESULT_STRING_DESCRIPTOR_UNSUPPORTED,
    USB_HOST_RESULT_TRANSFER_ABORTED,
    USB_HOST_RESULT_REQUEST_STALLED,
    USB_HOST_RESULT_PIPE_HANDLE_INVALID,
    USB_HOST_RESULT_END_OF_DEVICE_LIST,
    USB_HOST_RESULT_INTERFACE_UNKNOWN,
    USB_HOST_RESULT_PARAMETER_INVALID,
    USB_HOST_RESULT_CONFIGURATION_UNKNOWN,
    USB_HOST_RESULT_BUS_NOT_ENABLED,
    USB_HOST_RESULT_BUS_UNKNOWN,
    USB_HOST_RESULT_DEVICE_UNKNOWN,
    USB_HOST_RESULT_FAILURE,
    USB_HOST_RESULT_FALSE = 0,
    USB_HOST_RESULT_TRUE = 1,
    USB_HOST_RESULT_SUCCESS = USB_HOST_RESULT_TRUE
} USB_HOST_RESULT;
```

Members

Members	Description
USB_HOST_RESULT_REQUEST_BUSY = USB_HOST_RESULT_MIN	Indicates that the Host Layer cannot accept any requests at this point
USB_HOST_RESULT_STRING_DESCRIPTOR_UNSUPPORTED	The device does not support the request string descriptor
USB_HOST_RESULT_TRANSFER_ABORTED	Request was aborted
USB_HOST_RESULT_REQUEST_STALLED	Request was stalled
USB_HOST_RESULT_PIPE_HANDLE_INVALID	The specified pipe is not valid
USB_HOST_RESULT_END_OF_DEVICE_LIST	The end of the device list was reached.
USB_HOST_RESULT_INTERFACE_UNKNOWN	The specified interface is not available
USB_HOST_RESULT_PARAMETER_INVALID	A NULL parameter was passed to the function
USB_HOST_RESULT_CONFIGURATION_UNKNOWN	The specified configuration does not exist on this device.
USB_HOST_RESULT_BUS_NOT_ENABLED	A bus operation was requested but the bus was not operated
USB_HOST_RESULT_BUS_UNKNOWN	The specified bus does not exist in the system
USB_HOST_RESULT_DEVICE_UNKNOWN	The specified device does not exist in the system
USB_HOST_RESULT_FAILURE	An unknown failure has occurred
USB_HOST_RESULT_FALSE = 0	Indicates a false condition
USB_HOST_RESULT_TRUE = 1	Indicate a true condition
USB_HOST_RESULT_SUCCESS = USB_HOST_RESULT_TRUE	Indicates that the operation succeeded or the request was accepted and will be processed.

Description

USB Host Result

This enumeration defines the possible returns values of USB Host Layer API. A function may only return some of the values in this enumeration. Refer to function description for details on which values will be returned.

Remarks

None.

USB_HOST_STRING_REQUEST_COMPLETE_CALLBACK Type

USB Host Device String Descriptor Request Complete Callback Function Type

File

[usb_host.h](#)

C

```
typedef void (* USB_HOST_STRING_REQUEST_COMPLETE_CALLBACK)(USB_HOST_REQUEST_HANDLE requestHandle, size_t size, uintptr_t context);
```

Description

USB Host Device String Descriptor Request Complete Callback Function Type

This data type defines the required function signature of the USB Host Device String Descriptor Request Complete Callback Function. The application must specify a pointer to a function who's function signature (parameter and return value types) matches the type specified by this function pointer in order to a call backs from the Host Layer when the [USB_HOST_DeviceStringDescriptorGet\(\)](#) function has completed its operation. The description of the callback function parameters is given here.

requestHandle - a handle that is unique to this request. This will match the handle that was returned by the [USB_HOST_DeviceStringDescriptorGet\(\)](#) function.

size - size of the returned string descriptor. If the string descriptor could not be obtained, the size will be zero.

context - Value identifying the context of the application that was registered along with the event handling function.

Remarks

None.

USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
.driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned
```

```
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

USB_HOST_TPL_ENTRY Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```
typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ {\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ {\ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ {\ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
{\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ {\ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ {\ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ {\ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = \
initData, \ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ {\ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = \
initData, driver)\ {\ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver)\ {\ .id.vid_pid = { \
vid, pid }, driver)\ {\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
{\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \
.id.vid_pid = { vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ {\ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t \
value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned \
ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned \
ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver;
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;
```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The `tplFlag` member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the `TPL_FLAG_IGNORE_SUBCLASS` flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the `TPL_FLAG_VID_PID` flag cannot be combined with `TPL_FLAG_IGNORE_SUBCLASS`.

Remarks

None.

vid Enumeration

USB Host Layer TPL Table Entry Matching Criteria flag

File

[usb_host.h](#)

C

```

typedef enum {
    initData,
    classCode,
    subClassCode,
    protocolCode
} driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, subClassCode, 0xFF }, driver)\ \
\ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \
.tplFlags.ignoreClass = false, \ .tplFlags.ignoreClass = false, \ .tplFlags.ignoreProtocol = true, \
.hostClientDriverInitData = initData, driver)\ \ .id.cl_sc_p = { classCode, \ .hostClientDriverInitData = \
initData, driver)\ \ .id.cl_sc_p = { classCode, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreClass = false, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
\ .hostClientDriverInitData = initData, 0xFF, 0xFF, 0xFF }, driver)\ \ .id.vid_pid = { 0xFFFF, \
.tplFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL), \ .tplFlags.ignoreProtocol = true, \
.tplFlags.ignoreProtocol = true, \ .tplFlags.ignoreProtocol = true, \ .hostClientDriverInitData = initData, \
driver)\ \ .id.vid_pid = { vid, pid, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, driver)\ \ .id.vid_pid = { 0xFFFF, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 0, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, driver)\ \
\ .id.vid_pid = { vid, pid, \ .pidMask = mask, \ .hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { \
vid, pid }, \ .pidMask = mask, \ .tplFlags.driverType = (TPL_FLAG_VID_PID), \
.tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 0, \ .hostClientDriverInitData = initData, \
.hostClientDriverInitData = initData, driver)\ \ .id.vid_pid = { 0xFFFF, 0xFFFF }, \ .pidMask = 0x0000, \
.tplFlags.driverType = (TPL_FLAG_VID_PID), \ .tplFlags.ignoreVIDPID = 1, \ .tplFlags.ignoreVIDPID = 1, \
.hostClientDriverInitData = initData, \ .hostClientDriver = driver\ } typedef struct { union { uint32_t value; struct { uint16_t vid; uint16_t pid; } vid_pid; struct { uint8_t classCode; uint8_t subClassCode; \
uint8_t protocolCode; } cl_sc_p; } id; uint16_t pidMask; struct { unsigned driverType :1; unsigned ignoreClass :1; unsigned ignoreSubClass :1; unsigned ignoreProtocol :1; unsigned pidMasked :1; unsigned ignoreVIDPID :1; } tplFlags; void * hostClientDriverInitData; void * hostClientDriver; }
USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY, USB_HOST_TPL_ENTRY;

```

Description

USB Host Layer TPL Table Entry Matching Criteria flag

This enumeration defines the possible matching criteria flag that can be specified for a Host TPL table entry. The tplFlag member of the TPL table entry should be set to one or more of these flags. These flags define the criteria that the Host layer will use while matching the attached device to the TPL table entry. For example, if a device is specified by class, subclass and protocol specifying the TPL_FLAG_IGNORE_SUBCLASS flag will cause the Host layer to ignore the subclass while comparing the class, subclass and protocol of the attached device.

Multiple flags can be specified as a logically OR'ed combination. While combining multiple flags, VID and PID criteria flags cannot be combined with the Class, Subclass, Protocol flags. For example, the TPL_FLAG_VID_PID flag cannot be combined with TPL_FLAG_IGNORE_SUBCLASS.

Remarks

None.

USB_HOST_BUS_ALL Macro

USB Host Bus All

File

[usb_host.h](#)

C

```
#define USB_HOST_BUS_ALL ((USB_HOST_BUS)(0xFF))
```

Description

USB Host Bus All

This constant defines the value that should be passed to the [USB_HOST_BusSuspend\(\)](#), [USB_HOST_BusResume\(\)](#) and

USB_HOST_IsBusSuspended() function if all the USB segments must be addressed. Passing this constant to these functions will cause Suspend and Resume operation to affect all the USB segments and hence affect all connected devices.

Remarks

None.

USB_HOST_DEVICE_OBJ_HANDLE_INVALID Macro

Defines an invalid USB Device Object Handle.

File

[usb_host.h](#)

C

```
#define USB_HOST_DEVICE_OBJ_HANDLE_INVALID ((USB_HOST_DEVICE_OBJ_HANDLE)(-1))
```

Description

USB Host Invalid Device Object Handle

This constant defines an invalid USB Device Object Handle. The [USB_HOST_DeviceGetFirst\(\)](#) and the [USB_HOST_DeviceGetNext\(\)](#) functions return this value in the deviceObjHandle member of the [USB_HOST_DEVICE_INFO](#) object when there are no attached devices to report.

Remarks

None.

USB_HOST_DEVICE_STRING_LANG_ID_DEFAULT Macro

Defines the default Lang ID to be used while obtaining the string.

File

[usb_host.h](#)

C

```
#define USB_HOST_DEVICE_STRING_LANG_ID_DEFAULT (0)
```

Description

USB Host Device String Default Lang ID

This constant defines the default Lang ID. When then languageID parameter in the [USB_HOST_DeviceStringDescriptorGet\(\)](#) function is set to this value, the function will specify the default Lang ID while requesting the string from the device.

Remarks

None.

USB_HOST_REQUEST_HANDLE_INVALID Macro

USB Host Request Invalid Handle

File

[usb_host.h](#)

C

```
#define USB_HOST_REQUEST_HANDLE_INVALID ((USB_HOST_REQUEST_HANDLE)(-1))
```

Description

USB Host Request Invalid Handle

This constant defines an Invalid USB Host Request Handle. This handle is returned by the [USB_HOST_DeviceStringDescriptorGet\(\)](#) function when the request was not accepted.

Remarks

None.

USB_HOST_RESULT_MIN Macro

USB Host Result Minimum Constant.

File

[usb_host.h](#)

C

```
#define USB_HOST_RESULT_MIN -100
```

Description

USB Host Result Minimum Constant

Constant identifying the USB Host Result Minimum Value. This constant is used in the [USB_HOST_RESULT](#) enumeration.

Remarks

None.

Files**Files**

Name	Description
usb_host.h	USB Host Layer Interface Header
usb_host_config_template.h	USB host configuration template header file.

Description

This section lists the source and header files used by the library.

usb_host.h

USB Host Layer Interface Header

Enumerations

	Name	Description
	\ .tpFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL)	USB Host Layer TPL Table Entry Matching Criteria flag
	\ .tpFlags.driverType = (TPL_FLAG_VID_PID)	USB Host Layer TPL Table Entry Matching Criteria flag
	0	USB Host Layer TPL Table Entry Matching Criteria flag
	0x0000	USB Host Layer TPL Table Entry Matching Criteria flag
	0xFF	USB Host Layer TPL Table Entry Matching Criteria flag
	0xFF }	USB Host Layer TPL Table Entry Matching Criteria flag
	0xFFFF	USB Host Layer TPL Table Entry Matching Criteria flag
	0xFFFF }	USB Host Layer TPL Table Entry Matching Criteria flag
	1	USB Host Layer TPL Table Entry Matching Criteria flag
	classCode	USB Host Layer TPL Table Entry Matching Criteria flag
	false	USB Host Layer TPL Table Entry Matching Criteria flag
	initData	USB Host Layer TPL Table Entry Matching Criteria flag
	mask	USB Host Layer TPL Table Entry Matching Criteria flag
	pid	USB Host Layer TPL Table Entry Matching Criteria flag
	pid }	USB Host Layer TPL Table Entry Matching Criteria flag
	subClassCode	USB Host Layer TPL Table Entry Matching Criteria flag
	true	USB Host Layer TPL Table Entry Matching Criteria flag
	USB_HOST_DEVICE_STRING	Defines a defines types of strings that can be request through the USB_HOST_DeviceStringDescriptorGet() function.
	USB_HOST_EVENT	Defines the different events that the USB Host Layer can generate.
	USB_HOST_EVENT_RESPONSE	Host Layer Events Handler Function Response Type.
	USB_HOST_RESULT	USB Host Results.
	USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY	USB Host Layer TPL Table Entry Matching Criteria flag

	USB_HOST_TPL_ENTRY	USB Host Layer TPL Table Entry Matching Criteria flag
	vid	USB Host Layer TPL Table Entry Matching Criteria flag

Functions

	Name	Description
≡◊	USB_HOST_BusEnable	Starts host operations.
≡◊	USB_HOST_BusIsEnabled	Checks if the bus is enabled.
≡◊	USB_HOST_BusIsSuspended	Returns the suspend status of the bus.
≡◊	USB_HOST_BusResume	Resumes the bus.
≡◊	USB_HOST_BusSuspend	Suspends the bus.
≡◊	USB_HOST_Deinitialize	Deinitializes the specified instance of the USB Host Layer.
≡◊	USB_HOST_DeviceGetFirst	Returns information about the first attached device on the bus.
≡◊	USB_HOST_DeviceGetNext	Returns information about the next device on the bus.
≡◊	USB_HOST_DeviceIsSuspended	Returns the suspend state of the device is suspended.
≡◊	USB_HOST_DeviceResume	Resumes the selected device
≡◊	USB_HOST_DeviceSpeedGet	Returns the speed at which this device is operating.
≡◊	USB_HOST_DeviceStringDescriptorGet	Retrieves specified string descriptor from the device
≡◊	USB_HOST_DeviceSuspend	Suspends the specified device.
≡◊	USB_HOST_EventHandlerSet	USB Host Layer Event Handler Callback Function set function.
≡◊	USB_HOST_Initialize	Initializes the USB Host layer instance specified by the index.
≡◊	USB_HOST_Status	Gets the current status of the USB Host Layer.
≡◊	USB_HOST_Tasks	Maintains the USB Host Layer state machine.

Macros

	Name	Description
	USB_HOST_BUS_ALL	USB Host Bus All
	USB_HOST_DEVICE_OBJ_HANDLE_INVALID	Defines an invalid USB Device Object Handle.
	USB_HOST_DEVICE_STRING_LANG_ID_DEFAULT	Defines the default Lang ID to be used while obtaining the string.
	USB_HOST_REQUEST_HANDLE_INVALID	USB Host Request Invalid Handle
	USB_HOST_RESULT_MIN	USB Host Result Minimum Constant.

Structures

	Name	Description
	USB_HOST_DEVICE_INFO	Defines the data type that is used by the USB_HOST_DeviceGetFirst() and USB_HOST_DeviceGetNext() functions.
	USB_HOST_HCD	Defines the USB Host HCD Information object that is provided to the host layer.
	USB_HOST_INIT	Defines the data required to initialize a USB Host Layer instance.

Types

	Name	Description
	USB_HOST_BUS	Defines a USB Bus Data Type.
	USB_HOST_DEVICE_OBJ_HANDLE	Handle to an attached USB Device.
	USB_HOST_EVENT_HANDLER	USB Host Layer Event Handler Function Pointer Type
	USB_HOST_REQUEST_HANDLE	USB Host Request Handle Type
	USB_HOST_STRING_REQUEST_COMPLETE_CALLBACK	USB Host Device String Descriptor Request Complete Callback Function Type

Description

USB Host Layer Interface Definition

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the USB HOST layer.

File Name

usb_host.h

Company

Microchip Technology Inc.

usb_host_config_template.h

USB host configuration template header file.

Macros

	Name	Description
	USB_HOST_CONTROLLERS_NUMBER	Defines the number of USB Host Controllers that this Host Layer must manage.
	USB_HOST_DEVICE_INTERFACES_NUMBER	Defines the maximum number of interface that the attached device can contain in order for the USB Host Layer to process the device.
	USB_HOST_DEVICES_NUMBER	Defines the maximum number of devices to support.
	USB_HOST_HUB_SUPPORT_ENABLE	Defines if this USB Host application must support a Hub.
	USB_HOST_HUB_TIER_LEVEL	Defines the maximum tier of connected hubs to be supported.
	USB_HOST_PIPES_NUMBER	Defines the maximum number of pipes that the application will need.
	USB_HOST_TRANSFERS_NUMBER	Defines the maximum number of transfers that host layer should handle.

Description

USB Host Layer Configuration constants

This file contains USB host layer compile time options (macros) that are to be configured by the user. This file is a template file and must be used as an example only. This file must not be directly included in the project.

File Name

usb_host_config_template.h

Company

Microchip Technology Inc.

USB Audio v1.0 Host Client Driver Library

This section describes the USB Audio v1.0 Host Client Driver Library.

Introduction

Introduces the MPLAB Harmony USB Audio v1.0 Host Client Driver Library.

Description

The USB Audio v1.0 Host Client Driver in the MPLAB Harmony USB Host Stack allows USB Host applications to support and interact with Audio v1.0 USB devices. The USB Audio v1.0 Host Client Driver has the following features:

- Supports Audio v1.0 device with multiple streaming interfaces
- Designed to support multi-client operation
- RTOS ready
- Features an event driver non-clocking application interaction model
- Supports queuing of read and write data transfers

Using the Library

This topic describes the basic architecture of the USB Audio v1.0 Host Client Driver Library and provides information and examples on its use.

Abstraction Model

Describes the Abstraction Model of the USB Audio v1.0 Host Client Driver Library.

Description

The USB Audio v1.0 Host Client Driver interacts with Host Layer to control the attached Audio v1.0 device. The USB Host Layer attaches the Audio v1.0 Host Client Driver to the Audio v1.0 device when it meets the matching criteria specified in the USB Host TPL table. The Audio v1.0 Host Client Driver abstracts the details of sending Audio v1.0 class specific control transfer commands by providing easy to use non-blocking API to send these command. A command when issued is assigned a request handle. This request handle is returned in the event that is generated when the command has been processed, and can be used by the application to track the command.

While transferring data Audio Stream Data over the USB Audio v1.0 Host Client Driver abstracts details such as the Audio Streaming interface, endpoints and endpoint size. The USB Audio v1.0 Host Client Driver internally (and without application intervention) validates the Audio v1.0 class

specific device descriptors and opens isochronous pipes. While transferring data, multiple read and write requests can be queued. Each such request gets assigned a transfer handle. The transfer handle for a transfer request is returned along with the completion event for that transfer request. The data transfer routines are implemented in `usb_host_audio_v1_0.c`.

Library Overview

The USB Audio v1.0 Host Client Driver can be grouped functionally as shown in the following table.

Library Interface Section	Description
Audio Device access Functions	These functions allow application clients to perform audio control transfers, register event handlers and get the number of stream groups and the details of each audio stream. These functions are implemented in the <code>usb_host_audio_v1_0.c</code> file.
Audio Stream Access Functions	These functions allow the application client to open audio streams, set parameters of an audio stream, and perform data transfer operations on an audio stream. These functions are implemented in the <code>usb_host_audio_v1_0.c</code> file.

How the Library Works

Describes how the library works and how it should be used.

Description

The USB Audio v1.0 Host Client Driver provides the user application with an easy-to-use interface to the attached Audio v1.0 device. The USB Host Layer initializes the USB Audio v1.0 Host Client Driver when a device is attached. This process does not require application intervention. The following sections describe the steps and methods required for the user application to interact with the attached devices.

TPL Table Configuration for Audio v1.0 Devices

Describes how to configure TPL table options, which includes a code example.

Description

The Host Layer attaches the Audio v1.0 Host Client Driver to a device when the device class in the Interface descriptor matches the entry in the TPL table. When specifying the entry for the Audio v1.0 device, the entry for the Audio v1.0 device, the driver interface must be set to `USB_HOST_AUDIO_V1_0_INTERFACE`. This will attach the Audio v1.0 Host Client Driver to the device when the USB Host matches the TPL entry to the device. The following code shows possible TPL table options for matching Audio v1.0 Devices.

```
/* This code shows an example of TPL table entries for supporting Audio v1.0
 * devices. Note the driver interface is set to USB_HOST_AUDIO_V1_0_INTERFACE. This
 * will load the Audio v1.0 Host Client Driver when there is TPL match */

const USB_HOST_TPL_ENTRY USBTPLList[1] =
{
    /* This entry looks for any Audio v1.0 device. The Audio v1.0 Host Client Driver will
     * check if this is an Audio Streaming Device and will then load itself */
    TPL_INTERFACE_CLASS(USB_AUDIO_CLASS_CODE, NULL, USB_HOST_AUDIO_V1_0_INTERFACE),
};

};
```

Detecting Device Attach

Describes how to detect when a Audio v1.0 Device is attached, which includes a code example.

Description

The application will need to know when a Audio v1.0 Device is attached. To receive this attach event from the Audio v1.0 Host Client Driver, the application must register an Attach Event Handler by calling the `USB_HOST_AUDIO_V1_0_AttachEventHandlerSet` function. This function should be called before the `USB_HOST_BusEnable` function is called, else the application may miss Audio v1.0 attach events. It can be called multiple times to register multiple event handlers, each for different application clients that need to know about Audio v1.0 Device Attach events.

The total number of event handlers that can be registered is defined by `USB_HOST_AUDIO_V1_0_ATTACH_LISTENERS_NUMBER` configuration option in `system_config.h`. When a device is attached, the Audio v1.0 Host Client Driver will send the attach event to all the registered event handlers. In this event handler, the USB Audio v1.0 Host Client Driver will pass a `USB_HOST_AUDIO_V1_0_OBJ` that can be opened to gain access to the device. The following code shows an example of how to register attach event handlers.

```
/* This code shows an example of Audio v1.0 Attach Event Handler and how this
 * attach event handler can be registered with the Audio v1.0 Host Client Driver */
bool isAudioDeviceAttached = false;
USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj;
```

```

/* Audio attach event listener function */
void APP_USBHostAudioAttachEventListener
(
    USB_HOST_AUDIO_V1_0_OBJ audioObj,
    USB_HOST_AUDIO_V1_0_EVENT event,
    uintptr_t context
)
{
    /* This function gets called when the Audio v1.0 device is attached/detached. In this
     * example we let the application know that a device is attached and we
     * store the Audio v1.0 device object. This object will be required to open the
     * device. */
    switch (event)
    {
        case USB_HOST_AUDIO_V1_0_EVENT_ATTACH:
            if (isAudioDeviceAttached == false)
            {
                isAudioDeviceAttached = true;
                audioDeviceObj = audioObj;
            }
            else
            {
                /* This application supports only one Audio Device . Handle Error Here.*/
            }
            break;
        case USB_HOST_AUDIO_V1_0_EVENT_DETACH:
            if (isAudioDeviceAttached == true)
            {
                /* This means the device was detached. There is no event data
                 * associated with this event.*/
                isAudioDeviceAttached = false;
            }
            break;
    }
}

void APP_Tasks(void)
{
    switch (appData.state)
    {
        case APP_STATE_BUS_ENABLE:

            /* In this state the application enables the USB Host Bus. Note
             * how the Audio v1.0 Attach event handler is registered before the bus
             * is enabled. */

            USB_HOST_AUDIO_V1_0_AttachEventHandlerSet(APP_USBHostAudioAttachEventListener, (uintptr_t) 0);
            USB_HOST_BusEnable(0);
            appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:
            /* Here we wait for the bus enable operation to complete. */
            break;
    }
}

```

Obtaining Audio v1.0 Device Audio Stream Details

Describes how to obtain audio stream details, which includes a code example.

Description

The application will need to know more details about an attached audio device like Number of Audio Stream Groups and audio format details of each audio stream in audio stream group. Application will need to search through all of the audio streams and find if a suitable audio stream is available before it can open a stream and start communicating.

`USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet` function can be used to know how many stream groups are available in the attached Audio device. This function takes `USB_HOST_AUDIO_V1_0_OBJ` as an argument and returns `uint8_t` value as number of stream groups.

`USB_HOST_AUDIO_V1_0_StreamGetFirst` function can be used to find out audio format details of first audio stream in a Stream Groups. This function takes `USB_HOST_AUDIO_V1_0_OBJ`, stream group index and pointer to the `USB_HOST_AUDIO_V1_0_STREAM_INFO` as arguments. The stream index can any number between zero to number of stream groups returned by `USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet` function. The audio stream object returned as part of `USB_HOST_AUDIO_V1_0_STREAM_OBJ` structure.

`USB_HOST_AUDIO_V1_0_StreamGetNext` function can be used to find details about subsequent audio streams. When there are no more audio streams available in the specified audio stream group this function return `USB_HOST_AUDIO_V1_0_RESULT_END_OF_STREAM_LIST` error. It is application's responsibility to map and Audio Stream group and an audio stream.

If the application is looking for a audio stream with certain properties, application need compare audio stream properties with members of the `USB_HOST_AUDIO_V1_0_STREAM_INFO` structure returned by `USB_HOST_AUDIO_V1_0_StreamGetFirst` and `USB_HOST_AUDIO_V1_0_StreamGetNext` functions.

```

/* This code shows an example of getting details about audio stream
   in an attached Audio v1.0 device.*/

/* Specify the Audio Stream format details that this application supports */
const APP_USB_HOST_AUDIO_STREAM_FORMAT audioSpeakerStreamFormat =
{
    .streamDirection = USB_HOST_AUDIO_V1_0_DIRECTION_OUT,
    .format = USB_AUDIO_FORMAT_PCM,
    .nChannels = 2,
    .bitResolution = 16,
    .subFrameSize = 2,
    .samplingRate = 48000
};

bool isAudioDeviceAttached = false;
USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj;

/*****************/
/* Function to search for a specific Audio Stream */
/*****************/
USB_HOST_AUDIO_V1_0_STREAM_OBJ App_USBHostAudioSpeakerStreamFind(
{
    USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj,
    APP_USB_HOST_AUDIO_STREAM_FORMAT audioStream,
    uint8_t* numberofStreamGroups
}
{
    USB_HOST_AUDIO_V1_0_RESULT result;
    USB_HOST_AUDIO_V1_0_STREAM_INFO streamInfo;

    /* Get Number of Stream Groups */
    *numberofStreamGroups = USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet(audioDeviceObj);
    if (*numberofStreamGroups == 0)
    {
        return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;
    }
    /* Get the First Stream Information in the Stream Group */
    result = USB_HOST_AUDIO_V1_0_StreamGetFirst(appData.audioDeviceObj, 0, &streamInfo);
    if (result == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS)
    {
        /* Compare Audio Stream info */
        if ((streamInfo.format == audioStream.format)
            && (streamInfo.streamDirection == audioStream.streamDirection)
            && (streamInfo.nChannels == audioStream.nChannels)
            && (streamInfo.bitResolution == audioStream.bitResolution)
            && (streamInfo.subFrameSize == audioStream.subFrameSize))
        {
            return streamInfo.streamObj;
        }
    }
    return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;
}

/*****************/

```

```

/* Audio attach event listener function */
//****************************************************************************
void APP_USBHostAudioAttachEventListener
(
    USB_HOST_AUDIO_V1_0_OBJ audioObj,
    USB_HOST_AUDIO_V1_0_EVENT event,
    uintptr_t context
)
{
    /* This function gets called when the Audio v1.0 device is attached/detached. In this
     * example we let the application know that a device is attached and we
     * store the Audio v1.0 device object. This object will be required to open the
     * device. */
    switch (event)
    {
        case USB_HOST_AUDIO_V1_0_EVENT_ATTACH:
            if (isAudioDeviceAttached == false)
            {
                isAudioDeviceAttached = true;
                audioDeviceObj = audioObj;
            }
            else
            {
                /* This application supports only one Audio Device . Handle Error Here.*/
            }
            break;
        case USB_HOST_AUDIO_V1_0_EVENT_DETACH:
            if (isAudioDeviceAttached == true)
            {
                /* This means the device was detached. There is no event data
                 * associated with this event.*/
                isAudioDeviceAttached = false;
            }
            break;
    }
}

//****************************************************************************
/* Audio Tasks function */
//****************************************************************************
void APP_Tasks ( void )
{
    USB_HOST_AUDIO_V1_0_RESULT audioResult;
    USB_HOST_AUDIO_V1_0_STREAM_RESULT streamResult;

    /* Check the application's current state. */
    switch ( appData.state )
    {
        case APP_STATE_BUS_ENABLE:

            /* Register a callback for Audio Device Attach. */
            audioResult = USB_HOST_AUDIO_V1_0_AttachEventHandlerSet
            (
                &APP_USBHostAudioAttachEventListener,
                (uintptr_t)0
            );

            if (audioResult == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS )
            {
                /* Set Host Event Handler */
                USB_HOST_EventHandlerSet(APP_USBHostEventHandler, 0);
                USB_HOST_BusEnable(0);
                /* Advance application state */
                appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            }
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:

```

```

    if(USB_HOST_BusIsEnabled(0))
    {
        appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
    }
    break;

case APP_STATE_WAIT_FOR_DEVICE_ATTACH:
/* Check if an Audio Device has been attached */
if(appData.isAudioDeviceAttached == true)
{
    appData.nAudioStreamGroups = 0;
/* Find an Audio Stream matching to our requirement */
appData.ouStreamObj = App_USBHostAudioSpeakerStreamFind
(
    appData.audioDeviceObj,
    audioSpeakerStreamFormat,
    &appData.nAudioStreamGroups
);
if (appData.nAudioStreamGroups == 0)
{
    appData.state = APP_STATE_ERROR;
    break;
}
}
break;

default:
break;
}
}

```

Obtaining an Audio Stream

Describes how to open an audio stream, which includes a code example.

Description

Once application has identified which audio stream to use, application must open that audio stream by using `USB_HOST_AUDIO_V1_0_StreamOpen` function. This function takes audio stream object `USB_HOST_AUDIO_V1_0_STREAM_OBJ` as an argument which obtained by `USB_HOST_AUDIO_V1_0_StreamGetFirst` and `USB_HOST_AUDIO_V1_0_StreamGetNext` functions and returns audio stream handle `USB_HOST_AUDIO_V1_0_STREAM_HANDLE`. If the open function fails, it returns an invalid handle (`USB_HOST_AUDIO_V1_0_STREAM_HANDLE_INVALID`). Once opened successfully, a valid handle tracks the relationship between the client and the Audio Stream. This handle should be used with other Audio Stream functions.

An audio stream can be opened multiple times by different application clients. In an RTOS based application each client could running its own thread. Multiple clients can read write data to the one Audio stream. In such a case, the read and write requests are queued. The following code shows an example of how an Audio Stream is opened.

```

/* This code shows an example of opening an audio stream */

/* Specify the Audio Stream format details that this application supports */
const APP_USB_HOST_AUDIO_STREAM_FORMAT audioSpeakerStreamFormat =
{

    .streamDirection = USB_HOST_AUDIO_V1_0_DIRECTION_OUT,
    .format = USB_AUDIO_FORMAT_PCM,
    .nChannels = 2,
    .bitResolution = 16,
    .subFrameSize = 2,
    .samplingRate = 48000
};

bool isAudioDeviceAttached = false;
USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj;

/*****************************************/
/* Function to search for a specific Audio Stream */
/*****************************************/
USB_HOST_AUDIO_V1_0_STREAM_OBJ App_USBHostAudioSpeakerStreamFind
(
    USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj,

```

```

APP_USB_HOST_AUDIO_STREAM_FORMAT audioStream,
uint8_t* numberofStreamGroups
)
{
    USB_HOST_AUDIO_V1_0_RESULT result;
    USB_HOST_AUDIO_V1_0_STREAM_INFO streamInfo;

    /* Get Number of Stream Groups */
    *numberofStreamGroups = USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet(audioDeviceObj);
    if (*numberofStreamGroups == 0)
    {
        return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;
    }
    /* Get the First Stream Information in the Stream Group */
    result = USB_HOST_AUDIO_V1_0_StreamGetFirst(appData.audioDeviceObj, 0, &streamInfo);
    if (result == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS)
    {
        /* Compare Audio Stream info */
        if ((streamInfo.format == audioStream.format)
            && (streamInfo.streamDirection == audioStream.streamDirection)
            && (streamInfo.nChannels == audioStream.nChannels)
            && (streamInfo.bitResolution == audioStream.bitResolution)
            && (streamInfo.subFrameSize == audioStream.subFrameSize))
        {
            return streamInfo.streamObj;
        }
    }
    return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;
}

/*****************/
/* Audio attach event listener function */
/*****************/
void APP_USBHostAudioAttachEventListener
(
    USB_HOST_AUDIO_V1_0_OBJ audioObj,
    USB_HOST_AUDIO_V1_0_EVENT event,
    uintptr_t context
)
{
    /* This function gets called when the Audio v1.0 device is attached/detached. In this
     * example we let the application know that a device is attached and we
     * store the Audio v1.0 device object. This object will be required to open the
     * device. */
    switch (event)
    {
        case USB_HOST_AUDIO_V1_0_EVENT_ATTACH:
            if (isAudioDeviceAttached == false)
            {
                isAudioDeviceAttached = true;
                audioDeviceObj = audioObj;
            }
            else
            {
                /* This application supports only one Audio Device . Handle Error Here.*/
            }
            break;
        case USB_HOST_AUDIO_V1_0_EVENT_DETACH:
            if (isAudioDeviceAttached == true)
            {
                /* This means the device was detached. There is no event data
                 * associated with this event.*/
                isAudioDeviceAttached = false;
            }
            break;
    }
}
}

```

```
*****
/* Audio Tasks function */
*****
void APP_Tasks ( void )
{
    USB_HOST_AUDIO_V1_0_RESULT audioResult;
    USB_HOST_AUDIO_V1_0_STREAM_RESULT streamResult;

    /* Check the application's current state. */
    switch ( appData.state )
    {
        case APP_STATE_BUS_ENABLE:

            /* Register a callback for Audio Device Attach. */
            audioResult = USB_HOST_AUDIO_V1_0_AttachEventHandlerSet
                (
                    &APP_USBHostAudioAttachEventListen,
                    (uintptr_t)0
                );

            if (audioResult == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS )
            {
                /* Set Host Event Handler */
                USB_HOST_EventHandlerSet(APP_USBHostEventHandler, 0);
                USB_HOST_BusEnable(0);
                /* Advance application state */
                appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            }
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:
            if(USB_HOST_BusIsEnabled(0))
            {
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            }
            break;

        case APP_STATE_WAIT_FOR_DEVICE_ATTACH:
            /* Check if an Audio Device has been attached */
            if(appData.isAudioDeviceAttached == true)
            {
                appData.nAudioStreamGroups = 0;
                /* Find an Audio Stream matching to our requirement */
                appData.ouStreamObj = App_USBHostAudioSpeakerStreamFind
                    (
                        appData.audioDeviceObj,
                        audioSpeakerStreamFormat,
                        &appData.nAudioStreamGroups
                    );
                if (appData.nAudioStreamGroups == 0)
                {
                    appData.state = APP_STATE_ERROR;
                    break;
                }

                /* Open Audio Stream */
                appData.outStreamHandle = USB_HOST_AUDIO_V1_0_StreamOpen
                    (
                        appData.ouStreamObj
                    );

                if (appData.outStreamHandle == USB_HOST_AUDIO_V1_0_STREAM_HANDLE_INVALID)
                {
                    appData.state = APP_STATE_ERROR;
                    break;
                }
            }
            break;
    }
}
```

```

    default:
        break;
}
}

```

Audio Stream Event Handling

Describes audio stream event handling, which includes a code example.

Description

The Audio v1.0 streams presents an event driven interface to the application. The USB Audio v1.0 Host Client Driver requires the application client to set an event handler against each audio stream for meaningful operation.

A request to send a command or transfer data typically completes after the command request or transfer function has exited. The application must then use the Audio stream event to track the completion of this command or data transfer request. In a case where multiple data transfers are queued, the transfer handles can be used to identify the transfer requests.

The application must use the [USB_HOST_AUDIO_V1_0_StreamEventHandlerSet](#) function to register an audio stream handler. This event handler will be called when a command or data transfer event has occurred and should be registered before the request for command or a data transfer.

The following code shows an example of registering an audio stream event handler.

```

/* This code shows an example of Audio stream event handling */

/* Specify the Audio Stream format details that this application supports */
const APP_USB_HOST_AUDIO_STREAM_FORMAT audioSpeakerStreamFormat =
{

    .streamDirection = USB_HOST_AUDIO_V1_0_DIRECTION_OUT,
    .format = USB_AUDIO_FORMAT_PCM,
    .nChannels = 2,
    .bitResolution = 16,
    .subFrameSize = 2,
    .samplingRate = 48000
};

bool isAudioDeviceAttached = false;
USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj;

*****
* Audio Stream Event Handler function.
*****

USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE APP_USBHostAudioStreamEventHandler
(
    USB_HOST_AUDIO_V1_0_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_0_STREAM_EVENT event,
    void * eventData,
    uintptr_t context
)
{
    USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA * writeCompleteEventData;
    switch(event)
    {
        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE:
            break;
        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE:
            break;
        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE:
            break;
        default:
            break;
    }
    return USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE_NONE;
}

```

```
*****
/* Function to search for a specific Audio Stream */
*****
USB_HOST_AUDIO_V1_0_STREAM_OBJ App_USBHostAudioSpeakerStreamFind
(
    USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj,
    APP_USB_HOST_AUDIO_STREAM_FORMAT audioStream,
    uint8_t* numberofStreamGroups
)
{
    USB_HOST_AUDIO_V1_0_RESULT result;
    USB_HOST_AUDIO_V1_0_STREAM_INFO streamInfo;

    /* Get Number of Stream Groups */
    *numberofStreamGroups = USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet(audioDeviceObj);
    if (*numberofStreamGroups == 0)
    {
        return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;
    }
    /* Get the First Stream Information in the Stream Group */
    result = USB_HOST_AUDIO_V1_0_StreamGetFirst(appData.audioDeviceObj, 0, &streamInfo);
    if (result == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS)
    {
        /* Compare Audio Stream info */
        if ((streamInfo.format == audioStream.format)
            && (streamInfo.streamDirection == audioStream.streamDirection)
            && (streamInfo.nChannels == audioStream.nChannels)
            && (streamInfo.bitResolution == audioStream.bitResolution)
            && (streamInfo.subFrameSize == audioStream.subFrameSize))
        {
            return streamInfo.streamObj;
        }
    }
    return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;
}

*****
/* Audio attach event listener function */
*****
void APP_USBHostAudioAttachEventListener
(
    USB_HOST_AUDIO_V1_0_OBJ audioObj,
    USB_HOST_AUDIO_V1_0_EVENT event,
    uintptr_t context
)
{
    /* This function gets called when the Audio v1.0 device is attached/detached. In this
     * example we let the application know that a device is attached and we
     * store the Audio v1.0 device object. This object will be required to open the
     * device. */
    switch (event)
    {
        case USB_HOST_AUDIO_V1_0_EVENT_ATTACH:
            if (isAudioDeviceAttached == false)
            {
                isAudioDeviceAttached = true;
                audioDeviceObj = audioObj;
            }
            else
            {
                /* This application supports only one Audio Device . Handle Error Here.*/
            }
            break;
        case USB_HOST_AUDIO_V1_0_EVENT_DETACH:
            if (isAudioDeviceAttached == true)
            {
                /* This means the device was detached. There is no event data
                 * associated with this event.*/
                isAudioDeviceAttached = false;
            }
    }
}
```

```

        break;
    }
    break;
}
}

/*****************************************/
/* Audio Tasks function */
/*****************************************/
void APP_Tasks ( void )
{
    USB_HOST_AUDIO_V1_0_RESULT audioResult;
    USB_HOST_AUDIO_V1_0_STREAM_RESULT streamResult;

    /* Check the application's current state. */
    switch ( appData.state )
    {
        case APP_STATE_BUS_ENABLE:

            /* Register a callback for Audio Device Attach. */
            audioResult = USB_HOST_AUDIO_V1_0_AttachEventHandlerSet
                (
                    &APP_USBHostAudioAttachEventListenner,
                    (uintptr_t)0
                );

            if (audioResult == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS )
            {
                /* Set Host Event Handler */
                USB_HOST_EventHandlerSet(APP_USBHostEventHandler, 0);
                USB_HOST_BusEnable(0);
                /* Advance application state */
                appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            }
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:
            if(USB_HOST_BusIsEnabled(0))
            {
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            }
            break;

        case APP_STATE_WAIT_FOR_DEVICE_ATTACH:
            /* Check if an Audio Device has been attached */
            if(appData.isAudioDeviceAttached == true)
            {
                appData.nAudioStreamGroups = 0;
                /* Find an Audio Stream matching to our requirement */
                appData.ouStreamObj = App_USBHostAudioSpeakerStreamFind
                    (
                        appData.audioDeviceObj,
                        audioSpeakerStreamFormat,
                        &appData.nAudioStreamGroups
                    );
                if (appData.nAudioStreamGroups == 0)
                {
                    appData.state = APP_STATE_ERROR;
                    break;
                }

                /* Open Audio Stream */
                appData.outStreamHandle = USB_HOST_AUDIO_V1_0_StreamOpen
                    (
                        appData.ouStreamObj
                    );

                if (appData.outStreamHandle == USB_HOST_AUDIO_V1_0_STREAM_HANDLE_INVALID)
                {

```

```

        appData.state = APP_STATE_ERROR;
        break;
    }

    /* Set Stream Event Handler */
    streamResult = USB_HOST_AUDIO_V1_0_StreamEventHandlerSet
    (
        appData.outStreamHandle,
        APP_USBHostAudioStreamEventHandler,
        (uintptr_t)appData.ouStreamObj
    );

    if (streamResult != USB_HOST_AUDIO_V1_0_STREAM_SUCCESS)
    {
        appData.state = APP_STATE_ERROR;
        break;
    }
}
break;

default:
    break;
}
}

```

Enabling Audio Stream

Describes how to enable an audio stream, which includes a code example.

Description

An audio stream must be enabled before doing any data transfer operation. An audio stream enable or disable can be scheduled by using [USB_HOST_AUDIO_V1_0_StreamEnable](#) or [USB_HOST_AUDIO_V1_0_StreamDisable](#) functions. Return values of these function indicates if the request has been placed successfully or failed. When the audio stream enable request is completed, stream event handler generates an event [USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE](#). Similarly it generates an event [USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE](#) when stream disable is complete. The event data [USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_DATA](#) has details like request handle and termination status. The [requestStatus](#) member of the [USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_DATA](#) indicates if the request was success or failed. When audio stream multiple audio streams with an audio stream group cannot be enabled at the same time. The following code shows an example of how an Audio Stream is enabled.

```

/* This code shows an example of enabling an audio stream */

/* Specify the Audio Stream format details that this application supports */
const APP_USB_HOST_AUDIO_STREAM_FORMAT audioSpeakerStreamFormat =
{
    .streamDirection = USB_HOST_AUDIO_V1_0_DIRECTION_OUT,
    .format = USB_AUDIO_FORMAT_PCM,
    .nChannels = 2,
    .bitResolution = 16,
    .subFrameSize = 2,
    .samplingRate = 48000
};

bool isAudioDeviceAttached = false;
bool isStreamEnabled = false;
USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj;

/********************* *
 * Audio Stream Event Handler function.
***** */

USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE APP_USBHostAudioStreamEventHandler
(
    USB_HOST_AUDIO_V1_0_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_0_STREAM_EVENT event,
    void * eventData,
    uintptr_t context
)
```

```
)  
{  
    USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA * writeCompleteEventData;  
    switch(event)  
    {  
        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE:  
  
            break;  
  
        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE:  
            /* Check eventData result member to know if stream enable is complete */  
            isStreamEnabled = true;  
  
            break;  
        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE:  
  
            break;  
        default:  
            break;  
    }  
    return USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE_NONE;  
}  
/**************************************************************************/  
/* Function to search for a specific Audio Stream */  
/**************************************************************************/  
USB_HOST_AUDIO_V1_0_STREAM_OBJ App_USBHostAudioSpeakerStreamFind  
(  
    USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj,  
    APP_USB_HOST_AUDIO_STREAM_FORMAT audioStream,  
    uint8_t* numberofStreamGroups  
)  
{  
    USB_HOST_AUDIO_V1_0_RESULT result;  
    USB_HOST_AUDIO_V1_0_STREAM_INFO streamInfo;  
  
    /* Get Number of Stream Groups */  
    *numberofStreamGroups = USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet(audioDeviceObj);  
    if (*numberofStreamGroups == 0)  
    {  
        return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;  
    }  
    /* Get the First Stream Information in the Stream Group */  
    result = USB_HOST_AUDIO_V1_0_StreamGetFirst(appData.audioDeviceObj, 0, &streamInfo);  
    if (result == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS)  
    {  
        /* Compare Audio Stream info */  
        if ((streamInfo.format == audioStream.format)  
            && (streamInfo.streamDirection == audioStream.streamDirection)  
            && (streamInfo.nChannels == audioStream.nChannels)  
            && (streamInfo.bitResolution == audioStream.bitResolution)  
            && (streamInfo.subFrameSize == audioStream.subFrameSize))  
        {  
            return streamInfo.streamObj;  
        }  
    }  
    return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;  
}  
/**************************************************************************/  
/* Audio attach event listener function */  
/**************************************************************************/  
void APP_USBHostAudioAttachEventListener  
(  
    USB_HOST_AUDIO_V1_0_OBJ audioObj,  
    USB_HOST_AUDIO_V1_0_EVENT event,  
    uintptr_t context  
)  
{
```

```

/* This function gets called when the Audio v1.0 device is attached/detached. In this
 * example we let the application know that a device is attached and we
 * store the Audio v1.0 device object. This object will be required to open the
 * device. */
switch (event)
{
    case USB_HOST_AUDIO_V1_0_EVENT_ATTACH:
        if (isAudioDeviceAttached == false)
        {
            isAudioDeviceAttached = true;
            audioDeviceObj = audioObj;
        }
        else
        {
            /* This application supports only one Audio Device . Handle Error Here.*/
        }
    break;
    case USB_HOST_AUDIO_V1_0_EVENT_DETACH:
        if (isAudioDeviceAttached == true)
        {
            /* This means the device was detached. There is no event data
             * associated with this event.*/
            isAudioDeviceAttached = false;
        }
    break;
}
break;
}

/*****************************************/
/* Audio Tasks function */
/*****************************************/
void APP_Tasks ( void )
{
    USB_HOST_AUDIO_V1_0_RESULT audioResult;
    USB_HOST_AUDIO_V1_0_STREAM_RESULT streamResult;

    /* Check the application's current state. */
    switch ( appData.state )
    {
        case APP_STATE_BUS_ENABLE:

            /* Register a callback for Audio Device Attach. */
            audioResult = USB_HOST_AUDIO_V1_0_AttachEventHandlerSet
                (
                    &APP_USBHostAudioAttachEventListerner,
                    (uintptr_t)0
                );

            if (audioResult == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS )
            {
                /* Set Host Event Handler */
                USB_HOST_EventHandlerSet(APP_USBHostEventHandler, 0);
                USB_HOST_BusEnable(0);
                /* Advance application state */
                appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            }
        break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:
            if(USB_HOST_BusIsEnabled(0))
            {
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            }
        break;

        case APP_STATE_WAIT_FOR_DEVICE_ATTACH:
            /* Check if an Audio Device has been attached */
            if(appData.isAudioDeviceAttached == true)

```

```
{  
    appData.nAudioStreamGroups = 0;  
    /* Find an Audio Stream matching to our requirement */  
    appData.ouStreamObj = App_USBHostAudioSpeakerStreamFind  
    (  
        appData.audioDeviceObj,  
        audioSpeakerStreamFormat,  
        &appData.nAudioStreamGroups  
    );  
    if (appData.nAudioStreamGroups == 0)  
    {  
        appData.state = APP_STATE_ERROR;  
        break;  
    }  
  
    /* Open Audio Stream */  
    appData.outStreamHandle = USB_HOST_AUDIO_V1_0_StreamOpen  
    (  
        appData.ouStreamObj  
    );  
  
    if (appData.outStreamHandle == USB_HOST_AUDIO_V1_0_STREAM_HANDLE_INVALID)  
    {  
        appData.state = APP_STATE_ERROR;  
        break;  
    }  
  
    /* Set Stream Event Handler */  
    streamResult = USB_HOST_AUDIO_V1_0_StreamEventHandlerSet  
    (  
        appData.outStreamHandle,  
        APP_USBHostAudioStreamEventHandler,  
        (uintptr_t)appData.ouStreamObj  
    );  
  
    if (streamResult != USB_HOST_AUDIO_V1_0_STREAM_SUCCESS)  
    {  
        appData.state = APP_STATE_ERROR;  
        break;  
    }  
    appData.state = APP_STATE_ENABLE_AUDIO_STREAM;  
}  
break;  
  
case APP_STATE_ENABLE_AUDIO_STREAM:  
    isStreamEnableComplete = false;  
    /* Set default interface setting of the streaming interface */  
    streamResult = USB_HOST_AUDIO_V1_0_StreamEnable  
    (  
        appData.outStreamHandle,  
        &appData.requestHandle  
    );  
    if (streamResult != USB_HOST_AUDIO_V1_0_STREAM_SUCCESS)  
    {  
        appData.state = APP_STATE_ERROR;  
        break;  
    }  
    appData.state = APP_STATE_WAIT_FOR_ENABLE_AUDIO_STREAM;  
break;  
case APP_STATE_WAIT_FOR_ENABLE_AUDIO_STREAM:  
    if (isStreamEnabled == true)  
    {  
        /* stream enable complete */  
    }  
break;  
  
default:  
    break;  
}
```

```
}
```

Setting the Desired Audio Stream Sampling Rate

Describes how to set the desired audio stream sampling rate, which includes a code example.

Description

Sampling rate of an audio stream can be set using `USB_HOST_AUDIO_V1_0_StreamSamplingRateSet` function. Supported sampling rates for an audio stream is returned as part of `USB_HOST_AUDIO_V1_0_STREAM_INFO` by the `USB_HOST_AUDIO_V1_0_StreamGetFirst` and `USB_HOST_AUDIO_V1_0_StreamGetNext` functions. Return values of these function indicates if the request has been placed successfully or failed. When the set sampling rate request is completed the stream event handler generates an event `USB_HOST_AUDIO_V1_0_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE`. The event data `USB_HOST_AUDIO_V1_0_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE_DATA` has request handle and the `requestStatus` which indicates the set sampling request was accepted by the device or failed. The following code shows an example of how sampling rates can be set in an audio stream.

```
/* This code shows an example of Set sampling rate to an audio stream */

/* Specify the Audio Stream format details that this application supports */
const APP_USB_HOST_AUDIO_STREAM_FORMAT audioSpeakerStreamFormat =
{

    .streamDirection = USB_HOST_AUDIO_V1_0_DIRECTION_OUT,
    .format = USB_AUDIO_FORMAT_PCM,
    .nChannels = 2,
    .bitResolution = 16,
    .subFrameSize = 2,
    .samplingRate = 48000
};

bool isAudioDeviceAttached = false;
bool isStreamEnabled = false;
bool isSampleRateSetComplete = false;
USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj;

/*********************  

 * Audio Stream Event Handler function.  

 *******************/  

USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE APP_USBHostAudioStreamEventHandler
(
    USB_HOST_AUDIO_V1_0_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_0_STREAM_EVENT event,
    void * eventData,
    uintptr_t context
)
{
    USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA * writeCompleteEventData;
    switch(event)
    {
        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE:
            break;

        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE:
            /* Check eventData result member to know if stream enable is complete */
            isStreamEnabled = true;
            break;
        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE:
            break;

        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE:
            /* Check eventData result member to know if stream enable is complete */
            isSampleRateSetComplete = true;
    }
}
```

```

        break;

    default:
        break;
    }
    return USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE_NONE;
}
//*****************************************************************************
/* Function to search for a specific Audio Stream */
//*****************************************************************************
USB_HOST_AUDIO_V1_0_STREAM_OBJ App_USBHostAudioSpeakerStreamFind
(
    USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj,
    APP_USB_HOST_AUDIO_STREAM_FORMAT audioStream,
    uint8_t* numberofStreamGroups
)
{
    USB_HOST_AUDIO_V1_0_RESULT result;
    USB_HOST_AUDIO_V1_0_STREAM_INFO streamInfo;

    /* Get Number of Stream Groups */
    *numberofStreamGroups = USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet(audioDeviceObj);
    if (*numberofStreamGroups == 0)
    {
        return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;
    }
    /* Get the First Stream Information in the Stream Group */
    result = USB_HOST_AUDIO_V1_0_StreamGetFirst(appData.audioDeviceObj, 0, &streamInfo);
    if (result == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS)
    {
        /* Compare Audio Stream info */
        if ((streamInfo.format == audioStream.format)
            && (streamInfo.streamDirection == audioStream.streamDirection)
            && (streamInfo.nChannels == audioStream.nChannels)
            && (streamInfo.bitResolution == audioStream.bitResolution)
            && (streamInfo.subFrameSize == audioStream.subFrameSize))
        {
            return streamInfo.streamObj;
        }
    }
    return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;
}

//*****************************************************************************
/* Audio attach event listener function */
//*****************************************************************************
void APP_USBHostAudioAttachEventListener
(
    USB_HOST_AUDIO_V1_0_OBJ audioObj,
    USB_HOST_AUDIO_V1_0_EVENT event,
    uintptr_t context
)
{
    /* This function gets called when the Audio v1.0 device is attached/detached. In this
     * example we let the application know that a device is attached and we
     * store the Audio v1.0 device object. This object will be required to open the
     * device. */
    switch (event)
    {
        case USB_HOST_AUDIO_V1_0_EVENT_ATTACH:
            if (isAudioDeviceAttached == false)
            {
                isAudioDeviceAttached = true;
                audioDeviceObj = audioObj;
            }
            else
            {
                /* This application supports only one Audio Device . Handle Error Here.*/
            }
    }
}

```

```

break;
case USB_HOST_AUDIO_V1_0_EVENT_DETACH:
    if (isAudioDeviceAttached == true)
    {
        /* This means the device was detached. There is no event data
         * associated with this event.*/
        isAudioDeviceAttached = false;
        break;
    }
    break;
}

/*****************************************/
/* Audio Tasks function */
/*****************************************/
void APP_Tasks ( void )
{
    USB_HOST_AUDIO_V1_0_RESULT audioResult;
    USB_HOST_AUDIO_V1_0_STREAM_RESULT streamResult;

    /* Check the application's current state. */
    switch ( appData.state )
    {
        case APP_STATE_BUS_ENABLE:

            /* Register a callback for Audio Device Attach. */
            audioResult = USB_HOST_AUDIO_V1_0_AttachEventHandlerSet
                (
                    &APP_USBHostAudioAttachEventListener,
                    (uintptr_t)0
                );

            if (audioResult == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS )
            {
                /* Set Host Event Handler */
                USB_HOST_EventHandlerSet(APP_USBHostEventHandler, 0);
                USB_HOST_BusEnable(0);
                /* Advance application state */
                appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            }
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:
            if(USB_HOST_BusIsEnabled(0))
            {
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            }
            break;

        case APP_STATE_WAIT_FOR_DEVICE_ATTACH:
            /* Check if an Audio Device has been attached */
            if(appData.isAudioDeviceAttached == true)
            {
                appData.nAudioStreamGroups = 0;
                /* Find an Audio Stream matching to our requirement */
                appData.ouStreamObj = App_USBHostAudioSpeakerStreamFind
                    (
                        appData.audioDeviceObj,
                        audioSpeakerStreamFormat,
                        &appData.nAudioStreamGroups
                    );
                if (appData.nAudioStreamGroups == 0)
                {
                    appData.state = APP_STATE_ERROR;
                    break;
                }
            }

            /* Open Audio Stream */

```

```
appData.outStreamHandle = USB_HOST_AUDIO_V1_0_StreamOpen
(
    appData.outStreamObj
);

if (appData.outStreamHandle == USB_HOST_AUDIO_V1_0_STREAM_HANDLE_INVALID)
{
    appData.state = APP_STATE_ERROR;
    break;
}

/* Set Stream Event Handler */
streamResult = USB_HOST_AUDIO_V1_0_StreamEventHandlerSet
(
    appData.outStreamHandle,
    APP_USBHostAudioStreamEventHandler,
    (uintptr_t)appData.outStreamObj
);

if (streamResult != USB_HOST_AUDIO_V1_0_STREAM_SUCCESS)
{
    appData.state = APP_STATE_ERROR;
    break;
}
appData.state = APP_STATE_ENABLE_AUDIO_STREAM;
}
break;

case APP_STATE_ENABLE_AUDIO_STREAM:
isStreamEnableComplete = false;
/* Set default interface setting of the streaming interface */
streamResult = USB_HOST_AUDIO_V1_0_StreamEnable
(
    appData.outStreamHandle,
    &appData.requestHandle
);
if (streamResult != USB_HOST_AUDIO_V1_0_STREAM_SUCCESS)
{
    appData.state = APP_STATE_ERROR;
    break;
}
appData.state = APP_STATE_WAIT_FOR_ENABLE_AUDIO_STREAM;
break;
case APP_STATE_WAIT_FOR_ENABLE_AUDIO_STREAM:
if (isStreamEnabled == true)
{
    /* Set sampling rate 48000 Hz */
    isSampleRateSetComplete = false;
    streamResult = USB_HOST_AUDIO_V1_0_StreamSamplingRateSet
    (
        appData.outStreamHandle,
        &appData.requestHandle,
        48000
    );
    if (streamResult != USB_HOST_AUDIO_V1_0_STREAM_SUCCESS)
    {
        appData.state = APP_STATE_ERROR;
        break;
    }
    appData.state = APP_STATE_WAIT_FOR_SAMPLE_RATE_SET_COMPLETE;
}

break;
case APP_STATE_WAIT_FOR_SAMPLE_RATE_SET_COMPLETE:
if (isSampleRateSetComplete == true)
{
    /* Set sampling rate completed */
}
default:
```

```

        break;
    }
}

```

Audio Data Streaming

Describes how to transfer data to an audio stream, which includes a code example.

Description

The application can use the [USB_HOST_AUDIO_V1_0_StreamRead](#) and [USB_HOST_AUDIO_V1_0_StreamWrite](#) functions to transfer data to an Audio Stream. While calling these functions, the stream handle specifies the target Audio stream and the event handler function to which the events should be sent. It is possible for multiple clients to open the same audio stream and transfer data to the stream.

Calling the [USB_HOST_AUDIO_V1_0_StreamRead](#) and [USB_HOST_AUDIO_V1_0_StreamWrite](#) functions while a read/write transfer is already in progress will cause the transfer result to be queued. If the transfer was successfully queued or scheduled, the

[USB_HOST_AUDIO_V1_0_StreamRead](#) and [USB_HOST_AUDIO_V1_0_StreamWrite](#) functions will return a valid transfer handle. This transfer handle identifies the transfer request. The application clients can use the transfer handles to keep track of multiple queued transfers. When a transfer completes, the Audio stream handler generates an event. The following table shows the event and the event data associated with the event.

Table 1: Read

Function	USB_HOST_AUDIO_V1_0_StreamRead
Event	USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE
Event Data Type	USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE _DATA

Table 2: Write

Function	USB_HOST_AUDIO_V1_0_StreamWrite
Event	USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE
Event Data Type	USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE _DATA

The event data contains information on the amount of data transferred, completion status and the transfer handle of the transfer. The following code shows an example of reading and writing data.

/ This code shows an example of audio data streaming */*

```

/* PCM16 samples for 1Khz Sine Wave at 48 kHz Sample Rate */
uint16_t audioSamples[96] = {
    0x0000, 0x0000, //Sample 1
    0x10B4, 0x10B4, //Sample 2
    0x2120, 0x2120, //Sample 3
    0x30FB, 0x30FB, //Sample 4
    0x3FFF, 0x3FFF, //Sample 5
    0x4DEB, 0x4DEB, //Sample 6
    0x5A81, 0x5A81, //Sample 7
    0x658B, 0x658B, //Sample 8
    0x6ED9, 0x6ED9, //Sample 9
    0x7640, 0x7640, //Sample 10
    0x7BA2, 0x7BA2, //Sample 11
    0x7EE6, 0x7EE6, //Sample 12
    0x7FFF, 0x7FFF, //Sample 13
    0x7FE6, 0x7FE6, //Sample 14
    0x7BA2, 0x7BA2, //Sample 15
    0x7640, 0x7640, //Sample 16
    0x6ED9, 0x6ED9, //Sample 17
    0x658B, 0x658B, //Sample 18
    0x5A81, 0x5A81, //Sample 19
    0x4DEB, 0x4DEB, //Sample 20
    0x3FFF, 0x3FFF, //Sample 21
    0x30FB, 0x30FB, //Sample 22
    0x2120, 0x2120, //Sample 23
    0x10B4, 0x10B4, //Sample 24
    0x0000, 0x0000, //Sample 25
    0xEF4C, 0xEF4C, //Sample 26
    0xDEE0, 0xDEE0, //Sample 27
    0xCF05, 0xCF05, //Sample 28
    0xC001, 0xC001, //Sample 29
}

```

```

0xB215, 0xB215, //Sample 30
0xA57F, 0xA57F, //Sample 31
0x9A75, 0x9A75, //Sample 32
0x9127, 0x9127, //Sample 33
0x89C0, 0x89C0, //Sample 34
0x845E, 0x845E, //Sample 35
0x811A, 0x811A, //Sample 36
0x8001, 0x8001, //Sample 37
0x811A, 0x811A, //Sample 38
0x845E, 0x845E, //Sample 39
0x89C0, 0x89C0, //Sample 40
0x9127, 0x9127, //Sample 41
0x9A75, 0x9A75, //Sample 42
0xA57F, 0xA57F, //Sample 43
0xB215, 0xB215, //Sample 44
0xC001, 0xC001, //Sample 45
0xCF05, 0xCF05, //Sample 46
0xDEE0, 0xDEE0, //Sample 47
0xFF4C, 0xFF4C, //Sample 48
};

/* Specify the Audio Stream format details that this application supports */
const APP_USB_HOST_AUDIO_STREAM_FORMAT audioSpeakerStreamFormat =
{
    .streamDirection = USB_HOST_AUDIO_V1_0_DIRECTION_OUT,
    .format = USB_AUDIO_FORMAT_PCM,
    .nChannels = 2,
    .bitResolution = 16,
    .subFrameSize = 2,
    .samplingRate = 48000
};

bool isAudioDeviceAttached = false;
bool isStreamEnabled = false;
bool isAudioWriteCompleted = false;
USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj;
USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE transferHandleAudioWrite;

/**************************************************************************
 * Audio Stream Event Handler function.
 **************************************************************************/
USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE APP_USBHostAudioStreamEventHandler(
{
    USB_HOST_AUDIO_V1_0_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_0_STREAM_EVENT event,
    void * eventData,
    uintptr_t context
}
{
    USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA * writeCompleteEventData;
    switch(event)
    {
        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE:
            break;

        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE:
            /* Check eventData result member to know if stream enable is complete */
            isStreamEnabled = true;
            break;
        case USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE:
            /* This means the Write request completed. We can
             * find out if the request was successful. */
    }
}

```

```

        writeCompleteEventData =
            (USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA*)eventData;
        if(transferHandleAudioWrite == writeCompleteEventData->transferHandle)
        {
            isAudioWriteCompleted = true;
        }
        break;
    default:
        break;
    }
    return USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE_NONE;
}
//*****************************************************************************
/* Function to search for a specific Audio Stream */
//*****************************************************************************
USB_HOST_AUDIO_V1_0_STREAM_OBJ App_USBHostAudioSpeakerStreamFind(
{
    USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj,
    APP_USB_HOST_AUDIO_STREAM_FORMAT audioStream,
    uint8_t* numberofStreamGroups
)
{
    USB_HOST_AUDIO_V1_0_RESULT result;
    USB_HOST_AUDIO_V1_0_STREAM_INFO streamInfo;

    /* Get Number of Stream Groups */
    *numberofStreamGroups = USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet(audioDeviceObj);
    if (*numberofStreamGroups == 0)
    {
        return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;
    }
    /* Get the First Stream Information in the Stream Group */
    result = USB_HOST_AUDIO_V1_0_StreamGetFirst(appData.audioDeviceObj, 0, &streamInfo);
    if (result == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS)
    {
        /* Compare Audio Stream info */
        if ((streamInfo.format == audioStream.format)
            && (streamInfo.streamDirection == audioStream.streamDirection)
            && (streamInfo.nChannels == audioStream.nChannels)
            && (streamInfo.bitResolution == audioStream.bitResolution)
            && (streamInfo.subFrameSize == audioStream.subFrameSize))
        {
            return streamInfo.streamObj;
        }
    }
    return (USB_HOST_AUDIO_V1_0_STREAM_OBJ)0;
}

//*****************************************************************************
/* Audio attach event listener function */
//*****************************************************************************
void APP_USBHostAudioAttachEventListener(
{
    USB_HOST_AUDIO_V1_0_OBJ audioObj,
    USB_HOST_AUDIO_V1_0_EVENT event,
    uintptr_t context
)
{
    /* This function gets called when the Audio v1.0 device is attached/detached. In this
     * example we let the application know that a device is attached and we
     * store the Audio v1.0 device object. This object will be required to open the
     * device. */
    switch (event)
    {
        case USB_HOST_AUDIO_V1_0_EVENT_ATTACH:
            if (isAudioDeviceAttached == false)
            {
                isAudioDeviceAttached = true;
                audioDeviceObj = audioObj;
            }
    }
}

```

```

        }
    else
    {
        /* This application supports only one Audio Device . Handle Error Here.*/
    }
break;
case USB_HOST_AUDIO_V1_0_EVENT_DETACH:
    if (isAudioDeviceAttached == true)
    {
        /* This means the device was detached. There is no event data
         * associated with this event.*/
        isAudioDeviceAttached = false;
    }
break;
}
break;
}

/*****************************************/
/* Audio Tasks function */
/*****************************************/
void APP_Tasks ( void )
{
    USB_HOST_AUDIO_V1_0_RESULT audioResult;
    USB_HOST_AUDIO_V1_0_STREAM_RESULT streamResult;

    /* Check the application's current state. */
    switch ( appData.state )
    {
        case APP_STATE_BUS_ENABLE:

            /* Register a callback for Audio Device Attach. */
            audioResult = USB_HOST_AUDIO_V1_0_AttachEventHandlerSet
                (
                    &APP_USBHostAudioAttachEventListenner,
                    (uintptr_t)0
                );

            if (audioResult == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS )
            {
                /* Set Host Event Handler */
                USB_HOST_EventHandlerSet(APP_USBHostEventHandler, 0);
                USB_HOST_BusEnable(0);
                /* Advance application state */
                appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            }
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:
            if(USB_HOST_BusIsEnabled(0))
            {
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            }
            break;

        case APP_STATE_WAIT_FOR_DEVICE_ATTACH:
            /* Check if an Audio Device has been attached */
            if(appData.isAudioDeviceAttached == true)
            {
                appData.nAudioStreamGroups = 0;
                /* Find an Audio Stream matching to our requirement */
                appData.ouStreamObj = App_USBHostAudioSpeakerStreamFind
                    (
                        appData.audioDeviceObj,
                        audioSpeakerStreamFormat,
                        &appData.nAudioStreamGroups
                    );
                if (appData.nAudioStreamGroups == 0)
                {

```

```
        appData.state = APP_STATE_ERROR;
        break;
    }

    /* Open Audio Stream */
    appData.outStreamHandle = USB_HOST_AUDIO_V1_0_StreamOpen
    (
        appData.outStreamObj
    );

    if (appData.outStreamHandle == USB_HOST_AUDIO_V1_0_STREAM_HANDLE_INVALID)
    {
        appData.state = APP_STATE_ERROR;
        break;
    }

    /* Set Stream Event Handler */
    streamResult = USB_HOST_AUDIO_V1_0_StreamEventHandlerSet
    (
        appData.outStreamHandle,
        APP_USBHostAudioStreamEventHandler,
        (uintptr_t)appData.outStreamObj
    );

    if (streamResult != USB_HOST_AUDIO_V1_0_STREAM_SUCCESS)
    {
        appData.state = APP_STATE_ERROR;
        break;
    }
    appData.state = APP_STATE_ENABLE_AUDIO_STREAM;
}
break;

case APP_STATE_ENABLE_AUDIO_STREAM:
    isStreamEnableComplete = false;
    /* Set default interface setting of the streaming interface */
    streamResult = USB_HOST_AUDIO_V1_0_StreamEnable
    (
        appData.outStreamHandle,
        &appData.requestHandle
    );
    if (streamResult != USB_HOST_AUDIO_V1_0_STREAM_SUCCESS)
    {
        appData.state = APP_STATE_ERROR;
        break;
    }
    appData.state = APP_STATE_WAIT_FOR_ENABLE_AUDIO_STREAM;
break;
case APP_STATE_WAIT_FOR_ENABLE_AUDIO_STREAM:
    if (isStreamEnabled == true)
    {
        appData.state = APP_STATE_START_STREAM_DATA;
    }
break;
case APP_STATE_START_STREAM_DATA:
    isAudioWriteCompleted = false;
    appData.state = APP_STATE_WAIT_FOR_WRITE_COMPLETE;
    USB_HOST_AUDIO_V1_0_StreamWrite
    (
        appData.outStreamHandle,
        &transferHandleAudioWrite,
        (void*)&audioSamples,
        192
    );
break;

case APP_STATE_WAIT_FOR_WRITE_COMPLETE:
    if (appData.isAudioWriteCompleted)
    {
```

```

        isAudioWriteCompleted = false;
        USB_HOST_AUDIO_V1_0_StreamWrite
        (
            appData.outStreamHandle,
            &transferHandleAudioWrite,
            (void*)&audioSamples,
            192
        );
    }
    break;
}

default:
    break;
}
}

```

Sending Class Specific Control Transfers

Describes how to send class-specific control transfers to the connected device, which includes a code example.

Description

The Audio v1.0 Host Client Driver allows the application client to send Audio v1.0 Class specific commands to the connected device. These commands can be send using [USB_HOST_AUDIO_V1_0_ControlRequest](#) function.

This function is non-blocking. The functions will return before the actual command execution is complete. The return value indicates if the command was scheduled successfully, or if the driver is busy and cannot accept commands, or if the command failed due to an unknown reason. If the command failed because the driver was busy, it can be retried. If scheduled successfully, the function will return a valid request handle. This request handle is unique and tracks the requested command.

When the command related control transfer has completed, the Audio v1.0 Host Client Driver generates a callback function. The call back function is one of the argument to the [USB_HOST_AUDIO_V1_0_ControlRequest](#) function.

The following code shows an example of sending a Audio v1.0 class specific commands.

```

/* This code shows an example for Audio Control transfer */
bool isAudioDeviceAttached = false;
USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj;

/*************************************************************************/
/* Audio control request call back function   */
/*************************************************************************/
void App_USBAudioControlRequestCallback
(
    USB_HOST_AUDIO_V1_0_OBJ audioObj,
    USB_HOST_AUDIO_V1_0_REQUEST_HANDLE requestHandle,
    USB_HOST_AUDIO_V1_0_RESULT result,
    size_t size,
    uintptr_t context
)
{
    APP_USB_AUDIO_CONTROL_TRANSFER_ACTION controlAction = (APP_USB_AUDIO_CONTROL_TRANSFER_ACTION)context;
    switch (controlAction)
    {
        case APP_USB_AUDIO_MASTER_UNMUTE_SET:
            if (result == USB_HOST_AUDIO_V1_0_RESULT_SUCCESS)
            {
                appData.isMasterUnmuteSetComplete = true;
            }
            else
            {
                appData.muteStatus = 1;
            }

        break;
        default:
            break;
    }
}

```

```
*****
/* Function for sending Mute control to Audio device. */
*****
void APP_SendAudioMuteControl
(
    APP_USB_AUDIO_CONTROL_TRANSFER_ACTION action,
    uint32_t* mute
)
{
    USB_HOST_AUDIO_V1_0_RESULT result;
    USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST setupPacket;
    uint32_t status;

    /* Fill in Setup Packet */
    setupPacket.bmRequestType = ( USB_SETUP_DIRN_HOST_TO_DEVICE
        | USB_SETUP_TYPE_CLASS
        | USB_SETUP_RECIPIENT_INTERFACE
        ); //interface , Host to device , Standard;
    setupPacket.bRequest = USB_AUDIO_CS_SET_CUR;
    if (action == APP_USB_AUDIO_MASTER_MUTE_SET)
    {
        setupPacket.channelNumber = APP_USB_AUDIO_CHANNEL_MASTER;
        status = __builtin_disable_interrupts();
        *mute = 1;
        __builtin_mtc0(12,0,status);
    }
    else if (action == APP_USB_AUDIO_MASTER_UNMUTE_SET)
    {
        setupPacket.channelNumber = APP_USB_AUDIO_CHANNEL_MASTER;
        status = __builtin_disable_interrupts();
        *mute = 0;
        __builtin_mtc0(12,0,status);
    }

    setupPacket.controlSelector = USB_AUDIO_MUTE_CONTROL;
    setupPacket.featureUnitId = 0x02; //appData.featureUnitDescriptor->bUnitID;
    setupPacket.wLength = 1;
    result = USB_HOST_AUDIO_V1_0_ControlRequest
    (
        appData.audioDeviceObj,
        &appData.requestHandle,
        (USB_SETUP_PACKET *)&setupPacket,
        mute,
        App_USBAudioControlRequestCallback,
        (uintptr_t)action
    );
}

/*****
/* Audio attach event listener function */
*****
void APP_USBHostAudioAttachEventListener
(
    USB_HOST_AUDIO_V1_0_OBJ audioObj,
    USB_HOST_AUDIO_V1_0_EVENT event,
    uintptr_t context
)
{
    /* This function gets called when the Audio v1.0 device is attached/detached. In this
     * example we let the application know that a device is attached and we
     * store the Audio v1.0 device object. This object will be required to open the
     * device. */
    switch (event)
    {
        case USB_HOST_AUDIO_V1_0_EVENT_ATTACH:
            if (isAudioDeviceAttached == false)
            {

```

```

        isAudioDeviceAttached = true;
        audioDeviceObj = audioObj;
    }
else
{
    /* This application supports only one Audio Device . Handle Error Here.*/
}
break;
case USB_HOST_AUDIO_V1_0_EVENT_DETACH:
    if (isAudioDeviceAttached == true)
    {
        /* This means the device was detached. There is no event data
         * associated with this event.*/
        isAudioDeviceAttached = false;
        break;
    }
    break;
}

void APP_Tasks ( void )
{
    switch (appData.state)
    {
        case APP_STATE_BUS_ENABLE:

            /* In this state the application enables the USB Host Bus. Note
             * how the Audio v1.0 Attach event handler is registered before the bus
             * is enabled. */

            USB_HOST_AUDIO_V1_0_AttachEventHandlerSet(APP_USBHostAudioAttachEventListener, (uintptr_t) 0);
            USB_HOST_BusEnable(0);
            appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:
            if(USB_HOST_BusEnabled(0) != true)
            {
                return;
            }
            /* Here we wait for the bus enable operation to complete. */
            /* Unmute the Device */
            appData.isMasterUnmuteSetComplete = false;
            APP_SendAudioMuteControl
            (
                APP_USB_AUDIO_MASTER_UNMUTE_SET,
                (uint32_t*)&appData.muteStatus
            );
            appData.state = APP_STATE_AUDIO_WAIT_FOR_UNMUTE_COMPLETE;
            break;
        case APP_STATE_AUDIO_WAIT_FOR_UNMUTE_COMPLETE:
            if (appData.isMasterUnmuteSetComplete == true)
            {
                /* Audio Control request completed */
            }
    }
}

```

Configuring the Library

Describes how to configure the USB Audio v1.0 Host Client Driver.

Macros

	Name	Description
	USB_HOST_AUDIO_V1_ATTACH_LISTENERS_NUMBER	Defines the number of attach event listeners that can be registered with Audio v1.0 Host Client Driver.
	USB_HOST_AUDIO_V1_INSTANCES_NUMBER	Specifies the number of Audio v1.0 instances.
	USB_HOST_AUDIO_V1_STREAMING_INTERFACE_ALTERNATE_SETTINGS_NUMBER	Defines maximum number of alternate settings per Streaming interface provided by any Device that will be connected to this Audio Host.
	USB_HOST_AUDIO_V1_STREAMING_INTERFACES_NUMBER	Defines the maximum number of streaming interfaces could be present in an Audio v1.0 device that this Audio v1.0 Host Client Driver can support.

Description

The USB Audio v1.0 Host Client Driver requires configuration constants to be specified in `system_config.h` file. These constants define the build time configuration (functionality and static resources) of the USB Audio v1.0 Host Client Driver.

USB_HOST_AUDIO_V1_ATTACH_LISTENERS_NUMBER Macro

Defines the number of attach event listeners that can be registered with Audio v1.0 Host Client Driver.

File

`usb_host_audio_v1_0_config_template.h`

C

```
#define USB_HOST_AUDIO_V1_ATTACH_LISTENERS_NUMBER
```

Description

USB Host Audio v1.0 Attach Listeners Number

The USB Audio v1.0 Host Client Driver provides attach notification to listeners who have registered with the client driver via the `USB_HOST_AUDIO_V1_0_AttachEventHandlerSet()` function. The `USB_HOST_AUDIO_V1_0_ATTACH_LISTENERS_NUMBER` configuration constant defines the maximum number of event handlers that can be set. This number should be set to equal the number of entities that interested in knowing when a Audio v1.0 device is attached.

Remarks

None.

USB_HOST_AUDIO_V1_INSTANCES_NUMBER Macro

Specifies the number of Audio v1.0 instances.

File

`usb_host_audio_v1_0_config_template.h`

C

```
#define USB_HOST_AUDIO_V1_INSTANCES_NUMBER
```

Description

USB Host Audio v1.0 Maximum Number of Instances

This macro defines the number of instances of the Audio v1.0 host Driver. For example, if the application needs to implement two instances of the Audio v1.0 host Driver should be set to 2.

Remarks

None.

USB_HOST_AUDIO_V1_STREAMING_INTERFACE_ALTERNATE_SETTINGS_NUMBER Macro

Defines maximum number of alternate settings per Streaming interface provided by any Device that will be connected to this Audio Host.

File

[usb_host_audio_v1_0_config_template.h](#)

C

```
#define USB_HOST_AUDIO_V1_STREAMING_INTERFACE_ALTERNATE_SETTINGS_NUMBER
```

Description

USB Host Audio v1.0 Streaming interface alternate setting number

This configuration constant defines maximum number of Streaming interface alternate settings provided by any Device that will be connected to this Audio Host. The value of this constant should be at-least 1.

Remarks

Supporting multiple alternate settings per streaming interfaces requires more data memory and processing time.

Example

If the USB Audio v1.0 Host application must support a USB Audio Device with 2 alternate settings including Alternate Setting 0 then this constant should be defined to 3.

USB_HOST_AUDIO_V1_STREAMING_INTERFACES_NUMBER Macro

Defines the maximum number of streaming interfaces could be present in an Audio v1.0 device that this Audio v1.0 Host Client Driver can support.

File

[usb_host_audio_v1_0_config_template.h](#)

C

```
#define USB_HOST_AUDIO_V1_STREAMING_INTERFACES_NUMBER
```

Description

USB Host Audio v1.0 Streaming Interfaces Number

This configuration constant defines maximum number of streaming interfaces could be present in an Audio v1.0 device that this Audio v1.0 Host Client Driver can support. The value of this constant should be atleast 1.

Example 1 - If the USB Audio v1.0 Host application must support a USB Headset, this constant should be set 2 as an Audio Headset will have atleast 2 Audio Streaming interfaces, one for Host to Device streaming and one for Device to Host streaming.

Example 2 - If the USB Audio v1.0 Host application must support a USB Speaker, this constant should be set 1 as an Audio Speaker will have atleast 1 Audio Streaming interface.

Remarks

Supporting multiple streaming interfaces requires more data memory and processing time.

Building the Library

Describes the files to be included in the project while using the USB Audio v1.0 Host Client Driver.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/usb.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
usb_host_audio_v1_0.h	This header file should be included in any .c file that accesses the USB Audio v1.0 Host Client Driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/usb_host_audio_v1_0.c	This file implements the USB Audio v1.0 Host Client Driver interface and should be included in the project if the USB Audio v1.0 Host Client Driver operation is desired.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB Audio v1.0 Host Client Driver Library depends on the following modules:

- [USB Host Layer Library](#)

Library Interface

a) Audio Device Access Functions

Name	Description
USB_HOST_AUDIO_V1_AttachEventHandlerSet	Sets an attach/detach event handler.
USB_HOST_AUDIO_V1_0_ControlRequest	Schedules an Audio v1.0 control transfer.
USB_HOST_AUDIO_V1_ControlEntityGetFirst	Retrieves the handle to the first audio control entity
USB_HOST_AUDIO_V1_ControlEntityGetNext	Retrieves the handle to the next audio control entity.
USB_HOST_AUDIO_V1_DeviceObjHandleGet	Returns the device object handle for this Audio v1.0 Device.
USB_HOST_AUDIO_V1_EntityObjectGet	Retrieves the entity object for the entity ID.
USB_HOST_AUDIO_V1_EntityRequestCallbackSet	Registers an audio entity request callback function with the Audio v1.0 Client Driver.
USB_HOST_AUDIO_V1_EntityTypeGet	Returns the entity type of the audio control entity.
USB_HOST_AUDIO_V1_FeatureUnitChannelMuteExists	Returns "true" if mute control exists for the specified channel of the feature unit.
USB_HOST_AUDIO_V1_FeatureUnitChannelMuteGet	Schedules a get mute control request to the specified channel.
USB_HOST_AUDIO_V1_FeatureUnitChannelMuteSet	Schedules a set mute control request to the specified channel.
USB_HOST_AUDIO_V1_FeatureUnitChannelNumbersGet	Returns the number of channels.
USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeExists	Returns "true" if volume control exists for the specified channel of the feature unit.
USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeGet	Schedules a get current volume control request to the specified channel.
USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSet	Schedules a set current volume control request to the specified channel.
USB_HOST_AUDIO_V1_FeatureUnitIDGet	Returns ID of the Feature Unit.
USB_HOST_AUDIO_V1_FeatureUnitSourceIDGet	Returns the ID of the unit or terminal to which this feature unit is connected.
USB_HOST_AUDIO_V1_TerminalAssociationGet	Returns the associated terminal ID of the audio control terminal.
USB_HOST_AUDIO_V1_TerminalIDGet	Returns the terminal ID of the audio control entity.
USB_HOST_AUDIO_V1_TerminalInputChannelNumbersGet	Returns the number of logical output channels in the terminal's output audio channel cluster.
USB_HOST_AUDIO_V1_TerminalSourceIDGet	Returns the ID of the unit or terminal to which this terminal is connected.
USB_HOST_AUDIO_V1_TerminalTypeGet	Returns the terminal type of the audio control entity.

b) Audio Stream Access Functions

Name	Description
USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet	Gets the number of stream groups present in the attached Audio v1.0 Device.
USB_HOST_AUDIO_V1_StreamClose	Closes the audio stream.
USB_HOST_AUDIO_V1_StreamEventHandlerSet	Registers an event handler with the Audio v1.0 Client Driver stream.

≡◊	USB_HOST_AUDIO_V1_0_StreamDisable	Schedules an audio stream disable request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceGetFirst	Gets the first streaming interface object from the attached Audio Device.
≡◊	USB_HOST_AUDIO_V1_0_StreamEnable	Schedules an audio stream enable request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceGetNext	Gets the next streaming interface object from the attached Audio Device.
≡◊	USB_HOST_AUDIO_V1_0_StreamEventHandlerSet	Registers an event handler with the Audio v1.0 Client Driver stream.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceSet	Schedules a SET_INTERFACE request to the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_0_StreamGetFirst	Returns information about first audio stream in the specified audio stream group.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetFirst	Gets the first streaming interface setting object within an audio streaming interface.
≡◊	USB_HOST_AUDIO_V1_0_StreamGetNext	Returns information about the next audio stream in the specified audio stream group.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetNext	Gets the next streaming interface setting object within an audio streaming interface.
≡◊	USB_HOST_AUDIO_V1_0_StreamSamplingRateSet	Schedules an audio stream set sampling rate request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_StreamOpen	Opens the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_StreamRead	Schedules an audio stream read request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_StreamWrite	Schedules an audio stream write request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceBitResolutionGet	Returns the bit resolution of the specified streaming interface setting.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceChannelNumbersGet	Returns the number of channels of the specified streaming interface setting.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceDirectionGet	Returns the direction of the specified streaming interface setting.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceFormatTagGet	Returns the format tag of the specified streaming interface setting.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequenciesGet	Returns the sampling frequencies supported by the specified streaming interface setting.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequencyTypeGet	Returns the sampling frequency type of the specified streaming interface setting.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceSubFrameSizeGet	Returns the sub-frame size of the specified streaming interface setting.
≡◊	USB_HOST_AUDIO_V1_StreamingInterfaceTerminalLinkGet	Returns the terminal link of the specified streaming interface setting.
≡◊	USB_HOST_AUDIO_V1_StreamSamplingFrequencyGet	Schedules an audio stream get sampling rate request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_StreamSamplingFrequencySet	Schedules an audio stream set sampling rate request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_0_StreamRead	Schedules an audio stream read request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_0_StreamWrite	Schedules an audio stream write request for the specified audio stream.

c) Other Functions

	Name	Description
≡◊	USB_HOST_AUDIO_V1_TerminalInputChannelConfigGet	Returns a structure that describes the spatial location of the logical channels of in the terminal's output audio channel cluster.
≡◊	USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeRangeGet	Schedules a control request to the Audio Device feature unit to get the range supported by the volume control on the specified channel.
≡◊	USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSubRangeNumbersGet	Schedules a control request to an Audio Device feature unit to get the number of sub-ranges supported by the volume control on the specified channel.

d) Data Types and Constants

Name	Description
USB_HOST_AUDIO_V1_ATTACH_EVENT_HANDLER	USB Host Audio v1.0 Client Driver attach event handler function pointer type.
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ	Defines the type of the Audio v1.0 Host control entity object.
USB_HOST_AUDIO_V1_ENTITY_REQUEST_CALLBACK	USB Host Audio v1.0 class driver control transfer complete callback function pointer type.
USB_HOST_AUDIO_V1_EVENT	Identifies the possible events that the Audio v1.0 Class Driver attach event handler can generate.
USB_HOST_AUDIO_V1_OBJ	Defines the type of the Audio v1.0 Host client object.
USB_HOST_AUDIO_V1_REQUEST_HANDLE	USB Host Audio v1.0 Client Driver request handle.
USB_HOST_AUDIO_V1_RESULT	USB Host Audio v1.0 Class Driver result enumeration.
USB_HOST_AUDIO_V1_0_ATTACH_EVENT_HANDLER	USB Host Audio v1.0 Client Driver attach event handler function pointer type.
USB_HOST_AUDIO_V1_STREAM_DIRECTION	USB Host Audio v1.0 Class Driver stream direction.
USB_HOST_AUDIO_V1_STREAM_EVENT	Identifies the possible events that the Audio v1.0 Stream can generate.
USB_HOST_AUDIO_V1_STREAM_EVENT_HANDLER	USB Host Audio v1.0 Class Driver stream event handler function pointer type.
USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE_DATA	USB Host Audio v1.0 class stream control event data.
USB_HOST_AUDIO_V1_0_CONTROL_CALLBACK	USB Host Audio v1.0 Class Driver control transfer complete callback function pointer type.
USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA	USB Host Audio v1.0 class stream data transfer event data.
USB_HOST_AUDIO_V1_0_EVENT	Identifies the possible events that the Audio v1.0 Class Driver can generate.
USB_HOST_AUDIO_V1_STREAM_EVENT_RESPONSE	Returns the type of the USB Host Audio v1.0 stream event handler.
USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE_DATA	USB Host Audio v1.0 class stream control event data.
USB_HOST_AUDIO_V1_0_OBJ	Defines the type of the Audio v1.0 Host client object.
USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE_DATA	USB Host Audio v1.0 class stream data transfer event data.
USB_HOST_AUDIO_V1_0_REQUEST_HANDLE	USB Host Audio v1.0 Client Driver request handle.
USB_HOST_AUDIO_V1_STREAM_HANDLE	Defines the type of the Audio v1.0 Host stream handle.
USB_HOST_AUDIO_V1_0_RESULT	USB Host Audio v1.0 Class Driver audio result enumeration.
USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE	USB Host Audio v1.0 Class Driver stream data transfer handle.
USB_HOST_AUDIO_V1_0_STREAM_DIRECTION	USB Host Audio v1.0 Class Driver stream direction.
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ	Defines the type of the Audio v1.0 Host streaming interface object.
USB_HOST_AUDIO_V1_0_STREAM_EVENT	Identifies the possible events that the Audio v1.0 stream can generate.
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ	Defines the type of the Audio v1.0 Host streaming interface setting object.

	USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE_DATA	USB Host Audio v1.0 class stream control event data.
	USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_DATA	USB Host Audio v1.0 class stream control event data.
	USB_HOST_AUDIO_V1_0_STREAM_EVENT_HANDLER	USB Host Audio v1.0 Class Driver stream event handler function pointer type.
	USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_DATA	This is macro USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_DATA .
	USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE	Returns the type of the USB Audio v1.0 Host Client Driver event handler.
	USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA	USB Host Audio v1.0 class stream transfer event data.
	USB_HOST_AUDIO_V1_0_STREAM_HANDLE	Defines the type of the Audio v1.0 Host stream handle.
	USB_HOST_AUDIO_V1_INTERFACE	USB HOST Audio v1.0 Client Driver interface.
	USB_HOST_AUDIO_V1_0_STREAM_INFO	This is type USB_HOST_AUDIO_V1_0_STREAM_INFO .
	USB_HOST_AUDIO_V1_REQUEST_HANDLE_INVALID	USB Host Audio v1.0 Client Driver invalid request handle.
	USB_HOST_AUDIO_V1_0_STREAM_OBJ	Defines the type of the Audio v1.0 Host stream object.
	USB_HOST_AUDIO_V1_STREAM_HANDLE_INVALID	Defines Audio v1.0 Host stream invalid handle.
	USB_HOST_AUDIO_V1_0_STREAM_RESULT	USB Host Audio v1.0 stream result enumeration.
	USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE_INVALID	USB Host Audio v1.0 Class Driver invalid stream data transfer handle.
	USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE	USB Host Audio v1.0 Class Driver transfer handle.
	USB_HOST_AUDIO_V1_0_INTERFACE	USB HOST Audio Client Driver interface.
	USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID	USB Host Audio v1.0 Client Driver invalid request handle.
	USB_HOST_AUDIO_V1_0_STREAM_HANDLE_INVALID	Defines the type of the Audio v1.0 Host stream invalid handle.
	USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE_INVALID	USB Host Audio v1.0 Class Driver invalid transfer handle definition.
	USB_HOST_AUDIO_V1_0_AttachEventHandlerSet	Sets an attach/detach event handler.
	USB_HOST_AUDIO_V1_0_DeviceObjHandleGet	Returns the device object handle for this Audio v1.0 Device.
	USB_HOST_AUDIO_V1_0_DIRECTION_IN	This is macro USB_HOST_AUDIO_V1_0_DIRECTION_IN .
	USB_HOST_AUDIO_V1_0_DIRECTION_OUT	This is macro USB_HOST_AUDIO_V1_0_DIRECTION_OUT .
	USB_HOST_AUDIO_V1_0_EVENT_ATTACH	This is macro USB_HOST_AUDIO_V1_0_EVENT_ATTACH .
	USB_HOST_AUDIO_V1_0_EVENT_DETACH	This is macro USB_HOST_AUDIO_V1_0_EVENT_DETACH .
	USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE_NONE	Returns the type of the USB Host Audio v1.0 stream event handler.
	USB_HOST_AUDIO_V1_0_StreamClose	Closes the audio stream.
	USB_HOST_AUDIO_V1_0_StreamOpen	Opens the specified audio stream.
	USB_HOST_AUDIO_V1_SAMPLING_FREQUENCIES_NUMBER	This structure defines USB Host audio stream information structure.
	USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_GET_COMPLETE_DATA	USB Host Audio v1.0 class stream control event data.

Description

This section describes the Application Programming Interface (API) functions of the USB Audio v1.0 Host Client Driver Library.
Refer to each section for a detailed description.

a) Audio Device Access Functions

USB_HOST_AUDIO_V1_AttachEventHandlerSet Function

Sets an attach/detach event handler.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_AttachEventHandlerSet(USB_HOST_AUDIO_V1_ATTACH_EVENT_HANDLER
eventHandler, uintptr_t context);
```

Returns

- `USB_HOST_AUDIO_V1_RESULT_SUCCESS` - If the attach event handler was registered successfully
- `USB_HOST_AUDIO_V1_RESULT_FAILURE` - If the number of registered event handlers has exceeded `USB_HOST_AUDIO_V1_ATTACH_LISTENERS_NUMBER`

Description

This function will set an attach event handler. The attach event handler will be called when a Audio v1.0 Device has been attached or detached.
The context will be returned in the event handler. This function should be called before the bus has been enabled.

Remarks

This function should be called before the [USB_HOST_BusEnable](#) function is called.

Preconditions

None.

Parameters

Parameters	Description
<code>eventHandler</code>	Pointer to the attach event handler.
<code>context</code>	An application defined context that will be returned in the event handler.

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_AttachEventHandlerSet
(
    USB_HOST_AUDIO_V1_ATTACH_EVENT_HANDLER eventHandler,
    uintptr_t context
);
```

USB_HOST_AUDIO_V1_0_ControlRequest Function

Schedules an Audio v1.0 control transfer.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_0_RESULT USB_HOST_AUDIO_V1_0_ControlRequest(USB_HOST_AUDIO_V1_0_OBJ OBJ,
USB_HOST_AUDIO_V1_0_REQUEST_HANDLE * transferHandle, USB_SETUP_PACKET * setupPacket, void * data,
USB_HOST_AUDIO_V1_0_CONTROL_CALLBACK callback, uintptr_t context);
```

Returns

- `USB_HOST_AUDIO_V1_0_RESULT_SUCCESS` - The transfer was scheduled successfully. requestHandle will contain a valid transfer handle.
- `USB_HOST_AUDIO_V1_0_RESULT_FAILURE` - An unknown failure occurred. requestHandle will contain

[USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID](#).

- **USB_HOST_AUDIO_V1_0_RESULT_PARAMETER_INVALID** - The data pointer or requestHandle pointer is NULL

Description

This function schedules an Audio v1.0 control transfer. The audioObj parameter is an object of the Audio v1.0 Class Driver to which the audio control transfer is to be scheduled. The setupPacket parameter points to the SETUP command to be sent in the setup state of the control transfer. The size and the direction of the data stage is indicated by the SETUP packet. For control transfers where there is no data stage, data is ignored and can be NULL. In all other instances, data should point to the data to be transferred in the data stage of the control transfer.

If the transfer was scheduled successfully, requestHandle will contain a transfer handle that uniquely identifies this transfer. If the transfer could not be scheduled successfully, requestHandle will contain [USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID](#).

When the control transfer completes, the Audio v1.0 Client Driver will call the specified callback function. The context parameter specified here will be returned in the callback.

Remarks

None.

Preconditions

The Audio v1.0 Device should be attached.

Parameters

Parameters	Description
audioObj	Audio v1.0 client driver object
requestHandle	Output parameter that will contain the handle to this transfer
setupPacket	Pointer to the SETUP packet to be sent to the device in the SETUP stage of the control transfer
data	For control transfer with a data stage, this should point to data to be sent to the device (for a control write transfer) or point to the buffer that will receive data from the device (for a control read transfer). For control transfers that do not require a data stage, this parameter is ignored and can be NULL.
callback	Pointer to the callback function that will be called when the control transfer completes. If the callback function is NULL, there will be no notification of when the control transfer will complete.
context	User-defined context that is returned with the callback function

Function

```
USB_HOST_AUDIO_V1_0_RESULT USB_HOST_AUDIO_V1_0_ControlRequest
(
    USB_HOST_AUDIO_V1_0_OBJ audioObj,
    USB_HOST_AUDIO_V1_0_REQUEST_HANDLE * requestHandle,
    USB_SETUP_PACKET *setupPacket,
    void * data,
    USB_HOST_AUDIO_V1_0_CONTROL_CALLBACK callback,
    uintptr_t context
);
```

[USB_HOST_AUDIO_V1_ControlEntityGetFirst Function](#)

Retrieves the handle to the first audio control entity

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_ControlEntityGetFirst(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ * pEntityObject);
```

Returns

- **USB_HOST_AUDIO_V1_RESULT_SUCCESS** - The operation was successful
- **USB_HOST_AUDIO_V1_RESULT_END_OF_CONTROL_ENTITY** - No more audio control entities are available
- **USB_HOST_AUDIO_V1_RESULT_OBJ_INVALID** - The specified audio stream does not exist
- **USB_HOST_AUDIO_V1_RESULT_FAILURE** - An unknown failure occurred

Description

This function retrieves the handle to the first audio control entity.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 device object.
pEntityObject	pointer to the Audio control entity handle.

Function

```
USB_HOST_AUDIO_V1_REESULT USB_HOST_AUDIO_V1_ControlEntityGetFirst
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ * pEntityObject
);
```

USB_HOST_AUDIO_V1_ControlEntityGetNext Function

Retrieves the handle to the next audio control entity.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_ControlEntityGetNext(USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObjectCurrent, USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ *
    pEntityObject);
```

Returns

- `USB_HOST_AUDIO_V1_RESULT_SUCCESS` - The operation was successful
- `USB_HOST_AUDIO_V1_RESULT_END_OF_CONTROL_ENTITY` - No more audio control entities are available
- `USB_HOST_AUDIO_V1_RESULT_OBJ_INVALID` - The specified audio stream does not exist
- `USB_HOST_AUDIO_V1_RESULT_FAILURE` - An unknown failure occurred

Description

This function retrieves the handle to the next audio control entity.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 device object.
entityObjectCurrent	Handle to current audio control entity.
pEntityObject	pointer to audio control entity handle.

Function

```
USB_HOST_AUDIO_V1_REESULT USB_HOST_AUDIO_V1_ControlEntityGetNext
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObjectCurrent
```

```
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ * pEntityObject
);
```

USB_HOST_AUDIO_V1_DeviceObjHandleGet Function

Returns the device object handle for this Audio v1.0 Device.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_DEVICE_OBJ_HANDLE USB_HOST_AUDIO_V1_DeviceObjHandleGet(USB_HOST_AUDIO_V1_OBJ audioDeviceObj);
```

Returns

Will return a valid device object handle if the device is still connected to the system. Otherwise, the function will return [USB_HOST_DEVICE_OBJ_HANDLE_INVALID](#).

Description

This function returns the device object handle for this Audio v1.0 Device. This returned handle can be used by the application to perform device-level operations, such as obtaining the string descriptors.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
audioDeviceObj	Audio V1.0 device object handle returned in the USB_HOST_AUDIO_V1_ATTACH_EVENT_HANDLER function.

Function

```
USB_HOST_DEVICE_OBJ_HANDLE USB_HOST_AUDIO_V1_DeviceObjHandleGet
(
    USB_HOST_AUDIO_V1_OBJ audioDeviceObj
);
```

USB_HOST_AUDIO_V1_EntityObjectGet Function

Retrieves the entity object for the entity ID.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_EntityObjectGet(USB_HOST_AUDIO_V1_OBJ audioObj, uint8_t
entityId, USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ* entityObj);
```

Returns

- [USB_HOST_AUDIO_V1_RESULT_SUCCESS](#) - The operation was successful
- [USB_HOST_AUDIO_V1_RESULT_FAILURE](#) - The entity Id could not be found or an unknown failure occurred

Description

This function retrieves the entity object for the entity ID.

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityId	Entity ID
entityObject	Audio control entity object

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_EntityObjectGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    uint8_t entityId,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ* entityObj
);
```

USB_HOST_AUDIO_V1_EntityRequestCallbackSet Function

Registers an audio entity request callback function with the Audio v1.0 Client Driver.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_EntityRequestCallbackSet(USB_HOST_AUDIO_V1_OBJ audioDeviceObj,
USB_HOST_AUDIO_V1_ENTITY_REQUEST_CALLBACK appAudioEntityRequestCallback, uintptr_t context);
```

Returns

- USB_HOST_AUDIO_V1_RESULT_SUCCESS - The operation was successful
- USB_HOST_AUDIO_V1_RESULT_OBJ_INVALID - The specified audio object does not exist
- USB_HOST_AUDIO_V1_RESULT_FAILURE - An unknown failure occurred

Description

This function registers a callback function for the Audio v1.0 control entity requests. The Audio v1.0 Host Client Driver will call this callback function when an audio entity control request is completed.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
audioDeviceObj	Audio v1.0 device object.
appAudioEntityRequestCallback	A pointer to event handler function. If NULL, events will not be generated.
context	Application specific context that is returned in the event handler.

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_EntityRequestCallbackSet
(
    USB_HOST_AUDIO_V1_OBJ audioDeviceObj,
    USB_HOST_AUDIO_V1_CONTROL_EVENT_HANDLER appAudioEntityRequestCallback,
    uintptr_t context
);
```

USB_HOST_AUDIO_V1_EntityTypeGet Function

Returns the entity type of the audio control entity.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_AUDIO_V1_ENTITY_TYPE USB_HOST_AUDIO_V1_EntityTypeGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject);
```

Returns

USB_AUDIO_V1_ENTITY_TYPE.

Description

This function returns the entity type of the audio control entity. Prior to calling this function the entity object should be obtained by calling [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#).

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object

Function

```
USB_AUDIO_V1_ENTITY_TYPE USB_HOST_AUDIO_V1_EntityTypeGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject
);
```

USB_HOST_AUDIO_V1_FeatureUnitChannelMuteExists Function

Returns "true" if mute control exists for the specified channel of the feature unit.

File

[usb_host_audio_v1_0.h](#)

C

```
bool USB_HOST_AUDIO_V1_FeatureUnitChannelMuteExists(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject, uint8_t channel);
```

Returns

- true - Mute control exists on the specified channel
- false - Mute control does not exist on the specified channel

Description

This function returns "true" if mute control exists on the specified channel of the feature unit. Channel 0 indicates Master mute control. This function is only applicable to a feature unit. Prior to calling this function the entity object should be obtained by calling [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#).

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object
channel	Channel number

Function

```
bool USB_HOST_AUDIO_V1_FeatureUnitChannelMuteExists
```

```

(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject,
    uint8_t channel
);

```

USB_HOST_AUDIO_V1_FeatureUnitChannelMuteGet Function

Schedules a get mute control request to the specified channel.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelMuteGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject, USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle,
uint8_t channelNumber, bool * muteStatus);
```

Returns

- **USB_HOST_AUDIO_V1_RESULT_SUCCESS** - The request was scheduled successfully. requestHandle will contain a valid request handle.
- **USB_HOST_AUDIO_V1_RESULT_BUSY** - The control request mechanism is currently busy. Retry the request.
- **USB_HOST_AUDIO_V1_RESULT_FAILURE** - An unknown failure occurred. requestHandle will contain **USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID**.
- **USB_HOST_AUDIO_V1_RESULT_PARAMETER_INVALID** - The data pointer or requestHandle pointer is NULL

Description

This function schedules a get mute control request to the specified channel. Prior to calling this function the user should check if mute control exists on the specified channel by calling the [USB_HOST_AUDIO_V1_FeatureUnitChannelMuteExists](#) function.

If the request was scheduled successfully, the requestHandle parameter will contain a request handle that uniquely identifies this request. If the transfer could not be scheduled successfully, requestHandle will contain **USB_HOST_AUDIO_V1_REQUEST_HANDLE_INVALID**.

When the control request completes, the Audio v1.0 Client Driver will call the callback function that was set using the [USB_HOST_AUDIO_V1_EntityRequestCallbackSet](#) function. The context parameter specified here will be returned in the callback.

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object
requestHandle	Output parameter that will contain the handle to this request
channelNumber	Channel number
muteStatus	Output parameter that will contain Current Mute status when the request is completed and a callback is received

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelMuteGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject,
    USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle,
    uint8_t channelNumber,
    bool *muteStatus
);
```

USB_HOST_AUDIO_V1_FeatureUnitChannelMuteSet Function

Schedules a set mute control request to the specified channel.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelMuteSet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject, USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle,
uint8_t channelNumber, bool * muteStatus);
```

Returns

- USB_HOST_AUDIO_V1_RESULT_SUCCESS - The request was scheduled successfully. requestHandle will contain a valid request handle.
- USB_HOST_AUDIO_V1_RESULT_BUSY - The control request mechanism is currently busy. Retry the request.
- USB_HOST_AUDIO_V1_RESULT_FAILURE - An unknown failure occurred. requestHandle will contain [USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID](#).
- USB_HOST_AUDIO_V1_RESULT_PARAMETER_INVALID - The data pointer or requestHandle pointer is NULL

Description

This function schedules a set mute control request to the specified channel. Prior to calling this function the user should check if mute control exists on the specified channel by calling the [USB_HOST_AUDIO_V1_FeatureUnitChannelMuteExists](#) function.

If the request was scheduled successfully, the requestHandle parameter will contain a request handle that uniquely identifies this transfer. If the transfer could not be scheduled successfully, requestHandle will contain [USB_HOST_AUDIO_V1_REQUEST_HANDLE_INVALID](#).

When the control request completes, the Audio v1.0 Client Driver will call the callback function that was set using the [USB_HOST_AUDIO_V1_EntityRequestCallbackSet](#) function. The context parameter specified here will be returned in the callback.

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object
requestHandle	Output parameter that will contain the handle to this request
channelNumber	Channel Number
muteStatus	Value of mute control, where 1 mutes the channel and 0 removes unmutes

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelMuteSet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject,
    USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle,
    uint8_t channelNumber,
    bool *muteStatus
);
```

[USB_HOST_AUDIO_V1_FeatureUnitChannelNumbersGet Function](#)

Returns the number of channels.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_FeatureUnitChannelNumbersGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject);
```

Returns

The number of channels.

Description

This function returns the number of channels. This function is only applicable to a feature unit. Prior to calling this function the entity object should

be obtained by calling [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#).

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object

Function

```
uint8_t USB_HOST_AUDIO_V1_FeatureUnitChannelNumbersGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject
);
```

USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeExists Function

Returns "true" if volume control exists for the specified channel of the feature unit.

File

[usb_host_audio_v1_0.h](#)

C

```
bool USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeExists(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject, uint8_t channel);
```

Returns

- true - Volume control exists on the specified channel
- false - Volume control does not exist on the specified channel

Description

This function returns "true" if volume control exists on the specified channel of the feature unit. Channel 0 indicates master volume control. This function is only applicable to a feature unit. Prior to calling this function the entity object should be obtained by calling [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#).

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object
channel	Channel number

Function

```
bool USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeExists
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject,
    uint8_t channel
);
```

USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeGet Function

Schedules a get current volume control request to the specified channel.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject, USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle,
uint8_t channelNumber, uint16_t * volume);
```

Returns

- **USB_HOST_AUDIO_V1_RESULT_SUCCESS** - The request was scheduled successfully. requestHandle will contain a valid request handle.
- **USB_HOST_AUDIO_V1_RESULT_BUSY** - The control request mechanism is currently busy. Retry the request.
- **USB_HOST_AUDIO_V1_RESULT_FAILURE** - An unknown failure occurred. requestHandle will contain [USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID](#)
- **USB_HOST_AUDIO_V1_RESULT_PARAMETER_INVALID** - The data pointer or requestHandle pointer is NULL

Description

This function schedules a get current volume control request to the specified channel. Prior to calling this function the user should check if volume control exists on the specified channel by calling the [USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeExists](#) function.

If the request was scheduled successfully, the requestHandle parameter will contain a request handle that uniquely identifies this request. If the request could not be scheduled successfully, requestHandle will contain [USB_HOST_AUDIO_V1_REQUEST_HANDLE_INVALID](#).

When the control request completes, the Audio v1.0 Client Driver will call the callback function that was set using the [USB_HOST_AUDIO_V1_EntityRequestCallbackSet](#) function. The context parameter specified here will be returned in the callback.

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object
requestHandle	Output parameter that will contain the handle to this request
channelNumber	Channel number to which the volume control is addressed
volume	Output parameter that will contain the current volume when a request is completed and a callback is received

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject,
    USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle,
    uint8_t channelNumber,
    uint16_t *volume
);
```

[USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSet Function](#)

Schedules a set current volume control request to the specified channel.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject, USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle,
uint8_t channelNumber, uint16_t * volume);
```

Returns

- **USB_HOST_AUDIO_V1_RESULT_SUCCESS** - The request was scheduled successfully. requestHandle will contain a valid request handle.
- **USB_HOST_AUDIO_V1_RESULT_BUSY** - The control request mechanism is currently busy. Retry the request.

- **USB_HOST_AUDIO_V1_RESULT_FAILURE** - An unknown failure occurred. requestHandle will contain **USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID**.
- **USB_HOST_AUDIO_V1_RESULT_PARAMETER_INVALID** - The data pointer or requestHandle pointer is NULL

Description

This function schedules a set current volume request to the specified channel. Prior to calling this function the user should check if volume control exists on the specified channel by calling the [USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeExists](#) function.

If the request was scheduled successfully, the requestHandle parameter will contain a request handle that uniquely identifies this request. If the request could not be scheduled successfully, requestHandle will contain **USB_HOST_AUDIO_V1_REQUEST_HANDLE_INVALID**.

When the control request completes, the Audio v1.0 Client Driver will call the callback function that was set using the [USB_HOST_AUDIO_V1_EntityRequestCallbackSet](#) function. The context parameter specified here will be returned in the callback.

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object
requestHandle	Output parameter that will contain the handle to this request
channelNumber	Channel number to which the volume control is addressed
volume	Current volume control value that should be set in the Audio Device

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject,
    USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle,
    uint8_t channelNumber,
    uint16_t *volume
);
```

USB_HOST_AUDIO_V1_FeatureUnitIDGet Function

Returns ID of the Feature Unit.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_FeatureUnitIDGet(USB_HOST_AUDIO_V1_OBJ audioObj,
                                            USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject);
```

Returns

The ID of the feature unit.

Description

This function returns the ID of the D of the Feature Unit. This function is only applicable to Feature Unit. Prior to calling this function Entity Object should be obtained by calling the [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#) function.

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 device object.
entityObject	Audio control entity Object

Function

```
uint8_t USB_HOST_AUDIO_V1_FeatureUnitIDGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject
);
```

USB_HOST_AUDIO_V1_FeatureUnitSourceIDGet Function

Returns the ID of the unit or terminal to which this feature unit is connected.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_FeatureUnitSourceIDGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject);
```

Returns

The ID of the unit or terminal to which this feature unit is connected.

Description

This function returns the ID of the Unit or Terminal to which this feature unit is connected. This function is only applicable to a feature unit. Prior to calling this function the entity object should be obtained by calling [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#).

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object

Function

```
uint8_t USB_HOST_AUDIO_V1_FeatureUnitSourceIDGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject
);
```

USB_HOST_AUDIO_V1_TerminalAssociationGet Function

Returns the associated terminal ID of the audio control terminal.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_TerminalAssociationGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject);
```

Returns

The ID of the associated terminal.

Description

This function returns the ID of the associated terminal type of the audio control terminal. Prior to calling this function the entity object should be obtained by calling [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#).

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object

Function

```
uint8_t USB_HOST_AUDIO_V1_TerminalAssociationGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject
);
```

USB_HOST_AUDIO_V1_TerminalIDGet Function

Returns the terminal ID of the audio control entity.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_TerminalIDGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject);
```

Returns

The terminal ID of the audio control entity object.

Description

This function returns the Terminal ID of the Audio Control entity. Prior to calling this function the entity object should be obtained by calling [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#).

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object

Function

```
uint8_t USB_HOST_AUDIO_V1_TerminalIDGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject
);
```

USB_HOST_AUDIO_V1_TerminalInputChannelNumbersGet Function

Returns the number of logical output channels in the terminal's output audio channel cluster.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_TerminalInputChannelNumbersGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject);
```

Returns

The number of logical output channels in the terminal's output audio channel cluster.

Description

This function returns the number of logical output channels in the terminal's output audio channel cluster. This function is only applicable to an input terminal. Prior to calling this function the entity object should be obtained by calling [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#).

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 device object.
entityObject	Audio control entity object

Function

```
uint8_t USB_HOST_AUDIO_V1_TerminalInputChannelNumbersGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject
);
```

USB_HOST_AUDIO_V1_TerminalSourceIDGet Function

Returns the ID of the unit or terminal to which this terminal is connected.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_TerminalSourceIDGet(USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject);
```

Returns

The ID of the unit or terminal to which this terminal is connected.

Description

This function returns the ID of the unit or terminal to which this terminal is connected. This function is only applicable to an output terminal. Prior to calling this function the entity object should be obtained by calling [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#).

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object

Function

```
uint8_t USB_HOST_AUDIO_V1_TerminalSourceIDGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject
);
```

USB_HOST_AUDIO_V1_TerminalTypeGet Function

Returns the terminal type of the audio control entity.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_AUDIO_V1_TERMINAL_TYPE USB_HOST_AUDIO_V1_TerminalTypeGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject);
```

Returns

The terminal type.

Description

This function returns the Terminal type of the audio control entity. Prior to calling this function Entity Object should be obtained by calling the [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#) function.

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 device object
entityObject	Audio control entity Object

Function

```
USB_AUDIO_V1_TERMINAL_TYPE USB_HOST_AUDIO_V1_TerminalTypeGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject
);
```

b) Audio Stream Access Functions***USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet Function***

Gets the number of stream groups present in the attached Audio v1.0 Device.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet(USB_HOST_AUDIO_V1_0_OBJ audioObj);
```

Returns

A returned uint8_t indicates the number of audio stream groups present in the attached Audio v1.0 Device.

Description

This function will get number of stream groups present in the attached Audio v1.0 Device. The audio stream within an audio stream cannot be enabled at the same time.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
audioObj	Audio v1.0 Client Driver object

Function

```
uint8_t USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet
(
    USB_HOST_AUDIO_V1_0_OBJ audioObj
);
```

USB_HOST_AUDIO_V1_StreamClose Function

Closes the audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
void USB_HOST_AUDIO_V1_StreamClose(USB_HOST_AUDIO_V1_STREAM_HANDLE audioStreamHandle);
```

Returns

None.

Description

This function will close the open audio stream. This closes the association between the application entity that opened the audio stream and the audio stream. The audio stream handle becomes invalid.

Remarks

The device handle becomes invalid after calling this function.

Preconditions

None.

Parameters

Parameters	Description
audioSteamHandle	handle to the audio stream obtained from the USB_HOST_AUDIO_V1_StreamOpen function.

Function

```
void USB_HOST_AUDIO_V1_StreamClose
(
    USB_HOST_AUDIO_V1_STREAM_HANDLE audioSteamHandle
);
```

USB_HOST_AUDIO_V1_StreamEventHandlerSet Function

Registers an event handler with the Audio v1.0 Client Driver stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamEventHandlerSet(USB_HOST_AUDIO_V1_STREAM_HANDLE handle,
USB_HOST_AUDIO_V1_STREAM_EVENT_HANDLER appAudioHandler, uintptr_t context);
```

Returns

- **USB_HOST_AUDIO_V1_RESULT_SUCCESS** - The operation was successful
- **USB_HOST_AUDIO_V1_RESULT_HANDLE_INVALID** - The specified audio stream does not exist
- **USB_HOST_AUDIO_V1_RESULT_FAILURE** - An unknown failure occurred

Description

This function registers a client specific Audio v1.0 stream event handler. The Audio v1.0 Host Client Driver will call the appAudioHandler function specified as the second argument with relevant event and associated event data in response to audio stream data transfers that have been scheduled by the client.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
handle	The handle to the Audio v1.0 stream
eventHandler	A pointer to event handler function. If NULL, events will not be generated.
context	The application specific context that is returned in the event handler

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamEventHandlerSet
(
    USB_HOST_AUDIO_V1_STREAM_HANDLE handle,
    USB_HOST_AUDIO_V1_STREAM_EVENT_HANDLER appAudioHandler,
    uintptr_t context
);
```

USB_HOST_AUDIO_V1_0_StreamDisable Function

Schedules an audio stream disable request for the specified audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_0_STREAM_RESULT USB_HOST_AUDIO_V1_0_StreamDisable(USB_HOST_AUDIO_V1_0_STREAM_HANDLE
streamHandle, USB_HOST_AUDIO_V1_0_REQUEST_HANDLE * requestHandle);
```

Returns

- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_SUCCESS` - The operation was successful
- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_HANDLE_INVALID` - The specified audio stream does not exist
- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_FAILURE` - An unknown failure occurred

Description

This function schedules an audio stream disable request for the specified audio stream. A `USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE` event is generated when this request is completed. `USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE_DATA` returns the status and request handle of the request.

Remarks

None.

Preconditions

The audio stream should have been opened.

Parameters

Parameters	Description
streamHandle	Handle to the Audio v1.0 stream
requestHandle	Handle to the stream disable request

Function

```
USB_HOST_AUDIO_V1_0_STREAM_RESULT USB_HOST_AUDIO_V1_0_StreamDisable
(

```

```
USB_HOST_AUDIO_V1_0_STREAM_HANDLE streamHandle,
USB_HOST_AUDIO_V1_0_REQUEST_HANDLE * requestHandle
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceGetFirst Function

Gets the first streaming interface object from the attached Audio Device.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamingInterfaceGetFirst(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ* streamingInterfaceObj);
```

Returns

- `USB_HOST_AUDIO_V1_RESULT_SUCCESS` - The request completed successfully
- `USB_HOST_AUDIO_V1_RESULT_END_OF_STREAMING_INTERFACE` - No more streaming interfaces are available
- `USB_HOST_AUDIO_V1_RESULT_DEVICE_UNKNOWN` - Device is not attached
- `USB_HOST_AUDIO_V1_RESULT_OBJ_INVALID` - Audio Device object is invalid
- `USB_HOST_AUDIO_V1_RESULT_FAILURE` - An error has occurred

Description

This function will get the first streaming interface object from the attached Audio Device.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
<code>audioObj</code>	Audio v1.0 client driver object.
<code>streamingInterfaceObj</code>	Pointer to an audio streaming interface object.

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamingInterfaceGetFirst
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ* streamingInterfaceObj
);
```

USB_HOST_AUDIO_V1_0_StreamEnable Function

Schedules an audio stream enable request for the specified audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_0_STREAM_RESULT USB_HOST_AUDIO_V1_0_StreamEnable(USB_HOST_AUDIO_V1_0_STREAM_HANDLE
streamHandle, USB_HOST_AUDIO_V1_0_REQUEST_HANDLE * requestHandle);
```

Returns

- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_SUCCESS` - The operation was successful
- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_HANDLE_INVALID` - The specified audio stream does not exist
- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_FAILURE` - An unknown failure occurred

Description

This function schedules an audio stream enable request for the specified audio stream. An audio stream must be enable before scheduling any

data transfer with the stream. A `USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE` event is generated when this request is completed. `USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_DATA` returns the status and request handle of the request.

Remarks

None.

Preconditions

The audio stream should have been opened. Only one audio stream from an audio stream group can be enabled at a time.

Parameters

Parameters	Description
streamHandle	Handle to the audio v1.0 stream
requestHandle	Handle to the stream enable request

Function

```
USB_HOST_AUDIO_V1_0_STREAM_RESULT USB_HOST_AUDIO_V1_0_StreamEnable
(
    USB_HOST_AUDIO_V1_0_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_0_REQUEST_HANDLE * requestHandle
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceGetNext Function

Gets the next streaming interface object from the attached Audio Device.

File

`usb_host_audio_v1_0.h`

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamingInterfaceGetNext(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObjCurrent,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ* streamingInterfaceObjNext);
```

Returns

- `USB_HOST_AUDIO_V1_RESULT_SUCCESS` - The request completed successfully
- `USB_HOST_AUDIO_V1_RESULT_END_OF_STREAMING_INTERFACE` - No more streaming interfaces are available
- `USB_HOST_AUDIO_V1_RESULT_DEVICE_UNKNOWN` - Device is not attached
- `USB_HOST_AUDIO_V1_RESULT_OBJ_INVALID` - Audio Device object is invalid
- `USB_HOST_AUDIO_V1_RESULT_FAILURE` - An error has occurred

Description

This function will get the next streaming interface object from the attached Audio Device.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
audioObj	Audio Device object.
streamingInterfaceObjCurrent	Current audio streaming interface object.
streamingInterfaceObj	Pointer to audio streaming interface object.

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamingInterfaceGetNext
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObjCurrent
```

```
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ* streamingInterfaceObjNext
);
```

USB_HOST_AUDIO_V1_0_StreamEventHandlerSet Function

Registers an event handler with the Audio v1.0 Client Driver stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_0_STREAM_RESULT
USB_HOST_AUDIO_V1_0_StreamEventHandlerSet(USB_HOST_AUDIO_V1_0_STREAM_HANDLE handle,
USB_HOST_AUDIO_V1_0_STREAM_EVENT_HANDLER appAudioHandler, uintptr_t context);
```

Returns

- **USB_HOST_AUDIO_V1_0_STREAM_RESULT_SUCCESS** - The operation was successful
- **USB_HOST_AUDIO_V1_0_STREAM_RESULT_HANDLE_INVALID** - The specified audio stream does not exist
- **USB_HOST_AUDIO_V1_0_STREAM_RESULT_FAILURE** - An unknown failure occurred

Description

This function registers a client specific Audio v1.0 stream event handler. The Audio v1.0 Host Client Driver will call appAudioHandler function specified as 2nd argument with relevant event and associate event data, in response to audio stream data transfers that have been scheduled by the client.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
handle	A handle to the Audio v1.0 stream
eventHandler	A pointer to event handler function. If NULL, events will not be generated.
context	The application specific context that is returned in the event handler

Function

```
USB_HOST_AUDIO_V1_0_STREAM_RESULT USB_HOST_AUDIO_V1_0_StreamEventHandlerSet
(
    USB_HOST_AUDIO_V1_0_STREAM_HANDLE handle,
    USB_HOST_AUDIO_V1_0_STREAM_EVENT_HANDLER appAudioHandler,
    uintptr_t context
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceSet Function

Schedules a SET_INTERFACE request to the specified audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamingInterfaceSet(USB_HOST_AUDIO_V1_STREAM_HANDLE
streamHandle, USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj);
```

Returns

- **USB_HOST_AUDIO_V1_RESULT_SUCCESS** - The operation was successful
- **USB_HOST_AUDIO_V1_RESULT_HANDLE_INVALID** - The specified audio stream does not exist
- **USB_HOST_AUDIO_V1_RESULT_FAILURE** - An unknown failure occurred

Description

This function schedules an audio stream enable request for the specified audio stream. An audio stream must be enable before scheduling any data transfer with the stream. A USB_HOST_AUDIO_V1_STREAM_EVENT_ENABLE_COMPLETE event is generated when this request is completed. USB_HOST_AUDIO_V1_STREAM_EVENT_ENABLE_COMPLETE_DATA returns the status and request handle of the request.

Remarks

None.

Preconditions

The audio stream should have been opened. Only one audio stream from an audio stream group can be enabled at a time.

Parameters

Parameters	Description
streamHandle	Handle to the Audio v1.0 stream.
requestHandle	Handle to the stream enable request.

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamingInterfaceSet
(
    USB_HOST_AUDIO_V1_STREAM_HANDLE streamHandle,
    USB_INTERFACE_DESCRIPTOR* pInterfaceDesc,
    USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle
);
```

USB_HOST_AUDIO_V1_0_StreamGetFirst Function

Returns information about first audio stream in the specified audio stream group.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_0_RESULT USB_HOST_AUDIO_V1_0_StreamGetFirst(USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj,
uint8_t streamGroupIndex, USB_HOST_AUDIO_V1_0_STREAM_INFO * streamInfo);
```

Returns

- USB_HOST_AUDIO_V1_0_STREAM_RESULT_SUCCESS - The operation was successful
- USB_HOST_AUDIO_V1_0_RESULT_OBJ_INVALID - The specified Audio v1.0 client driver object does not exist
- USB_HOST_AUDIO_V1_0_STREAM_RESULT_FAILURE - An unknown failure occurred

Description

This function returns information about the first audio stream in the specified audio stream group. The stream group index is parameter to this function and it can be any value starting from zero to the number of stream groups minus one. Number of stream groups can be obtained by using the [USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet](#) function.

The streamInfo object is an out parameter to this function.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached to the Host.

Parameters

Parameters	Description
audioDeviceObj	Audio v1.0 Client Driver object
streamGroupIndex	Stream group index
streamInfo	Pointer to the streamInfo object

Function

```
USB_HOST_AUDIO_V1_0_RESULT USB_HOST_AUDIO_V1_0_StreamGetFirst
```

```

(
    USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj,
    uint8_t streamGroupIndex,
    USB_HOST_AUDIO_V1_0_STREAM_INFO * streamInfo
);

```

USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetFirst Function

Gets the first streaming interface setting object within an audio streaming interface.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetFirst(USB_HOST_AUDIO_V1_OBJ
audioObj, USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ * interfaceSettingObj);
```

Returns

- `USB_HOST_AUDIO_V1_RESULT_SUCCESS` - The request completed successfully
- `USB_HOST_AUDIO_V1_RESULT_END_OF_INTERFACE_SETTINGS` - No more streaming interface settings are available
- `USB_HOST_AUDIO_V1_RESULT_DEVICE_UNKNOWN` - Device is not attached
- `USB_HOST_AUDIO_V1_RESULT_OBJ_INVALID` - Audio Device object is invalid
- `USB_HOST_AUDIO_V1_RESULT_FAILURE` - An error has occurred

Description

This function gets the first streaming interface setting object within an audio streaming interface.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
<code>audioObj</code>	Audio device object.
<code>streamingInterfaceObj</code>	Audio streaming interface object.
<code>interfaceSettingObj</code>	Pointer to the audio streaming interface setting object.

Function

```

USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetFirst
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ *interfaceSettingObj
);

```

USB_HOST_AUDIO_V1_0_StreamGetNext Function

Returns information about the next audio stream in the specified audio stream group.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_0_RESULT USB_HOST_AUDIO_V1_0_StreamGetNext(USB_HOST_AUDIO_V1_0_STREAM_OBJ audioStreamObj,
USB_HOST_AUDIO_V1_0_STREAM_INFO * streamInfo);
```

Returns

- USB_HOST_AUDIO_V1_0_STREAM_RESULT_SUCCESS - The operation was successful
- USB_HOST_AUDIO_V1_0_RESULT_OBJ_INVALID - The specified Audio v1.0 client driver object does not exist
- USB_HOST_AUDIO_V1_0_STREAM_RESULT_FAILURE - An unknown failure occurred
- USB_HOST_AUDIO_V1_0_RESULT_END_OF_STREAM_LIST - There are no more audio streams in the stream group

Description

This function returns information about next audio stream in the specified Audio stream group. The [USB_HOST_AUDIO_V1_0_StreamGetFirst](#) function should have been called at least once on the same audio stream group before calling this function. Then, calling this function repeatedly on the stream group will return information about the next audio stream in the stream group. When there are no more audio streams to report, the function returns `USB_HOST_AUDIO_V1_0_RESULT_END_OF_STREAM_LIST`.

Calling the [USB_HOST_AUDIO_V1_0_StreamGetFirst](#) function on the stream group index after the [USB_HOST_AUDIO_V1_0_StreamGetNext](#) function has been called will cause the Audio v1.0 Client Driver to reset the audio stream group to point to the first stream in the stream group.

Remarks

None.

Preconditions

The [USB_HOST_AUDIO_V1_0_StreamGetFirst](#) function must have been called before calling this function.

Parameters

Parameters	Description
<code>audioStreamObj</code>	Present audio stream object
<code>streamInfo</code>	Pointer to the <code>streamInfo</code> object

Function

```
USB_HOST_AUDIO_V1_0_RESULT USB_HOST_AUDIO_V1_0_StreamGetNext
(
    USB_HOST_AUDIO_V1_0_STREAM_OBJ audioStreamObj,
    USB_HOST_AUDIO_V1_0_STREAM_INFO * streamInfo
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetNext Function

Gets the next streaming interface setting object within an audio streaming interface.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetNext(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObjCurrent,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ * interfaceSettingObjNext);
```

Returns

- `USB_HOST_AUDIO_V1_RESULT_SUCCESS` - The request completed successfully
- `USB_HOST_AUDIO_V1_RESULT_END_OF_INTERFACE_SETTINGS` - No more streaming interface settings are available
- `USB_HOST_AUDIO_V1_RESULT_DEVICE_UNKNOWN` - Device is not attached
- `USB_HOST_AUDIO_V1_RESULT_OBJ_INVALID` - Audio Device object is invalid
- `USB_HOST_AUDIO_V1_RESULT_FAILURE` - An error has occurred

Description

This function gets the next streaming interface setting object within an audio streaming interface.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
audioObj	Audio Device object
streamingInterfaceObj	Audio streaming interface object
interfaceSettingObjCurrent	Current audio streaming interface setting object
interfaceSettingObjNext	Pointer to the next audio streaming interface setting object

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetNext
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObjCurrent,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ *interfaceSettingObjNext
);
```

USB_HOST_AUDIO_V1_0_StreamSamplingRateSet Function

Schedules an audio stream set sampling rate request for the specified audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_0_STREAM_RESULT
USB_HOST_AUDIO_V1_0_StreamSamplingRateSet(USB_HOST_AUDIO_V1_0_STREAM_HANDLE streamHandle,
USB_HOST_AUDIO_V1_0_REQUEST_HANDLE * requestHandle, uint32_t * samplingRate);
```

Returns

- USB_HOST_AUDIO_V1_0_STREAM_RESULT_SUCCESS - The operation was successful
- USB_HOST_AUDIO_V1_0_STREAM_RESULT_HANDLE_INVALID - The specified audio stream does not exist
- USB_HOST_AUDIO_V1_0_STREAM_RESULT_FAILURE - An unknown failure occurred

Description

This function schedules an audio stream set sampling rate request for the specified audio stream. A USB_HOST_AUDIO_V1_0_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE event is generated when this request is completed. USB_HOST_AUDIO_V1_0_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE_DATA returns the status and request handle of the request.

Remarks

None.

Preconditions

The audio stream should have been opened.

Parameters

Parameters	Description
streamHandle	Handle to the Audio v1.0 stream
requestHandle	Handle to the stream set sampling rate request
samplingRate	Pointer to the sampling rate

Function

```
USB_HOST_AUDIO_V1_0_STREAM_RESULT USB_HOST_AUDIO_V1_0_StreamSamplingRateSet
(
    USB_HOST_AUDIO_V1_0_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_0_REQUEST_HANDLE requestHandle,
    uint32_t* samplingRate
);
```

USB_HOST_AUDIO_V1_StreamOpen Function

Opens the specified audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_STREAM_HANDLE USB_HOST_AUDIO_V1_StreamOpen(USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ  
audiostreamingInterfaceObj);
```

Returns

Will return a valid handle if the audio stream could be opened successfully. Otherwise, [USB_HOST_AUDIO_V1_RESULT_HANDLE_INVALID](#) is returned. The function will return a valid handle if the stream is ready to be opened.

Description

This function will open the specified audio stream. Once opened, the audio stream can be accessed via the handle that this function returns. The `audiostreamingInterfaceObj` parameter is the value returned in the [USB_HOST_AUDIO_V1_StreamingInterfaceGetFirst](#) or [USB_HOST_AUDIO_V1_StreamingInterfaceGetNext](#) functions.

Remarks

None.

Preconditions

The audio streaming interface object should be valid.

Parameters

Parameters	Description
<code>audiostreamingInterfaceObj</code>	Audio streaming interface object

Function

```
USB_HOST_AUDIO_V1_STREAM_HANDLE USB_HOST_AUDIO_V1_StreamOpen  
(  
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ audiostreamingInterfaceObj  
)
```

USB_HOST_AUDIO_V1_StreamRead Function

Schedules an audio stream read request for the specified audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamRead(USB_HOST_AUDIO_V1_STREAM_HANDLE streamHandle,  
USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE * transferHandle, void * source, size_t length);
```

Returns

- [USB_HOST_AUDIO_V1_RESULT_SUCCESS](#) - The operation was successful
- [USB_HOST_AUDIO_V1_RESULT_HANDLE_INVALID](#) - The specified audio stream does not exist
- [USB_HOST_AUDIO_V1_RESULT_FAILURE](#) - An unknown failure occurred

Description

This function schedules an audio stream read request for the specified audio stream. A [USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE](#) event is generated when this request is completed. [USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA](#) returns the status and request handle of the request.

Remarks

None.

Preconditions

The audio stream should have been opened and enabled. The direction of the audio stream should be USB_HOST_AUDIO_V1_DIRECTION_IN.

Parameters

Parameters	Description
streamHandle	Handle to the Audio v1.0 stream
transferHandle	Handle to the stream read transfer request
source	Pointer to the buffer containing data to be read from the device
length	Amount of data to read (in bytes)

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamRead
(
    USB_HOST_AUDIO_V1_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE * transferHandle,
    void * source,
    size_t length
);
```

USB_HOST_AUDIO_V1_StreamWrite Function

Schedules an audio stream write request for the specified audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamWrite(USB_HOST_AUDIO_V1_STREAM_HANDLE streamHandle,
USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE * transferHandle, void * source, size_t length);
```

Returns

- USB_HOST_AUDIO_V1_RESULT_SUCCESS - The operation was successful
- USB_HOST_AUDIO_V1_RESULT_HANDLE_INVALID - The specified audio stream does not exist
- USB_HOST_AUDIO_V1_RESULT_FAILURE - An unknown failure occurred

Description

This function schedules an audio stream write request for the specified audio stream. A USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE event is generated when this request is completed. [USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE_DATA](#) returns the status and request handle of the request.

Remarks

None.

Preconditions

The audio stream should have been opened and enabled. The direction of the audio stream should be USB_HOST_AUDIO_V1_DIRECTION_OUT.

Parameters

Parameters	Description
streamHandle	Handle to the Audio v1.0 stream
transferHandle	Handle to the stream write transfer request
source	Pointer to the buffer containing data to be written to the device
length	Amount of data to write (in bytes)

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamWrite
(
    USB_HOST_AUDIO_V1_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE * transferHandle,
```

```
void * source,
size_t length
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceBitResolutionGet Function

Returns the bit resolution of the specified streaming interface setting.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_StreamingInterfaceBitResolutionGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj);
```

Returns

The bit resolution size of the audio streaming interface setting.

Description

This function returns the bit resolution size of the specified streaming interface setting.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
audioObj	Audio Device object
streamingInterfaceObj	Audio streaming interface object
interfaceSettingObj	Audio streaming interface setting object

Function

```
uint8_t USB_HOST_AUDIO_V1_StreamingInterfaceBitResolutionGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceChannelNumbersGet Function

Returns the number of channels of the specified streaming interface setting.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_StreamingInterfaceChannelNumbersGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj);
```

Returns

The number of channels present in the audio streaming interface setting.

Description

This function returns the number of channels of the specified streaming interface setting.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
audioObj	Audio Device object
streamingInterfaceObj	Audio streaming interface object
interfaceSettingObj	Audio streaming interface setting object

Function

```
uint8_t USB_HOST_AUDIO_V1_StreamingInterfaceChannelNumbersGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceDirectionGet Function

Returns the direction of the specified streaming interface setting.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_STREAM_DIRECTION USB_HOST_AUDIO_V1_StreamingInterfaceDirectionGet(USB_HOST_AUDIO_V1_OBJ
audioObj, USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj);
```

Returns

- `USB_HOST_AUDIO_V1_DIRECTION_OUT` - Host to Device
- `USB_HOST_AUDIO_V1_DIRECTION_IN` - Device to Host

Description

This function returns the direction of the specified streaming interface setting.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
audioObj	Audio Device object
streamingInterfaceObj	Audio streaming interface object
interfaceSettingObj	Audio streaming interface setting object

Function

```
USB_HOST_AUDIO_V1_STREAM_DIRECTION USB_HOST_AUDIO_V1_StreamingInterfaceDirectionGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceFormatTagGet Function

Returns the format tag of the specified streaming interface setting.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_AUDIO_V1_FORMAT_TAG USB_HOST_AUDIO_V1_StreamingInterfaceFormatTagGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj);
```

Returns

The format tag of the audio streaming interface setting.

Description

This function returns the format tag link of the specified streaming interface setting.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
audioObj	Audio Device object
streamingInterfaceObj	Audio streaming interface object
interfaceSettingObj	Audio streaming interface setting object

Function

```
uint8_t USB_HOST_AUDIO_V1_StreamingInterfaceFormatTagGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequenciesGet Function

Returns the sampling frequencies supported by the specified streaming interface setting.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t* USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequenciesGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj);
```

Returns

A pointer to the sampling frequencies supported by the audio streaming interface setting.

Description

This function returns the sampling frequencies supported by the specified streaming interface setting.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
audioObj	Audio Device object
streamingInterfaceObj	Audio streaming interface object
interfaceSettingObj	Audio streaming interface setting object

Function

```
uint8_t* USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequenciesGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequencyTypeGet Function

Returns the sampling frequency type of the specified streaming interface setting.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequencyTypeGet(USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj);
```

Returns

The sampling frequency type of the audio streaming interface setting.

- 0 - Continuous Sampling frequency is supported
- 1 to 255 - The number of discrete sampling frequencies supported by the audio streaming interface

Description

This function returns the sampling frequency type of the specified streaming interface setting.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
audioObj	Audio Device object
streamingInterfaceObj	Audio streaming interface object
interfaceSettingObj	Audio streaming interface setting object

Function

```
uint8_t USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequencyTypeGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceSubFrameSizeGet Function

Returns the sub-frame size of the specified streaming interface setting.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_StreamingInterfaceSubFrameSizeGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj);
```

Returns

The sub-frame size of the audio streaming interface setting.

Description

This function returns the sub-frame size of the specified streaming interface setting.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
audioObj	Audio Device object
streamingInterfaceObj	Audio streaming interface object
interfaceSettingObj	Audio streaming interface setting object

Function

```
uint8_t USB_HOST_AUDIO_V1_StreamingInterfaceSubFrameSizeGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj
);
```

USB_HOST_AUDIO_V1_StreamingInterfaceTerminalLinkGet Function

Returns the terminal link of the specified streaming interface setting.

File

[usb_host_audio_v1_0.h](#)

C

```
uint8_t USB_HOST_AUDIO_V1_StreamingInterfaceTerminalLinkGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj);
```

Returns

The terminal link of the audio streaming interface setting.

Description

This function returns the terminal link of the specified streaming interface setting.

Remarks

None.

Preconditions

The Audio v1.0 Device should have been attached.

Parameters

Parameters	Description
audioObj	Audio Device object
streamingInterfaceObj	Audio streaming interface object
interfaceSettingObj	Audio streaming interface setting object

Function

```
uint8_t USB_HOST_AUDIO_V1_StreamingInterfaceTerminalLinkGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ streamingInterfaceObj,
    USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ interfaceSettingObj
);
```

USB_HOST_AUDIO_V1_StreamSamplingFrequencyGet Function

Schedules an audio stream get sampling rate request for the specified audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamSamplingFrequencyGet(USB_HOST_AUDIO_V1_STREAM_HANDLE
streamHandle, USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle, uint32_t * samplingFrequency);
```

Returns

- `USB_HOST_AUDIO_V1_RESULT_SUCCESS` - The operation was successful
- `USB_HOST_AUDIO_V1_RESULT_HANDLE_INVALID` - The specified audio stream does not exist
- `USB_HOST_AUDIO_V1_RESULT_FAILURE` - An unknown failure occurred

Description

This function schedules an audio stream set sampling rate request for the specified audio stream. A `USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE` event is generated when this request is completed. `USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE_DATA` returns the status and request handle of the request.

Remarks

None.

Preconditions

The audio stream should have been opened.

Parameters

Parameters	Description
streamHandle	Handle to the Audio v1.0 stream
requestHandle	Handle to the stream set sampling rate request
samplingRate	Pointer to the sampling rate

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamSamplingFrequencyGet
(
    USB_HOST_AUDIO_V1_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_REQUEST_HANDLE requestHandle,
    uint32_t *samplingFrequency
)
```

USB_HOST_AUDIO_V1_StreamSamplingFrequencySet Function

Schedules an audio stream set sampling rate request for the specified audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamSamplingFrequencySet(USB_HOST_AUDIO_V1_STREAM_HANDLE
streamHandle, USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle, uint32_t * samplingFrequency);
```

Returns

- `USB_HOST_AUDIO_V1_RESULT_SUCCESS` - The operation was successful
- `USB_HOST_AUDIO_V1_RESULT_HANDLE_INVALID` - The specified audio stream does not exist
- `USB_HOST_AUDIO_V1_RESULT_FAILURE` - An unknown failure occurred

Description

This function schedules an audio stream set sampling rate request for the specified audio stream. A `USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE` event is generated when this request is completed. `USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE_DATA` returns the status and request handle of the request.

Remarks

None.

Preconditions

The audio stream should have been opened.

Parameters

Parameters	Description
<code>streamHandle</code>	Handle to the Audio v1.0 stream
<code>requestHandle</code>	Handle to the stream set sampling rate request
<code>samplingRate</code>	Pointer to the sampling rate

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_StreamSamplingFrequencySet
(
    USB_HOST_AUDIO_V1_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_REQUEST_HANDLE requestHandle,
    uint32_t *samplingFrequency
)
```

USB_HOST_AUDIO_V1_0_StreamRead Function

Schedules an audio stream read request for the specified audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_0_StreamRead(USB_HOST_AUDIO_V1_STREAM_HANDLE streamHandle,
USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE * transferHandle, void * source, size_t length);
```

Returns

- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_SUCCESS` - The operation was successful
- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_HANDLE_INVALID` - The specified audio stream does not exist
- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_FAILURE` - An unknown failure occurred

Description

This function schedules an audio stream read request for the specified audio stream. A

`USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE` event is generated when this request is completed.
`USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_DATA` returns the status and request handle of the request.

Remarks

None.

Preconditions

The audio stream should have been opened and enabled. The direction of the audio stream should be
`USB_HOST_AUDIO_V1_0_DIRECTION_IN`.

Parameters

Parameters	Description
streamHandle	Handle to the Audio v1.0 stream
transferHandle	Handle to the stream read transfer request
source	Pointer to the buffer containing data to be read from the device
length	Amount of data to read (in bytes)

Function

```
USB_HOST_AUDIO_V1_0_STREAM_RESULT USB_HOST_AUDIO_V1_0_StreamRead
(
    USB_HOST_AUDIO_V1_0_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE * transferHandle,
    void * source,
    size_t length
);
```

USB_HOST_AUDIO_V1_0_StreamWrite Function

Schedules an audio stream write request for the specified audio stream.

File

`usb_host_audio_v1_0.h`

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_0_StreamWrite(USB_HOST_AUDIO_V1_STREAM_HANDLE streamHandle,
USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE * transferHandle, void * source, size_t length);
```

Returns

- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_SUCCESS` - The operation was successful
- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_HANDLE_INVALID` - The specified audio stream does not exist
- `USB_HOST_AUDIO_V1_0_STREAM_RESULT_FAILURE` - An unknown failure occurred

Description

This function schedules an audio stream write request for the specified audio stream. A
`USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE` event is generated when this request is completed.
`USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA` returns the status and request handle of the request.

Remarks

None.

Preconditions

The audio stream should have been opened and enabled. The direction of the audio stream should be
`USB_HOST_AUDIO_V1_0_DIRECTION_OUT`.

Parameters

Parameters	Description
streamHandle	Handle to the Audio v1.0 stream
transferHandle	Handle to the stream write transfer request
source	Pointer to the buffer containing data to be written to the device

length	Amount of data to write (in bytes)
--------	------------------------------------

Function

```
USB_HOST_AUDIO_V1_0_STREAM_RESULT USB_HOST_AUDIO_V1_0_StreamWrite
(
    USB_HOST_AUDIO_V1_0_STREAM_HANDLE streamHandle,
    USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE * transferHandle,
    void * source,
    size_t length
);
```

c) Other Functions***USB_HOST_AUDIO_V1_TerminalInputChannelConfigGet Function***

Returns a structure that describes the spatial location of the logical channels of in the terminal's output audio channel cluster.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_AUDIO_CHANNEL_CONFIG USB_HOST_AUDIO_V1_TerminalInputChannelConfigGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject);
```

Returns

The structure that describes the spatial location of the logical channels.

Description

This function returns a structure that describes the spatial location of the logical channels of in the terminal's output audio channel cluster. This function is only applicable to an input terminal. Prior to calling this function the entity object should be obtained by calling [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#), [USB_HOST_AUDIO_V1_ControlEntityGetNext](#), or [USB_HOST_AUDIO_V1_EntityObjectGet](#).

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 device object
entityObject	Audio control entity object

Function

```
USB_AUDIO_CHANNEL_CONFIG USB_HOST_AUDIO_V1_TerminalInputChannelConfigGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject
);
```

USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeRangeGet Function

Schedules a control request to the Audio Device feature unit to get the range supported by the volume control on the specified channel.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeRangeGet(USB_HOST_AUDIO_V1_OBJ audioObj,
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject, USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle,
uint8_t channelNumber, void * data, size_t size);
```

Returns

- USB_HOST_AUDIO_V1_RESULT_SUCCESS - The request was scheduled successfully. requestHandle will contain a valid request handle.
- USB_HOST_AUDIO_V1_RESULT_BUSY - The control request mechanism is currently busy. Retry the request.
- USB_HOST_AUDIO_V1_RESULT_FAILURE - An unknown failure occurred. requestHandle will contain [USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID](#).
- USB_HOST_AUDIO_V1_RESULT_PARAMETER_INVALID - The data pointer or requestHandle pointer is NULL

Description

This function schedules a control request to the Audio Device feature unit to get the range supported by the volume control on the specified channel.

Prior to calling this function the user should call the [USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSubRangeNumbersGet](#) function to know how many sub-ranges are supported.

Users should calculate the 'size' parameter of this function, as follows:

```
size = Size of number of ranges + nSubRanges * (Size (MIN) + Size (MAX) + Size of (RES))
```

If the request was scheduled successfully, the requestHandle parameter will contain a request handle that uniquely identifies this request. If the request could not be scheduled successfully, requestHandle will contain [USB_HOST_AUDIO_V1_REQUEST_HANDLE_INVALID](#).

When the control request completes, the Audio v1.0 Client Driver will call the callback function that was set using the [USB_HOST_AUDIO_V1_EntityRequestCallbackSet](#) function. The context parameter specified here will be returned in the callback.

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object
requestHandle	Output parameter that will contain the handle to this request
channelNumber	Channel number to which the volume control is addressed
nSubRanges	Output parameter that will contain the number of sub-ranges when the request is completed and a callback is received

Function

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeRangeGet
(
    USB_HOST_AUDIO_V1_OBJ audioObj,
    USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject,
    USB_HOST_AUDIO_V1_REQUEST_HANDLE * requestHandle,
    uint8_t channelNumber,
    void * data,
    size_t size
);
```

USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSubRangeNumbersGet Function

Schedules a control request to an Audio Device feature unit to get the number of sub-ranges supported by the volume control on the specified channel.

File

[usb_host_audio_v1_0.h](#)

C

```
USB_HOST_AUDIO_V1_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSubRangeNumbersGet(USB_HOST_AUDIO_V1_OBJ
audioObj, USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ entityObject, USB_HOST_AUDIO_V1_REQUEST_HANDLE *
requestHandle, uint8_t channelNumber, uint16_t * nSubRanges);
```

Returns

- USB_HOST_AUDIO_V1_RESULT_SUCCESS - The request was scheduled successfully. requestHandle will contain a valid request handle.
- USB_HOST_AUDIO_V1_RESULT_BUSY - The control request mechanism is currently busy. Retry the request.
- USB_HOST_AUDIO_V1_RESULT_FAILURE - An unknown failure occurred. requestHandle will contain

[USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID](#).

- [USB_HOST_AUDIO_V1_RESULT_PARAMETER_INVALID](#) - The data pointer or requestHandle pointer is NULL

Description

This function schedules a control request to the Audio Device feature unit to get the number of sub-ranges supported by the volume control on the specified channel. Prior to calling this function the user should check if volume control exists on the specified channel by calling the [USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeExists](#) function.

If the request was scheduled successfully, the requestHandle parameter will contain a request handle that uniquely identifies this request. If the request could not be scheduled successfully, requestHandle will contain [USB_HOST_AUDIO_V1_REQUEST_HANDLE_INVALID](#).

When the control request completes, the Audio v1.0 Client Driver will call the callback function that was set using the [USB_HOST_AUDIO_V1_EntityRequestCallbackSet](#) function. The context parameter specified here will be returned in the callback.

Remarks

None.

Parameters

Parameters	Description
audioObj	USB Host Audio v1.0 Device object
entityObject	Audio control entity object
requestHandle	Output parameter that will contain the handle to this request
channelNumber	Channel number to which the volume control is addressed
nSubRanges	Output parameter that will contain the number of sub-ranges when the request is completed and a callback is received

Function

```
USB\_HOST\_AUDIO\_V1\_RESULT USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSubRangeNumbersGet
(
    USB\_HOST\_AUDIO\_V1\_OBJ audioObj,
    USB\_HOST\_AUDIO\_V1\_CONTROL\_ENTITY\_OBJ entityObject,
    USB\_HOST\_AUDIO\_V1\_REQUEST\_HANDLE * requestHandle,
    uint8_t channelNumber,
    uint16_t *nSubRanges
);
```

d) Data Types and Constants**[USB_HOST_AUDIO_V1_ATTACH_EVENT_HANDLER](#) Type**

USB Host Audio v1.0 Client Driver attach event handler function pointer type.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef void (* USB\_HOST\_AUDIO\_V1\_ATTACH\_EVENT\_HANDLER)(USB\_HOST\_AUDIO\_V1\_OBJ audioObj,
USB\_HOST\_AUDIO\_V1\_EVENT event, uintptr\_t context);
```

Description

USB Host Audio v1.0 Client Driver Attach Event Handler Function Pointer Type.

This data type defines the required function signature of the USB Host Audio v1.0 Client Driver attach event handling callback function. The application must register a pointer to the Audio v1.0 Client Driver attach events handling function whose function signature (parameter and return value types) match the types specified by this function pointer to receive attach and detach events callbacks from the Audio v1.0 Client Driver. The application should use the [USB_HOST_AUDIO_V1_AttachEventHandlerSet](#) function to register an attach event handler. The client driver will call this function with the relevant event parameters. The descriptions of the event handler function parameters are as follows:

- audioObj - Audio Device object to which this event is directed
- event - Event indicates if it is an Attach or Detach
- context - Value identifying the context of the application that was registered with the event handling function

Remarks

None.

USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ Type

Defines the type of the Audio v1.0 Host control entity object.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef uintptr_t USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ;
```

Description

USB Host Audio v1.0 Control Entity Object

This data type defines the type of the object returned by the [USB_HOST_AUDIO_V1_ControlEntityGetFirst](#) or [USB_HOST_AUDIO_V1_ControlEntityGetNext](#) functions. This application uses this object to get more information about that audio control entity.

Remarks

None.

USB_HOST_AUDIO_V1_ENTITY_REQUEST_CALLBACK Type

USB Host Audio v1.0 class driver control transfer complete callback function pointer type.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef void (* USB_HOST_AUDIO_V1_ENTITY_REQUEST_CALLBACK)(USB_HOST_AUDIO_V1_OBJ audioObj,  
USB_HOST_AUDIO_V1_REQUEST_HANDLE requestHandle, USB_HOST_AUDIO_V1_RESULT result, size_t size, uintptr_t  
context);
```

Description

USB Host Audio v1.0 Class driver Control Transfer Complete Callback Function Pointer type

This data type defines the required function signature of the USB Host Audio v1.0 class driver control transfer complete callback function. The client must provide a pointer to a control transfer complete callback function whose function signature (parameter and return value types) must match the types specified by this function pointer to receive notification when a control transfer has completed. The application should use the [USB_HOST_AUDIO_V1_EntityRequestCallbackSet](#) function to register an entity control request callback. The Audio v1.0 client driver will call this function with the relevant event parameters. The descriptions of the event handler function parameters are as follows:

- audioObj - Audio v1.0 client driver object associated with this event
- requestHandle - Request handle of the control transfer request that caused this event
- result - Completion result of the control transfer. This will be `USB_HOST_AUDIO_V1_RESULT_SUCCESS` if the control transfer completed successfully, `USB_HOST_AUDIO_V1_RESULT_FAILURE` if an unknown failure occurred, or `USB_HOST_AUDIO_V1_RESULT_REQUEST_STALLED` if the request was stalled.
- size - Size of the data stage that was transferred
- context - Value identifying the context of the application that was provided when the `USB_HOST_AUDIO_V1_ControlRequest` function was called

Remarks

None.

USB_HOST_AUDIO_V1_EVENT Enumeration

Identifies the possible events that the Audio v1.0 Class Driver attach event handler can generate.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef enum {  
    USB_HOST_AUDIO_V1_EVENT_ATTACH,
```

```
    USB_HOST_AUDIO_V1_EVENT_DETACH
} USB_HOST_AUDIO_V1_EVENT;
```

Members

Members	Description
USB_HOST_AUDIO_V1_EVENT_ATTACH	This event occurs when the Host layer has detected the Audio v1.0 Class Driver instance from a USB Audio v1.0 Device. There is no event data associated with this event.
USB_HOST_AUDIO_V1_EVENT_DETACH	This event occurs when host layer has detached the Audio v1.0 Class Driver instance from a USB Audio v1.0 Device. This can happen if the device itself was detached or if the device configuration was changed. There is no event data associated with this event.

Description

Audio v1.0 Class Driver Events

This enumeration identifies the possible events that the Audio v1.0 Class Driver attach event handler can generate. The application should register an event handler using the [USB_HOST_AUDIO_V1_AttachEventHandlerSet](#) function to receive Audio v1.0 Class Driver Attach events.

USB_HOST_AUDIO_V1_OBJ Type

Defines the type of the Audio v1.0 Host client object.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef uintptr_t USB_HOST_AUDIO_V1_OBJ;
```

Description

USB Host Audio v1.0 Object

This data type defines the type of the Audio Host client object. This type is returned by the client driver attach event handler and is used by the application to open the attached Audio v1.0 Device.

Remarks

None.

USB_HOST_AUDIO_V1_REQUEST_HANDLE Type

USB Host Audio v1.0 Client Driver request handle.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef uintptr_t USB_HOST_AUDIO_V1_REQUEST_HANDLE;
```

Description

USB Host Audio v1.0 Client Driver Request Handle

This handle is returned by the Audio v1.0 Host client driver entity control functions and audio stream control request functions. Applications should use this handle to track a request.

Remarks

None.

USB_HOST_AUDIO_V1_RESULT Enumeration

USB Host Audio v1.0 Class Driver result enumeration.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef enum {
    USB_HOST_AUDIO_V1_RESULT_FAILURE,
```

```

USB_HOST_AUDIO_V1_RESULT_BUSY,
USB_HOST_AUDIO_V1_RESULT_REQUEST_STALLED,
USB_HOST_AUDIO_V1_RESULT_INVALID_PARAMETER,
USB_HOST_AUDIO_V1_RESULT_DEVICE_UNKNOWN,
USB_HOST_AUDIO_V1_RESULT_HANDLE_INVALID,
USB_HOST_AUDIO_V1_RESULT_TRANSFER_ABORTED,
USB_HOST_AUDIO_V1_RESULT_OBJ_INVALID,
USB_HOST_AUDIO_V1_RESULT_END_OF_CONTROL_ENTITY,
USB_HOST_AUDIO_V1_RESULT_END_OF_STREAMING_INTERFACE,
USB_HOST_AUDIO_V1_RESULT_END_OF_INTERFACE_SETTINGS,
USB_HOST_AUDIO_V1_RESULT_SUCCESS
} USB_HOST_AUDIO_V1_RESULT;

```

Members

Members	Description
USB_HOST_AUDIO_V1_RESULT_FAILURE	An unknown failure has occurred
USB_HOST_AUDIO_V1_RESULT_BUSY	The transfer or request could not be scheduled because internal queues are full. The request or transfer should be retried
USB_HOST_AUDIO_V1_RESULT_REQUEST_STALLED	The request was stalled
USB_HOST_AUDIO_V1_RESULT_INVALID_PARAMETER	A required parameter was invalid
USB_HOST_AUDIO_V1_RESULT_DEVICE_UNKNOWN	The associated device does not exist in the system.
USB_HOST_AUDIO_V1_RESULT_HANDLE_INVALID	The specified handle is not valid
USB_HOST_AUDIO_V1_RESULT_TRANSFER_ABORTED	The transfer or requested was aborted
USB_HOST_AUDIO_V1_RESULT_OBJ_INVALID	The specified Audio v1.0 object is invalid
USB_HOST_AUDIO_V1_RESULT_END_OF_CONTROL_ENTITY	No more audio control entity
USB_HOST_AUDIO_V1_RESULT_END_OF_STREAMING_INTERFACE	No more streaming interface settings present in the audio device
USB_HOST_AUDIO_V1_RESULT_END_OF_INTERFACE_SETTINGS	No more interface alternate settings are present in the audio streaming interface
USB_HOST_AUDIO_V1_RESULT_SUCCESS	Indicates that the operation succeeded or the request was accepted and will be processed.

Description

USB Host Audio v1.0 Class Driver Result enumeration.

This enumeration lists the possible USB Host Audio v1.0 Class Driver operation results. These values are returned by Audio v1.0 Class Driver functions.

Remarks

None.

USB_HOST_AUDIO_V1_0_ATTACH_EVENT_HANDLER Type

USB Host Audio v1.0 Client Driver attach event handler function pointer type.

File

[usb_host_audio_v1_0.h](#)

C

```

typedef void (* USB_HOST_AUDIO_V1_0_ATTACH_EVENT_HANDLER)(USB_HOST_AUDIO_V1_0_OBJ audioObj,
USB_HOST_AUDIO_V1_0_EVENT event, uintptr_t context);

```

Description

USB Host Audio v1.0 Client Driver Attach Event Handler Function Pointer Type.

This data type defines the required function signature of the USB Host Audio v1.0 Client Driver attach event handling callback function. The application must register a pointer to a Audio v1.0 Client Driver attach events handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive attach and detach events call backs from the Audio v1.0 Client Driver. The client driver will invoke this function with event relevant parameters. The descriptions of the event handler function parameters are as follows:

- audioObj - Handle of the client to which this event is directed
- event - Event indicates if it is an attach or detach
- context - Value identifying the context of the application that was registered with the event handling function

Remarks

None.

USB_HOST_AUDIO_V1_STREAM_DIRECTION Enumeration

USB Host Audio v1.0 Class Driver stream direction.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef enum {
    USB_HOST_AUDIO_V1_DIRECTION_OUT,
    USB_HOST_AUDIO_V1_DIRECTION_IN
} USB_HOST_AUDIO_V1_STREAM_DIRECTION;
```

Members

Members	Description
USB_HOST_AUDIO_V1_DIRECTION_OUT	Stream Direction Host to Device
USB_HOST_AUDIO_V1_DIRECTION_IN	Stream Direction Device to Host

Description

USB Host Audio v1.0 Class Driver Stream Direction

This enumeration lists the possible audio stream directions.

Remarks

None.

USB_HOST_AUDIO_V1_STREAM_EVENT Enumeration

Identifies the possible events that the Audio v1.0 Stream can generate.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef enum {
    USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE,
    USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE,
    USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE,
    USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_SET_COMPLETE,
    USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_GET_COMPLETE,
    USB_HOST_AUDIO_V1_STREAM_EVENT_DETACH
} USB_HOST_AUDIO_V1_STREAM_EVENT;
```

Members

Members	Description
USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE	This event occurs when a Audio v1.0 stream read operation has completed (i.e., when the data has been received from the connected Audio v1.0 stream). This event is generated after the application calls the USB_HOST_AUDIO_V1_StreamRead function. The eventData parameter in the event callback function will be of a pointer to a USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA structure. This contains details about the transfer handle associated with this read request, the amount of data read and the termination status of the read request.

USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE	This event occurs when an Audio v1.0 stream write operation has completed (i.e., when the data has been written to the connected Audio v1.0 stream). This event is generated after the application calls the USB_HOST_AUDIO_V1_StreamWrite function. The eventData parameter in the event callback function will be of a pointer to a USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE_DATA structure. This contains details about the transfer handle associated with this write request, the amount of data written and the termination status of the write request.
USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE	This event occurs when an audio streaming set interface request has been completed. This event is generated after the application calls the USB_HOST_AUDIO_V1_StreamingInterfaceSet function. The eventData parameter in the event callback function will be of a pointer to a USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE_DATA . This contains details about the request handle associated with the interface set request and the termination status of the request.
USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_SET_COMPLETE	This event occurs when an Audio v1.0 sampling frequency set request has been completed. This event is generated after the application calls the USB_HOST_AUDIO_V1_StreamSamplingFrequencySet function. The eventData parameter in the event callback function will be of a pointer to a USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_SET_COMPLETE_DATA . This contains details about the request handle associated with this sampling frequency set request and the termination status of the request.
USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_GET_COMPLETE	This event occurs when an Audio v1.0 sampling frequency get request has been completed. This event is generated after the application calls the USB_HOST_AUDIO_V1_StreamingFrequencyGet function. The eventData parameter in the event callback function will be of a pointer to a USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_GET_COMPLETE_DATA . This contains details about the request handle associated with this sampling frequency get request and the termination status of the request.
USB_HOST_AUDIO_V1_STREAM_EVENT_DETACH	This event occurs when an audio stream is detached from the Host. This can happen if the Audio device itself was detached, or if the Audio device configuration was changed. There is no event data associated with this event.

Description

Audio v1.0 Stream Events

This enumeration identifies the possible events that the Audio v1.0 Stream can generate. The application should register an event handler using the [USB_HOST_AUDIO_V1_StreamEventHandlerSet](#) function to receive Audio v1.0 stream events.

An event may have data associated with it. Events that are generated due to a transfer of data between the host and device are accompanied by data structures that provide the status of the transfer termination. For example, the [USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE](#) event is accompanied by a pointer to a [USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA](#) data structure. The transferStatus member of this data structure indicates the success or failure of the transfer. A transfer may fail due to the device not responding on the bus, or if the device stalls any stages of the transfer. The event description provides details on the nature of the event and the data that is associated with the event.

USB_HOST_AUDIO_V1_STREAM_EVENT_HANDLER Type

USB Host Audio v1.0 Class Driver stream event handler function pointer type.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef USB_HOST_AUDIO_V1_STREAM_EVENT_RESPONSE (*
USB_HOST_AUDIO_V1_STREAM_EVENT_HANDLER)(USB_HOST_AUDIO_V1_STREAM_HANDLE handle,
USB_HOST_AUDIO_V1_STREAM_EVENT event, void * eventData, uintptr_t context);
```

Description

USB Host Audio v1.0 Class Driver Stream Event Handler Function Pointer Type.

This data type defines the required function signature of the USB Host Audio v1.0 Class Driver Stream event handling callback function. The application must register a pointer to the Audio v1.0 Class Driver stream events handling function whose function signature (parameter and return value types) match the types specified by this function pointer to receive event callbacks from the Audio v1.0 Class Driver. The application should use the [USB_HOST_AUDIO_V1_StreamEventHandlerSet](#) function to register an audio stream event handler. The class driver will call this function with the relevant event parameters. The descriptions of the stream event handler function parameters are as follows:

- handle - Handle to the Audio v1.0 stream
- event - Type of event generated
- eventData - This parameter should be type casted to an event specific pointer type based on the event that has occurred. Refer to the [USB_HOST_AUDIO_V1_STREAM_EVENT](#) enumeration description for more information.
- context - Value identifying the context of the application that was registered with the event handling function

Remarks

None.

[USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE_DATA](#) Structure

USB Host Audio v1.0 class stream control event data.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef struct {
    USB_HOST_AUDIO_V1_REQUEST_HANDLE requestHandle;
    USB_HOST_AUDIO_V1_RESULT requestStatus;
} USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE_DATA,
USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE_DATA,
USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_GET_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_AUDIO_V1_REQUEST_HANDLE requestHandle;	Transfer handle of this transfer
USB_HOST_AUDIO_V1_RESULT requestStatus;	Transfer termination status

Description

USB Host Audio v1.0 Class Stream Control Event Data.

This data type defines the data structure returned by the Audio V1.0 stream in conjunction with the following events:

- [USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE](#)
- [USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_SET_COMPLETE](#)
- [USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_GET_COMPLETE](#)

Remarks

None.

[USB_HOST_AUDIO_V1_0_CONTROL_CALLBACK](#) Type

USB Host Audio v1.0 Class Driver control transfer complete callback function pointer type.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef void (* USB_HOST_AUDIO_V1_0_CONTROL_CALLBACK)(USB_HOST_AUDIO_V1_0_OBJ audioObj,
USB_HOST_AUDIO_V1_0_REQUEST_HANDLE requestHandle, USB_HOST_AUDIO_V1_0_RESULT result, size_t size, uintptr_t context);
```

Description

USB Host Audio v1.0 Class driver Control Transfer Complete Callback Function Pointer type

This data type defines the required function signature of the USB Host Audio v1.0 Class Driver control transfer complete callback function. The client must provide a pointer to a control transfer complete callback function whose function signature (parameter and return value types) must match the types specified by this function pointer to receive notification when a control transfer has completed. The pointer to the callback function must be specified in [USB_HOST_AUDIO_V1_0_ControlRequest](#) function. The Audio v1.0 client driver will invoke this function with event relevant parameters. The descriptions of the event handler function parameters are as follows:

- audioObj - Audio v1.0 client driver object associated with this event
- requestHandle - Request handle of the control transfer request that caused this event
- result - Completion result of the control transfer. This will be USB_HOST_AUDIO_V1_0_RESULT_SUCCESS if the control transfer completed successfully, USB_HOST_AUDIO_V1_0_RESULT_FAILURE if an unknown failure occurred, or USB_HOST_AUDIO_V1_0_RESULT_REQUEST_STALLED if the request was stalled.

size - Size of the data stage that was transferred context - Value identifying the context of the application that was provided when the [USB_HOST_AUDIO_V1_0_ControlRequest](#) function was called.

Remarks

None.

USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA Structure

USB Host Audio v1.0 class stream data transfer event data.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef struct {
    USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE transferHandle;
    size_t length;
    USB_HOST_AUDIO_V1_RESULT result;
} USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA, USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE transferHandle;	Transfer handle of this transfer
size_t length;	Amount of data transferred
USB_HOST_AUDIO_V1_RESULT result;	Transfer termination status

Description

USB Host Audio v1.0 Class Stream Data Transfer Event Data.

This data type defines the data structure returned by the Audio V1.0 stream in conjunction with the following events:

- [USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA](#)
- [USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE_DATA](#)

Remarks

None.

USB_HOST_AUDIO_V1_0_EVENT Macro

Identifies the possible events that the Audio v1.0 Class Driver can generate.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_EVENT USB_HOST_AUDIO_V1_EVENT
```

Description

Audio v1.0 Class Driver Events

This enumeration identifies the possible events that the Audio v1.0 Class Driver can generate. The application should register an event handler using the [USB_HOST_AUDIO_V1_0_AttachEventHandlerSet](#) function to receive Audio v1.0 Class Driver events.

USB_HOST_AUDIO_V1_STREAM_EVENT_RESPONSE Enumeration

Returns the type of the USB Host Audio v1.0 stream event handler.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef enum {
    USB_HOST_AUDIO_V1_STREAM_EVENT_RESPONSE_NONE
} USB_HOST_AUDIO_V1_STREAM_EVENT_RESPONSE;
```

Members

Members	Description
USB_HOST_AUDIO_V1_STREAM_EVENT_RESPONSE_NONE	This means no response is required

Description

USB Host Audio v1.0 Stream Event Handler Return Type

This enumeration lists the possible return values of the USB Host Audio v1.0 stream event handler.

Remarks

None.

USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE_DATA Structure

USB Host Audio v1.0 class stream control event data.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef struct {
    USB_HOST_AUDIO_V1_REQUEST_HANDLE requestHandle;
    USB_HOST_AUDIO_V1_RESULT requestStatus;
} USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE_DATA,
USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE_DATA,
USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_GET_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_AUDIO_V1_REQUEST_HANDLE requestHandle;	Transfer handle of this transfer
USB_HOST_AUDIO_V1_RESULT requestStatus;	Transfer termination status

Description

USB Host Audio v1.0 Class Stream Control Event Data.

This data type defines the data structure returned by the Audio V1.0 stream in conjunction with the following events:

- [USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE](#)
- [USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_SET_COMPLETE](#)
- [USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_GET_COMPLETE](#)

Remarks

None.

USB_HOST_AUDIO_V1_0_OBJ Macro

Defines the type of the Audio v1.0 Host client object.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_OBJ USB_HOST_AUDIO_V1_OBJ
```

Description

USB Host Audio v1.0 Object

This type defines the type of the Audio Host client object. This type is returned by the attach event handler and is used by the application to open the attached Audio v1.0 Device.

Remarks

None.

USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE_DATA Structure

USB Host Audio v1.0 class stream data transfer event data.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef struct {
    USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE transferHandle;
    size_t length;
    USB_HOST_AUDIO_V1_RESULT result;
} USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA, USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE transferHandle;	Transfer handle of this transfer
size_t length;	Amount of data transferred
USB_HOST_AUDIO_V1_RESULT result;	Transfer termination status

Description

USB Host Audio v1.0 Class Stream Data Transfer Event Data.

This data type defines the data structure returned by the Audio V1.0 stream in conjunction with the following events:

- [USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA](#)
- [USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE_DATA](#)

Remarks

None.

USB_HOST_AUDIO_V1_0_REQUEST_HANDLE Macro

USB Host Audio v1.0 Client Driver request handle.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_REQUEST_HANDLE USB_HOST_AUDIO_V1_REQUEST_HANDLE
```

Description

USB Host Audio v1.0 Client Driver Request Handle

This is returned by the Audio v1.0 Client Driver command routines and should be used by the application to track the command especially in cases

where transfers are queued.

Remarks

None.

USB_HOST_AUDIO_V1_STREAM_HANDLE Type

Defines the type of the Audio v1.0 Host stream handle.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef uintptr_t USB_HOST_AUDIO_V1_STREAM_HANDLE;
```

Description

USB Host Audio stream handle

This data type defines the type of the handle returned by [USB_HOST_AUDIO_V1_StreamOpen](#) function. The application uses this handle to interact with an Audio Stream.

Remarks

None.

USB_HOST_AUDIO_V1_0_RESULT Enumeration

USB Host Audio v1.0 Class Driver audio result enumeration.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef enum {
    USB_HOST_AUDIO_V1_0_RESULT_BUSY = USB_HOST_AUDIO_V1_0_RESULT_TRANSFER_ABORTED,
    USB_HOST_AUDIO_V1_0_RESULT_REQUEST_STALLED,
    USB_HOST_AUDIO_V1_0_RESULT_OBJ_INVALID,
    USB_HOST_AUDIO_V1_0_RESULT_END_OF_STREAM_LIST,
    USB_HOST_AUDIO_V1_0_RESULT_PARAMETER_INVALID,
    USB_HOST_AUDIO_V1_0_RESULT_DEVICE_UNKNOWN,
    USB_HOST_AUDIO_V1_0_RESULT_FAILURE,
    USB_HOST_AUDIO_V1_0_RESULT_FALSE = 0,
    USB_HOST_AUDIO_V1_0_RESULT_TRUE = 1,
    USB_HOST_AUDIO_V1_0_RESULT_SUCCESS = USB_HOST_RESULT_TRUE
} USB_HOST_AUDIO_V1_0_RESULT;
```

Members

Members	Description
<code>USB_HOST_AUDIO_V1_0_RESULT_BUSY = USB_HOST_AUDIO_V1_0_RESULT_TRANSFER_ABORTED</code>	The transfer or request could not be scheduled because internal <ul style="list-style-type: none"> queues are full. The request or transfer should be retried
<code>USB_HOST_AUDIO_V1_0_RESULT_REQUEST_STALLED</code>	The request was stalled
<code>USB_HOST_AUDIO_V1_0_RESULT_OBJ_INVALID</code>	The specified Audio v1.0 Object is Invalid
<code>USB_HOST_AUDIO_V1_0_RESULT_END_OF_STREAM_LIST</code>	No more audio stream present in the Device
<code>USB_HOST_AUDIO_V1_0_RESULT_PARAMETER_INVALID</code>	A required parameter was invalid
<code>USB_HOST_AUDIO_V1_0_RESULT_DEVICE_UNKNOWN</code>	The specified device does not exist in the system
<code>USB_HOST_AUDIO_V1_0_RESULT_FAILURE</code>	An unknown failure has occurred
<code>USB_HOST_AUDIO_V1_0_RESULT_FALSE = 0</code>	Indicates a false condition
<code>USB_HOST_AUDIO_V1_0_RESULT_TRUE = 1</code>	Indicate a true condition
<code>USB_HOST_AUDIO_V1_0_RESULT_SUCCESS = USB_HOST_RESULT_TRUE</code>	Indicates that the operation succeeded or the request was accepted and will be processed.

Description

USB Host Audio v1.0 Class Driver Result enumeration.

This enumeration lists the possible USB Host Audio v1.0 Class Driver operation results. These values are returned by Audio v1.0 Class Driver functions.

Remarks

None.

USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE Type

USB Host Audio v1.0 Class Driver stream data transfer handle.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef uintptr_t USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE;
```

Description

USB Host Audio v1.0 Class Driver Stream Data Transfer Handle

This handle is returned by the Audio v1.0 Class driver stream data transfer functions and should be used by the application to track the transfer, especially in cases where transfers are queued.

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_DIRECTION Macro

USB Host Audio v1.0 Class Driver stream direction.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_STREAM_DIRECTION USB_HOST_AUDIO_V1_STREAM_DIRECTION
```

Description

USB Host Audio v1.0 Class Driver Stream Direction

This macro defines the stream direction of the USB Host Audio v1.0 Class Driver.

Remarks

None.

USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ Type

Defines the type of the Audio v1.0 Host streaming interface object.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef uintptr_t USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ;
```

Description

USB Host Audio v1.0 Streaming interface Object

This data type defines the type of the Audio v1.0 Host streaming interface object. This type is returned by the `USB_AUDIO_V1_StreamingInterfaceGetFirst` and `USB_AUDIO_V1_StreamingInterfaceGetNext` functions.

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_EVENT Enumeration

Identifies the possible events that the Audio v1.0 stream can generate.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef enum {
    USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE,
    USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE,
    USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE,
    USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE,
    USB_HOST_AUDIO_V1_0_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE
} USB_HOST_AUDIO_V1_0_STREAM_EVENT;
```

Members

Members	Description
USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE	<p>This event occurs when a Audio v1.0 stream read operation has completed (i.e., when the data has been received from the connected Audio v1.0 stream). This event is generated after the application calls the USB_HOST_AUDIO_V1_0_StreamRead function. The eventData parameter in the event callback function will be of a pointer to a USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_DATA structure. This contains details about the transfer handle associated with this read request, the amount of data read and the termination status of the read request.</p>
USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE	<p>This event occurs when an Audio v1.0 stream write operation has completed (i.e., when the data has been written to the connected Audio v1.0 stream). This event is generated after the application calls the USB_HOST_AUDIO_V1_0_StreamWrte function. The eventData parameter in the event callback function will be of a pointer to a USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA structure. This contains details about the transfer handle associated with this write request, the amount of data written and the termination status of the write request.</p>
USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE	<p>This event occurs when an Audio v1.0 stream enable request has been completed. This event is generated after the application calls the USB_HOST_AUDIO_V1_0_StreamEnable function. The eventData parameter in the event callback function will be of a pointer to a USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_DATA. This contains details about the request handle associated with this stream enable request and the termination status of the Stream Enable request.</p>
USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE	<p>This event occurs when an Audio v1.0 stream disable request has been completed. This event is generated after the application calls the USB_HOST_AUDIO_V1_0_StreamDisable function. The eventData parameter in the event callback function will be of a pointer to a USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE_DATA. This contains details about the request handle associated with this stream disable request and the termination status of the Stream Disable request.</p>
USB_HOST_AUDIO_V1_0_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE	<p>This event occurs when an Audio v1.0 sampling rate set request has been completed. This event is generated after the application calls the USB_HOST_AUDIO_V1_0_StreamSamplingRateSet function. The eventData parameter in the event callback function will be of a pointer to a USB_HOST_AUDIO_V1_0_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE_DATA. This contains details about the request handle associated with this Sampling Rate Set request and the termination status of the stream disable request.</p>

Description

Audio v1.0 Stream Events

This enumeration identifies the possible events that the Audio v1.0 stream can generate. The application should register an event handler using the [USB_HOST_AUDIO_V1_0_StreamEventHandlerSet](#) function to receive Audio v1.0 stream events.

An event may have data associated with it. Events that are generated due to a transfer of data between the Host and Device are accompanied by data structures that provide the status of the transfer termination. For example, the [USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE](#) event is accompanied by a pointer to a [USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_DATA](#) data structure. The transferStatus member of this data structure indicates the success or failure of the transfer. A transfer may fail due to the Device not responding on the bus if the Device stalls any stages of the transfer or due to NAK time-outs. The event description provides details on the nature of the event and the data that is associated with the event.

[**USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ**](#) Type

Defines the type of the Audio v1.0 Host streaming interface setting object.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef uintptr_t USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ;
```

Description

USB Host Audio v1.0 Streaming Interface Setting Object

This data type defines the type of the Audio v1.0 Host streaming interface setting object. This type is returned by the [USB_AUDIO_V1_StreamingInterfaceSettingGetFirst](#) and [USB_AUDIO_V1_StreamingInterfaceSettingGetNext](#) functions.

Remarks

None.

[**USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE_DATA**](#) Structure

USB Host Audio v1.0 class stream control event data.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef struct {
    USB_HOST_AUDIO_V1_0_REQUEST_HANDLE requestHandle;
    USB_HOST_AUDIO_V1_0_RESULT requestStatus;
} USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE_DATA,
```

USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_DATA;

Members

Members	Description
USB_HOST_AUDIO_V1_0_REQUEST_HANDLE requestHandle;	Transfer handle of this transfer
USB_HOST_AUDIO_V1_0_RESULT requestStatus;	Transfer termination status

Description

USB Host Audio v1.0 Class Stream Control Event Data.

This data type defines the data structure returned by the Audio V1.0 Client Driver in conjunction with the following events:

- [USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_DATA](#)
- [USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE_DATA](#)

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_DATA Structure

USB Host Audio v1.0 class stream control event data.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef struct {
    USB_HOST_AUDIO_V1_0_REQUEST_HANDLE requestHandle;
    USB_HOST_AUDIO_V1_0_RESULT requestStatus;
} USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_DATA,
USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_AUDIO_V1_0_REQUEST_HANDLE requestHandle;	Transfer handle of this transfer
USB_HOST_AUDIO_V1_0_RESULT requestStatus;	Transfer termination status

Description

USB Host Audio v1.0 Class Stream Control Event Data.

This data type defines the data structure returned by the Audio V1.0 Client Driver in conjunction with the following events:

- [USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_DATA](#)
- [USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE_DATA](#)

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_EVENT_HANDLER Type

USB Host Audio v1.0 Class Driver stream event handler function pointer type.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE (*
USB_HOST_AUDIO_V1_0_STREAM_EVENT_HANDLER)(USB_HOST_AUDIO_V1_0_STREAM_HANDLE handle,
USB_HOST_AUDIO_V1_0_STREAM_EVENT event, void * eventData, uintptr_t context);
```

Description

USB Host Audio v1.0 Class Driver Stream Event Handler Function Pointer Type.

This data type defines the required function signature of the USB Host Audio v1.0 Class Driver stream event handling callback function. The application must register a pointer to a Audio v1.0 Class Driver stream events handling function whose function signature (parameter and return value types) match the types specified by this function pointer to receive event call backs from the Audio v1.0 Class Driver. The class driver will call this function with relevant event parameters. The descriptions of the event handler function parameters are as follows:

- handle - Handle to the Audio v1.0 stream
- event - Type of event generated
- eventData - This parameter should be type casted to an event specific pointer type based on the event that has occurred. Refer to the [USB_HOST_AUDIO_V1_0_STREAM_EVENT](#) enumeration description for more information.
- context - Value identifying the context of the application that was registered with the event handling function

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_DATA Macro**File**

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_DATA  
USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA
```

Description

This is macro USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_DATA.

USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE Macro

Returns the type of the USB Audio v1.0 Host Client event handler.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE USB_HOST_AUDIO_V1_STREAM_EVENT_RESPONSE
```

Description

USB Host Audio v1.0 Event Handler Return Type

This enumeration lists the possible return values of the USB Audio v1.0 Host Client Driver event handler.

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA Macro

USB Host Audio v1.0 class stream transfer event data.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA  
USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE_DATA
```

Description

USB Host Audio v1.0 Class Stream Transfer Event Data.

This data type defines the data structure returned by the Audio V1.0 Client Driver in conjunction with the following events:

- [USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_DATA](#)
- [USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA](#)

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_HANDLE Macro

Defines the type of the Audio v1.0 Host stream handle.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_STREAM_HANDLE USB_HOST_AUDIO_V1_STREAM_HANDLE
```

Description

USB Host Audio stream handle

This type defines the type of the handle returned by the [USB_HOST_AUDIO_V1_0_StreamOpen](#) function. This application uses this handle to interact with an audio stream.

Remarks

None.

USB_HOST_AUDIO_V1_INTERFACE Macro

USB HOST Audio v1.0 Client Driver interface.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_INTERFACE
```

Description

USB HOST Audio V1 Client Driver Interface

This macro should be used by the application in the TPL table while adding support for the USB Audio v1.0 Host Client Driver.

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_INFO Structure**File**

[usb_host_audio_v1_0.h](#)

C

```
typedef struct {
    USB_HOST_AUDIO_V1_0_STREAM_OBJ streamObj;
    USB_AUDIO_FORMAT_CODE format;
    USB_HOST_AUDIO_V1_0_STREAM_DIRECTION streamDirection;
    uint8_t nChannels;
    uint8_t subFrameSize;
    uint8_t bitResolution;
    uint8_t nSamplingRates;
    uint32_t tSamFreq[USB_HOST_AUDIO_V1_SAMPLING_FREQUENCIES_NUMBER];
} USB_HOST_AUDIO_V1_0_STREAM_INFO;
```

Members

Members	Description
USB_HOST_AUDIO_V1_0_STREAM_OBJ streamObj;	Audio Stream Object. Clients need to pass this object when opening this audio stream using USB_HOST_AUDIO_V1_0_StreamOpen function.
USB_AUDIO_FORMAT_CODE format;	Audio Format code for this Stream
USB_HOST_AUDIO_V1_0_STREAM_DIRECTION streamDirection;	Stream direction
uint8_t nChannels;	Number of physical channels in the audio stream
uint8_t subFrameSize;	Number of bytes occupied by one audio sub-frame
uint8_t bitResolution;	Number of effectively used bits from the available bits in an audio sub-frame
uint8_t nSamplingRates;	Indicates how the sampling frequency can be programmed: 0: Continuous sampling frequency 1..255: Number of discrete sampling frequencies supported by Audio stream
uint32_t tSamFreq[USB_HOST_AUDIO_V1_SAMPLING_FREQUENCIES_NUMBER];	Supported sampling Frequencies

Description

This is type `USB_HOST_AUDIO_V1_0_STREAM_INFO`.

USB_HOST_AUDIO_V1_REQUEST_HANDLE_INVALID Macro

USB Host Audio v1.0 Client Driver invalid request handle.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_REQUEST_HANDLE_INVALID ((USB_HOST_AUDIO_V1_REQUEST_HANDLE)(-1))
```

Description

USB Host Audio v1.0 Client Driver Invalid Request Handle

This handle is returned by the Audio v1.0 Client driver command routines when the request could not be scheduled.

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_OBJ Type

Defines the type of the Audio v1.0 Host stream object.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef uintptr_t USB_HOST_AUDIO_V1_0_STREAM_OBJ;
```

Description

USB Host Audio v1.0 Stream Object

This type defines the type of the Audio v1.0 Host stream object. This type is returned by `USB_AUDIO_V1_0_StreamGetFirst` and `USB_AUDIO_V1_0_StreamGetNext` as part of [`USB_HOST_AUDIO_V1_0_STREAM_INFO`](#) structure.

Remarks

None.

USB_HOST_AUDIO_V1_STREAM_HANDLE_INVALID Macro

Defines Audio v1.0 Host stream invalid handle.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_STREAM_HANDLE_INVALID ((USB_HOST_AUDIO_V1_STREAM_HANDLE)(-1))
```

Description

USB Host Audio stream Invalid handle

This handle is returned by the [`USB_HOST_AUDIO_V1_StreamOpen`](#) function when a stream open has failed.

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_RESULT Enumeration

USB Host Audio v1.0 stream result enumeration.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef enum {
    USB_HOST_AUDIO_V1_0_STREAM_RESULT_REQUEST_BUSY = USB_HOST_RESULT_REQUEST_BUSY,
    USB_HOST_AUDIO_V1_0_STREAM_RESULT_TRANSFER_ABORTED,
    USB_HOST_AUDIO_V1_0_STREAM_RESULT_REQUEST_STALLED,
    USB_HOST_AUDIO_V1_0_STREAM_RESULT_HANDLE_INVALID,
    USB_HOST_AUDIO_V1_0_STREAM_RESULT_END_OF_DEVICE_LIST,
    USB_HOST_AUDIO_V1_0_STREAM_RESULT_INTERFACE_UNKNOWN,
    USB_HOST_AUDIO_V1_0_STREAM_RESULT_PARAMETER_INVALID,
    USB_HOST_AUDIO_V1_0_STREAM_RESULT_CONFIGURATION_UNKNOWN,
}
```

```

USB_HOST_AUDIO_V1_0_STREAM_RESULT_BUS_NOT_ENABLED,
USB_HOST_AUDIO_V1_0_STREAM_RESULT_BUS_UNKNOWN,
USB_HOST_AUDIO_V1_0_STREAM_RESULT_UNKNOWN,
USB_HOST_AUDIO_V1_0_STREAM_RESULT_FAILURE,
USB_HOST_AUDIO_V1_0_STREAM_RESULT_FALSE = 0,
USB_HOST_AUDIO_V1_0_STREAM_RESULT_TRUE = 1,
USB_HOST_AUDIO_V1_0_STREAM_SUCCESS = USB_HOST_RESULT_TRUE
} USB_HOST_AUDIO_V1_0_STREAM_RESULT;

```

Members

Members	Description
USB_HOST_AUDIO_V1_0_STREAM_RESULT_REQUEST_BUSY = USB_HOST_RESULT_REQUEST_BUSY	The transfer or request could not be scheduled because internal • queues are full. The request or transfer should be retried
USB_HOST_AUDIO_V1_0_STREAM_RESULT_TRANSFER_ABORTED	Request was aborted
USB_HOST_AUDIO_V1_0_STREAM_RESULT_REQUEST_STALLED	Request was stalled
USB_HOST_AUDIO_V1_0_STREAM_RESULT_HANDLE_INVALID	The specified Stream Handle is not valid
USB_HOST_AUDIO_V1_0_STREAM_RESULT_END_OF_DEVICE_LIST	The end of the device list was reached.
USB_HOST_AUDIO_V1_0_STREAM_RESULT_INTERFACE_UNKNOWN	The specified interface is not available
USB_HOST_AUDIO_V1_0_STREAM_RESULT_PARAMETER_INVALID	A NULL parameter was passed to the function
USB_HOST_AUDIO_V1_0_STREAM_RESULT_CONFIGURATION_UNKNOWN	The specified configuration does not exist on this device.
USB_HOST_AUDIO_V1_0_STREAM_RESULT_BUS_NOT_ENABLED	A bus operation was requested but the bus was not operated
USB_HOST_AUDIO_V1_0_STREAM_RESULT_BUS_UNKNOWN	The specified bus does not exist in the system
USB_HOST_AUDIO_V1_0_STREAM_RESULT_UNKNOWN	The specified audio stream does not exist in the system
USB_HOST_AUDIO_V1_0_STREAM_RESULT_FAILURE	An unknown failure has occurred
USB_HOST_AUDIO_V1_0_STREAM_RESULT_FALSE = 0	Indicates a false condition
USB_HOST_AUDIO_V1_0_STREAM_RESULT_TRUE = 1	Indicate a true condition
USB_HOST_AUDIO_V1_0_STREAM_SUCCESS = USB_HOST_RESULT_TRUE	Indicates that the operation succeeded or the request was accepted and will be processed.

Description

USB Host Audio v1.0 Stream Result enumeration.

This enumeration lists the possible USB Host Audio v1.0 stream operation results. These values are returned by Audio v1.0 stream functions.

Remarks

None.

USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE_INVALID Macro

USB Host Audio v1.0 Class Driver invalid stream data transfer handle.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE_INVALID ((USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE)(-1))
```

Description

USB Host Audio v1.0 Class Driver Invalid Stream Data Transfer Handle Definition

This macro defines a USB Host Audio v1.0 Class Driver invalid stream data transfer handle. An invalid transfer handle is returned by the Audio v1.0 Class Driver stream data transfer routines when the request was not successful.

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE Macro

USB Host Audio v1.0 Class Driver transfer handle.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE
```

Description

USB Host Audio v1.0 Class Driver Transfer Handle

This is returned by the Audio v1.0 Class Driver command and data transfer routines and should be used by the application to track the transfer especially in cases where transfers are queued.

Remarks

None.

USB_HOST_AUDIO_V1_0_INTERFACE Macro

USB HOST Audio Client Driver interface.

File

```
usb_host_audio_v1_0.h
```

C

```
#define USB_HOST_AUDIO_V1_0_INTERFACE (void*)USB_HOST_AUDIO_V1_INTERFACE
```

Description

USB HOST Audio Client Driver Interface

This macro should be used by the application in the TPL table while adding support for the USB Audio Host Client Driver.

Remarks

None.

USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID Macro

USB Host Audio v1.0 Client Driver invalid request handle.

File

```
usb_host_audio_v1_0.h
```

C

```
#define USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID ((USB_HOST_AUDIO_V1_0_REQUEST_HANDLE)(-1))
```

Description

USB Host Audio v1.0 Client Driver Invalid Request Handle

This is returned by the Audio v1.0 Client Driver command routines when the request could not be scheduled.

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_HANDLE_INVALID Macro

Defines the type of the Audio v1.0 Host stream invalid handle.

File

```
usb_host_audio_v1_0.h
```

C

```
#define USB_HOST_AUDIO_V1_0_STREAM_HANDLE_INVALID USB_HOST_AUDIO_V1_STREAM_HANDLE_INVALID
```

Description

USB Host Audio stream Invalid handle

This is returned by the [USB_HOST_AUDIO_V1_0_StreamOpen](#) function when a stream open request has failed.

Remarks

None.

USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE_INVALID Macro

USB Host Audio v1.0 Class Driver invalid transfer handle definition.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE_INVALID USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE_INVALID
```

Description

USB Host Audio v1.0 Class Driver Invalid Transfer Handle Definition

This macro defines a USB Host Audio v1.0 Class Driver invalid transfer handle. A invalid transfer handle is returned by the Audio v1.0 Class Driver data and command transfer routines when the request was not successful.

Remarks

None.

USB_HOST_AUDIO_V1_0_AttachEventHandlerSet Macro

Sets an attach/detach event handler.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_AttachEventHandlerSet USB_HOST_AUDIO_V1_AttachEventHandlerSet
```

Returns

- `USB_HOST_AUDIO_V1_0_RESULT_SUCCESS` - if the attach event handler was registered successfully
- `USB_HOST_AUDIO_V1_0_RESULT_FAILURE` - if the number of registered event handlers has exceeded `USB_HOST_AUDIO_V1_0_ATTACH_LISTENERS_NUMBER`

Description

This function will set an attach event handler. The attach event handler will be called when a Audio v1.0 device has been attached or detached. The context will be returned in the event handler. This function should be called before the bus has been enabled.

Remarks

This function should be called before the [USB_HOST_BusEnable](#) function is called.

Preconditions

None.

Parameters

Parameters	Description
<code>eventHandler</code>	Pointer to the attach event handler
<code>context</code>	An application defined context that will be returned in the event handler

Function

```
USB_HOST_AUDIO_V1_0_RESULT USB_HOST_AUDIO_V1_0_AttachEventHandlerSet
(
    USB_HOST_AUDIO_V1_0_ATTACH_EVENT_HANDLER eventHandler,
    uintptr_t context
);
```

USB_HOST_AUDIO_V1_0_DeviceObjHandleGet Macro

Returns the device object handle for this Audio v1.0 Device.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_DeviceObjHandleGet USB_HOST_AUDIO_V1_DeviceObjHandleGet
```

Returns

This function will return a valid device object handle if the device is still connected to the system. Otherwise, [USB_HOST_DEVICE_OBJ_HANDLE_INVALID](#) is returned.

Description

This function returns the device object handle for this Audio v1.0 Device. This returned device object handle can be used by the application to perform device-level operations such as getting the string descriptors.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
audioDeviceObj	Audio V1.0 device object handle returned in the USB_HOST_AUDIO_V1_0_ATTACH_EVENT_HANDLER function.

Function

```
USB_HOST_DEVICE_OBJ_HANDLE USB_HOST_AUDIO_V1_0_DeviceObjHandleGet
(
    USB_HOST_AUDIO_V1_0_OBJ audioDeviceObj
);
```

USB_HOST_AUDIO_V1_0_DIRECTION_IN Macro**File**

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_DIRECTION_IN USB_HOST_AUDIO_V1_DIRECTION_IN
```

Description

This is macro `USB_HOST_AUDIO_V1_0_DIRECTION_IN`.

USB_HOST_AUDIO_V1_0_DIRECTION_OUT Macro**File**

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_DIRECTION_OUT USB_HOST_AUDIO_V1_DIRECTION_OUT
```

Description

This is macro `USB_HOST_AUDIO_V1_0_DIRECTION_OUT`.

USB_HOST_AUDIO_V1_0_EVENT_ATTACH Macro**File**

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_EVENT_ATTACH USB_HOST_AUDIO_V1_EVENT_ATTACH
```

Description

This is macro USB_HOST_AUDIO_V1_0_EVENT_ATTACH.

USB_HOST_AUDIO_V1_0_EVENT_DETACH Macro**File**

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_EVENT_DETACH USB_HOST_AUDIO_V1_EVENT_DETACH
```

Description

This is macro USB_HOST_AUDIO_V1_0_EVENT_DETACH.

USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE_NONE Macro

Returns the type of the USB Host Audio v1.0 stream event handler.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE_NONE USB_HOST_AUDIO_V1_STREAM_EVENT_RESPONSE_NONE
```

Description

USB Host Audio v1.0 Stream Event Handler Return Type

This enumeration lists the possible return values of the USB Host Audio v1.0 stream event handler.

Remarks

None.

USB_HOST_AUDIO_V1_0_StreamClose Macro

Closes the audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_0_StreamClose USB_HOST_AUDIO_V1_StreamClose
```

Returns

None.

Description

This function will close the open audio stream. This closes the association between the application entity that opened the audio stream and the audio stream. The audio stream handle becomes invalid.

Remarks

The device handle becomes invalid after calling this function.

Preconditions

None.

Parameters

Parameters	Description
audioSteamHandle	handle to the audio stream obtained from the USB_HOST_AUDIO_V1_0_StreamOpen function.

Function

```
void USB_HOST_AUDIO_V1_0_StreamClose
(
    USB\_HOST\_AUDIO\_V1\_0\_STREAM\_HANDLE audioSteamHandle
);
```

[USB_HOST_AUDIO_V1_0_StreamOpen Macro](#)

Opens the specified audio stream.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB\_HOST\_AUDIO\_V1\_0\_StreamOpen USB_HOST_AUDIO_V1_0_StreamOpen
```

Returns

This function will return a valid handle if the audio stream could be opened successfully; otherwise, it will return [USB_HOST_AUDIO_V1_0_STREAM_RESULT_HANDLE_INVALID](#). The function will return a valid handle if the stream is ready to be opened.

Description

This function will open the specified audio stream. Once opened, the audio stream can be accessed via the handle which this function returns. The [audioStreamObj](#) parameter is the value returned in the [USB_HOST_AUDIO_V1_0_StreamGetFirst](#) or [USB_HOST_AUDIO_V1_0_StreamGetNext](#) functions.

Remarks

None.

Preconditions

The audio stream object should be valid.

Parameters

Parameters	Description
audioStreamObj	Audio stream object

Section

Audio Stream Access Functions

Function

```
USB\_HOST\_AUDIO\_V1\_0\_STREAM\_HANDLE USB_HOST_AUDIO_V1_0_StreamOpen
(
    USB\_HOST\_AUDIO\_V1\_0\_STREAM\_OBJ audioStreamObj
);
```

USB_HOST_AUDIO_V1_SAMPLING_FREQUENCIES_NUMBER Macro

This structure defines USB Host audio stream information structure.

File

[usb_host_audio_v1_0.h](#)

C

```
#define USB_HOST_AUDIO_V1_SAMPLING_FREQUENCIES_NUMBER USB_HOST_AUDIO_V1_0_SAMPLING_FREQUENCIES_NUMBER
```

Description

USB Host Audio stream Info table structure

This structure is an out parameter to the functions [USB_HOST_AUDIO_V1_0_StreamGetFirst](#) and [USB_HOST_AUDIO_V1_0_StreamGetNext](#) functions. This structure contains information about an audio stream in the attached Audio Device. This structure contains the stream object, audio format, etc.

Remarks

None.

USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_GET_COMPLETE_DATA Structure

USB Host Audio v1.0 class stream control event data.

File

[usb_host_audio_v1_0.h](#)

C

```
typedef struct {
    USB_HOST_AUDIO_V1_REQUEST_HANDLE requestHandle;
    USB_HOST_AUDIO_V1_RESULT requestStatus;
} USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE_DATA,
USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE_DATA,
USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_GET_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_AUDIO_V1_REQUEST_HANDLE requestHandle;	Transfer handle of this transfer
USB_HOST_AUDIO_V1_RESULT requestStatus;	Transfer termination status

Description

USB Host Audio v1.0 Class Stream Control Event Data.

This data type defines the data structure returned by the Audio V1.0 stream in conjunction with the following events:

- [USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE](#)
- [USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_SET_COMPLETE](#)
- [USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_FREQUENCY_GET_COMPLETE](#)

Remarks

None.

Files**Files**

Name	Description
usb_host_audio_v1_0.h	USB Host Audio v1_0 Class Driver Interface Header
usb_host_audio_v1_0_config_template.h	USB host Audio v1.0 Class configuration definitions template

Description

This section lists the source and header files used by the library.

usb_host_audio_v1_0.h

USB Host Audio v1_0 Class Driver Interface Header

Enumerations

	Name	Description
	USB_HOST_AUDIO_V1_0_RESULT	USB Host Audio v1.0 Class Driver audio result enumeration.
	USB_HOST_AUDIO_V1_0_STREAM_EVENT	Identifies the possible events that the Audio v1.0 stream can generate.
	USB_HOST_AUDIO_V1_0_STREAM_RESULT	USB Host Audio v1.0 stream result enumeration.
	USB_HOST_AUDIO_V1_EVENT	Identifies the possible events that the Audio v1.0 Class Driver attach event handler can generate.
	USB_HOST_AUDIO_V1_RESULT	USB Host Audio v1.0 Class Driver result enumeration.
	USB_HOST_AUDIO_V1_STREAM_DIRECTION	USB Host Audio v1.0 Class Driver stream direction.
	USB_HOST_AUDIO_V1_STREAM_EVENT	Identifies the possible events that the Audio v1.0 Stream can generate.
	USB_HOST_AUDIO_V1_STREAM_EVENT_RESPONSE	Returns the type of the USB Host Audio v1.0 stream event handler.

Functions

	Name	Description
≡◊	USB_HOST_AUDIO_V1_0_ControlRequest	Schedules an Audio v1.0 control transfer.
≡◊	USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet	Gets the number of stream groups present in the attached Audio v1.0 Device.
≡◊	USB_HOST_AUDIO_V1_0_StreamDisable	Schedules an audio stream disable request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_0_StreamEnable	Schedules an audio stream enable request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_0_StreamEventHandlerSet	Registers an event handler with the Audio v1.0 Client Driver stream.
≡◊	USB_HOST_AUDIO_V1_0_StreamGetFirst	Returns information about first audio stream in the specified audio stream group.
≡◊	USB_HOST_AUDIO_V1_0_StreamGetNext	Returns information about the next audio stream in the specified audio stream group.
≡◊	USB_HOST_AUDIO_V1_0_StreamRead	Schedules an audio stream read request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_0_StreamSamplingRateSet	Schedules an audio stream set sampling rate request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_0_StreamWrite	Schedules an audio stream write request for the specified audio stream.
≡◊	USB_HOST_AUDIO_V1_AttachEventHandlerSet	Sets an attach/detach event handler.
≡◊	USB_HOST_AUDIO_V1_ControlEntityGetFirst	Retrieves the handle to the first audio control entity
≡◊	USB_HOST_AUDIO_V1_ControlEntityGetNext	Retrieves the handle to the next audio control entity.
≡◊	USB_HOST_AUDIO_V1_DeviceObjHandleGet	Returns the device object handle for this Audio v1.0 Device.
≡◊	USB_HOST_AUDIO_V1_EntityObjectGet	Retrieves the entity object for the entity ID.
≡◊	USB_HOST_AUDIO_V1_EntityRequestCallbackSet	Registers an audio entity request callback function with the Audio v1.0 Client Driver.
≡◊	USB_HOST_AUDIO_V1_EntityTypeGet	Returns the entity type of the audio control entity.
≡◊	USB_HOST_AUDIO_V1_FeatureUnitChannelMuteExists	Returns "true" if mute control exists for the specified channel of the feature unit.
≡◊	USB_HOST_AUDIO_V1_FeatureUnitChannelMuteGet	Schedules a get mute control request to the specified channel.
≡◊	USB_HOST_AUDIO_V1_FeatureUnitChannelMuteSet	Schedules a set mute control request to the specified channel.
≡◊	USB_HOST_AUDIO_V1_FeatureUnitChannelNumbersGet	Returns the number of channels.
≡◊	USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeExists	Returns "true" if volume control exists for the specified channel of the feature unit.
≡◊	USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeGet	Schedules a get current volume control request to the specified channel.
≡◊	USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeRangeGet	Schedules a control request to the Audio Device feature unit to get the range supported by the volume control on the specified channel.

USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSet	Schedules a set current volume control request to the specified channel.
USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSubRangeNumbersGet	Schedules a control request to an Audio Device feature unit to get the number of sub-ranges supported by the volume control on the specified channel.
USB_HOST_AUDIO_V1_FeatureUnitIDGet	Returns ID of the Feature Unit.
USB_HOST_AUDIO_V1_FeatureUnitSourceIDGet	Returns the ID of the unit or terminal to which this feature unit is connected.
USB_HOST_AUDIO_V1_StreamClose	Closes the audio stream.
USB_HOST_AUDIO_V1_StreamEventHandlerSet	Registers an event handler with the Audio v1.0 Client Driver stream.
USB_HOST_AUDIO_V1_StreamingInterfaceBitResolutionGet	Returns the bit resolution of the specified streaming interface setting.
USB_HOST_AUDIO_V1_StreamingInterfaceChannelNumbersGet	Returns the number of channels of the specified streaming interface setting.
USB_HOST_AUDIO_V1_StreamingInterfaceDirectionGet	Returns the direction of the specified streaming interface setting.
USB_HOST_AUDIO_V1_StreamingInterfaceFormatTagGet	Returns the format tag of the specified streaming interface setting.
USB_HOST_AUDIO_V1_StreamingInterfaceGetFirst	Gets the first streaming interface object from the attached Audio Device.
USB_HOST_AUDIO_V1_StreamingInterfaceGetNext	Gets the next streaming interface object from the attached Audio Device.
USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequenciesGet	Returns the sampling frequencies supported by the specified streaming interface setting.
USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequencyTypeGet	Returns the sampling frequency type of the specified streaming interface setting.
USB_HOST_AUDIO_V1_StreamingInterfaceSet	Schedules a SET_INTERFACE request to the specified audio stream.
USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetFirst	Gets the first streaming interface setting object within an audio streaming interface.
USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetNext	Gets the next streaming interface setting object within an audio streaming interface.
USB_HOST_AUDIO_V1_StreamingInterfaceSubFrameSizeGet	Returns the sub-frame size of the specified streaming interface setting.
USB_HOST_AUDIO_V1_StreamingInterfaceTerminalLinkGet	Returns the terminal link of the specified streaming interface setting.
USB_HOST_AUDIO_V1_StreamOpen	Opens the specified audio stream.
USB_HOST_AUDIO_V1_StreamRead	Schedules an audio stream read request for the specified audio stream.
USB_HOST_AUDIO_V1_StreamSamplingFrequencyGet	Schedules an audio stream get sampling rate request for the specified audio stream.
USB_HOST_AUDIO_V1_StreamSamplingFrequencySet	Schedules an audio stream set sampling rate request for the specified audio stream.
USB_HOST_AUDIO_V1_StreamWrite	Schedules an audio stream write request for the specified audio stream.
USB_HOST_AUDIO_V1_TerminalAssociationGet	Returns the associated terminal ID of the audio control terminal.
USB_HOST_AUDIO_V1_TerminalIDGet	Returns the terminal ID of the audio control entity.
USB_HOST_AUDIO_V1_TerminalInputChannelConfigGet	Returns a structure that describes the spatial location of the logical channels of in the terminal's output audio channel cluster.
USB_HOST_AUDIO_V1_TerminalInputChannelNumbersGet	Returns the number of logical output channels in the terminal's output audio channel cluster.
USB_HOST_AUDIO_V1_TerminalSourceIDGet	Returns the ID of the unit or terminal to which this terminal is connected.
USB_HOST_AUDIO_V1_TerminalTypeGet	Returns the terminal type of the audio control entity.

Macros

Name	Description
USB_HOST_AUDIO_V1_0_AttachEventHandlerSet	Sets an attach/detach event handler.

	USB_HOST_AUDIO_V1_0_DeviceObjHandleGet	Returns the device object handle for this Audio v1.0 Device.
	USB_HOST_AUDIO_V1_0_DIRECTION_IN	This is macro USB_HOST_AUDIO_V1_0_DIRECTION_IN .
	USB_HOST_AUDIO_V1_0_DIRECTION_OUT	This is macro USB_HOST_AUDIO_V1_0_DIRECTION_OUT .
	USB_HOST_AUDIO_V1_0_EVENT	Identifies the possible events that the Audio v1.0 Class Driver can generate.
	USB_HOST_AUDIO_V1_0_EVENT_ATTACH	This is macro USB_HOST_AUDIO_V1_0_EVENT_ATTACH .
	USB_HOST_AUDIO_V1_0_EVENT_DETACH	This is macro USB_HOST_AUDIO_V1_0_EVENT_DETACH .
	USB_HOST_AUDIO_V1_0_INTERFACE	USB HOST Audio Client Driver interface.
	USB_HOST_AUDIO_V1_0_OBJ	Defines the type of the Audio v1.0 Host client object.
	USB_HOST_AUDIO_V1_0_REQUEST_HANDLE	USB Host Audio v1.0 Client Driver request handle.
	USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID	USB Host Audio v1.0 Client Driver invalid request handle.
	USB_HOST_AUDIO_V1_0_STREAM_DIRECTION	USB Host Audio v1.0 Class Driver stream direction.
	USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_DATA	This is macro USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_DATA .
	USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE	Returns the type of the USB Audio v1.0 Host Client Driver event handler.
	USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE_NONE	Returns the type of the USB Host Audio v1.0 stream event handler.
	USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_DATA	USB Host Audio v1.0 class stream transfer event data.
	USB_HOST_AUDIO_V1_0_STREAM_HANDLE	Defines the type of the Audio v1.0 Host stream handle.
	USB_HOST_AUDIO_V1_0_STREAM_HANDLE_INVALID	Defines the type of the Audio v1.0 Host stream invalid handle.
	USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE	USB Host Audio v1.0 Class Driver transfer handle.
	USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE_INVALID	USB Host Audio v1.0 Class Driver invalid transfer handle definition.
	USB_HOST_AUDIO_V1_0_StreamClose	Closes the audio stream.
	USB_HOST_AUDIO_V1_0_StreamOpen	Opens the specified audio stream.
	USB_HOST_AUDIO_V1_INTERFACE	USB HOST Audio v1.0 Client Driver interface.
	USB_HOST_AUDIO_V1_REQUEST_HANDLE_INVALID	USB Host Audio v1.0 Client Driver invalid request handle.
	USB_HOST_AUDIO_V1_SAMPLING_FREQUENCIES_NUMBER	This structure defines USB Host audio stream information structure.
	USB_HOST_AUDIO_V1_STREAM_HANDLE_INVALID	Defines Audio v1.0 Host stream invalid handle.
	USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE_INVALID	USB Host Audio v1.0 Class Driver invalid stream data transfer handle.

Structures

Name	Description
USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE_DATA	USB Host Audio v1.0 class stream control event data.
USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_DATA	USB Host Audio v1.0 class stream control event data.
USB_HOST_AUDIO_V1_0_STREAM_INFO	This is type USB_HOST_AUDIO_V1_0_STREAM_INFO .
USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE_DATA	USB Host Audio v1.0 class stream control event data.
USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA	USB Host Audio v1.0 class stream data transfer event data.
USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_GET_COMPLETE_DATA	USB Host Audio v1.0 class stream control event data.
USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE_DATA	USB Host Audio v1.0 class stream control event data.

	USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE_DATA	USB Host Audio v1.0 class stream data transfer event data.
--	--	--

Types

Name	Description
USB_HOST_AUDIO_V1_0_ATTACH_EVENT_HANDLER	USB Host Audio v1.0 Client Driver attach event handler function pointer type.
USB_HOST_AUDIO_V1_0_CONTROL_CALLBACK	USB Host Audio v1.0 Class Driver control transfer complete callback function pointer type.
USB_HOST_AUDIO_V1_0_STREAM_EVENT_HANDLER	USB Host Audio v1.0 Class Driver stream event handler function pointer type.
USB_HOST_AUDIO_V1_0_STREAM_OBJ	Defines the type of the Audio v1.0 Host stream object.
USB_HOST_AUDIO_V1_ATTACH_EVENT_HANDLER	USB Host Audio v1.0 Client Driver attach event handler function pointer type.
USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ	Defines the type of the Audio v1.0 Host control entity object.
USB_HOST_AUDIO_V1_ENTITY_REQUEST_CALLBACK	USB Host Audio v1.0 class driver control transfer complete callback function pointer type.
USB_HOST_AUDIO_V1_OBJ	Defines the type of the Audio v1.0 Host client object.
USB_HOST_AUDIO_V1_REQUEST_HANDLE	USB Host Audio v1.0 Client Driver request handle.
USB_HOST_AUDIO_V1_STREAM_EVENT_HANDLER	USB Host Audio v1.0 Class Driver stream event handler function pointer type.
USB_HOST_AUDIO_V1_STREAM_HANDLE	Defines the type of the Audio v1.0 Host stream handle.
USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE	USB Host Audio v1.0 Class Driver stream data transfer handle.
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ	Defines the type of the Audio v1.0 Host streaming interface object.
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ	Defines the type of the Audio v1.0 Host streaming interface setting object.

Description

USB Host Audio v1.0 Class Driver Interface Definition

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the USB Host Audio v1.0 Class Driver.

File Name

`usb_host_audio_v1_0.h`

Company

Microchip Technology Inc.

`usb_host_audio_v1_0_config_template.h`

USB host Audio v1.0 Class configuration definitions template

Macros

Name	Description
USB_HOST_AUDIO_V1_ATTACH_LISTENERS_NUMBER	Defines the number of attach event listeners that can be registered with Audio v1.0 Host Client Driver.
USB_HOST_AUDIO_V1_INSTANCES_NUMBER	Specifies the number of Audio v1.0 instances.
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_ALTERNATE_SETTINGS_NUMBER	Defines maximum number of alternate settings per Streaming interface provided by any Device that will be connected to this Audio Host.
USB_HOST_AUDIO_V1_STREAMING_INTERFACES_NUMBER	Defines the maximum number of streaming interfaces could be present in an Audio v1.0 device that this Audio v1.0 Host Client Driver can support.

Description

USB Host Audio v1.0 Class Configuration Definitions

This file contains configurations macros needed to configure the Audio v1.0 host Driver. This file is a template file only. It should not be included by the application. The configuration macros defined in the file should be defined in the configuration specific system_config.h.

File Name

[usb_host_cdc_config_template.h](#)

Company

Microchip Technology Inc.

USB CDC Host Library

This section describes the USB CDC Host Library.

Introduction

Introduces the MPLAB Harmony USB CDC Host Library.

Description

The CDC Host Client Driver in the MPLAB Harmony USB Host Stack allows USB Host applications to support and interact with Communications Device Class (CDC) USB devices. The CDC Host Client Driver has the following features:

- Supports CDC ACM devices
- Supports CDC device matching at both the device descriptor and interface descriptor level
- Supports composite CDC devices (multiple CDC interfaces or CDC with other device classes)
- Designed to support multi-client operation
- RTOS ready
- An event driver non-clocking application interaction model
- Allows the application to send CDC ACM commands to the device
- Supports queuing of read and write data transfers

Using the Library

This topic describes the basic architecture of the USB CDC Host Client Driver Library and provides information and examples on its use.

Abstraction Model

Describes the Abstraction Model of the USB CDC Host Client Driver Library.

Description

The CDC Host Client Driver interacts with the Host Layer to control the attached CDC device. The USB Host Layer attaches the CDC Host Client Driver to the CDC device when it meets the matching criteria specified in the USB Host TPL table. The CDC Host Client Driver abstracts the details of sending CDC class specific control transfer commands by providing easy to use non-blocking API to send these command. A command, when issued, is assigned a request handle. This request handle is returned in the event that is generated when the command has been processed, and can be used by the application to track the command. The class specific command functions are implemented in `usb_host_cdc_acm.c`.

While transferring data over the data interface, the CDC Host Client Driver abstracts details such as the bulk interface, endpoints and endpoint size. The CDC Host Client Driver internally (and without application intervention) validates the CDC class specific device descriptors and opens communication pipes. While transferring data, multiple read and write requests can be queued. Each such request gets assigned a transfer handle. The transfer handle for a transfer request is returned along with the completion event for that transfer request. The data transfer routines are implemented in `usb_host_cdc.c`.

Library Overview

The USB CDC Host Client Driver API is grouped functionally, as shown in the following table.

Library Interface Section	Description
Client Access Functions	These functions allow application clients to open, close the client and register event handlers. These functions are implemented in <code>usb_host_cdc.c</code> .

Data Transfer Functions	These functions allow the application client to transfer data to the attached device. These functions are implemented in <code>usb_host_cdc.c</code> .
CDC Class-specific Command Functions	These functions allow the application to send class specific control transfer requests to the application. These functions are implemented in <code>usb_host_cdc_acm.c</code> .

How the Library Works

Describes how the Library works and how it should be used.

Description

The CDC Host Client Driver provides the user application with an easy-to-use interface to the attached CDC device. The USB Host Layer initializes the CDC Host Client Driver when a device is attached. This process does not require application intervention.

The following sections describe the steps and methods required for the user application to interact with the attached devices:

- TPL Table Configuration for CDC Devices
- Detecting Device Attach
- Opening the CDC Host Client Driver
- Sending Class-specific Control Transfers
- Reading and Writing Data
- Event Handling

TPL Table Configuration for CDC Devices

Provides information on configuring the TPL table for CDC devices.

Description

The Host Layer attaches the CDC Host Client Driver to a device when the device class, subclass, protocol in the device descriptor or when the class, subclass and protocol fields in the Interface Association Descriptor (IAD) or Interface descriptor matches the entry in the TPL table. When specifying the entry for the CDC device, the entry for the CDC device, the driver interface must be set to `USB_HOST_CDC_INTERFACE`. This will attach the CDC Host Client Driver to the device when the USB Host matches the TPL entry to the device. The following code shows possible TPL table options for matching CDC Devices.

Example:

```
/* This code shows an example of TPL table entries for supporting CDC
 * devices. Note the driver interface is set to USB_HOST_CDC_INTERFACE. This
 * will load the CDC Host Client Driver when there is TPL match */

const USB_HOST_TPL_ENTRY USBTPLList[1] =
{
    /* This entry looks for any CDC device. The CDC Host Client Driver will
     * check if this is an ACM device and will then load itself */
    TPL_INTERFACE_CLASS_SUBCLASS_PROTOCOL(USB_CDC_CLASS_CODE, USB_CDC_SUBCLASS_CODE, 0x0, NULL,
    USB_HOST_CDC_INTERFACE),

    /* This entry looks specifically for the communications class protocol.
     * This entry should be used if the host application is expected to support
     * CDC as a part of an composite device */
    TPL_INTERFACE_CLASS_SUBCLASS_PROTOCOL(USB_CDC_COMMUNICATIONS_INTERFACE_CLASS_CODE,
    USB_CDC_SUBCLASS_ABSTRACT_CONTROL_MODEL, USB_CDC_PROTOCOL_AT_V250, NULL, USB_HOST_CDC_INTERFACE),
};


```

Detecting Device Attach

Describes how to register an Attach Event Handler.

Description

The application will need to know when a CDC Device is attached. To receive this attach event from the CDC Host Client Driver, the application must register an Attach Event Handler by calling the `USB_HOST_CDC_AttachEventHandlerSet` function. This function should be called before the `USB_HOST_BusEnable` function is called, else the application may miss CDC attach events. It can be called multiple times to register multiple event handlers, each for different application clients that need to know about CDC Device Attach events.

The total number of event handlers that can be registered is defined by `USB_HOST_CDC_ATTACH_LISTENERS_NUMBER` configuration option in `system_config.h`. When a device is attached, the CDC Host Client Driver will send the attach event to all the registered event handlers. In

this event handler, the CDC Host Client Driver will pass a [USB_HOST_CDC_OBJ](#) that can be opened to gain access to the device. The following code shows an example of how to register attach event handlers.

Example:

```
/* This code shows an example of CDC Attach Event Handler and how this
 * attach event handler can be registered with the CDC Host Client Driver */

void APP_USBHostCDCAttachEventListener(USB_HOST_CDC_OBJ cdcObj, uintptr_t context)
{
    /* This function gets called when the CDC device is attached. In this
     * example we let the application know that a device is attached and we
     * store the CDC device object. This object will be required to open the
     * device. */

    appData.deviceIsAttached = true;
    appData.cdcObj = cdcObj;
}

void APP_Tasks(void)
{
    switch (appData.state)
    {
        case APP_STATE_BUS_ENABLE:

            /* In this state the application enables the USB Host Bus. Note
             * how the CDC Attach event handler is registered before the bus
             * is enabled. */

            USB_HOST_CDC_AttachEventHandlerSet(APP_USBHostCDCAttachEventListener, (uintptr_t) 0);
            USB_HOST_BusEnable(0);
            appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:
            /* Here we wait for the bus enable operation to complete. */
            break;
    }
}
```

Opening the CDC Host Client Driver

Describes how an application can open the CDC Host Client Driver.

Description

The application must open the CDC Host Client Driver to communicate and control the attached device. The device can be opened by using the [USB_HOST_CDC_Open](#) function and specifying the [USB_HOST_CDC_OBJ](#) object that was returned in the attached event handler. If the open function fails, it returns an invalid handle ([USB_HOST_CDC_HANDLE_INVALID](#)). Once opened successfully, a valid handle tracks the relationship between the client and the CDC Host Client Driver. This handle should be used with other CDC Host Client Driver functions to specify the instance of the CDC Host Client Driver being accessed.

A CDC Host Client Driver instance can be opened multiple times by different application clients. In an ROTS based application each client could running its own thread. Multiple clients can read write data to the one CDC device. In such a case, the read and write requests are queued. The following code shows an example of how the CDC Driver is opened.

Example:

```
/* This code shows an example of the how to open the CDC Host Client
 * driver. The application state machine waits for a device attach and then
 * opens the CDC Host Client Driver. */

void APP_USBHostCDCAttachEventListener(USB_HOST_CDC_OBJ cdcObj, uintptr_t context)
{
    /* This function gets called when the CDC device is attached. Update the
     * application data structure to let the application know that this device
     * is attached */

    appData.deviceIsAttached = true;
    appData.cdcObj = cdcObj;
}
```

```

void APP_Tasks(void)
{
    switch (appData.state)
    {
        case APP_STATE_BUS_ENABLE:

            /* In this state the application enables the USB Host Bus. Note
             * how the CDC Attach event handler are registered before the bus
             * is enabled. */

            USB_HOST_EventHandlerSet(APP_USBHostEventHandler, (uintptr_t)0);
            USB_HOST_CDC_AttachEventHandlerSet(APP_USBHostCDCAttachEventListener, (uintptr_t) 0);
            USB_HOST_BusEnable(0);
            appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:

            /* In this state we wait for the Bus enable to complete */
            if(USB_HOST_BusIsEnabled(0))
            {
                appData.state = APP_STATE_WAIT_FOR_DEVICE_ATTACH;
            }
            break;

        case APP_STATE_WAIT_FOR_DEVICE_ATTACH:

            /* In this state the application is waiting for the device to be
             * attached */
            if(appData.deviceIsAttached)
            {
                /* A device is attached. We can open this device */
                appData.state = APP_STATE_OPEN_DEVICE;
                appData.deviceIsAttached = false;
            }
            break;

        case APP_STATE_OPEN_DEVICE:

            /* In this state the application opens the attached device */
            appData.cdcHostHandle = USB_HOST_CDC_Open(appData.cdcObj);
            if(appData.cdcHostHandle != USB_HOST_CDC_HANDLE_INVALID)
            {
                /* The driver was opened successfully. Set the event handler
                 * and then go to the next state. */
                USB_HOST_CDC_EventHandlerSet(appData.cdcHostHandle,
                                             APP_USBHostCDCEventHandler,
                                             (uintptr_t)0);
                appData.state = APP_STATE_SET_LINE_CODING;
            }
            break;

        default:
            break;
    }
}

```

Sending Class-specific Control Transfers

Describes how the application client can send CDC Class-specific commands to the connected device.

Description

The CDC Host Client Driver allows the application client to send CDC Class specific commands to the connected device. These commands allows the application client to:

- Set the device line coding (USB_HOST_CDC_LineCodingSet)
- Retrieve the device line coding (USB_HOST_CDC_LineCodingGet)
- Set the device control line state (USB_HOST_CDC_ControlLineStateSet)

- Ask the device to send a break signal (USB_HOST_CDC_BreakSend)

These functions are non-blocking. The functions will return before the actual command execution is complete. The return value indicates if the command was scheduled successfully, or if the driver is busy and cannot accept commands, or if the command failed due to an unknown reason. If the command failed because the driver was busy, it can be retried. If scheduled successfully, the function will return a valid request handle. This request handle is unique and tracks the requested command.

When the command related control transfer has completed, the CDC Host Client Driver generates a command specific completion event. This event is accompanied by a data structure that contains information about the completion of the command. The request handler generated at the time of calling the command request function is also returned along with the event. The request handle expires after the event handler exits. The following tables show the command functions, along with the respective events and the type of the event related data.

Table 1: Set Line Coding

Function	USB_HOST_CDC_ACN_LineCodingSet
Event	USB_HOST_CDC_EVENT_ACN_SET_LINE_CODING_COMPLETE
Event Data Type	USB_HOST_CDC_EVENT_ACN_SET_LINE_CODING_COMPLETE_DATA

Table 2: Get Line Coding

Function	USB_HOST_CDC_ACN_LineCodingGet
Event	USB_HOST_CDC_EVENT_ACN_GET_LINE_CODING_COMPLETE
Event Data Type	USB_HOST_CDC_EVENT_ACN_GET_LINE_CODING_COMPLETE_DATA

Table 3: Set Control Line State

Function	USB_HOST_CDC_ACN_ControlLineStateSet
Event	USB_HOST_CDC_EVENT_ACN_CONTROL_LINE_STATE_SET_COMPLETE
Event Data Type	USB_HOST_CDC_EVENT_ACN_CONTROL_LINE_STATE_SET_COMPLETE_DATA

Table 4: Send Break

Function	USB_HOST_CDC_ACN_SendBreak
Event	USB_HOST_CDC_EVENT_ACN_SEND_BREAK_COMPLETE
Event Data Type	USB_HOST_CDC_EVENT_ACN_SEND_BREAK_COMPLETE_DATA

The following code shows an example of sending a CDC class specific commands. Refer to the [Event Handling](#) section for details on setting the event handler function.

Example:

```
/* This code shows an example of how to send CDC Class specific command
 * requests. The event handling related to each command is also shown. */

USB_HOST_CDC_EVENT_RESPONSE APP_USBHostCDCEventHandler
(
    USB_HOST_CDC_HANDLE cdcHandle,
    USB_HOST_CDC_EVENT event,
    void * eventData,
    uintptr_t context
)
{
    /* This function is called when a CDC Host event has occurred. A pointer to
     * this function is registered after opening the device. See the call to
     * USB_HOST_CDC_EventHandlerSet() function. */

    USB_HOST_CDC_EVENT_ACN_SET_LINE_CODING_COMPLETE_DATA * setLineCodingEventData;
    USB_HOST_CDC_EVENT_ACN_SET_CONTROL_LINE_STATE_COMPLETE_DATA * setControlLineStateEventData;
    USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA * writeCompleteEventData;
    USB_HOST_CDC_EVENT_READ_COMPLETE_DATA * readCompleteEventData;

    switch(event)
    {
        case USB_HOST_CDC_EVENT_ACN_SET_LINE_CODING_COMPLETE:

            /* This means the application requested Set Line Coding request is
             * complete. */
            setLineCodingEventData = (USB_HOST_CDC_EVENT_ACN_SET_LINE_CODING_COMPLETE_DATA *) (eventData);
```

```
    appData.controlRequestDone = true;
    appData.controlRequestResult = setLineCodingEventData->result;
    break;
```

```
    case USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE:
```

```
        /* This means the application requested Set Control Line State
         * request has completed. */
        setControlLineStateEventData = (USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA
*) (eventData);
        appData.controlRequestDone = true;
        appData.controlRequestResult = setControlLineStateEventData->result;
        break;
```

```
    default:
        break;
```

```
}
```

```
}
```

```
void APP_Tasks(void)
{
    switch(appData.state)
    {
        /* The application states that enable the bus and wait for device attach are
         * not shown here for brevity */

        case APP_STATE_OPEN_DEVICE:

            /* In this state the application opens the attached device */
            appData.cdcHostHandle = USB_HOST_CDC_Open(appData.cdcObj);
            if(appData.cdcHostHandle != USB_HOST_CDC_HANDLE_INVALID)
            {
                /* The driver was opened successfully. Set the event handler
                 * and then go to the next state. */
                USB_HOST_CDC_EventHandlerSet(appData.cdcHostHandle, APP_USBHostCDCEventHandler,
(uintptr_t)0);
                appData.state = APP_STATE_SET_LINE_CODING;
            }
            break;
```

```
        case APP_STATE_SET_LINE_CODING:

            /* Here we set the Line coding. The control request done flag will
             * be set to true when the control request has completed. */

            appData.controlRequestDone = false;
            result = USB_HOST_CDC_ACM_LineCodingSet(appData.cdcHostHandle, NULL,
&appData.cdcHostLineCoding);

            if(result == USB_HOST_CDC_RESULT_SUCCESS)
            {
                /* We wait for the set line coding to complete */
                appData.state = APP_STATE_WAIT_FOR_SET_LINE_CODING;
            }
            break;
```

```
        case APP_STATE_WAIT_FOR_SET_LINE_CODING:

            if(appData.controlRequestDone)
            {
                if(appData.controlRequestResult != USB_HOST_CDC_RESULT_SUCCESS)
                {
                    /* The control request was not successful. */
                    appData.state = APP_STATE_ERROR;
                }
                else
```

```

    {
        /* Next we set the Control Line State */
        appData.state = APP_STATE_SEND_SET_CONTROL_LINE_STATE;
    }
}

break;

case APP_STATE_SEND_SET_CONTROL_LINE_STATE:

    /* Here we set the control line state */
    appData.controlRequestDone = false;
    result = USB_HOST_CDC_ACM_ControlLineStateSet(appData.cdcHostHandle, NULL,
                                                &appData.controlLineState);

    if(result == USB_HOST_CDC_RESULT_SUCCESS)
    {
        /* We wait for the set line coding to complete */
        appData.state = APP_STATE_WAIT_FOR_SET_CONTROL_LINE_STATE;
    }

    break;

case APP_STATE_WAIT_FOR_SET_CONTROL_LINE_STATE:

    /* Here we wait for the control line state set request to complete */
    if(appData.controlRequestDone)
    {
        if(appData.controlRequestResult != USB_HOST_CDC_RESULT_SUCCESS)
        {
            /* The control request was not successful. */
            appData.state = APP_STATE_ERROR;
        }
        else
        {
            /* Next we set the Control Line State */
            appData.state = APP_STATE_SEND_PROMPT_TO_DEVICE;
        }
    }

    break;

default:
    break;
}
}

```

Reading and Writing Data

Describes how to transfer data to the attached CDC device.

Description

The application can use the [USB_HOST_CDC_Read](#) and [USB_HOST_CDC_Write](#) functions to transfer data to the attached CDC device. While calling these function, the client handle specifies the target CDC device and the event handler function to which the events should be sent. It is possible for multiple client to open the same instance of the CDC Host Client Driver instance and transfer data to the attached CDC Device.

Calling the [USB_HOST_CDC_Read](#) and [USB_HOST_CDC_Write](#) functions while a read/write transfer is already in progress will cause the transfer result to be queued. If the transfer was successfully queued or scheduled, the [USB_HOST_CDC_Read](#) and [USB_HOST_CDC_Write](#) functions will return a valid transfer handle. This transfer handle identifies the transfer request. The application clients can use the transfer handles to keep track of multiple queued transfers. When a transfer completes, the CDC Host Client Driver generates an event. The following tables shows the event and the event data associated with the event.

Table 1: Read

Function	USB_HOST_CDC_Read
Event	USB_HOST_CDC_EVENT_READ_COMPLETE
Event Data Type	USB_HOST_CDC_EVENT_READ_COMPLETE_DATA

Table 2: Write

Function	USB_HOST_CDC_ACm_LineCodingGet
Event	USB_HOST_CDC_EVENT_READ_COMPLETE
Event Data Type	USB_HOST_CDC_EVENT_READ_COMPLETE_DATA

The event data contains information on the amount of data transferred, completion status and the transfer handle of the transfer. The following code shows an example of reading and writing data.

Example:

```
/* In this code example, the USB_HOST_CDC_Read and the USB_HOST_CDC_Write
 * functions are used to read and write data. The event related to the read and
 * write operations are handled in the APP_USBHostCDCEventHandler function. */

USB_HOST_CDC_EVENT_RESPONSE APP_USBHostCDCEventHandler
(
    USB_HOST_CDC_HANDLE cdcHandle,
    USB_HOST_CDC_EVENT event,
    void * eventData,
    uintptr_t context
)
{
    /* This function is called when a CDC Host event has occurred. A pointer to
     * this function is registered after opening the device. */

    USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA * writeCompleteEventData;
    USB_HOST_CDC_EVENT_READ_COMPLETE_DATA * readCompleteEventData;

    switch(event)
    {
        case USB_HOST_CDC_EVENT_WRITE_COMPLETE:

            /* This means an application requested write has completed */
            appData.writeTransferDone = true;
            writeCompleteEventData = (USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA *) (eventData);
            appData.writeTransferResult = writeCompleteEventData->result;
            break;

        case USB_HOST_CDC_EVENT_READ_COMPLETE:

            /* This means an application requested write has completed */
            appData.readTransferDone = true;
            readCompleteEventData = (USB_HOST_CDC_EVENT_READ_COMPLETE_DATA *) (eventData);
            appData.readTransferResult = readCompleteEventData->result;
            break;

        default:
            break;
    }

    return(USB_HOST_CDC_EVENT_RESPONSE_NONE);
}

void APP_Tasks(void)
{
    switch(appData.state)
    {
        /* The application states that wait for device attach and open the CDC
         * Host Client Driver are not shown here for brevity */

        case APP_STATE_SEND_PROMPT_TO_DEVICE:

            /* The prompt is sent to the device here. The write transfer done
             * flag is updated in the event handler. */

            appData.writeTransferDone = false;
            result = USB_HOST_CDC_Write(appData.cdcHostHandle, NULL, prompt, 8);

            if(result == USB_HOST_CDC_RESULT_SUCCESS)
            {

```

```

        appData.state = APP_STATE_WAIT_FOR_PROMPT_SEND_COMPLETE;
    }
    break;

case APP_STATE_WAIT_FOR_PROMPT_SEND_COMPLETE:

/* Here we check if the write transfer is done */
if(appData.writeTransferDone)
{
    if(appData.writeTransferResult == USB_HOST_CDC_RESULT_SUCCESS)
    {
        /* Now to get data from the device */
        appData.state = APP_STATE_GET_DATA_FROM_DEVICE;
    }
    else
    {
        /* Try sending the prompt again. */
        appData.state = APP_STATE_SEND_PROMPT_TO_DEVICE;
    }
}
break;

case APP_STATE_GET_DATA_FROM_DEVICE:

/* Here we request data from the device */
appData.readTransferDone = false;
result = USB_HOST_CDC_Read(appData.cdcHostHandle, NULL, appData.indataArray, 1);
if(result == USB_HOST_CDC_RESULT_SUCCESS)
{
    appData.state = APP_STATE_WAIT_FOR_DATA_FROM_DEVICE;
}
break;

case APP_STATE_WAIT_FOR_DATA_FROM_DEVICE:

/* Wait for data from device. */
if(appData.readTransferDone)
{
    if(appData.readTransferResult == USB_HOST_CDC_RESULT_SUCCESS)
    {
        /* Do something with the data here */
    }
}
break;

default:
    break;
}
}

```

Event Handling

Describes how to set event handlers.

Description

The CDC Host Client Driver presents an event driven interface to the application. The CDC Host Client Driver requires the application client to set two event handlers for meaningful operation:

- The Attach event handler is not client specific and is registered before the [USB_HOST_BusEnable](#) function is called. This event handler and the attach event is discussed in the Detecting Device Attach section.
- The client specific command, data transfer and detach events. The CDC Class specific command request events are discussed in the Sending Class Specific Control Transfers section. The data transfer related events are discussed in the Reading and Writing Data section. Some general points about these events are discussed below.

A request to send a command or transfer data typically completes after the command request or transfer function has exited. The application must then use the CDC Host Client Driver event to track the completion of this command or data transfer request. In a case where multiple data transfers are queued, the transfer handles can be used to identify the transfer requests.

The application must use the `USB_HOST_CDC_EventHandlerSet` function to register a client specific event handler. This event handler will be called when a command, data transfer or detach event has occurred and should be registered before the client request for command or a data transfer. The following code shows an example of registering an event handler.

Example:

```
/* This code shows an example of setting an event handler and an example
 * event handler. For the full set of events that the CDC Host Client generates,
 * refer to USB_HOST_CDC_EVENT enumeration description */

USB_HOST_CDC_EVENT_RESPONSE APP_USBHostCDCEventHandler
(
    USB_HOST_CDC_HANDLE cdcHandle,
    USB_HOST_CDC_EVENT event,
    void * eventData,
    uintptr_t context
)
{
    /* This function is called when a CDC Host event has occurred. A pointer to
     * this function is registered after opening the device. See the call to
     * USB_HOST_CDC_EventHandlerSet() function. */

    USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA * setLineCodingEventData;
    USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA * setControlLineStateEventData;
    USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA * writeCompleteEventData;
    USB_HOST_CDC_EVENT_READ_COMPLETE_DATA * readCompleteEventData;

    switch(event)
    {
        case USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE:
            /* This means the application requested Set Line Coding request is
             * complete. */
            setData.setLineCodingEventData = (USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA *) (eventData);
            appData.controlRequestDone = true;
            appData.controlRequestResult = setLineCodingEventData->result;
            break;

        case USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE:
            /* This means the application requested Set Control Line State
             * request has completed. */
            setData.setControlLineStateEventData = (USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA *)
                (eventData);
            appData.controlRequestDone = true;
            appData.controlRequestResult = setControlLineStateEventData->result;
            break;

        case USB_HOST_CDC_EVENT_WRITE_COMPLETE:
            /* This means an application requested write has completed */
            appData.writeTransferDone = true;
            writeCompleteEventData = (USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA *) (eventData);
            appData.writeTransferResult = writeCompleteEventData->result;
            break;

        case USB_HOST_CDC_EVENT_READ_COMPLETE:
            /* This means an application requested write has completed */
            appData.readTransferDone = true;
            readCompleteEventData = (USB_HOST_CDC_EVENT_READ_COMPLETE_DATA *) (eventData);
            appData.readTransferResult = readCompleteEventData->result;
            break;

        case USB_HOST_CDC_EVENT_DEVICE_DETACHED:
            /* The device was detached */
            appData.deviceWasDetached = true;
            break;

        default:
    }
}
```

```

        break;
    }

    return(USB_HOST_CDC_EVENT_RESPONE_NONE);
}

void APP_Tasks(void)
{
    switch(appData.state)
    {
        /* The application states that enable the bus and wait for device attach
         * are not shown here for brevity */
        case APP_STATE_OPEN_DEVICE:

            /* In this state the application opens the attached device */
            appData.cdcHostHandle = USB_HOST_CDC_Open(appData.cdcObj);
            if(appData.cdcHostHandle != USB_HOST_CDC_HANDLE_INVALID)
            {
                /* The driver was opened successfully. Set the event handler
                 * and then go to the next state. */
                USB_HOST_CDC_EventHandlerSet(appData.cdcHostHandle, APP_USBHostCDCEventHandler,
(uintptr_t)0);
                appData.state = APP_STATE_SET_LINE_CODING;
            }
            break;

        default:
            break;
    }
}

```

Configuring the Library

Describes how to configure the USB CDC Host Library.

Macros

Name	Description
USB_HOST_CDC_ATTACH_LISTENERS_NUMBER	Defines the number of attach event listeners that can be registered with CDC Host Client Driver.
USB_HOST_CDC_INSTANCES_NUMBER	Specifies the number of CDC instances.

Description

The CDC Host Client Driver requires configuration constants to be specified in `system_config.h` file. These constants define the build time configuration (functionality and static resources) of the CDC Host Client Driver.

USB_HOST_CDC_ATTACH_LISTENERS_NUMBER Macro

Defines the number of attach event listeners that can be registered with CDC Host Client Driver.

File

`usb_host_cdc_config_template.h`

C

```
#define USB_HOST_CDC_ATTACH_LISTENERS_NUMBER
```

Description

USB Host CDC Attach Listeners Number

The USB CDC Host Client Driver provides attach notification to listeners who have registered with the client driver via the `USB_HOST_CDC_AttachEventHandlerSet()` function. The `USB_HOST_CDC_ATTACH_LISTENERS_NUMBER` configuration constant defines the maximum number of event handlers that can be set. This number should be set to equal the number of entities that interested in knowing when a CDC device is attached.

Remarks

None.

USB_HOST_CDC_INSTANCES_NUMBER Macro

Specifies the number of CDC instances.

File

[usb_host_cdc_config_template.h](#)

C

```
#define USB_HOST_CDC_INSTANCES_NUMBER
```

Description

USB Host CDC Maximum Number of Instances

This macro defines the number of instances of the CDC host Driver. For example, if the application needs to implement two instances of the CDC host Driver should be set to 2.

Remarks

None.

Building the Library

Describes the files to be included in the project while using the USB CDC Host Client Driver.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/usb.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
usb_host_cdc.h	This header file should be included in any .c file that accesses the CDC Host Client Driver API.
sub_host_cdc_acm.h	This header file should be included in any .c file that accesses the CDC Host Client Driver command request API.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/usb_host_cdc.c	This file implements the CDC Host Client Driver interface and should be included in the project if the CDC Host Client Driver operation is desired.
/src/dynamic/usb_host_cdc_acm.c	This file implements the CDC Host Client Driver command request functions and should be included if any class specific function must be called.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB CDC Host Library depends on the following modules:

- [USB Host Layer Library](#)

Library Interface**a) Client Access Functions**

	Name	Description
≡◊	USB_HOST_CDC_Open	This function opens the specified CDC device.
≡◊	USB_HOST_CDC_Close	This function closes the CDC device.
≡◊	USB_HOST_CDC_AttachEventHandlerSet	This function will set an attach event handler.
≡◊	USB_HOST_CDC_EventHandlerSet	Registers an event handler with the CDC Host Client Driver.
≡◊	USB_HOST_CDC_DeviceObjHandleGet	This function returns the Device Object Handle for this CDC device.

b) Data Transfer Functions

	Name	Description
≡◊	USB_HOST_CDC_Read	This function will read data from the attached device.
≡◊	USB_HOST_CDC_SerialStateNotificationGet	This function will request Serial State Notification from the attached device.
≡◊	USB_HOST_CDC_Write	This function will write data to the attached device.

c) CDC Class-specific Functions

	Name	Description
≡◊	USB_HOST_CDC_ACM_BreakSend	This function sends a request to the attached device to update its break duration.
≡◊	USB_HOST_CDC_ACM_ControlLineStateSet	This function sends a request to the attached device to set its Control Line State.
≡◊	USB_HOST_CDC_ACM_LineCodingGet	This function sends a request to the attached device to get its Line Coding.
≡◊	USB_HOST_CDC_ACM_LineCodingSet	This function sends a request to the attached device to set its Line Coding.

d) Data Types and Constants

	Name	Description
	USB_HOST_CDC_EVENT	Identifies the possible events that the CDC Class Driver can generate.
	USB_HOST_CDC_RESULT	USB Host CDC Client Driver Result enumeration.
	USB_HOST_CDC_TRANSFER_HANDLE	USB Host CDC Client Driver Transfer Handle
	USB_HOST_CDC_TRANSFER_HANDLE_INVALID	USB Host CDC Client Driver Invalid Transfer Handle Definition.
	USB_HOST_CDC_ATTACH_EVENT_HANDLER	USB Host CDC Client Driver Attach Event Handler Function Pointer Type.
	USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA	USB Host CDC Client Driver Command Event Data.
	USB_HOST_CDC_HANDLE	Defines the type of the CDC Host Client Driver Handle
	USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA	USB Host CDC Client Driver Command Event Data.
	USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA	USB Host CDC Client Driver Command Event Data.
	USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA	USB Host CDC Client Driver Command Event Data.
	USB_HOST_CDC_EVENT_HANDLER	USB Host CDC Client Driver Event Handler Function Pointer Type.
	USB_HOST_CDC_EVENT_READ_COMPLETE_DATA	USB Host CDC Client Driver Event Data.
	USB_HOST_CDC_EVENT_RESPONSE	Return type of the USB CDC Host Client Driver Event Handler.
	USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION RECEIVED DATA	USB Host CDC Client Driver Event Data.
	USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA	USB Host CDC Client Driver Event Data.
	USB_HOST_CDC_OBJ	Defines the type of the CDC Host Client Object.
	USB_HOST_CDC_REQUEST_HANDLE	USB Host CDC Client Driver Request Handle
	USB_HOST_CDC_INTERFACE	USB HOST CDC Client Driver Interface

	USB_HOST_CDC_REQUEST_HANDLE_INVALID	USB Host CDC Client Driver Invalid Request Handle
	USB_HOST_CDC_HANDLE_INVALID	Defines an Invalid CDC Client Driver Handle.

Description

a) Client Access Functions

USB_HOST_CDC_Open Function

This function opens the specified CDC device.

File

[usb_host_cdc.h](#)

C

```
USB_HOST_CDC_HANDLE USB_HOST_CDC_Open(USB_HOST_CDC_OBJ cdcDeviceObj);
```

Returns

Will return a valid handle if the device could be opened successfully, else will return [USB_HOST_CDC_HANDLE_INVALID](#). The function will return a valid handle if the device is ready to be opened.

Description

This function will open the specified CDC device. Once opened, the CDC device can be accessed via the handle which this function returns. The `cdcDeviceObj` parameter is the value returned in the [USB_HOST_CDC_ATTACH_EVENT_HANDLER](#) event handling function.

Remarks

None.

Preconditions

The client handle should be valid.

Example

Parameters

Parameters	Description
<code>cdcDeviceObj</code>	CDC device object handle returned in the USB_HOST_CDC_ATTACH_EVENT_HANDLER function.

Function

```
USB_HOST_CDC_HANDLE USB_HOST_CDC_Open
(
    USB_HOST_CDC_OBJ cdcDeviceObj
);
```

USB_HOST_CDC_Close Function

This function closes the CDC device.

File

[usb_host_cdc.h](#)

C

```
void USB_HOST_CDC_Close(USB_HOST_CDC_HANDLE cdcDeviceHandle);
```

Returns

None.

Description

This function will close the open CDC device. This closes the association between the application entity that opened the device and device. The driver handle becomes invalid.

Remarks

The device handle becomes invalid after calling this function.

Preconditions

None.

Example

Parameters

Parameters	Description
cdcDeviceHandle	handle to the CDC device obtained from the USB_HOST_CDC_Open() function.

Function

```
void USB_HOST_CDC_Close
(
    USB_HOST_CDC_HANDLE cdcDeviceHandle
);
```

[USB_HOST_CDC_AttachEventHandlerSet Function](#)

This function will set an attach event handler.

File

[usb_host_cdc.h](#)

C

```
USB_HOST_CDC_RESULT USB_HOST_CDC_AttachEventHandlerSet(USB_HOST_CDC_ATTACH_EVENT_HANDLER eventHandler,
uintptr_t context);
```

Returns

USB_HOST_CDC_RESULT_SUCCESS - if the attach event handler was registered successfully.

USB_HOST_CDC_RESULT_FAILURE - if the number of registered event handlers has exceeded [USB_HOST_CDC_ATTACH_LISTENERS_NUMBER](#).

Description

This function will set an attach event handler. The attach event handler will be called when a CDC device has been attached. The context will be returned in the event handler. This function should be called before the bus has been enabled.

Remarks

Function should be called before [USB_HOST_BusEnable\(\)](#) function is called.

Preconditions

None.

Example

Parameters

Parameters	Description
eventHandler	pointer to the attach event handler
context	an application defined context that will be returned in the event handler.

Function

```
USB_HOST_CDC_RESULT USB_HOST_CDC_AttachEventHandlerSet
(
    USB_HOST_CDC_ATTACH_EVENT_HANDLER eventHandler,
```

```
uintptr_t context
);
```

USB_HOST_CDC_EventHandlerSet Function

Registers an event handler with the CDC Host Client Driver.

File

[usb_host_cdc.h](#)

C

```
USB_HOST_CDC_RESULT USB_HOST_CDC_EventHandlerSet(USB_HOST_CDC_HANDLE handle, USB_HOST_CDC_EVENT_HANDLER
eventHandler, uintptr_t context);
```

Returns

USB_HOST_CDC_RESULT_SUCCESS - The operation was successful
 USB_HOST_CDC_RESULT_HANDLE_INVALID - The specified instance does not exist.
 USB_HOST_CDC_RESULT_FAILURE - An unknown failure occurred.

Description

This function registers a client specific CDC Host Client Driver event handler. The CDC Host Client Driver will call this function with relevant event and associated event data, in response to command requests and data transfers that have been scheduled by the client.

Remarks

None.

Preconditions

None.

Example

Parameters

Parameters	Description
handle	handle to the CDC Host Client Driver.
eventHandler	A pointer to event handler function. If NULL, then events will not be generated.
context	Application specific context that is returned in the event handler.

Function

```
USB_HOST_CDC_RESULT USB_HOST_CDC_EventHandlerSet
(
    USB_HOST_CDC_HANDLE handle,
    USB_HOST_CDC_EVENT_HANDLER eventHandler,
    uintptr_t context
);
```

USB_HOST_CDC_DeviceObjHandleGet Function

This function returns the Device Object Handle for this CDC device.

File

[usb_host_cdc.h](#)

C

```
USB_HOST_DEVICE_OBJ_HANDLE USB_HOST_CDC_DeviceObjHandleGet(USB_HOST_CDC_OBJ cdcDeviceObj);
```

Returns

Will return a valid device object handle if the device is still connected to the system. Will return an [USB_HOST_DEVICE_OBJ_HANDLE_INVALID](#) otherwise.

Description

This function returns the Device Object Handle for this CDC device. This returned Device Object Handle can be used by the application to perform

device level operations such as getting the string descriptors.

Remarks

None.

Preconditions

None.

Example

Parameters

Parameters	Description
cdcDeviceObj	CDC device object handle returned in the USB_HOST_CDC_ATTACH_EVENT_HANDLER function.

Function

```
USB_HOST_DEVICE_OBJ_HANDLE USB_HOST_CDC_DeviceObjHandleGet
(
    USB_HOST_CDC_OBJ cdcDeviceObj
);
```

b) Data Transfer Functions

USB_HOST_CDC_Read Function

This function will read data from the attached device.

File

[usb_host_cdc.h](#)

C

```
USB_HOST_CDC_RESULT USB_HOST_CDC_Read(USB_HOST_CDC_HANDLE handle, USB_HOST_CDC_TRANSFER_HANDLE * transferHandle, void * data, size_t size);
```

Returns

USB_HOST_CDC_RESULT_SUCCESS - The operation was successful.

USB_HOST_CDC_RESULT_DEVICE_UNKNOWN - The device that this request was targeted to does not exist in the system.

USB_HOST_CDC_RESULT_BUSY - The request could not be scheduled at this time. The client should try again.

USB_HOST_CDC_RESULT_INVALID_PARAMETER - An input parameter was NULL.

USB_HOST_CDC_RESULT_FAILURE - An unknown failure occurred.

USB_HOST_CDC_RESULT_HANDLE_INVALID - The client handle is not valid.

Description

This function will read data from the attached CDC device. The function will try to read size amount of bytes but will stop reading when the device terminates the USB transfer (sends a short packet or a ZLP). If the request was accepted, transferHandle will contain a valid transfer handle, else it will contain [USB_HOST_CDC_TRANSFER_HANDLE_INVALID](#). The completion of the request will be indicated by the [USB_HOST_CDC_EVENT_READ_COMPLETE](#) event. The transfer handle will be returned in the event.

Remarks

None.

Preconditions

The client handle should be valid.

Example

Parameters

Parameters	Description
handle	handle to the CDC device instance to which the request should be sent.

transferHandle	Pointer to USB_HOST_CDC_TRANSFER_HANDLE type of a variable. This will contain a valid transfer handle if the request was successful.
data	pointer to the buffer where the received data will be stored. The contents of the buffer will be valid only when the USB_HOST_CDC_EVENT_READ_COMPLETE event has occurred.
size	size of the data buffer. Only these many bytes or less will be read.

Function

```
USB_HOST_CDC_RESULT USB_HOST_CDC_Read
(
    USB_HOST_CDC_HANDLE handle,
    USB_HOST_CDC_TRANSFER_HANDLE * transferHandle,
    void * data,
    size_t size
);
```

[USB_HOST_CDC_SerialStateNotificationGet Function](#)

This function will request Serial State Notification from the attached device.

File

[usb_host_cdc.h](#)

C

```
USB_HOST_CDC_RESULT USB_HOST_CDC_SerialStateNotificationGet(USB_HOST_CDC_HANDLE handle,
    USB_HOST_CDC_TRANSFER_HANDLE * transferHandle, USB_CDC_SERIAL_STATE * serialState);
```

Returns

[USB_HOST_CDC_RESULT_SUCCESS](#) - The operation was successful.
[USB_HOST_CDC_RESULT_DEVICE_UNKNOWN](#) - The device that this request was targeted to does not exist in the system.
[USB_HOST_CDC_RESULT_BUSY](#) - The request could not be scheduled at this time. The client should try again.
[USB_HOST_CDC_RESULT_INVALID_PARAMETER](#) - An input parameter was NULL.
[USB_HOST_CDC_RESULT_FAILURE](#) - An unknown failure occurred.
[USB_HOST_CDC_RESULT_HANDLE_INVALID](#) - The client handle is not valid.

Description

This function will request Serial State Notification from the attached device. If the request was accepted, transferHandle will contain a valid transfer handle, else it will contain [USB_HOST_CDC_TRANSFER_HANDLE_INVALID](#). The completion of the request will be indicated by the [USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION_RECEIVED](#) event. The transfer handle will be returned in the event.

Remarks

None.

Preconditions

The client handle should be valid.

Example

Parameters

Parameters	Description
handle	handle to the CDC device instance to which the request should be sent.
transferHandle	Pointer to USB_HOST_CDC_TRANSFER_HANDLE type of a variable. This will contain a valid transfer handle if the request was successful.
serialState	Pointer to the serial state data structure where the received serial state will be stored.

Function

```
USB_HOST_CDC_RESULT USB_HOST_CDC_SerialStateNotificationGet
(
    USB_HOST_CDC_HANDLE handle,
    USB_HOST_CDC_TRANSFER_HANDLE * transferHandle,
```

```
USB_CDC_SERIAL_STATE * serialState
);
```

USB_HOST_CDC_Write Function

This function will write data to the attached device.

File

[usb_host_cdc.h](#)

C

```
USB_HOST_CDC_RESULT USB_HOST_CDC_Write(USB_HOST_CDC_HANDLE handle, USB_HOST_CDC_TRANSFER_HANDLE * transferHandle, void * data, size_t size);
```

Returns

USB_HOST_CDC_RESULT_SUCCESS - The operation was successful.
 USB_HOST_CDC_RESULT_DEVICE_UNKNOWN - The device that this request was targeted to does not exist in the system.
 USB_HOST_CDC_RESULT_BUSY - The request could not be scheduled at this time. The client should try again.
 USB_HOST_CDC_RESULT_INVALID_PARAMETER - An input parameter was NULL.
 USB_HOST_CDC_RESULT_FAILURE - An unknown failure occurred.
 USB_HOST_CDC_RESULT_HANDLE_INVALID - The client handle is not valid.

Description

This function will write data to the attached CDC device. The function will write size amount of bytes. If the request was accepted, transferHandle will contain a valid transfer handle, else it will contain [USB_HOST_CDC_TRANSFER_HANDLE_INVALID](#). The completion of the request will be indicated by the [USB_HOST_CDC_EVENT_WRITE_COMPLETE](#) event. The transfer handle will be returned in the event.

Remarks

None.

Preconditions

The client handle should be valid.

Example

Parameters

Parameters	Description
handle	handle to the CDC device instance to which the request should be sent.
transferHandle	Pointer to USB_HOST_CDC_TRANSFER_HANDLE type of a variable. This will contain a valid transfer handle if the request was successful.
data	pointer to the buffer containing the data to be written. The contents of the buffer should not be changed till the USB_HOST_CDC_EVENT_WRITE_COMPLETE event has occurred.
size	Number of bytes to write.

Function

```
USB_HOST_CDC_RESULT USB_HOST_CDC_Write
(
    USB_HOST_CDC_HANDLE handle,
    USB_HOST_CDC_TRANSFER_HANDLE * transferHandle,
    void * data,
    size_t size
);
```

c) CDC Class-specific Functions

USB_HOST_CDC_ACm_BreakSend Function

This function sends a request to the attached device to update its break duration.

File

[usb_host_cdc_acm.h](#)

C

```
USB_HOST_CDC_RESULT USB_HOST_CDC_ACm_BreakSend(USB_HOST_CDC_HANDLE handle, USB_HOST_CDC_REQUEST_HANDLE * requestHandle, uint16_t breakDuration);
```

Returns

USB_HOST_CDC_RESULT_SUCCESS - The operation was successful.

USB_HOST_CDC_RESULT_DEVICE_UNKNOWN - The device that this request was targeted to does not exist in the system.

USB_HOST_CDC_RESULT_BUSY - The request could not be scheduled at this time. The client should try again.

USB_HOST_CDC_RESULT_INVALID_PARAMETER - An input parameter was NULL.

USB_HOST_CDC_RESULT_FAILURE - An unknown failure occurred.

USB_HOST_CDC_RESULT_HANDLE_INVALID - The client handle is not valid.

Description

This function sends a request to the attached to update its break duration. The function schedules a SEND BREAK control transfer. If successful, the transferHandle parameter will contain a valid request handle, else it will contain [USB_HOST_CDC_REQUEST_HANDLE_INVALID](#). When completed, the CDC client driver will generate a [USB_HOST_CDC_EVENT_ACm_SEND_BREAK_COMPLETE](#) event.

Remarks

None.

Preconditions

The client handle should be valid.

Example

Parameters

Parameters	Description
handle	handle to the CDC device instance to which the request should be sent.
requestHandle	Pointer to USB_HOST_CDC_REQUEST_HANDLE type of a variable. This will contain a valid transfer handle if the request was successful.
breakDuration	Break duration.

Function

```
USB_HOST_CDC_RESULT USB_HOST_CDC_ACm_BreakSend
(
    USB_HOST_CDC_HANDLE handle,
    USB_HOST_CDC_REQUEST_HANDLE * requestHandle,
    uint16_t breakDuration
);
```

USB_HOST_CDC_ACm_ControlLineStateSet Function

This function sends a request to the attached device to set its Control Line State.

File

[usb_host_cdc_acm.h](#)

C

```
USB_HOST_CDC_RESULT USB_HOST_CDC_ACm_ControlLineStateSet(USB_HOST_CDC_HANDLE handle,
USB_HOST_CDC_REQUEST_HANDLE * requestHandle, USB_CDC_CONTROL_LINE_STATE * controlLineState);
```

Returns

USB_HOST_CDC_RESULT_SUCCESS - The operation was successful.
 USB_HOST_CDC_RESULT_DEVICE_UNKNOWN - The device that this request was targeted to does not exist in the system.
 USB_HOST_CDC_RESULT_BUSY - The request could not be scheduled at this time. The client should try again.
 USB_HOST_CDC_RESULT_INVALID_PARAMETER - An input parameter was NULL.
 USB_HOST_CDC_RESULT_FAILURE - An unknown failure occurred.
 USB_HOST_CDC_RESULT_HANDLE_INVALID - The client handle is not valid.

Description

This function sends a request to the attached to set its Control Line State. The function schedules a SET CONTROL LINE STATE control transfer. If successful, the requestHandle parameter will contain a valid request handle, else it will contain [USB_HOST_CDC_REQUEST_HANDLE_INVALID](#). When completed, the CDC client driver will generate a USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE event.

Remarks

None.

Preconditions

The client handle should be valid.

Example

Parameters

Parameters	Description
handle	handle to the CDC device instance to which the request should be sent.
requestHandle	Pointer to USB_HOST_CDC_REQUEST_HANDLE type of a variable. This will contain a valid transfer handle if the request was successful.
controlLineState	Pointer to the control line state data structure.

Function

```

USB\_HOST\_CDC\_RESULT USB_HOST_CDC_ACM_ControlLineStateSet
(
  USB\_HOST\_CDC\_HANDLE handle,
  USB\_HOST\_CDC\_REQUEST\_HANDLE * requestHandle,
  USB\_CDC\_HOST\_CONTROL\_LINE\_STATE * controlLineState
);

```

[USB_HOST_CDC_ACM_LineCodingGet Function](#)

This function sends a request to the attached device to get its Line Coding.

File

[usb_host_cdc_acm.h](#)

C

```

USB\_HOST\_CDC\_RESULT USB\_HOST\_CDC\_ACM\_LineCodingGet(USB\_HOST\_CDC\_HANDLE handle, USB\_HOST\_CDC\_REQUEST\_HANDLE
* requestHandle, USB\_CDC\_LINE\_CODING * lineCoding);

```

Returns

USB_HOST_CDC_RESULT_SUCCESS - The operation was successful.
 USB_HOST_CDC_RESULT_DEVICE_UNKNOWN - The device that this request was targeted to does not exist in the system.
 USB_HOST_CDC_RESULT_BUSY - The request could not be scheduled at this time. The client should try again.
 USB_HOST_CDC_RESULT_INVALID_PARAMETER - An input parameter was NULL.
 USB_HOST_CDC_RESULT_FAILURE - An unknown failure occurred.
 USB_HOST_CDC_RESULT_HANDLE_INVALID - The client handle is not valid.

Description

This function sends a request to the attached device to get its line coding. The function schedules a GET LINE CODING control transfer. If

successful, the requestHandle parameter will contain a valid request handle, else it will contain [USB_HOST_CDC_REQUEST_HANDLE_INVALID](#). When completed, the CDC client driver will generate a [USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE](#) event.

Remarks

None.

Preconditions

The client handle should be valid.

Example

Parameters

Parameters	Description
handle	handle to the CDC device instance to which the request should be sent.
requestHandle	Pointer to USB_HOST_CDC_REQUEST_HANDLE type of a variable. This will contain a valid transfer handle if the request was successful.
lineCoding	Pointer to the line coding data structure where the obtained line coding will be stored. The contents of this data structure will be valid only when the USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE event has been generated.

Function

```
USB_HOST_CDC_RESULT USB_HOST_CDC_ACM_LineCodingGet
(
    USB_HOST_CDC_HANDLE handle,
    USB_HOST_CDC_REQUEST_HANDLE * requestHandle,
    USB_CDC_LINE_CODING * lineCoding
);
```

USB_HOST_CDC_ACM_LineCodingSet Function

This function sends a request to the attached device to set its Line Coding.

File

[usb_host_cdc_acm.h](#)

C

```
USB_HOST_CDC_RESULT USB_HOST_CDC_ACM_LineCodingSet(USB_HOST_CDC_HANDLE handle, USB_HOST_CDC_REQUEST_HANDLE
* requestHandle, USB_CDC_LINE_CODING * lineCoding);
```

Returns

[USB_HOST_CDC_RESULT_SUCCESS](#) - The operation was successful.
[USB_HOST_CDC_RESULT_DEVICE_UNKNOWN](#) - The device that this request was targeted to does not exist in the system.
[USB_HOST_CDC_RESULT_BUSY](#) - The request could not be scheduled at this time. The client should try again.
[USB_HOST_CDC_RESULT_INVALID_PARAMETER](#) - An input parameter was NULL.
[USB_HOST_CDC_RESULT_FAILURE](#) - An unknown failure occurred.
[USB_HOST_CDC_RESULT_HANDLE_INVALID](#) - The client handle is not valid.

Description

This function sends a request to the attached device to set its line coding. The function schedules a SET LINE CODING control transfer. If successful, the requestHandle parameter will contain a valid request handle, else it will contain [USB_HOST_CDC_REQUEST_HANDLE_INVALID](#). When completed, the CDC client driver will generate a [USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE](#) event.

Remarks

None.

Preconditions

The client handle should be valid.

Example**Parameters**

Parameters	Description
handle	handle to the CDC device instance to which the request should be sent.
requestHandle	Pointer to USB_HOST_CDC_REQUEST_HANDLE type of a variable. This will contain a valid transfer handle if the request was successful.
lineCoding	Pointer to the line coding data structure containing the line coding to be set. The contents of this data structure should not be changed until the USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE event has been generated.

Function

```
USB_HOST_CDC_RESULT USB_HOST_CDC_ACM_LineCodingSet
(
    USB_HOST_CDC_HANDLE handle,
    USB_HOST_CDC_REQUEST_HANDLE * requestHandle,
    USB_CDC_LINE_CODING * lineCoding
);
```

d) Data Types and Constants**USB_HOST_CDC_EVENT Enumeration**

Identifies the possible events that the CDC Class Driver can generate.

File

[usb_host_cdc.h](#)

C

```
typedef enum {
    USB_HOST_CDC_EVENT_READ_COMPLETE,
    USB_HOST_CDC_EVENT_WRITE_COMPLETE,
    USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE,
    USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE,
    USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE,
    USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE,
    USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION_RECEIVED,
    USB_HOST_CDC_EVENT_DEVICE_DETACHED
} USB_HOST_CDC_EVENT;
```

Members

Members	Description
USB_HOST_CDC_EVENT_READ_COMPLETE	This event occurs when a CDC Client Driver Read operation has completed i.e when the data has been received from the connected CDC device. This event is generated after the application calls the USB_HOST_CDC_Read function. The eventData parameter in the event call back function will be of a pointer to a USB_HOST_CDC_EVENT_READ_COMPLETE_DATA structure. This contains details about the transfer handle associated with this read request, the amount of data read and the termination status of the read request.
USB_HOST_CDC_EVENT_WRITE_COMPLETE	This event occurs when a CDC Client Driver Write operation has completed i.e when the data has been written to the connected CDC device. This event is generated after the application calls the USB_HOST_CDC_Write function. The eventData parameter in the event call back function will be a pointer to a USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA structure. This contains details about the transfer handle associated with this write request, the amount of data written and the termination status of the write request.

USB_HOST_CDC_EVENT_ACm_SEND_BREAK_COMPLETE	This event occurs when a CDC Client Driver Send Break request has completed. This event is generated after the application calls the USB_HOST_CDC_ACm_BreakSend function and the device has either acknowledged or stalled the request. The eventData parameter in the event call back function will be of a pointer to a USB_HOST_CDC_EVENT_ACm_SET_CONTROL_LINE_STATE_COMPLETE_DATA structure. This contains details about the transfer handle associated with this request, the amount of data sent and the termination status of the set request.
USB_HOST_CDC_EVENT_ACm_SET_CONTROL_LINE_STATE_COMPLETE	This event occurs when a CDC Client Driver Set Control Line State request has completed. This event is generated after the application calls the USB_HOST_CDC_ACm_ControlLineStateSet function and the device has either acknowledged or stalled the request. The eventData parameter in the event call back function will be of a pointer to a USB_HOST_CDC_EVENT_ACm_SET_CONTROL_LINE_STATE_COMPLETE_DATA structure. This contains details about the transfer handle associated with this request, the amount of data sent and the termination status of the set request.
USB_HOST_CDC_EVENT_ACm_SET_LINE_CODING_COMPLETE	This event occurs when a CDC Client Driver Set Line Coding request has completed. This event is generated after the application calls the USB_HOST_CDC_ACm_LineCodingSet function and the device either acknowledged or stalled the request. The eventData parameter in the event call back function will be of a pointer to a USB_HOST_CDC_EVENT_ACm_SET_LINE_CODING_COMPLETE_DATA structure. This contains details about the transfer handle associated with this request, the amount of data sent and the termination status of the set request.
USB_HOST_CDC_EVENT_ACm_GET_LINE_CODING_COMPLETE	This event occurs when a CDC Client Driver Get Line Coding request has completed. This event is generated after the application calls the USB_HOST_CDC_ACm_LineCodingGet function and the device sends the line coding to the host. The eventData parameter in the event call back function will be of a pointer to a USB_HOST_CDC_EVENT_ACm_GET_LINE_CODING_COMPLETE_DATA structure. This contains details about the transfer handle associated with this request, the amount of data received and the termination status of the get request.
USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION_RECEIVED	This event occurs when a CDC Client Driver Serial State Notification Get operation has completed. This event is generated after the application calls the USB_HOST_CDC_SerialStateNotificationGet and the device sends a serial state notification to the host. The eventData parameter in the event call back function will be of a pointer to a USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION_RECEIVED_DATA structure. This contains details about the transfer handle associated with this request, the amount of data received and the termination status of the get request.
USB_HOST_CDC_EVENT_DEVICE_DETACHED	This event occurs when the device that this client was connected to has <ul style="list-style-type: none"> • been detached. The client should close the CDC instance. There is no • event data associated with this event

Description

CDC Class Driver Events

This enumeration identifies the possible events that the CDC Class Driver can generate. The application should register an event handler using the [USB_HOST_CDC_EventHandlerSet](#) function to receive CDC Class Driver events.

An event may have data associated with it. Events that are generated due to a transfer of data between the host and device are accompanied by data structures that provide the status of the transfer termination. For example, the [USB_HOST_CDC_EVENT_ACm_SET_LINE_CODING_COMPLETE](#) event is accompanied by a pointer to a [USB_HOST_CDC_EVENT_ACm_SET_LINE_CODING_COMPLETE_DATA](#) data structure. The transferStatus member of this data structure indicates the success or failure of the transfer. A transfer may fail due to device not responding on the bus, if the device stalls any stages of the

transfer or due to NAK timeouts. The event description provides details on the nature of the event and the data that is associated with the event.

Remarks

None.

USB_HOST_CDC_RESULT Enumeration

USB Host CDC Client Driver Result enumeration.

File

[usb_host_cdc.h](#)

C

```
typedef enum {
    USB_HOST_CDC_RESULT_FAILURE,
    USB_HOST_CDC_RESULT_BUSY,
    USB_HOST_CDC_RESULT_REQUEST_STALLED,
    USB_HOST_CDC_RESULT_INVALID_PARAMETER,
    USB_HOST_CDC_RESULT_DEVICE_UNKNOWN,
    USB_HOST_CDC_RESULT_ABORTED,
    USB_HOST_CDC_RESULT_HANDLE_INVALID,
    USB_HOST_CDC_RESULT_SUCCESS
} USB_HOST_CDC_RESULT;
```

Members

Members	Description
USB_HOST_CDC_RESULT_FAILURE	An unknown failure has occurred
USB_HOST_CDC_RESULT_BUSY	The transfer or request could not be scheduled because internal • queues are full. The request or transfer should be retried
USB_HOST_CDC_RESULT_REQUEST_STALLED	The request was stalled
USB_HOST_CDC_RESULT_INVALID_PARAMETER	A required parameter was invalid
USB_HOST_CDC_RESULT_DEVICE_UNKNOWN	The associated device does not exist in the system.
USB_HOST_CDC_RESULT_ABORTED	The transfer or requested was aborted
USB_HOST_CDC_RESULT_HANDLE_INVALID	The specified handle is not valid
USB_HOST_CDC_RESULT_SUCCESS	The operation was successful

Description

USB Host CDC Client Driver Result.

This enumeration lists the possible results the CDC client driver uses. Only some results are applicable to some functions and events. Refer to the event and function documentation for more details.

Remarks

None.

USB_HOST_CDC_TRANSFER_HANDLE Type

USB Host CDC Client Driver Transfer Handle

File

[usb_host_cdc.h](#)

C

```
typedef uintptr_t USB_HOST_CDC_TRANSFER_HANDLE;
```

Description

USB Host CDC Client Driver Transfer Handle

This is returned by the CDC Client driver data transfer routines and should be used by the application to track the transfer especially in cases where transfers are queued.

Remarks

None.

USB_HOST_CDC_TRANSFER_HANDLE_INVALID Macro

USB Host CDC Client Driver Invalid Transfer Handle Definition.

File

[usb_host_cdc.h](#)

C

```
#define USB_HOST_CDC_TRANSFER_HANDLE_INVALID ((USB_HOST_CDC_TRANSFER_HANDLE)(-1))
```

Description

USB Host CDC Client Driver Invalid Transfer Handle Definition

This definition defines a USB Host CDC Client Driver Invalid Transfer Handle. A Invalid Transfer Handle is returned by the CDC Client Driver data transfer routines when the request was not successful.

Remarks

None.

USB_HOST_CDC_ATTACH_EVENT_HANDLER Type

USB Host CDC Client Driver Attach Event Handler Function Pointer Type.

File

[usb_host_cdc.h](#)

C

```
typedef void (* USB_HOST_CDC_ATTACH_EVENT_HANDLER)(USB_HOST_CDC_OBJ cdcObjHandle, uintptr_t context);
```

Description

USB Host CDC Client Driver Attach Event Handler Function Pointer Type.

This data type defines the required function signature of the USB Host CDC Client Driver attach event handling callback function. The application must register a pointer to a CDC Client Driver attach events handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive attach event call backs from the CDC Client Driver. The client driver will invoke this function with event relevant parameters. The description of the event handler function parameters is given here.

cdcObjHandle - Handle of the client to which this event is directed.

context - Value identifying the context of the application that was registered along with the event handling function.

Remarks

None.

USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA Structure

USB Host CDC Client Driver Command Event Data.

File

[usb_host_cdc.h](#)

C

```
typedef struct {
    USB_HOST_CDC_REQUEST_HANDLE requestHandle;
    USB_HOST_CDC_RESULT result;
    size_t length;
} USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_CDC_REQUEST_HANDLE requestHandle;	Request handle of this request

USB_HOST_CDC_RESULT result;	Termination status
size_t length;	Size of the data transferred in the request

Description

USB Host CDC Client Driver Command Event Data.

This data type defines the data structure returned by the driver along with the following events:

[USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA](#),

Remarks

None.

USB_HOST_CDC_HANDLE Type

Defines the type of the CDC Host Client Driver Handle

File

[usb_host_cdc.h](#)

C

```
typedef uintptr_t USB_HOST_CDC_HANDLE;
```

Description

USB Host CDC Client Driver Handle

This type defines the type of the handle returned by [USB_HOST_CDC_Open\(\)](#) function. This application uses this handle to specify the instance of the CDC client driver being accessed while calling a CDC Client driver function.

Remarks

None.

USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA Structure

USB Host CDC Client Driver Command Event Data.

File

[usb_host_cdc.h](#)

C

```
typedef struct {
    USB_HOST_CDC_REQUEST_HANDLE requestHandle;
    USB_HOST_CDC_RESULT result;
    size_t length;
} USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_CDC_REQUEST_HANDLE requestHandle;	Request handle of this request
USB_HOST_CDC_RESULT result;	Termination status
size_t length;	Size of the data transferred in the request

Description

USB Host CDC Client Driver Command Event Data.

This data type defines the data structure returned by the driver along with the following events:

[USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA](#),

Remarks

None.

USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA Structure

USB Host CDC Client Driver Command Event Data.

File

[usb_host_cdc.h](#)

C

```
typedef struct {
    USB_HOST_CDC_REQUEST_HANDLE requestHandle;
    USB_HOST_CDC_RESULT result;
    size_t length;
} USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_CDC_REQUEST_HANDLE requestHandle;	Request handle of this request
USB_HOST_CDC_RESULT result;	Termination status
size_t length;	Size of the data transferred in the request

Description

USB Host CDC Client Driver Command Event Data.

This data type defines the data structure returned by the driver along with the following events:

[USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA](#),

Remarks

None.

USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA Structure

USB Host CDC Client Driver Command Event Data.

File

[usb_host_cdc.h](#)

C

```
typedef struct {
    USB_HOST_CDC_REQUEST_HANDLE requestHandle;
    USB_HOST_CDC_RESULT result;
    size_t length;
} USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA,
USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_CDC_REQUEST_HANDLE requestHandle;	Request handle of this request
USB_HOST_CDC_RESULT result;	Termination status
size_t length;	Size of the data transferred in the request

Description

USB Host CDC Client Driver Command Event Data.

This data type defines the data structure returned by the driver along with the following events:

[USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA](#),
[USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA](#),

Remarks

None.

USB_HOST_CDC_EVENT_HANDLER Type

USB Host CDC Client Driver Event Handler Function Pointer Type.

File

[usb_host_cdc.h](#)

C

```
typedef USB_HOST_CDC_EVENT_RESPONSE (* USB_HOST_CDC_EVENT_HANDLER)(USB_HOST_CDC_HANDLE cdcHandle,  
USB_HOST_CDC_EVENT event, void * eventData, uintptr_t context);
```

Description

USB Host CDC Client Driver Event Handler Function Pointer Type.

This data type defines the required function signature of the USB Host CDC Client Driver event handling callback function. The application must register a pointer to a CDC Client Driver events handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event call backs from the CDC Client Driver. The class driver will invoke this function with event relevant parameters. The description of the event handler function parameters is given here.

handle - Handle of the client to which this event is directed.

event - Type of event generated.

eventData - This parameter should be type casted to a event specific pointer type based on the event that has occurred. Refer to the [USB_HOST_CDC_EVENT](#) enumeration description for more details.

context - Value identifying the context of the application that was registered along with the event handling function.

Remarks

None.

USB_HOST_CDC_EVENT_READ_COMPLETE_DATA Structure

USB Host CDC Client Driver Event Data.

File

[usb_host_cdc.h](#)

C

```
typedef struct {  
    USB_HOST_CDC_TRANSFER_HANDLE transferHandle;  
    USB_HOST_CDC_RESULT result;  
    size_t length;  
} USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION_RECEIVED_DATA, USB_HOST_CDC_EVENT_READ_COMPLETE_DATA,  
USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_CDC_TRANSFER_HANDLE transferHandle;	Transfer handle of this transfer
USB_HOST_CDC_RESULT result;	Termination transfer status
size_t length;	Size of the data transferred in the request

Description

USB Host CDC Client Driver Event Data.

This data type defines the data structure returned by the driver along with the following events:
[USB_HOST_CDC_EVENT_READ_COMPLETE_DATA](#), [USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA](#),

Remarks

None.

USB_HOST_CDC_EVENT_RESPONSE Enumeration

Return type of the USB CDC Host Client Driver Event Handler.

File

[usb_host_cdc.h](#)

C

```
typedef enum {
    USB_HOST_CDC_EVENT_RESPONSE_NONE
} USB_HOST_CDC_EVENT_RESPONSE;
```

Members

Members	Description
USB_HOST_CDC_EVENT_RESPONSE_NONE	This means no response is required

Description

USB Host CDC Event Handler Return Type

This enumeration list the possible return values of the USB CDC Host Client Driver Event Handler.

Remarks

None.

USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION RECEIVED DATA Structure

USB Host CDC Client Driver Event Data.

File

[usb_host_cdc.h](#)

C

```
typedef struct {
    USB_HOST_CDC_TRANSFER_HANDLE transferHandle;
    USB_HOST_CDC_RESULT result;
    size_t length;
} USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION_RECEIVED_DATA, USB_HOST_CDC_EVENT_READ_COMPLETE_DATA,
USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_CDC_TRANSFER_HANDLE transferHandle;	Transfer handle of this transfer
USB_HOST_CDC_RESULT result;	Termination transfer status
size_t length;	Size of the data transferred in the request

Description

USB Host CDC Client Driver Event Data.

This data type defines the data structure returned by the driver along with the following events:

[USB_HOST_CDC_EVENT_READ_COMPLETE_DATA](#), [USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA](#),

Remarks

None.

USB_HOST_CDC_EVENT_WRITE_COMPLETE DATA Structure

USB Host CDC Client Driver Event Data.

File

[usb_host_cdc.h](#)

C

```
typedef struct {
    USB_HOST_CDC_TRANSFER_HANDLE transferHandle;
    USB_HOST_CDC_RESULT result;
    size_t length;
} USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION_RECEIVED_DATA, USB_HOST_CDC_EVENT_READ_COMPLETE_DATA,
USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA;
```

Members

Members	Description
USB_HOST_CDC_TRANSFER_HANDLE transferHandle;	Transfer handle of this transfer
USB_HOST_CDC_RESULT result;	Termination transfer status
size_t length;	Size of the data transferred in the request

Description

USB Host CDC Client Driver Event Data.

This data type defines the data structure returned by the driver along with the following events:

[USB_HOST_CDC_EVENT_READ_COMPLETE_DATA](#), [USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA](#),

Remarks

None.

USB_HOST_CDC_OBJ Type

Defines the type of the CDC Host Client Object.

File

[usb_host_cdc.h](#)

C

```
typedef uintptr_t USB_HOST_CDC_OBJ;
```

Description

USB Host CDC Object

This type defines the type of the CDC Host Client Object. This type is returned by the Attach Event Handler and is used by the application to open the attached CDC Device.

Remarks

None.

USB_HOST_CDC_REQUEST_HANDLE Type

USB Host CDC Client Driver Request Handle

File

[usb_host_cdc.h](#)

C

```
typedef uintptr_t USB_HOST_CDC_REQUEST_HANDLE;
```

Description

USB Host CDC Client Driver Request Handle

This is returned by the CDC Client driver command routines and should be used by the application to track the command especially in cases where transfers are queued.

Remarks

None.

USB_HOST_CDC_INTERFACE Macro

USB HOST CDC Client Driver Interface

File

[usb_host_cdc.h](#)

C

```
#define USB_HOST_CDC_INTERFACE
```

Description

USB HOST CDC Client Driver Interface

This macro should be used by the application in TPL table while adding support for the USB CDC Host Client Driver.

Remarks

None.

USB_HOST_CDC_REQUEST_HANDLE_INVALID Macro

USB Host CDC Client Driver Invalid Request Handle

File

[usb_host_cdc.h](#)

C

```
#define USB_HOST_CDC_REQUEST_HANDLE_INVALID ((USB_HOST_CDC_REQUEST_HANDLE)(-1))
```

Description

USB Host CDC Client Driver Invalid Request Handle

This is returned by the CDC Client driver command routines when the request could not be scheduled.

Remarks

None.

USB_HOST_CDC_HANDLE_INVALID Macro

Defines an Invalid CDC Client Driver Handle.

File

[usb_host_cdc.h](#)

C

```
#define USB_HOST_CDC_HANDLE_INVALID ((USB_HOST_CDC_HANDLE)(-1))
```

Description

USB Host CDC Client Driver Invalid Handle

This type defines an Invalid CDC Client Driver Handle. The [USB_HOST_CDC_Open\(\)](#) function returns an invalid handle when it fails to open the specified CDC device instance.

Remarks

None.

Files**Files**

Name	Description
usb_host_cdc.h	USB Host CDC Client Driver Interface Header
usb_host_cdc_acm.h	USB Host CDC Client Driver Interface Header
usb_host_cdc_config_template.h	USB host CDC Class configuration definitions template

Description

usb_host_cdc.h

USB Host CDC Client Driver Interface Header

Enumerations

	Name	Description
	USB_HOST_CDC_EVENT	Identifies the possible events that the CDC Class Driver can generate.
	USB_HOST_CDC_EVENT_RESPONSE	Return type of the USB CDC Host Client Driver Event Handler.
	USB_HOST_CDC_RESULT	USB Host CDC Client Driver Result enumeration.

Functions

	Name	Description
≡◊	USB_HOST_CDC_AttachEventHandlerSet	This function will set an attach event handler.
≡◊	USB_HOST_CDC_Close	This function closes the CDC device.
≡◊	USB_HOST_CDC_DeviceObjHandleGet	This function returns the Device Object Handle for this CDC device.
≡◊	USB_HOST_CDC_EventHandlerSet	Registers an event handler with the CDC Host Client Driver.
≡◊	USB_HOST_CDC_Open	This function opens the specified CDC device.
≡◊	USB_HOST_CDC_Read	This function will read data from the attached device.
≡◊	USB_HOST_CDC_SerialStateNotificationGet	This function will request Serial State Notification from the attached device.
≡◊	USB_HOST_CDC_Write	This function will write data to the attached device.

Macros

	Name	Description
	USB_HOST_CDC_HANDLE_INVALID	Defines an Invalid CDC Client Driver Handle.
	USB_HOST_CDC_INTERFACE	USB HOST CDC Client Driver Interface
	USB_HOST_CDC_REQUEST_HANDLE_INVALID	USB Host CDC Client Driver Invalid Request Handle
	USB_HOST_CDC_TRANSFER_HANDLE_INVALID	USB Host CDC Client Driver Invalid Transfer Handle Definition.

Structures

	Name	Description
	USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA	USB Host CDC Client Driver Command Event Data.
	USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA	USB Host CDC Client Driver Command Event Data.
	USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE_DATA	USB Host CDC Client Driver Command Event Data.
	USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA	USB Host CDC Client Driver Command Event Data.
	USB_HOST_CDC_EVENT_READ_COMPLETE_DATA	USB Host CDC Client Driver Event Data.
	USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION RECEIVED DATA	USB Host CDC Client Driver Event Data.
	USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA	USB Host CDC Client Driver Event Data.

Types

	Name	Description
	USB_HOST_CDC_ATTACH_EVENT_HANDLER	USB Host CDC Client Driver Attach Event Handler Function Pointer Type.
	USB_HOST_CDC_EVENT_HANDLER	USB Host CDC Client Driver Event Handler Function Pointer Type.
	USB_HOST_CDC_HANDLE	Defines the type of the CDC Host Client Driver Handle
	USB_HOST_CDC_OBJ	Defines the type of the CDC Host Client Object.
	USB_HOST_CDC_REQUEST_HANDLE	USB Host CDC Client Driver Request Handle
	USB_HOST_CDC_TRANSFER_HANDLE	USB Host CDC Client Driver Transfer Handle

Description

USB Host CDC Client Driver Interface Definition

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the USB Host CDC

Client Driver.

File Name

`usb_host_cdc.h`

Company

Microchip Technology Inc.

usb_host_cdc_acm.h

USB Host CDC Client Driver Interface Header

Functions

	Name	Description
≡◊	<code>USB_HOST_CDC_ACM_BreakSend</code>	This function sends a request to the attached device to update its break duration.
≡◊	<code>USB_HOST_CDC_ACM_ControlLineStateSet</code>	This function sends a request to the attached device to set its Control Line State.
≡◊	<code>USB_HOST_CDC_ACM_LineCodingGet</code>	This function sends a request to the attached device to get its Line Coding.
≡◊	<code>USB_HOST_CDC_ACM_LineCodingSet</code>	This function sends a request to the attached device to set its Line Coding.

Description

USB Host CDC Client Driver Interface Definition

This header file contains the CDC ACM specific function prototypes and definitions of the data types and constants that make up the interface to the USB Host CDC Client Driver.

File Name

`usb_host_cdc_acm.h`

Company

Microchip Technology Inc.

usb_host_cdc_config_template.h

USB host CDC Class configuration definitions template

Macros

	Name	Description
	<code>USB_HOST_CDC_ATTACH_LISTENERS_NUMBER</code>	Defines the number of attach event listeners that can be registered with CDC Host Client Driver.
	<code>USB_HOST_CDC_INSTANCES_NUMBER</code>	Specifies the number of CDC instances.

Description

USB Host CDC Class Configuration Definitions

This file contains configurations macros needed to configure the CDC host Driver. This file is a template file only. It should not be included by the application. The configuration macros defined in the file should be defined in the configuration specific system_config.h.

File Name

`usb_host_cdc_config_template.h`

Company

Microchip Technology Inc.

USB HID Host Mouse Driver Library

This section describes the USB HID Host Mouse Driver Library.

Introduction

Introduces the MPLAB Harmony USB HID Host Mouse Driver Library.

Description

The HID Host Mouse Driver in the MPLAB Harmony USB Host Stack allows USB Host Applications to support and interact with USB Mouse devices. The USB HID Host Mouse Driver has the following features:

- Supports USB HID Mouse devices
- Supports HID device matching at both device descriptor and interface descriptor level
- Supports both Boot and Non Boot interface USB Mouse devices
- Performs parsing of Mouse Report descriptor by using USB HID Host client driver APIs
- Supports detection of Mouse X, Y, and Z movements, as well as button click events

Using the Library

This topic describes the basic architecture of the USB HID Host Mouse Driver Library and provides information and examples on its use.

Library Overview

The USB HID Host Mouse Driver can be grouped functionally as shown in the following table.

Library Interface Section	Description
Mouse Access Functions	These functions allow application to register event handlers with the mouse driver. These functions are implemented in <code>usb_host_hid_mouse.c</code> .

Abstraction Model

Describes the Abstraction Model of the USB HID Host Mouse Driver Library.

Description

The USB HID Host Mouse Driver interacts with the USB Host HID Client Driver to control the attached HID device. The USB Host Layer attaches the USB HID Host Client Driver to the HID device when it meets the matching criteria specified in the USB Host TPL table.

The USB HID Host Client driver notifies the mouse driver of device attach and detach information and with report receive events with relevant event data. On a report receive event, the USB Host HID Mouse Driver obtains all of the field information present in the Report Descriptor of the mouse device and uses that field information and the INTERRUPT IN data received to understand mouse parameter values.

How the Library Works

Describes how the library works and how it should be used.

Description

The USB HID Host Mouse Driver provides the user application with an easy to use interface to the attached HID device. The USB Host Layer initializes the USB HID Host Client Driver when a device is attached. This process does not require application intervention. The following sections describe the steps and methods required for the user application to interact with the attached mouse devices through the USB Host HID Mouse Driver.

HID Device TPL Table Configuration

Provides information on configuring the TPL table for HID devices.

Description

The Host Layer attaches the USB HID Host Client Driver to a device when the device class, subclass, protocol in the device descriptor or when the class, subclass and protocol fields in the Interface descriptor matches the entry in the TPL table. When specifying the entry for the HID device along with the Usage driver, the driver interface must be set to `USB_HOST_HID_INTERFACE` and the usage driver interface must be set to `usageDriverInterface`. `usageDriverInterface` must be properly initialized to capture the Mouse driver APIs. This will attach the USB HID Host Mouse Driver to the device when the USB Host HID Client Driver is attached. The following code shows possible TPL table options for matching HID Devices.

Example:

```
/* This code shows an example of TPL table entries for supporting HID mouse devices.
 * Note that the driver interface is set to USB_HOST_HID_INTERFACE. This
 * will load the HID Host Client Driver when there is TPL match. Usage driver
 * interface is initialized with appropriate function pointer for Mouse driver.
 * This facilitates subsequent loading of Mouse driver post HID client driver.
 */
```

```

USB_HOST_HID_USAGE_DRIVER_INTERFACE usageDriverInterface =
{
    .initialize = NULL,
    .deinitialize = NULL,
    .usageDriverEventHandler = _USB_HOST_HID_MOUSE_EventHandler,
    .usageDriverTask = _USB_HOST_HID_MOUSE_Task
};

USB_HOST_HID_USAGE_DRIVER_TABLE_ENTRY usageDriverTableEntry[1] =
{
    {
        .usage = USB_HID_USAGE_MOUSE,
        .initializeData = NULL,
        .interface = &usageDriverInterface
    }
};
const USB_HOST_TPL_ENTRY USBTPLList[1] =
{
    /* This entry looks for any HID Mouse device */
    TPL_INTERFACE_CLASS_SUBCLASS_PROTOCOL(0x03, 0x01, 0x02, usageDriverTableEntry,
                                         USB_HOST_HID_INTERFACE) ,
};

```

Detecting Device Attach

Describe how to detect when a HID mouse device is attached.

Description

The application will need to know when a HID mouse device is attached. To receive this attach event from the USB HID Host Mouse Driver, the application must register an Attach Event Handler by calling the [USB_HOST_HID_MOUSE_EventHandlerSet](#) function. This function should be called before calling the [USB_HOST_BusEnable](#) function; otherwise, the application may miss HID attach events.

Mouse Data Event Handling

Describes mouse data event handling, which includes a code example.

Description

No extra event handler is required to be registered to receive mouse data. A call to function [USB_HOST_HID_MOUSE_EventHandlerSet](#) once is adequate to receive mouse data as well.

The mouse button state along with the X, Y, and Z relative coordinate positions are provided by the USB Host HID Mouse Driver. The data type is [USB_HOST_HID_MOUSE_DATA](#) and is defined in [usb_host_hid_mouse.h](#). The following code shows an event handler example.

Example:

```

/* This code shows an example of HID Mouse Event Handler */

void APP_USBHostHIDMouseEventHandler
(
    USB_HOST_HID_MOUSE_HANDLE handle,
    USB_HOST_HID_MOUSE_EVENT event,
    void * pData
)
{
    /* This function gets called in the following scenarios:
     1. USB Mouse is Attached
     2. USB Mouse is detached
     3. USB Mouse data has been obtained.
    */

    switch ( event )
    {
        case USB_HOST_HID_MOUSE_EVENT_ATTACH:
            /* Mouse Attached */
            appData.state = APP_STATE_DEVICE_ATTACHED;
            break;

        case USB_HOST_HID_MOUSE_EVENT_DETACH:
    }
}

```

```

/* Mouse Detached */
appData.state = APP_STATE_DEVICE_DETACHED;
break;

case USB_HOST_HID_MOUSE_EVENT_REPORT_RECEIVED:
/* Mouse data event */
appData.state = APP_STATE_READ_HID;
/* Mouse Data from device */
memcpy(&appData.data, pData, sizeof(appData.data));

/* Now the Mouse data has been obtained. This is a parsed data
in a simple format defined by USB_HOST_HID_MOUSE_DATA type.
*/
break;
}

}

void APP_Tasks(void)
{
    switch (appData.state)
    {
        case APP_STATE_BUS_ENABLE:

            /* In this state the application enables the USB Host Bus. Note
            * how the USB Mouse event handler is registered before the bus
            * is enabled. */

            USB_HOST_HID_MOUSE_EventHandlerSet(APP_USBHostHIDMouseEventHandler);
            USB_HOST_BusEnable(0);
            appData.state = APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE;
            break;

        case APP_STATE_WAIT_FOR_BUS_ENABLE_COMPLETE:
            /* Here we wait for the bus enable operation to complete. */
            break;
    }
}

```

Configuring the Library

Describes how to configure the USB HID Host Mouse Driver.

Macros

Name	Description
USB_HID_GLOBAL_PUSH_POP_STACK_SIZE	Specifies the Global PUSH POP stack size supported.
USB_HOST_HID_INSTANCES_NUMBER	Specifies the number of HID instances.
USB_HOST_HID_INTERRUPT_IN_ENDPOINTS_NUMBER	Specifies the maximum number of INTERRUPT IN endpoints supported.
USB_HOST_HID_MOUSE_BUTTONS_NUMBER	Specifies the number of Mouse buttons supported.
USB_HOST_HID_USAGE_DRIVER_SUPPORT_NUMBER	Specifies the number of Usage driver registered

Description

The USB HID Host Mouse Driver requires configuration constants to be specified in the `system_config.h` file. These constants define the build time configuration (functionality and static resources) of the USB HID Host Mouse Driver.

USB_HID_GLOBAL_PUSH_POP_STACK_SIZE Macro

Specifies the Global PUSH POP stack size supported.

File

`usb_host_hid_config_template.h`

C

```
#define USB_HID_GLOBAL_PUSH_POP_STACK_SIZE
```

Description

USB Host HID Global PUSH POP stack size supported

This macro defines the size of the Global PUSH POP stack per HID driver instance. If the application wants to support HID device having 2 continuous PUSH item or 2 continuous POP item in the Report Descriptor, then the value should be set to 2.

Remarks

None.

USB_HOST_HID_INSTANCES_NUMBER Macro

Specifies the number of HID instances.

File

[usb_host_hid_config_template.h](#)

C

```
#define USB_HOST_HID_INSTANCES_NUMBER
```

Description

USB host HID Maximum Number of instances

This macro defines the number of instances of the HID host Driver. For example, if the application needs to implement two instances of the HID host Driver, value should be set to 2.

Remarks

None.

USB_HOST_HID_INTERRUPT_IN_ENDPOINTS_NUMBER Macro

Specifies the maximum number of INTERRUPT IN endpoints supported.

File

[usb_host_hid_config_template.h](#)

C

```
#define USB_HOST_HID_INTERRUPT_IN_ENDPOINTS_NUMBER
```

Description

USB Host HID Maximum Number of INTERRUPT IN endpoints supported

This macro defines the number of INTERRUPT IN endpoints supported by USB Host HID driver. If the application needs to work with HID device which has 2 INTERRUPT IN endpoints, the value should be set to 2.

Remarks

None.

USB_HOST_HID_MOUSE_BUTTONS_NUMBER Macro

Specifies the number of Mouse buttons supported.

File

[usb_host_hid_config_template.h](#)

C

```
#define USB_HOST_HID_MOUSE_BUTTONS_NUMBER
```

Description

USB Host HID number of Mouse buttons supported

This macro defines the number of Mouse buttons supported. If the application wants to support HID Mouse device having 5 buttons, then the value should be set to 5.

Remarks

None.

USB_HOST_HID_USAGE_DRIVER_SUPPORT_NUMBER Macro

Specifies the number of Usage driver registered

File

[usb_host_hid_config_template.h](#)

C

```
#define USB_HOST_HID_USAGE_DRIVER_SUPPORT_NUMBER
```

Description

USB Host HID number of Usage driver registered

This macro defines the number of Usage driver registered with USB Host Hid driver. If the application wants HID driver to support 2 HID device having different usage, then the value should be set to 2.

Remarks

None.

Building the Library

Describes the files to be included in the project while using the USB HID Host Mouse Driver.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/usb.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
usb_host_hid_mouse.h	This header file should be included in any .c file that accesses the YSB HID Host Mouse Driver API.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/usb_host_hid.c	This file implements the USB HID Host Client Driver interface and should be included in the project if any usage driver operation is desired.
/src/dynamic/usb_host_hid_mouse.c	This file implements the USB HID Host Mouse Driver interface and should be included in the project if any usage driver operation is desired.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB HID Host Mouse Driver Library depends on the following modules:

- [USB Host Layer Library](#)
- [USB Host HID Client Driver Library](#)

Library Interface

a) Mouse Access Functions

	Name	Description
≡	USB_HOST_HID_MOUSE_EventHandlerSet	This function registers application callback function with the mouse driver.
≡	_USB_HOST_HID_MOUSE_EventHandler	This is function _USB_HOST_HID_MOUSE_EventHandler.
≡	_USB_HOST_HID_MOUSE_Task	This is function _USB_HOST_HID_MOUSE_Task.

b) Data Types and Constants

	Name	Description
	USB_HOST_HID_MOUSE_DATA	Defines the USB Host HID mouse data object.
	USB_HOST_HID_MOUSE_EVENT	Defines the possible USB HOST HID mouse driver events.
	USB_HOST_HID_MOUSE_EVENT_HANDLER	USB HOST mouse driver event handler function pointer type.
	USB_HOST_HID_MOUSE_HANDLE	USB HOST HID mouse driver instance handle.
	USB_HOST_HID_MOUSE_RESULT	USB Host HID mouse driver results.
	USB_HOST_HID_MOUSE_RESULT_MIN	USB Host HID mouse driver result minimum constant.
	USB_HOST_HID_MOUSE_HANDLE_INVALID	This is macro USB_HOST_HID_MOUSE_HANDLE_INVALID.

Description

This section describes the Application Programming Interface (API) functions of the USB HID Host Mouse Driver Library.

The USB Mouse driver does not require explicit API call by the application to obtain Mouse data. The data in the appropriate format is sent to the application during an application event handler function call.

a) Mouse Access Functions

[USB_HOST_HID_MOUSE_EventHandlerSet Function](#)

This function registers application callback function with the mouse driver.

File

[usb_host_hid_mouse.h](#)

C

```
USB_HOST_HID_MOUSE_RESULT USB_HOST_HID_MOUSE_EventHandlerSet(USB_HOST_HID_MOUSE_EVENT_HANDLER
appMouseEventHandler);
```

Returns

Returns data structure of [USB_HOST_HID_MOUSE_RESULT](#) type. [USB_HOST_HID_MOUSE_RESULT_INVALID_PARAMETER](#): Invalid Parameter [USB_HOST_HID_MOUSE_RESULT_FAILURE](#): On failure [USB_HOST_HID_MOUSE_RESULT_SUCCESS](#): On success

Description

This function registers application callback function with the mouse driver. Any subsequent mouse events is passed to the application by calling the registered application function. The function prototype should be of the [USB_HOST_HID_MOUSE_EVENT_HANDLER](#) type.

Remarks

This function should be called before the USB bus is enabled.

Preconditions

This function should be called before the USB bus is enabled.

Parameters

Parameters	Description
appMouseEventHandler	Function pointer to the application function.

Function

```
USB_HOST_HID_MOUSE_RESULT USB_HOST_HID_MOUSE_EventHandlerSet
()
```

```
USB_HOST_HID_MOUSE_EVENT_HANDLER appMouseEventHandler
);
```

_USB_HOST_HID_MOUSE_EventHandler Function

File

[usb_host_hid_mouse.h](#)

C

```
void _USB_HOST_HID_MOUSE_EventHandler(USB_HOST_HID_OBJ_HANDLE handle, USB_HOST_HID_EVENT event, void * eventData);
```

Description

This is function _USB_HOST_HID_MOUSE_EventHandler.

_USB_HOST_HID_MOUSE_Task Function

File

[usb_host_hid_mouse.h](#)

C

```
void _USB_HOST_HID_MOUSE_Task(USB_HOST_HID_OBJ_HANDLE handle);
```

Description

This is function _USB_HOST_HID_MOUSE_Task.

b) Data Types and Constants

USB_HOST_HID_MOUSE_DATA Structure

Defines the USB Host HID mouse data object.

File

[usb_host_hid_mouse.h](#)

C

```
typedef struct {
    USB_HID_BUTTON_STATE buttonState[USB_HOST_HID_MOUSE_BUTTONS_NUMBER];
    USB_HID_BUTTON_ID buttonID[USB_HOST_HID_MOUSE_BUTTONS_NUMBER];
    int16_t xMovement;
    int16_t yMovement;
    int16_t zMovement;
} USB_HOST_HID_MOUSE_DATA;
```

Members

Members	Description
USB_HID_BUTTON_STATE buttonState[USB_HOST_HID_MOUSE_BUTTONS_NUMBER];	Button state for the buttons. USB_HOST_HID_MOUSE_BUTTONS_NUMBER is system configurable option. The actual number of buttons in the mouse needs to be <= USB_HOST_HID_MOUSE_BUTTONS_NUMBER
int16_t xMovement;	Applicable for 2D Mouse Y - Coordinate displacement
int16_t yMovement;	Applicable for 2D Mouse Z - Coordinate displacement
int16_t zMovement;	Applicable only for 3D Mouse

Description

USB Host HID Mouse Data Object

This structure defines the USB Host HID mouse data object.

Remarks

None.

USB_HOST_HID_MOUSE_EVENT Enumeration

Defines the possible USB HOST HID mouse driver events.

File

[usb_host_hid_mouse.h](#)

C

```
typedef enum {
    USB_HOST_HID_MOUSE_EVENT_ATTACH = 0,
    USB_HOST_HID_MOUSE_EVENT_DETACH,
    USB_HOST_HID_MOUSE_EVENT_REPORT_RECEIVED
} USB_HOST_HID_MOUSE_EVENT;
```

Members

Members	Description
USB_HOST_HID_MOUSE_EVENT_ATTACH = 0	Mouse has been attached
USB_HOST_HID_MOUSE_EVENT_DETACH	Mouse has been detached
USB_HOST_HID_MOUSE_EVENT_REPORT_RECEIVED	Mouse IN Report data available

Description

USB HOST HID Mouse Driver Events

This enumeration lists the possible mouse events that the mouse driver can provide to the application. Some of these events have event data associated with them.

USB_HOST_HID_MOUSE_EVENT_HANDLER Type

USB HOST mouse driver event handler function pointer type.

File

[usb_host_hid_mouse.h](#)

C

```
typedef void (* USB_HOST_HID_MOUSE_EVENT_HANDLER)(USB_HOST_HID_MOUSE_HANDLE handle,
USB_HOST_HID_MOUSE_EVENT event, void *pData);
```

Description

USB HOST Mouse Driver Event Handler Function Pointer Type.

This defines the USB HOST HID mouse driver event handler function pointer type. Application must register a function of this type to receive HID mouse events. Registration should happen before USB BUS is enabled by the application.

USB_HOST_HID_MOUSE_HANDLE Type

USB HOST HID mouse driver instance handle.

File

[usb_host_hid_mouse.h](#)

C

```
typedef uintptr_t USB_HOST_HID_MOUSE_HANDLE;
```

Description

USB HOST HID Mouse Driver Instance Handle

This defines a USB Host HID mouse driver handle.

Remarks

None.

USB_HOST_HID_MOUSE_RESULT Enumeration

USB Host HID mouse driver results.

File

[usb_host_hid_mouse.h](#)

C

```
typedef enum {
    USB_HOST_HID_MOUSE_RESULT_FAILURE = USB_HOST_HID_MOUSE_RESULT_MIN,
    USB_HOST_HID_MOUSE_RESULT_INVALID_PARAMETER,
    USB_HOST_HID_MOUSE_RESULT_SUCCESS = 0
} USB_HOST_HID_MOUSE_RESULT;
```

Members

Members	Description
USB_HOST_HID_MOUSE_RESULT_FAILURE = USB_HOST_HID_MOUSE_RESULT_MIN	An unknown failure occurred
USB_HOST_HID_MOUSE_RESULT_INVALID_PARAMETER	Invalid or NULL parameter passed
USB_HOST_HID_MOUSE_RESULT_SUCCESS = 0	Indicates that the operation succeeded or the request was accepted and will be processed.

Description

USB Host HID MOUSE Result

This enumeration defines the possible returns values of USB Host HID mouse driver API. A function may only return some of the values in this enumeration. Refer to function description for details on which values will be returned.

Remarks

None.

USB_HOST_HID_MOUSE_RESULT_MIN Macro

USB Host HID mouse driver result minimum constant.

File

[usb_host_hid_mouse.h](#)

C

```
#define USB_HOST_HID_MOUSE_RESULT_MIN -50
```

Description

USB Host HID Mouse Driver Result Minimum Constant

This constant identifies the minimum value of the USB Host HID mouse driver and is used in the [USB_HOST_HID_MOUSE_RESULT](#) enumeration.

Remarks

None.

USB_HOST_HID_MOUSE_HANDLE_INVALID Macro**File**

[usb_host_hid_mouse.h](#)

C

```
#define USB_HOST_HID_MOUSE_HANDLE_INVALID ((USB_HOST_HID_MOUSE_HANDLE)(-1))
```

Description

This is macro USB_HOST_HID_MOUSE_HANDLE_INVALID.

Files**Files**

Name	Description
usb_host_hid_mouse.h	USB Host HID Mouse Driver Definition Header

[usb_host_hid_config_template.h](#)

USB host HID Class configuration definitions template

Description

This section lists the source and header files used by the library.

usb_host_hid_mouse.h

USB Host HID Mouse Driver Definition Header

Enumerations

	Name	Description
≡	USB_HOST_HID_MOUSE_EVENT	Defines the possible USB HOST HID mouse driver events.
≡	USB_HOST_HID_MOUSE_RESULT	USB Host HID mouse driver results.

Functions

	Name	Description
≡	_USB_HOST_HID_MOUSE_EventHandler	This is function _USB_HOST_HID_MOUSE_EventHandler.
≡	_USB_HOST_HID_MOUSE_Task	This is function _USB_HOST_HID_MOUSE_Task.
≡	USB_HOST_HID_MOUSE_EventHandlerSet	This function registers application callback function with the mouse driver.

Macros

	Name	Description
≡	USB_HOST_HID_MOUSE_HANDLE_INVALID	This is macro USB_HOST_HID_MOUSE_HANDLE_INVALID.
≡	USB_HOST_HID_MOUSE_RESULT_MIN	USB Host HID mouse driver result minimum constant.

Structures

	Name	Description
≡	USB_HOST_HID_MOUSE_DATA	Defines the USB Host HID mouse data object.

Types

	Name	Description
≡	USB_HOST_HID_MOUSE_EVENT_HANDLER	USB HOST mouse driver event handler function pointer type.
≡	USB_HOST_HID_MOUSE_HANDLE	USB HOST HID mouse driver instance handle.

Description

USB HOST HID Mouse Driver Interface Definition

This header file contains the function prototypes and definitions of the data types and constants that make up the interface between HID Mouse driver and top level application.

File Name

usb_host_hid_mouse.h

Company

Microchip Technology Inc.

usb_host_hid_config_template.h

USB host HID Class configuration definitions template

Macros

	Name	Description
≡	USB_HID_GLOBAL_PUSH_POP_STACK_SIZE	Specifies the Global PUSH POP stack size supported.
≡	USB_HOST_HID_INSTANCES_NUMBER	Specifies the number of HID instances.
≡	USB_HOST_HID_INTERRUPT_IN_ENDPOINTS_NUMBER	Specifies the maximum number of INTERRUPT IN endpoints supported.
≡	USB_HOST_HID_MOUSE_BUTTONS_NUMBER	Specifies the number of Mouse buttons supported.
≡	USB_HOST_HID_USAGE_DRIVER_SUPPORT_NUMBER	Specifies the number of Usage driver registered

Description

USB Host HID Class Configuration Definitions

This file contains configurations macros needed to configure the HID host Driver. This file is a template file only. It should not be included by the application. The configuration macros defined in the file should be defined in the configuration specific system_config.h.

File Name

usb_host_hid_config_template.h

Company

Microchip Technology Inc.

USB Hub Host Client Driver Library

This section describes the USB Hub Host Client Driver Library.

Introduction

Introduces the MPLAB Harmony USB Hub Host Client Driver Library.

Description

The USB Hub Host Client Driver in the MPLAB Harmony USB Host Stack allows USB Host Applications to interact with a USB Hub and thus manage multiple USB devices simultaneously in one application. The key features of the Hub Host Client Driver include:

- Allows multiple USB devices to be connected to the host and hence allow the USB Host application to interact simultaneously with multiple USB devices.
- Implemented as per Chapter 11 of the USB 2.0 specification.
- Support multiple Hub tiers. A Hub can be connected to another Hub.
- Does not require application intervention for its operation. The application does not have to call an Hub Driver API.

Abstraction Model

Describes the Abstraction Model of the USB Hub Host Client Driver Library.

Description

The USB Hub Host Client Driver abstracts the complexities of Hub operation and presents a simple interface to the Host Layer. The interface allows the Host Layer to perform port operations such as port reset, port suspend and port resume. The port interface offered by the Hub Host Client Driver is the same as that offered by the root hub driver. In that, the Host Layer does not differentiate between an external hub and the root hub.

The USB Hub Host Client Driver does not have any application callable API. It only interacts with the Host Layer. The USB Hub Host Client Driver performs the task of powering up the ports, detecting device attach and detach and notifying the same to the Host Layer and detecting over current conditions. The USB Hub Host Client Driver performs the control transfers required for these tasks.

Library Overview

The USB Hub Host Client Driver does not contain any application callable functions.

Using the Library

This topic describes the basic architecture of the USB Hub Host Client Driver Library and provides information and examples on its use.

How the Library Works

Describes how the Library works and how it should be used.

Description

The USB Hub Host Client Driver does not contain any application callable functions. The only step that the application code must implement is to enable USB Host Layer Hub support and to provision the USB Hub Host Client Driver in the TPL table.

The USB Host Layer enables Hub Support when the [USB_HOST_HUB_SUPPORT_ENABLE](#) configuration macro is defined in system_config.h. Refer to the [Configuring the Library](#) section of the USB Host Layer Library Help Topic for more information.

Hub TPL Table Configuration

Provides information on configuring the TPL table for adding Hub support.

Description

The Host Layer attaches the USB Hub Host Client Driver to a Hub device only if the TPL table contains an entry to enable this feature. The driver interface for such a TPL entry should point to [USB_HOST_HUB_INTERFACE](#). The following code shows an example of the TPL entry for the adding Hub support to the application.

Example:

```
/* This code shows an example of how to initialize the TPL table to
 * support a USB Hub Host Client Driver */

#include "usb/usb_host_hub.h"
#include "usb/usb_hub.h"

const USB_HOST_TPL_ENTRY USBTPLList[ 2 ] =
{
    TPL_INTERFACE_CLASS_SUBCLASS_PROTOCOL(USB_HUB_CLASS_CODE, 0x00, 0x00, NULL, USB_HOST_HUB_INTERFACE),

    /* A high-speed hub will report the number of transaction translators in the
     * protocol field. We can ignore this and let the host layer load the hub
     * driver for a high-speed hub */

    TPL_INTERFACE_CLASS (USB_HUB_CLASS_CODE, NULL, USB_HOST_HUB_INTERFACE)
};


```

USB Hub Host Client Driver Test Results

Provides test results for the USB Hub Host Client Driver.

Description

The following table lists the commercially available USB hubs, which have been tested to successfully enumerate and operate with the USB Hub Host Client Driver in the MPLAB Harmony USB Host Stack. Note that if the Hub you are using is not included in the table, this indicates that this Hub has not been tested with the USB Hub Host Client Driver. However, the Hub could still potentially work with the USB Hub Host Client Driver. The hubs were tested with the hub_msd USB Host demonstration in the latest version of the MPLAB Harmony USB Host Stack.

Hub Model	Number of Ports	VID	PID
Belkin USB 2.0	4	0x050D	0x0233
QHMPL	4	0x1A40	0x0101
Portronics	3	0x1A40	0x0101
Sanda	4	0x05E3	0x0606
iBall	4	0x1A40	0x0101

Configuring the Library

Describes how to configure the USB Hub Host Client Driver.

Macros

Name	Description
USB_HOST_HUB_INSTANCES_NUMBER	Specifies the number of Hub to be supported in the system.
USB_HOST_HUB_PORTS_NUMBER	Specifies the number of ports per Hub.

Description

The USB Hub Host Client Driver requires configuration constants to be specified in `system_config.h` file. These constants define the build time configuration (functionality and static resources) of the Hub Host Client Driver.

USB_HOST_HUB_INSTANCES_NUMBER Macro

Specifies the number of Hub to be supported in the system.

File

[usb_host_hub_config_template.h](#)

C

```
#define USB_HOST_HUB_INSTANCES_NUMBER
```

Description

USB Host Hub Instances Number

This configuration constant defines the total number of hubs to be supported in the application. This includes hubs connected across multiple USBs. If the hub connected to the host exceed this number, then the additional hubs will not be enumerated.

Remarks

Increasing number of Hubs to be supported will also increase memory consumption.

USB_HOST_HUB_PORTS_NUMBER Macro

Specifies the number of ports per Hub.

File

[usb_host_hub_config_template.h](#)

C

```
#define USB_HOST_HUB_PORTS_NUMBER
```

Description

USB Host Hub Number of Ports per Hub

This configuration macros specifies the number of Ports per Hub. If any Hub connected to host will have a maximum of 4 ports, then this number should be set to 4. A hub with more ports than the value defined by this constant will not be supported.

Remarks

Supporting a hub with more ports increases the memory requirement.

Building the Library

Describes the files to be included in the project while using the USB Hub Host Client Driver.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/usb.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
usb_host_hub.h	This header file should be included in any .c file that accesses the USB Hub Host Client Driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/usb_host_hub.c	This file implements the USB Hub Host Client Driver interface and should be included in the project if the USB Hub Host Client Driver operation is desired.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB Hub Host Client Driver Library depends on the following modules:

- [USB Host Layer Library](#)

Library Interface

Data Types and Constants

	Name	Description
	USB_HOST_HUB_INTERFACE	USB Hub Host Client Driver Interface Pointer.

Description

This section describes the Application Programming Interface (API) functions of the USB Hub Host Client Driver Library.

Refer to each section for a detailed description.

Data Types and Constants

USB_HOST_HUB_INTERFACE Macro

USB Hub Host Client Driver Interface Pointer.

File

[usb_host_hub.h](#)

C

```
#define USB_HOST_HUB_INTERFACE
```

Description

USB Hub Host Client Driver Interface Pointer

This constant is a pointer to a table of function pointers that define the interface between the Hub Host Client Driver and the USB Host Layer. This constant should be used while adding support for the Hub Driver in TPL table.

Remarks

None.

Files

Files

Name	Description
usb_host_hub.h	USB Host Hub Client Driver Interface Header
usb_host_hub_config_template.h	USB host CDC Class configuration definitions template

Description

This section lists the source and header files used by the library.

usb_host_hub.h

USB Host Hub Client Driver Interface Header

Macros

	Name	Description
	USB_HOST_HUB_INTERFACE	USB Hub Host Client Driver Interface Pointer.

Description

USB Host Hub Client Driver Interface Definition

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the USB HOST Hub Client Driver.

File Name

`usb_host_hub.h`

Company

Microchip Technology Inc.

`usb_host_hub_config_template.h`

USB host CDC Class configuration definitions template

Macros

	Name	Description
	<code>USB_HOST_HUB_INSTANCES_NUMBER</code>	Specifies the number of Hub to be supported in the system.
	<code>USB_HOST_HUB_PORTS_NUMBER</code>	Specifies the number of ports per Hub.

Description

USB Host Hub Configuration Definitions

This file contains configurations macros needed to configure the Hub Driver. This file is a template file only. It should not be included by the application. The configuration macros defined in the file should be defined in the configuration specific `system_config.h`.

File Name

`usb_host_hub_config_template.h`

Company

Microchip Technology Inc.

USB MSD Host Client Driver Library

This section describes the USB MSD Host Client Driver Library.

Introduction

Introduces the MPLAB Harmony USB Mass Storage Device (MSD) Host Client Driver Library.

Description

The USB MSD Host Client Driver in the MPLAB Harmony USB Host Stack allows USB Host Applications to support and interact with Mass Storage Class (MSC) USB devices. Examples of such devices are USB Pen Drives and USB Card readers. The USB MSD Host Client Driver along with the SCSI Block Storage Driver Library implement a multi-layer solution to reading and writing to mass storage USB device that implement the SCSI command protocol. The USB MSD Host Client Driver has the following features:

- Implements the Bulk Only Transport (BOT) protocol in the USB MSD specification
- Supports multiple instances, which allows the application to interact with multiple storage devices
- Supports multi-LUN devices such as USB Card Reader
- Automatically (without application intervention) attaches the SCSI Block Driver to an identified device
- Implements automatic clearing of endpoint stall conditions
- Implements all three stages of a BOT transfer and provide a simple event driver transfer interface to the top-level application (which is typically a block storage driver library such as the SCSI Block Storage Driver Library)
- Typically operates without application intervention. The BOT transfers are typically invoked by the SCSI Block Storage Driver Library.

Using the Library

This topic describes the basic architecture of the USB MSD Host Client Drier Library and provides information and examples on its use.

Library Overview

Provides an overview of the USB Host MSD Library.

Description

The USB MSD Host Client Driver can be grouped functionally as shown in the following table.

Library Interface Section	Descriptions
Data Transfer Functions	These functions allow the application client to transfer data to the attached device.

Abstraction Model

Describes the Abstraction Model of the USB MSD Host Client Driver Library.

Description

The USB MSD Host Client Driver provides the transport for SCSI commands that implements media read, write and control operations. It abstracts the details of initiating and completing a BOT transfer and performing error handling and presents a simple event driven interface to the top-level block storage command driver library.

The USB MSD Host Client Driver uses the USB Host data transfer and pipe management routines to implement the three stages of a BOT transfer. The library accepts a SCSI command from the SCSI Block Storage driver and transports this command in the command block of the Command Block Wrapper in the CBW stage of the BOT transfer. If the command requires a data stage, the USB MSD Host Client Driver library will transfer data between the USB Host and the device. The USB MSD Host Client Driver will then terminate the BOT transfer by requesting for the Command Status Wrapper (CSW) from the device.

If the device stalls any stage of the transfer, the USB MSD Host Client Driver will clear the stall and will automatically initiate the CSW stage to complete the transfer. The transfer result is communicated to the top level block storage driver library through a callback mechanism.

The USB Host layer will attach the USB MSD Host Client Driver to a mass storage device based on a TPL entry match. The USB MSD Host Client Driver will then open data and communication control pipes to the device. It will first get the number of logical units (LUN) that the device contains. It will then initialize the SCSI block storage driver for each reported LUN and mark the device state as being ready for data transfers.

How the Library Works

Describes how the library works and how it should be used.

Description

The USB MSD Host Client Driver provides the top level block storage driver with an easy to use, event driven, interface to transport the block storage command and data between the block storage command driver library and a compliant mass storage device. The USB MSD Host Client Driver in the MPLAB Harmony USB Host stack immediately supports mass storage devices that advertise support of the SCSI command set. Indeed, most of the commercially available USB storage devices such as USB pen driver and USB card readers respond to SCSI command requests.

The process of initializing the SCSI block storage driver library when a device is attached is performed automatically, by the USB MSD Host Client Driver. This does not require user application intervention. The following sections describe the TPL table design (application responsibility) and USB MSD Host Client Driver data transfer function (typically called by the SCSI block storage driver library).

MSD TPL Table Configuration

Describes TPL table design for matching MSD devices.

Description

The Host Layer attaches the MSD Host Client Driver to a device when the class, subclass and protocol fields in the Interface Association Descriptor (IAD) or Interface descriptor match the entry in the TPL table. When specifying the entry for the MSD device, the driver interface must be set to [USB_HOST_MSD_INTERFACE](#). This will attach the USB MSD Host Client Driver to the device when the USB Host matches the TPL entry to the device. The following code shows a TPL table design for matching MSD Devices.

Example:

```
/* This code shows an example TPL table entry for supporting a Mass
 * Storage Device */

const USB_HOST_TPL_ENTRY USBTPLList[ 1 ] =
{
    TPL_INTERFACE_CLASS_SUBCLASS_PROTOCOL(USB_MSD_CLASS_CODE,
        USB_MSD_SUBCLASS_CODE_SCSI_TRANSPARENT_COMMAND_SET, USB_MSD_PROTOCOL, NULL,
        USB_HOST_MSD_INTERFACE)
```

{ ;

Data Transfer

Describes how to transfer data, which includes a code example.

Description

The USB MSD Host Client Driver data transfer function is typically called by the SCSI Block Storage Driver Library. The [USB_HOST_MSD_Transfer](#) function allows the SCSI Block Storage Driver to transport SCSI commands to the mass storage device. The cdb parameter and the cdbLength parameter of the function specify the command and its size respectively. If the command requires the transport of data, then data must contain the pointer to the buffer and size specifies the amount of data expected to be transported. When the BOT transfer complete, the USB MSD Host Client Diver will call the callback function. The following code snippet shows an example of using the [USB_HOST_MSD_Transfer](#) function.

Example:

```
/* This code shows usage of the USB_HOST_MSD_Transfer function. The SCSI Block
 * Driver Library uses this function to send a SCSI Inquiry Command to the
 * device. Note how the commandCompleted flag in the SCSI instance object
 * tracks the completion of the transfer. This flag is updated in the transfer
 * callback. */

void _USB_HOST_SCSI_TransferCallback
(
    USB_HOST_MSD_LUN_HANDLE lunHandle,
    USB_HOST_MSD_TRANSFER_HANDLE transferHandle,
    USB_HOST_MSD_RESULT result,
    size_t size,
    uintptr_t context
)
{
    int scsiObjIndex;
    USB_HOST_SCSI_OBJ * scsiObj;
    USB_HOST_SCSI_COMMAND_OBJ * commandObj;
    USB_HOST_SCSI_EVENT event;

    /* Get the SCSI object index from the lunHandle */
    scsiObjIndex = _USB_HOST_SCSI_LUNHandleToSCSIInstance(lunHandle);

    /* Get the pointer to the SCSI object */
    scsiObj = &gUSBHostSCSIObj[scsiObjIndex];

    /* Pointer to the command object */
    commandObj = &scsiObj->commandObj;

    /* The processed size */
    commandObj->size = size;

    /* The result of the command */
    commandObj->result = result;

    /* Let the main state machine know that the command is completed */
    commandObj->commandCompleted = true;

    /* The rest of the code is not shown here for the sake of brevity */
}

void USB_HOST_SCSI_Tasks(USB_HOST_MSD_LUN_HANDLE lunHandle)
{
    switch(scsiObj->state)
    {
        /* For the sake of brevity, only one SCSI command is show here */
        case USB_HOST_SCSI_STATE_INQUIRY_RESPONSE:

            /* We get the SCSI Enquiry response. Although there isn't much
             * that we can do with this data */
            _USB_HOST_SCSI_InquiryResponseCommand(scsiObj->commandObj.cdb);

            /* The commandCompleted flag will be updated in the callback.
             * Update the state and send the command. */
    }
}
```

```

scsiObj->commandObj.inUse = true;
scsiObj->commandObj.commandCompleted = false;
scsiObj->commandObj.generateEvent = false;

result = USB_HOST_MSD_Transfer(scsiObj->lunHandle,
                               scsiObj->commandObj.cdb, 6, scsiObj->buffer, 36,
                               USB_HOST_MSD_TRANSFER_DIRECTION_DEVICE_TO_HOST,
                               _USB_HOST_SCSI_TransferCallback, (uintptr_t)(scsiObj));

if(result == USB_HOST_MSD_RESULT_SUCCESS)
{
    scsiObj->state = USB_HOST_SCSI_STATE_WAIT_INQUIRY_RESPONSE;
}

breakdefault:
    break;

}
}

```

Configuring the Library

Describes how to configure the USB Host Layer.

Macros

	Name	Description
	USB_HOST_MSD_INSTANCES_NUMBER	Defines the maximum number of MSD devices to be supported by this host application.
	USB_HOST_MSD_LUNS_NUMBER	Defines the maximum number of MSD Device LUNs to be supported in the application.

Description

The USB MSD Host Client Driver requires configuration constants to be specified in `system_config.h` file. These constants define the build time configuration (functionality and static resources) of the USB MSD Host Client Driver.

USB_HOST_MSD_INSTANCES_NUMBER Macro

Defines the maximum number of MSD devices to be supported by this host application.

File

`usb_host_msd_config_template.h`

C

```
#define USB_HOST_MSD_INSTANCES_NUMBER
```

Description

USB Host MSD Client Driver Instances Number.

This constant defines the maximum number of MSD devices to be supported by this host application. For example, if 3 USB Pen Drives need to be supported, then this value should be 3. This value cannot be greater than `USB_HOST_DEVICES_NUMBER`, which defines the maximum number of devices to be supported in the application. If this constant is less than `USB_HOST_DEVICES_NUMBER`, then only `USB_HOST_MSD_INSTANCES_NUMBER` of MSD devices will be enumerated.

Remarks

None

USB_HOST_MSD_LUNS_NUMBER Macro

Defines the maximum number of MSD Device LUNs to be supported in the application.

File

`usb_host_msd_config_template.h`

C

```
#define USB_HOST_MSD_LUNS_NUMBER
```

Description

USB Host MSD Client Driver LUNs Number.

An MSD device may have multiple storage units, each addressable through a LUN number. An example is a USB Card reader with multiple card slots. Each card slot has a LUN number. The `USB_HOST_MSD_LUNS_NUMBER` constant defines the maximum number of such logical units that can be managed by the USB Host application. This number should at least be equal to `USB_HOST_MSD_INSTANCES_NUMBER`. To configure this value, consider an example of an application that will support a maximum of 2 USB Storage device. These 2 storage devices are expected to have at the most 3 LUNs each. Then the `USB_HOST_MSD_LUNS_NUMBER` constant should be set to 6 (2 devices and 3 LUNs per device)/

Remarks

None

Building the Library

Describes the files to be included in the project while using the MSD Host Function Driver.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/usb`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
usb_host_msd.h	This header file should be included in any .c file that accesses the USB Host MSD Client Driver API.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must always be included in the MPLAB X IDE project to build this library.

Source File Name	Description
<code>/src/dynamic/usb_host_msd.c</code>	This file implements the USB Host MSD Client Driver interface and should be included in the project if USB Host MSD Client Driver operation is desired.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies

The USB MSD Host Library depends on the following modules:

- [USB Host Layer Library](#)

Library Interface**a) Data Transfer Functions**

	Name	Description
≡	USB_HOST_MSD_Transfer	This function schedules a MSD BOT transfer.
≡	USB_HOST_MSD_TransferErrorTasks	This function maintains the MSD transfer error handling state machine.

b) Data Types and Constants

	Name	Description
	USB_HOST_MSD_RESULT	USB HOST MSD Result
	USB_HOST_MSD_TRANSFER_CALLBACK	USB HOST MSD Transfer Complete Callback
	USB_HOST_MSD_TRANSFER_DIRECTION	USB HOST MSD Transfer Direction.
	USB_HOST_MSD_TRANSFER_HANDLE	USB HOST MSD Transfer Handle
	USB_HOST_MSD_INTERFACE	USB HOST MSD Client Driver Interface
	USB_HOST_MSD_TRANSFER_HANDLE_INVALID	USB HOST MSD Transfer Handle Invalid
	USB_HOST_MSD_LUN_HANDLE	USB HOST MSD LUN Handle
	USB_HOST_MSD_LUN_HANDLE_INVALID	USB HOST MSD LUN Handle Invalid
	USB_HOST_MSD_ERROR_CODE	USB Host MSD Error Codes.

Description

a) Data Transfer Functions

USB_HOST_MSD_Transfer Function

This function schedules a MSD BOT transfer.

File

[usb_host_msd.h](#)

C

```
USB_HOST_MSD_RESULT USB_HOST_MSD_Transfer(USB_HOST_MSD_LUN_HANDLE lunHandle, uint8_t * cdb, uint8_t
cdbLength, void * data, size_t size, USB_HOST_MSD_TRANSFER_DIRECTION transferDirection,
USB_HOST_MSD_TRANSFER_CALLBACK callback, uintptr_t context);
```

Returns

[USB_HOST_MSD_RESULT_FAILURE](#) - An unknown failure occurred. [USB_HOST_MSD_RESULT_BUSY](#) - The transfer cannot be scheduled right now. The caller should retry. [USB_HOST_MSD_RESULT_LUN_HANDLE_INVALID](#) - This LUN does not exist in the system. [USB_HOST_MSD_RESULT_SUCCESS](#) - The transfer request was scheduled.

Description

This function schedules a MSD BOT transfer. The command to be executed is specified in the cdb. This should be pointer to a 16 byte command descriptor block. The actual length of the command is specified by cdbLength. If there is data to be transferred, the pointer to the buffer is specified by data. The size of the buffer is specified in size. When the transfer completes, the callback function will be called. The context will be returned in the callback function.

Remarks

This is a local function and should not be called directly by the application.

Preconditions

None.

Parameters

Parameters	Description
cdb	pointer to the command to be executed. Should be a pointer to a 16 byte array. Unused bytes should be zero-padded.
cdbLength	Actual size of the command.
data	pointer to the data buffer if a data stage is involved.
size	size of the data buffer.
callback	callback function to be called when the transfer has completed.
transferDirection	specifies the direction of the MSD transfer.
context	caller defined context that is returned in the callback function.

Function

```
USB_HOST_MSD_RESULT USB_HOST_MSD_Transfer
(
    uint8_t * cdb,
    uint8_t cdbLength,
    void * data,
    size_t size,
    USB_HOST_MSD_TRANSFER_CALLBACK callback,
    uintptr_t context
)
```

USB_HOST_MSD_TransferErrorTasks Function

This function maintains the MSD transfer error handling state machine.

File

[usb_host_msd.h](#)

C

```
void USB_HOST_MSD_TransferErrorTasks(USB_HOST_MSD_LUN_HANDLE lunHandle);
```

Returns

None.

Description

This function maintains the MSD transfer error handling state machine. This function should be called periodically after the [USB_HOST_MSD_Transfer](#) function has been called to schedule a transfer. The function should be called periodically atleast till the transfer completion event has been received. Calling this function while a BOT transfer is in progress allows the MSD Host Client driver to perform BOT error handling in a non-blocking manner.

Calling this function when there is no BOT transfer in progress will not have any effect. In case of BOT error handling, calling this function will eventually result in a BOT transfer event. It is not necessary to call this function after this event has occurred (till the next BOT transfer has been scheduled).

Remarks

While running in an RTOS application, this function should be called in the same thread that requested the BOT Transfer and operating the logical unit.

Preconditions

The lunHandle should be valid.

Parameters

Parameters	Description
lunHandle	handle to valid LUN.

Function

```
void USB_HOST_MSD_TransferErrorTasks
(
    USB_HOST_MSD_LUN_HANDLE lunHandle,
);
```

b) Data Types and Constants

USB_HOST_MSD_RESULT Enumeration

USB HOST MSD Result

File

[usb_host_msd.h](#)

C

```
typedef enum {
    USB_HOST_MSD_RESULT_COMMAND_PASSED = 0,
    USB_HOST_MSD_RESULT_COMMAND_FAILED = 1,
    USB_HOST_MSD_RESULT_COMMAND_PHASE_ERROR = 2,
    USB_HOST_MSD_RESULT_SUCCESS,
    USB_HOST_MSD_RESULT_FAILURE,
    USB_HOST_MSD_RESULT_BUSY,
    USB_HOST_MSD_RESULT_LUN_HANDLE_INVALID,
    USB_HOST_MSD_RESULT_COMMAND_STALLED
} USB_HOST_MSD_RESULT;
```

Members

Members	Description
USB_HOST_MSD_RESULT_COMMAND_PASSED = 0	MSD Command result was success. The command issued to the MSD device <ul style="list-style-type: none"> • passed.
USB_HOST_MSD_RESULT_COMMAND_FAILED = 1	MSD Command failed. The command issued to the MSD device failed. The <ul style="list-style-type: none"> • device BOT state machine is in sync with the host. The data residue • length is valid.
USB_HOST_MSD_RESULT_COMMAND_PHASE_ERROR = 2	MSD Command failed with phase error. The command issued to the MSD device <ul style="list-style-type: none"> • has failed. The failure reason is unknown. The MSD Host Client driver has • reset the device BOT state machine.
USB_HOST_MSD_RESULT_SUCCESS	The operation was successful
USB_HOST_MSD_RESULT_FAILURE	An unknown failure has occurred.
USB_HOST_MSD_RESULT_BUSY	The request cannot be accepted at this time
USB_HOST_MSD_RESULT_LUN_HANDLE_INVALID	The specified LUN is not valid
USB_HOST_MSD_RESULT_COMMAND_STALLED	The MSD request was stalled

Description

USB HOST MSD Result

This enumeration defines the possible return values of different USB HOST MSD Client driver function call. Refer to the specific function documentation for details on the return values.

Remarks

None.

USB_HOST_MSD_TRANSFER_CALLBACK Type

USB HOST MSD Transfer Complete Callback

File

[usb_host_msd.h](#)

C

```
typedef void (* USB_HOST_MSD_TRANSFER_CALLBACK)(USB_HOST_MSD_LUN_HANDLE lunHandle,
                                                USB_HOST_MSD_TRANSFER_HANDLE transferHandle, USB_HOST_MSD_RESULT result, size_t size, uintptr_t context);
```

Description

USB HOST MSD Transfer Complete Callback

This type defines the type of the callback function that the application must register in the [USB_HOST_MSD_Transfer](#) function to receive notification when a transfer has completed. The callback function will be called with the following parameters.

lunHandle - The handle to the LUN from where this notification originated.

transferHandle - the handle to the MSD transfer.

result - result of the transfer.

size - of the transfer.

context - context that specified when this transfer was scheduled.

Remarks

None.

USB_HOST_MSD_TRANSFER_DIRECTION Enumeration

USB HOST MSD Transfer Direction.

File

[usb_host_msd.h](#)

C

```
typedef enum {
    USB_HOST_MSD_TRANSFER_DIRECTION_HOST_TO_DEVICE = 0x00,
    USB_HOST_MSD_TRANSFER_DIRECTION_DEVICE_TO_HOST = 0x80
} USB_HOST_MSD_TRANSFER_DIRECTION;
```

Members

Members	Description
USB_HOST_MSD_TRANSFER_DIRECTION_HOST_TO_DEVICE = 0x00	Data moves from host to device
USB_HOST_MSD_TRANSFER_DIRECTION_DEVICE_TO_HOST = 0x80	Data moves from device to host

Description

USB HOST Transfer Direction

This enumeration specifies the direction of the data stage.

Remarks

None.

USB_HOST_MSD_TRANSFER_HANDLE Type

USB HOST MSD Transfer Handle

File

[usb_host_msd.h](#)

C

```
typedef uintptr_t USB_HOST_MSD_TRANSFER_HANDLE;
```

Description

USB HOST MSD Transfer Handle

This type defines a USB Host MSD Transfer Handle.

Remarks

None.

USB_HOST_MSD_INTERFACE Macro

USB HOST MSD Client Driver Interface

File

[usb_host_msd.h](#)

C

```
#define USB_HOST_MSD_INTERFACE
```

Description

USB HOST MSD Client Driver Interface

This macro should be used by the application in TPL table while adding support for the USB MSD Host Client Driver.

Remarks

None.

USB_HOST_MSD_TRANSFER_HANDLE_INVALID Macro

USB HOST MSD Transfer Handle Invalid

File

[usb_host_msd.h](#)

C

```
#define USB_HOST_MSD_TRANSFER_HANDLE_INVALID
```

Description

USB HOST MSD Transfer Handle Invalid

This value defines an invalid Transfer Handle.

Remarks

None.

USB_HOST_MSD_LUN_HANDLE Type

USB HOST MSD LUN Handle

File

[usb_host_msd.h](#)

C

```
typedef uintptr_t USB_HOST_MSD_LUN_HANDLE;
```

Description

USB HOST MSD LUN Handle

This type defines a MSD LUN Handle. This handle is used by SCSI driver to identify the LUN.

Remarks

None.

USB_HOST_MSD_LUN_HANDLE_INVALID Macro

USB HOST MSD LUN Handle Invalid

File

[usb_host_msd.h](#)

C

```
#define USB_HOST_MSD_LUN_HANDLE_INVALID
```

Description

USB HOST MSD LUN Handle Invalid

This value defines an invalid LUN Handle.

Remarks

None.

USB_HOST_MSD_ERROR_CODE Enumeration

USB Host MSD Error Codes.

File

[usb_host_msd.h](#)

C

```
typedef enum {
```

```

USB_HOST_MSD_ERROR_CODE_INSUFFICIENT_INSTANCES = 1,
USB_HOST_MSD_ERROR_CODE_NOT_FOUND_BULK_IN_ENDPOINT,
USB_HOST_MSD_ERROR_CODE_NOT_FOUND_BULK_OUT_ENDPOINT,
USB_HOST_MSD_ERROR_CODE_FAILED_PIPE_OPEN,
USB_HOST_MSD_ERROR_CODE_FAILED_GET_MAX_LUN,
USB_HOST_MSD_ERROR_CODE_FAILED_BOT_TRANSFER,
USB_HOST_MSD_ERROR_CODE_FAILED_RESET_RECOVERY,
USB_HOST_MSD_ERROR_CODE_CBW_STALL_RESET_RECOVERY,
USB_HOST_MSD_ERROR_CODE_TRANSFER_BUSY,
USB_HOST_MSD_ERROR_CODE_CSW_PHASE_ERROR,
USB_HOST_MSD_ERROR_CODE_CSW_UNKNOWN_ERROR
} USB_HOST_MSD_ERROR_CODE;

```

Members

Members	Description
USB_HOST_MSD_ERROR_CODE_INSUFFICIENT_INSTANCES = 1	<p>This error occurs when the number of MSD instances defined via USB_HOST_MSD_INSTANCES_NUMBER (in system_config.h) is insufficient. For</p> <ul style="list-style-type: none"> example, this error would occur if the value of USB_HOST_MSD_INSTANCES_NUMBER is 2, two MSC devices are already connected and third MSC device is connected to the host. The object identifier in this case will be the USB_HOST_DEVICE_OBJ_HANDLE value.
USB_HOST_MSD_ERROR_CODE_NOT_FOUND_BULK_IN_ENDPOINT	<p>This error occurs when the driver descriptor parser could not find a Bulk</p> <ul style="list-style-type: none"> IN endpoint in the interface descriptor. The object identifier in this case will be the USB_HOST_DEVICE_OBJ_HANDLE value.
USB_HOST_MSD_ERROR_CODE_NOT_FOUND_BULK_OUT_ENDPOINT	<p>This error occurs when the driver descriptor parser could not find a Bulk</p> <ul style="list-style-type: none"> OUT endpoint in the interface descriptor. The object identifier in this case will be USB_HOST_DEVICE_OBJ_HANDLE value.
USB_HOST_MSD_ERROR_CODE_FAILED_PIPE_OPEN	<p>This error occurs when the driver could not open a Bulk pipe. This</p> <ul style="list-style-type: none"> typically happens either due to a host layer error or due to insufficient number of pipes (which is configured via USB_HOST_PIPES_NUMBER). The object identifier in this case will be USB_HOST_DEVICE_OBJ_HANDLE value.
USB_HOST_MSD_ERROR_CODE_FAILED_GET_MAX_LUN	<p>This error occurs when the Get Max LUN request issued by the driver fails</p> <ul style="list-style-type: none"> for any reason. The object identifier in this case will be the MSC device instance index.
USB_HOST_MSD_ERROR_CODE_FAILED_BOT_TRANSFER	<p>This error occurs when any stage of the BOT has failed due to bus error</p> <ul style="list-style-type: none"> or an unknown failure. The object identifier in this case will be the MSC device instance index.
USB_HOST_MSD_ERROR_CODE_FAILED_RESET_RECOVERY	<p>This error occurs when the MSD Reset Recovery procedure has failed. A MSC</p> <ul style="list-style-type: none"> device should not fail a MSD Reset Recovery procedure. The object identifier in this case will be the device instance index.
USB_HOST_MSD_ERROR_CODE_CBW_STALL_RESET_RECOVERY	<p>This error code indicates a condition where the CBW stage of the BOT was</p> <ul style="list-style-type: none"> stalled and the driver is about to launch MSD reset recovery. The object identifier in this case is the MSC Device instance index. This code is generated from an interrupt context. The driver may continue to function normally post this condition.

USB_HOST_MSD_ERROR_CODE_TRANSFER_BUSY	This error code indicates a condition where the BOT transfer could not be initiated because a transfer is already in progress. The identifier in this case is the MSC Device Instance Index. The driver may continue to function normally post this condition. This condition may occur several times.
USB_HOST_MSD_ERROR_CODE_CSW_PHASE_ERROR	This error code indicates a condition where the BOT transfer failed due to a phase error in the CSW stage of the BOT. The identifier in this case is the MSC Device instance index. This code is generated from an interrupt context. The driver may continue to function normally post this condition.
USB_HOST_MSD_ERROR_CODE_CSW_UNKNOWN_ERROR	This error code indicates that a condition where an unknown error has occurred during the CSW stage of the BOT. The identifier in this case is the MSC Device instance index. This code is generated from an interrupt context. The driver may continue to function normally post this condition.

Description

USB Host MSD Error Codes.

This enumeration defines the codes that the MSD Client Driver returns for possible errors that lead to the device being placed in an error state. The MSD client driver will not operate on a device which is in an error state. The errors are returned in the USB_HOST_MSD_ErrorCallback function.

Remarks

None.

Files

Files

Name	Description
usb_host_msd.h	USB Host MSD Class Driver Interface Header
usb_host_msd_config_template.h	USB Host MSD configuration template header file

Description

usb_host_msd.h

USB Host MSD Class Driver Interface Header

Enumerations

	Name	Description
	USB_HOST_MSD_ERROR_CODE	USB Host MSD Error Codes.
	USB_HOST_MSD_RESULT	USB HOST MSD Result
	USB_HOST_MSD_TRANSFER_DIRECTION	USB HOST MSD Transfer Direction.

Functions

	Name	Description
	USB_HOST_MSD_Transfer	This function schedules a MSD BOT transfer.
	USB_HOST_MSD_TransferErrorTasks	This function maintains the MSD transfer error handling state machine.

Macros

	Name	Description
	USB_HOST_MSD_INTERFACE	USB HOST MSD Client Driver Interface
	USB_HOST_MSD_LUN_HANDLE_INVALID	USB HOST MSD LUN Handle Invalid
	USB_HOST_MSD_TRANSFER_HANDLE_INVALID	USB HOST MSD Transfer Handle Invalid

Types

	Name	Description
	USB_HOST_MSD_LUN_HANDLE	USB HOST MSD LUN Handle
	USB_HOST_MSD_TRANSFER_CALLBACK	USB HOST MSD Transfer Complete Callback
	USB_HOST_MSD_TRANSFER_HANDLE	USB HOST MSD Transfer Handle

Description

USB Host MSD Class Driver Interface Definition

This header file contains the function prototypes and definitions of the data types and constants that make up the interface to the USB Host MSD Class Driver.

File Name

usb_host_msd.h

Company

Microchip Technology Inc.

usb_host_msd_config_template.h

USB Host MSD configuration template header file

Macros

	Name	Description
	USB_HOST_MSD_INSTANCES_NUMBER	Defines the maximum number of MSD devices to be supported by this host application.
	USB_HOST_MSD_LUNS_NUMBER	Defines the maximum number of MSD Device LUNs to be supported in the application.

Description

USB Host MSD function driver compile time options

This file contains USB host MSD function driver compile time options(macros) that has to be configured by the user. This file is a template file and must be used as an example only. This file must not be directly included in the project.

File Name

usb_host_msd_config_template.h

Company

Microchip Technology Inc.

Index

\

\.tpFlags.driverType = (TPL_FLAG_CLASS_SUBCLASS_PROTOCOL) enumeration 287

\.tpFlags.driverType = (TPL_FLAG_VID_PID) enumeration 288

—

_USB_DEVICE_IRP structure 152

_USB_HOST_HID_MOUSE_EventHandler function 449

_USB_HOST_HID_MOUSE_Task function 449

_USB_HOST_IRP structure 156

_USB_HOST_IRP_STATUS enumeration 157

0

0 enumeration 289

0x0000 enumeration 290

0xFF } enumeration 291

0xFF enumeration 290

0xFFFF } enumeration 293

0xFFFF enumeration 292

1

1 enumeration 293

A

a) Functions 62

Abstract Control Model (ACM) 165

Abstraction Model 24, 53, 79, 164, 199, 231, 241, 263, 311, 409, 443, 453, 458

 Generic USB Device Library 241

 USB CDC Device Library 164

 USB Device Layer Library 79

 USB MSD Device Library 231

Application Client Interaction 86

Audio Data Streaming 330

Audio Stream Event Handling 319

B

b) Data Types and Constants 67

BOS Descriptor Support 96

Building the Library 32, 61, 99, 176, 211, 236, 247, 273, 339, 420, 447, 455, 461

 Generic USB Device Library 247

 USB Audio 2.0 Device Library 61

 USB Audio Device Library 32

 USB Audio v1.0 Host Client Driver Library 339

 USB CDC Host Library 420

 USB Device Layer Library 99

 USB HID Device Library 211

 USB HID Host Mouse Driver Library 447

 USB Host Layer Library 273

 USB Hub Host Client Driver Library 455

 USB MSD Device Library 236

 USB MSD Host Library 461

C

Calling the Device Layer API 14

classCode enumeration 294

Configuring the Library 30, 60, 97, 175, 210, 235, 247, 270, 337, 419,

445, 454, 460

 Generic USB Device Library 247

 USB Audio 2.0 Device Library 60

 USB Audio Device Library 30

 USB Device Layer Library 97

Configuring the Stack 14

Control Transfer Events 243

Creating Your Own USB Device 7

D

Data Transfer 235, 459

Demonstration Application Logic 20

Detecting Device Attach 312, 410, 444

Device Layer Control Transfers 89

Device Layer Task Routines 85

Device Stack Configuration 22

E

Enabling Audio Stream 322

Endpoint Data Transfer 246

Endpoint Data Transfer Events 245

Endpoint Management 245

Event Handling 15, 21, 26, 55, 87, 167, 203, 242, 270, 417

F

false enumeration 295

Files 51, 77, 160, 197, 229, 240, 309, 404, 440, 451, 456, 468

 USB Audio 1.0 Device Library 51

 USB Audio 2.0 Device Library 77

 USB Audio v1.0 Host Client Driver Library 404

 USB CDC Device Library 197

 USB Device Layer Library 160

 USB HID Device Library 229

 USB HID Host Mouse Driver Library 451

 USB Host Layer Library 309

 USB Hub Host Client Driver Library 456

 USB MSD Device Library 240

Files to Include

 USB CDC Device Library 176

 USB Device Stack Porting 13

Function Driver Registration Table 83

Fuse Configuration and Initialization 19

G

Generic USB Device Library 241

Getting Started

 USB Device Library 3

 USB Host Library 249

H

Handling Endpoint 0 (EP0) Packets 16

HID Device TPL Table Configuration 443

Host Layer - Application Interaction 267

Host Layer Initialization 263

How the Library Works 24, 54, 81, 166, 201, 232, 242, 263, 312, 410, 443, 453, 458

 Generic USB Device Library 242

 USB Device Layer Library 81

 USB MSD Device Library 232

Hub TPL Table Configuration 454

- I**
- initData enumeration 296
 - Initializing and Communicating with the Endpoint 16
 - Initializing the Device Layer 84
 - Initializing the Library 25, 54
 - Initializing the USB Device Stack 13
 - Introduction 3, 13, 23, 53, 79, 163, 199, 231, 241, 249, 252, 262, 311, 409, 442, 453, 457
 - Generic USB Device Library 241
 - USB Audio 1.0 Device Library 23
 - USB Audio 2.0 Device Library 53
 - USB Audio v1.0 Host Client Driver Library 311
 - USB CDC Device Library 163
 - USB CDC Host Library 409
 - USB Device Layer Library 79
 - USB Device Library - Getting Started 3
 - USB HID Device Library 199
 - USB HID Host Mouse Driver Library 442
 - USB Host Layer Library 262
 - USB Host Library - Getting Started 249
 - USB Hub Host Client Driver Library 453
 - USB MSD Device Library 231
 - USB MSD Host Library 457
 - Invoking the Tasks Routine 22
- L**
- Library Architecture
 - USB Device Library 3
 - USB Host Library 249
 - Library Configuration
 - USB CDC Device Library 175
 - USB HID Device Library 210
 - Library Initialization 81, 166, 201, 232, 242
 - USB CDC Device Library 166
 - USB HID Device Library 201
 - Library Interface 33, 62, 100, 176, 211, 237, 248, 274, 340, 421, 448, 456, 461
 - Generic USB Device Library 248
 - USB Audio 1.0 Device Library 33
 - USB Audio 2.0 Device Library 62
 - USB Audio v1.0 Host Client Driver Library 340
 - USB CDC Device Library 176
 - USB Device Layer Library 100
 - USB HID Device Library 211
 - USB HID Host Mouse Driver Library 448
 - USB Host Layer Library 274
 - USB Hub Host Client Driver Library 456
 - USB MSD Device Library 237
 - Library Overview 24, 54, 80, 164, 201, 232, 242, 263, 312, 409, 443, 453, 458
 - Generic USB Device Library 242
 - USB Audio 1.0 Device Library 24
 - USB Audio 2.0 Device Library 54
 - USB Audio v1.0 Host Client Driver Library 312
 - USB CDC Device Library 164
 - USB Device Layer Library 80
 - USB HID Device Library 201
 - USB HID Host Mouse Driver Library 443
- M**
- USB Hub Host Client Driver Library 453
 - USB MSD Host Library 458
- O**
- mask enumeration 296
 - Master Descriptor Table 81
 - MLA and MPLAB Harmony USB Device Stack Files 18
 - Mouse Data Event Handling 444
 - MSD TPL Table Configuration 458
- P**
- pid } enumeration 298
 - pid enumeration 297
 - Prerequisites 17
- R**
- Reading and Writing Data 415
 - Receiving a Report 210
 - Receiving Data 174
- S**
- Sending a Report 209
 - Sending Class Specific Control Transfers 335
 - Sending Class-specific Control Transfers 412
 - Sending Data 172
 - Setting the Desired Audio Stream Sampling Rate 326
 - Source Code Analysis 19
 - Source Files to Include 13
 - String Descriptor Table 92
 - subClassCode enumeration 299
- T**
- TPL Table Configuration for Audio v1.0 Devices 312
 - TPL Table Configuration for CDC Devices 410
 - Transferring Data 30, 59
 - true enumeration 300
- U**
- USB Audio 1.0 Device Library 23
 - USB Audio 2.0 Device Library 53
 - USB Audio v1.0 Host Client Driver Library 311
 - USB CDC Device Library 163
 - USB CDC Host Client Driver 258
 - USB CDC Host Library 409
 - USB Device Descriptors 22
 - USB Device Layer Library 79
 - USB Device Library 3
 - USB Device Library - Application Interaction 5
 - USB Device Library - Getting Started 3
 - USB Device Library Architecture 3
 - USB Device Stack in MLA and MPLAB Harmony 18
 - USB Device Stack Porting Example 17
 - USB Device Stack Porting Guide 13
 - USB HID Device Library 199
 - USB HID Host Mouse Driver Library 442

USB Host Layer 252
 USB Host Layer Library 262
 USB Host Library 249
 USB Host Library - Application Interaction 251
 USB Host Library - Getting Started 249
 USB Host Library Architecture 249
 USB Host Library Migration Guide 251
 USB Hub Host Client Driver Library 453
 USB Hub Host Client Driver Test Results 454
 USB Libraries Help 2
 USB MSD Device Library 231
 USB MSD Host Client Driver and SCSI Block Storage Driver 254
 USB MSD Host Client Driver Library 457
 usb_common.h 162
 USB_DATA_DIRECTION enumeration 158
 usb_device.h 160
 USB_DEVICE_ActiveConfigurationGet function 119
 USB_DEVICE_ActiveSpeedGet function 119
 USB_DEVICE_Attach function 117
 USB_DEVICE_AUDIO_EVENT enumeration 39
 USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_CUR type 48
 USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MAX type 48
 USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MEM type 48
 USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_MIN type 48
 USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_GET_RES type 49
 USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_CUR type 49
 USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MAX type 49
 USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MEM type 49
 USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_MIN type 50
 USB_DEVICE_AUDIO_EVENT_DATA_CONTROL_SET_RES type 50
 USB_DEVICE_AUDIO_EVENT_DATA_ENTITY_GET_STAT type 51
 USB_DEVICE_AUDIO_EVENT_DATA_INTERFACE_SETTING_CHANGED structure 50
 USB_DEVICE_AUDIO_EVENT_DATA_READ_COMPLETE structure 45
 USB_DEVICE_AUDIO_EVENT_DATA_WRITE_COMPLETE structure 45
 USB_DEVICE_AUDIO_EVENT_HANDLER type 46
 USB_DEVICE_AUDIO_EVENT_RESPONSE type 46
 USB_DEVICE_AUDIO_EVENT_RESPONSE_NONE macro 39
 USB_DEVICE_AUDIO_EventHandlerSet function 34
 USB_DEVICE_AUDIO_FUNCTION_DRIVER macro 48
 USB_DEVICE_AUDIO_INDEX type 46
 USB_DEVICE_AUDIO_INIT structure 50
 USB_DEVICE_AUDIO_INSTANCES_NUMBER macro 31
 USB_DEVICE_AUDIO_MAX_ALTERNATE_SETTING macro 31
 USB_DEVICE_AUDIO_MAX_STREAMING_INTERFACES macro 31
 USB_DEVICE_AUDIO_QUEUE_DEPTH_COMBINED macro 32
 USB_DEVICE_AUDIO_Read function 35
 USB_DEVICE_AUDIO_RESULT enumeration 47
 USB_DEVICE_AUDIO_TRANSFER_ABORT_NOTIFY macro 51
 USB_DEVICE_AUDIO_TRANSFER_HANDLE type 47
 USB_DEVICE_AUDIO_TRANSFER_HANDLE_INVALID macro 39
 USB_DEVICE_AUDIO_TransferCancel function 36
 usb_device_audio_v1_0.h 51
 usb_device_audio_v1_0_config_template.h 53
 usb_device_audio_v2_0.h 77
 usb_device_audio_v2_0_config_template.h 78
 USB_DEVICE_AUDIO_V2_EVENT enumeration 68
 USB_DEVICE_AUDIO_V2_EVENT_DATA_READ_COMPLETE structure 72
 USB_DEVICE_AUDIO_V2_EVENT_DATA_SET_ALTERNATE_INTERFACE structure 73
 USB_DEVICE_AUDIO_V2_EVENT_DATA_WRITE_COMPLETE structure 73
 USB_DEVICE_AUDIO_V2_EVENT_HANDLER type 74
 USB_DEVICE_AUDIO_V2_EVENT_RESPONSE type 74
 USB_DEVICE_AUDIO_V2_EVENT_RESPONSE_NONE macro 76
 USB_DEVICE_AUDIO_V2_EventHandlerSet function 62
 USB_DEVICE_AUDIO_V2_FUNCTION_DRIVER macro 77
 USB_DEVICE_AUDIO_V2_INDEX type 74
 USB_DEVICE_AUDIO_V2_INIT structure 75
 USB_DEVICE_AUDIO_V2_INSTANCES_NUMBER macro 60
 USB_DEVICE_AUDIO_V2_MAX_ALTERNATE_SETTING macro 60
 USB_DEVICE_AUDIO_V2_MAX_STREAMING_INTERFACES macro 60
 USB_DEVICE_AUDIO_V2_QUEUE_DEPTH_COMBINED macro 61
 USB_DEVICE_AUDIO_V2_Read function 63
 USB_DEVICE_AUDIO_V2_RESULT enumeration 75
 USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE type 76
 USB_DEVICE_AUDIO_V2_TRANSFER_HANDLE_INVALID macro 77
 USB_DEVICE_AUDIO_V2_TransferCancel function 65
 USB_DEVICE_AUDIO_V2_Write function 66
 USB_DEVICE_AUDIO_Write function 37
 USB_DEVICE_BOS_DESCRIPTOR_SUPPORT_ENABLE macro 159
 usb_device_cdc.h 197
 usb_device_cdc_config_template.h 199
 USB_DEVICE_CDC_EVENT enumeration 186
 USB_DEVICE_CDC_EVENT_DATA_READ_COMPLETE structure 190
 USB_DEVICE_CDC_EVENT_DATA_SEND_BREAK structure 192
 USB_DEVICE_CDC_EVENT_DATA_SERIAL_STATE_NOTIFICATION_COMPLETE structure 190
 USB_DEVICE_CDC_EVENT_DATA_WRITE_COMPLETE structure 191
 USB_DEVICE_CDC_EVENT_HANDLER type 191
 USB_DEVICE_CDC_EVENT_RESPONSE type 192
 USB_DEVICE_CDC_EVENT_RESPONSE_NONE macro 195
 USB_DEVICE_CDC_EventHandlerSet function 177
 USB_DEVICE_CDC_FUNCTION_DRIVER macro 195
 USB_DEVICE_CDC_INDEX type 193
 USB_DEVICE_CDC_INDEX_0 macro 196
 USB_DEVICE_CDC_INDEX_1 macro 196
 USB_DEVICE_CDC_INDEX_2 macro 196
 USB_DEVICE_CDC_INDEX_3 macro 196
 USB_DEVICE_CDC_INDEX_4 macro 197
 USB_DEVICE_CDC_INDEX_5 macro 197
 USB_DEVICE_CDC_INDEX_6 macro 197
 USB_DEVICE_CDC_INDEX_7 macro 197
 USB_DEVICE_CDC_INIT structure 193
 USB_DEVICE_CDC_INSTANCES_NUMBER macro 175
 USB_DEVICE_CDC_QUEUE_DEPTH_COMBINED macro 175
 USB_DEVICE_CDC_Read function 181
 USB_DEVICE_CDC_RESULT enumeration 193
 USB_DEVICE_CDC_SerialStateNotificationSend function 185
 USB_DEVICE_CDC_TRANSFER_FLAGS enumeration 194
 USB_DEVICE_CDC_TRANSFER_HANDLE type 195
 USB_DEVICE_CDC_TRANSFER_HANDLE_INVALID macro 195
 USB_DEVICE_CDC_Write function 183

USB_DEVICE_CLIENT_STATUS enumeration 145
USB_DEVICE_ClientStatusGet function 108
USB_DEVICE_Close function 107
usb_device_config_template.h 163
USB_DEVICE_CONFIGURATION_DESCRIPTOR_TABLE type 146
USB_DEVICE_CONTROL_STATUS enumeration 137
USB_DEVICE_CONTROL_TRANSFER_RESULT enumeration 137
USB_DEVICE_ControlReceive function 130
USB_DEVICE_ControlSend function 132
USB_DEVICE_ControlStatus function 134
USB_DEVICE_Deinitialize function 104
USB_DEVICE_Detach function 118
USB_DEVICE_DRIVER_INITIALIZE_EXPLICIT macro 159
USB_DEVICE_ENDPOINT_QUEUE_DEPTH_COMBINED macro 97
USB_DEVICE_EndpointDisable function 122
USB_DEVICE_EndpointEnable function 123
USB_DEVICE_EndpointIsEnabled function 125
USB_DEVICE_EndpointIsStalled function 120
USB_DEVICE_EndpointRead function 126
USB_DEVICE_EndpointStall function 121
USB_DEVICE_EndpointStallClear function 122
USB_DEVICE_EndpointTransferCancel function 127
USB_DEVICE_EndpointWrite function 128
USB_DEVICE_EP0_BUFFER_SIZE macro 99
USB_DEVICE_EVENT enumeration 138
USB_DEVICE_EVENT_DATA_CONFIGURED structure 144
USB_DEVICE_EVENT_DATA_ENDPOINT_READ_COMPLETE structure 148
USB_DEVICE_EVENT_DATA_ENDPOINT_WRITE_COMPLETE structure 149
USB_DEVICE_EVENT_DATA_SET_DESCRIPTOR structure 149
USB_DEVICE_EVENT_DATA_SOF structure 150
USB_DEVICE_EVENT_DATA_SYNCH_FRAME structure 150
USB_DEVICE_EVENT_HANDLER type 146
USB_DEVICE_EVENT_RESPONSE type 146
USB_DEVICE_EVENT_RESPONSE_NONE macro 145
USB_DEVICE_EventHandlerSet function 109
USB_DEVICE_FUNCTION_REGISTRATION_TABLE structure 147
USB_DEVICE_HANDLE type 142
USB_DEVICE_HANDLE_INVALID macro 145
usb_device_hid.h 229
usb_device_hid_config_template.h 231
USB_DEVICE_HID_EVENT enumeration 217
USB_DEVICE_HID_EVENT_DATA_GET_IDLE structure 224
USB_DEVICE_HID_EVENT_DATA_GET_REPORT structure 221
USB_DEVICE_HID_EVENT_DATA_REPORT_RECEIVED structure 222
USB_DEVICE_HID_EVENT_DATA_REPORT_SENT structure 222
USB_DEVICE_HID_EVENT_DATA_SET_IDLE structure 223
USB_DEVICE_HID_EVENT_DATA_SET_PROTOCOL structure 225
USB_DEVICE_HID_EVENT_DATA_SET_REPORT structure 223
USB_DEVICE_HID_EVENT_HANDLER type 225
USB_DEVICE_HID_EVENT_RESPONSE type 225
USB_DEVICE_HID_EVENT_RESPONSE_NONE macro 227
USB_DEVICE_HID_EventHandlerSet function 212
USB_DEVICE_HID_FUNCTION_DRIVER macro 227
USB_DEVICE_HID_INDEX type 224
USB_DEVICE_HID_INDEX_0 macro 228
USB_DEVICE_HID_INDEX_1 macro 228
USB_DEVICE_HID_INDEX_2 macro 228
USB_DEVICE_HID_INDEX_3 macro 228
USB_DEVICE_HID_INDEX_4 macro 229
USB_DEVICE_HID_INDEX_5 macro 229
USB_DEVICE_HID_INDEX_6 macro 229
USB_DEVICE_HID_INDEX_7 macro 229
USB_DEVICE_HID_INIT structure 226
USB_DEVICE_HID_INSTANCES_NUMBER macro 210
USB_DEVICE_HID_QUEUE_DEPTH_COMINED macro 211
USB_DEVICE_HID_ReportReceive function 213
USB_DEVICE_HID_ReportSend function 215
USB_DEVICE_HID_RESULT enumeration 226
USB_DEVICE_HID_TRANSFER_HANDLE type 224
USB_DEVICE_HID_TRANSFER_HANDLE_INVALID macro 227
USB_DEVICE_HID_TransferCancel function 216
USB_DEVICE_INDEX_0 macro 135
USB_DEVICE_INDEX_1 macro 136
USB_DEVICE_INDEX_2 macro 136
USB_DEVICE_INDEX_3 macro 136
USB_DEVICE_INDEX_4 macro 136
USB_DEVICE_INDEX_5 macro 136
USB_DEVICE_INIT structure 142
USB_DEVICE_Initialize function 102
USB_DEVICE_INSTANCES_NUMBER macro 97
USB_DEVICE_IRP structure 152
USB_DEVICE_IRP_FLAG enumeration 153
USB_DEVICE_IRP_STATUS enumeration 153
USB_DEVICE_IsSuspended function 113
USB_DEVICE_MASTER_DESCRIPTOR structure 147
USB_DEVICE_MICROSOFT_OS_DESCRIPTOR_SUPPORT_ENABLE macro 99
usb_device_msd.h 240
usb_device_msd_config_template.h 240
USB_DEVICE_MSD_FUNCTION_DRIVER macro 239
USB_DEVICE_MSD_INIT structure 238
USB_DEVICE_MSD_INSTANCES_NUMBER macro 235
USB_DEVICE_MSD_LUNS_NUMBER macro 236
USB_DEVICE_MSD_MEDIA_FUNCTIONS structure 237
USB_DEVICE_MSD_MEDIA_INIT_DATA structure 239
USB_DEVICE_Open function 106
USB_DEVICE_POWER_STATE enumeration 143
USB_DEVICE_PowerStateSet function 111
USB_DEVICE_REMOTE_WAKEUP_STATUS enumeration 144
USB_DEVICE_RemoteWakeUpStart function 114
USB_DEVICE_RemoteWakeUpStartTimed function 115
USB_DEVICE_RemoteWakeUpStatusGet function 112
USB_DEVICE_RemoteWakeUpStop function 115
USB_DEVICE_RESULT enumeration 150
USB_DEVICE_SET_DESCRIPTOR_EVENT_ENABLE macro 98
USB_DEVICE_SOF_EVENT_ENABLE macro 98
USB_DEVICE_StateGet function 116
USB_DEVICE_Status function 104
USB_DEVICE_STRING_DESCRIPTOR_TABLE_ADVANCED_ENABLE macro 160
USB_DEVICE_STRING_DESCRIPTOR_TABLE type 148
USB_DEVICE_SYNCH_FRAME_EVENT_ENABLE macro 98
USB_DEVICE_Tasks function 105
USB_DEVICE_Tasks_ISR function 106

USB_DEVICE_Tasks_ISR_USBDMA function 106
 USB_DEVICE_TRANSFER_FLAGS enumeration 151
 USB_DEVICE_TRANSFER_HANDLE type 151
 USB_DEVICE_TRANSFER_HANDLE_INVALID macro 152
 USB_ENDPOINT type 154
 USB_ENDPOINT_AND_DIRECTION macro 158
 USB_ERROR enumeration 154
 USB_HID_GLOBAL_PUSH_POP_STACK_SIZE macro 445
 usb_host.h 309
 usb_host_audio_v1_0.h 405
 USB_HOST_AUDIO_V1_0_ATTACH_EVENT_HANDLER type 383
 USB_HOST_AUDIO_V1_0_AttachEventHandlerSet macro 400
 usb_host_audio_v1_0_config_template.h 408
 USB_HOST_AUDIO_V1_0_CONTROL_CALLBACK type 386
 USB_HOST_AUDIO_V1_0_ControlRequest function 344
 USB_HOST_AUDIO_V1_0_DeviceObjHandleGet macro 400
 USB_HOST_AUDIO_V1_0_DIRECTION_IN macro 401
 USB_HOST_AUDIO_V1_0_DIRECTION_OUT macro 401
 USB_HOST_AUDIO_V1_0_EVENT macro 387
 USB_HOST_AUDIO_V1_0_EVENT_ATTACH macro 401
 USB_HOST_AUDIO_V1_0_EVENT_DETACH macro 402
 USB_HOST_AUDIO_V1_0_INTERFACE macro 399
 USB_HOST_AUDIO_V1_0_NumberOfStreamGroupsGet function 358
 USB_HOST_AUDIO_V1_0_OBJ macro 389
 USB_HOST_AUDIO_V1_0_REQUEST_HANDLE macro 389
 USB_HOST_AUDIO_V1_0_REQUEST_HANDLE_INVALID macro 399
 USB_HOST_AUDIO_V1_0_RESULT enumeration 390
 USB_HOST_AUDIO_V1_0_STREAM_DIRECTION macro 391
 USB_HOST_AUDIO_V1_0_STREAM_EVENT enumeration 391
 USB_HOST_AUDIO_V1_0_STREAM_EVENT_DISABLE_COMPLETE_DATA structure 393
 USB_HOST_AUDIO_V1_0_STREAM_EVENT_ENABLE_COMPLETE_D ATA structure 394
 USB_HOST_AUDIO_V1_0_STREAM_EVENT_HANDLER type 394
 USB_HOST_AUDIO_V1_0_STREAM_EVENT_READ_COMPLETE_D ATA macro 394
 USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE macro 395
 USB_HOST_AUDIO_V1_0_STREAM_EVENT_RESPONSE_NONE macro 402
 USB_HOST_AUDIO_V1_0_STREAM_EVENT_WRITE_COMPLETE_D ATA macro 395
 USB_HOST_AUDIO_V1_0_STREAM_HANDLE macro 395
 USB_HOST_AUDIO_V1_0_STREAM_HANDLE_INVALID macro 399
 USB_HOST_AUDIO_V1_0_STREAM_INFO structure 396
 USB_HOST_AUDIO_V1_0_STREAM_OBJ type 397
 USB_HOST_AUDIO_V1_0_STREAM_RESULT enumeration 397
 USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE macro 398
 USB_HOST_AUDIO_V1_0_STREAM_TRANSFER_HANDLE_INVALID macro 400
 USB_HOST_AUDIO_V1_0_StreamClose macro 402
 USB_HOST_AUDIO_V1_0_StreamDisable function 360
 USB_HOST_AUDIO_V1_0_StreamEnable function 361
 USB_HOST_AUDIO_V1_0_StreamEventHandlerSet function 363
 USB_HOST_AUDIO_V1_0_StreamGetFirst function 364
 USB_HOST_AUDIO_V1_0_StreamGetNext function 365
 USB_HOST_AUDIO_V1_0_StreamOpen macro 403
 USB_HOST_AUDIO_V1_0_StreamRead function 376
 USB_HOST_AUDIO_V1_0_StreamSamplingRateSet function 367
 USB_HOST_AUDIO_V1_0_StreamWrite function 377
 USB_HOST_AUDIO_V1_ATTACH_EVENT_HANDLER type 380
 USB_HOST_AUDIO_V1_ATTACH_LISTENERS_NUMBER macro 338
 USB_HOST_AUDIO_V1_AttachEventHandlerSet function 344
 USB_HOST_AUDIO_V1_CONTROL_ENTITY_OBJ type 381
 USB_HOST_AUDIO_V1_ControlEntityGetFirst function 345
 USB_HOST_AUDIO_V1_ControlEntityGetNext function 346
 USB_HOST_AUDIO_V1_DeviceObjHandleGet function 347
 USB_HOST_AUDIO_V1_ENTITY_REQUEST_CALLBACK type 381
 USB_HOST_AUDIO_V1_EntityObjectGet function 347
 USB_HOST_AUDIO_V1_EntityRequestCallbackSet function 348
 USB_HOST_AUDIO_V1_EntityTypeGet function 348
 USB_HOST_AUDIO_V1_EVENT enumeration 381
 USB_HOST_AUDIO_V1_FeatureUnitChannelMuteExists function 349
 USB_HOST_AUDIO_V1_FeatureUnitChannelMuteGet function 350
 USB_HOST_AUDIO_V1_FeatureUnitChannelMuteSet function 350
 USB_HOST_AUDIO_V1_FeatureUnitChannelNumbersGet function 351
 USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeExists function 352
 USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeGet function 352
 USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeRangeGet function 378
 USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSet function 353
 USB_HOST_AUDIO_V1_FeatureUnitChannelVolumeSubRangeNumbers Get function 379
 USB_HOST_AUDIO_V1_FeatureUnitIDGet function 354
 USB_HOST_AUDIO_V1_FeatureUnitSourceIDGet function 355
 USB_HOST_AUDIO_V1_INSTANCES_NUMBER macro 338
 USB_HOST_AUDIO_V1_INTERFACE macro 396
 USB_HOST_AUDIO_V1_OBJ type 382
 USB_HOST_AUDIO_V1_REQUEST_HANDLE type 382
 USB_HOST_AUDIO_V1_REQUEST_HANDLE_INVALID macro 396
 USB_HOST_AUDIO_V1_RESULT enumeration 382
 USB_HOST_AUDIO_V1_SAMPLING_FREQUENCIES_NUMBER macro 404
 USB_HOST_AUDIO_V1_STREAM_DIRECTION enumeration 384
 USB_HOST_AUDIO_V1_STREAM_EVENT enumeration 384
 USB_HOST_AUDIO_V1_STREAM_EVENT_HANDLER type 385
 USB_HOST_AUDIO_V1_STREAM_EVENT_INTERFACE_SET_COMPLETE_DATA structure 386
 USB_HOST_AUDIO_V1_STREAM_EVENT_READ_COMPLETE_DATA structure 387
 USB_HOST_AUDIO_V1_STREAM_EVENT_RESPONSE enumeration 388
 USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_GET_COMPLETE_DATA structure 404
 USB_HOST_AUDIO_V1_STREAM_EVENT_SAMPLING_RATE_SET_COMPLETE DATA structure 388
 USB_HOST_AUDIO_V1_STREAM_EVENT_WRITE_COMPLETE_DATA structure 389
 USB_HOST_AUDIO_V1_STREAM_HANDLE type 390
 USB_HOST_AUDIO_V1_STREAM_HANDLE_INVALID macro 397
 USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE type 391
 USB_HOST_AUDIO_V1_STREAM_TRANSFER_HANDLE_INVALID macro 398

USB_HOST_AUDIO_V1_StreamClose function 359
USB_HOST_AUDIO_V1_StreamEventHandlerSet function 359
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_ALTERNATE_SET_TINGS_NUMBER
macro 338
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_OBJ type 391
USB_HOST_AUDIO_V1_STREAMING_INTERFACE_SETTING_OBJ
type 393
USB_HOST_AUDIO_V1_STREAMING_INTERFACES_NUMBER macro
339
USB_HOST_AUDIO_V1_StreamingInterfaceBitResolutionGet function
370
USB_HOST_AUDIO_V1_StreamingInterfaceChannelNumbersGet
function 370
USB_HOST_AUDIO_V1_StreamingInterfaceDirectionGet function 371
USB_HOST_AUDIO_V1_StreamingInterfaceFormatTagGet function 372
USB_HOST_AUDIO_V1_StreamingInterfaceGetFirst function 361
USB_HOST_AUDIO_V1_StreamingInterfaceGetNext function 362
USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequenciesGet
function 372
USB_HOST_AUDIO_V1_StreamingInterfaceSamplingFrequencyTypeGe
t
function 373
USB_HOST_AUDIO_V1_StreamingInterfaceSet function 363
USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetFirst function 365
USB_HOST_AUDIO_V1_StreamingInterfaceSettingGetNext function 366
USB_HOST_AUDIO_V1_StreamingInterfaceSubFrameSizeGet function
374
USB_HOST_AUDIO_V1_StreamingInterfaceTerminalLinkGet function
374
USB_HOST_AUDIO_V1_StreamOpen function 368
USB_HOST_AUDIO_V1_StreamRead function 368
USB_HOST_AUDIO_V1_StreamSamplingFrequencyGet function 375
USB_HOST_AUDIO_V1_StreamSamplingFrequencySet function 376
USB_HOST_AUDIO_V1_StreamWrite function 369
USB_HOST_AUDIO_V1_TerminalAssociationGet function 355
USB_HOST_AUDIO_V1_TerminalIDGet function 356
USB_HOST_AUDIO_V1_TerminalInputChannelConfigGet function 378
USB_HOST_AUDIO_V1_TerminalInputChannelNumbersGet function
356
USB_HOST_AUDIO_V1_TerminalSourceIDGet function 357
USB_HOST_AUDIO_V1_TerminalTypeGet function 358
USB_HOST_BUS type 300
USB_HOST_BUS_ALL macro 307
USB_HOST_BusEnable function 277
USB_HOST_BusIsEnabled function 278
USB_HOST_BusIsSuspended function 278
USB_HOST_BusResume function 279
USB_HOST_BusSuspend function 279
usb_host_cdc.h 441
usb_host_cdc_acm.h 442
USB_HOST_CDC_ACM_BreakSend function 428
USB_HOST_CDC_ACM_ControlLineStateSet function 428
USB_HOST_CDC_ACM_LineCodingGet function 429
USB_HOST_CDC_ACM_LineCodingSet function 430
USB_HOST_CDC_ATTACH_EVENT_HANDLER type 434
USB_HOST_CDC_ATTACH_LISTENERS_NUMBER macro 419
USB_HOST_CDC_AttachEventHandlerSet function 423
USB_HOST_CDC_Close function 422
usb_host_cdc_config_template.h 442
USB_HOST_CDC_DeviceObjHandleGet function 424
USB_HOST_CDC_EVENT enumeration 431
USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_D
ATA
structure 434
USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE_DATA
structure 435
USB_HOST_CDC_EVENT_ACM_SET_CONTROL_LINE_STATE_COMPLETE
DATA
structure 436
USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_D
ATA
structure 436
USB_HOST_CDC_EVENT_HANDLER type 437
USB_HOST_CDC_EVENT_READ_COMPLETE_DATA structure 437
USB_HOST_CDC_EVENT_RESPONSE enumeration 438
USB_HOST_CDC_EVENT_SERIAL_STATE_NOTIFICATION_RECEIVE
D_DATA
structure 438
USB_HOST_CDC_EVENT_WRITE_COMPLETE_DATA structure 438
USB_HOST_CDC_EventHandlerSet function 424
USB_HOST_CDC_HANDLE type 435
USB_HOST_CDC_HANDLE_INVALID macro 440
USB_HOST_CDC_INSTANCES_NUMBER macro 420
USB_HOST_CDC_INTERFACE macro 440
USB_HOST_CDC_OBJ type 439
USB_HOST_CDC_Open function 422
USB_HOST_CDC_Read function 425
USB_HOST_CDC_REQUEST_HANDLE type 439
USB_HOST_CDC_REQUEST_HANDLE_INVALID macro 440
USB_HOST_CDC_RESULT enumeration 433
USB_HOST_CDC_SerialStateNotificationGet function 426
USB_HOST_CDC_TRANSFER_HANDLE type 433
USB_HOST_CDC_TRANSFER_HANDLE_INVALID macro 434
USB_HOST_CDC_Write function 427
usb_host_config_template.h 311
USB_HOST_CONTROLLERS_NUMBER macro 270
USB_HOST_Deinitialize function 275
USB_HOST_DEVICE_INFO structure 301
USB_HOST_DEVICE_INTERFACES_NUMBER macro 271
USB_HOST_DEVICE_OBJ_HANDLE type 301
USB_HOST_DEVICE_OBJ_HANDLE_INVALID macro 308
USB_HOST_DEVICE_STRING enumeration 301
USB_HOST_DEVICE_STRING_LANG_ID_DEFAULT macro 308
USB_HOST_DeviceGetFirst function 280
USB_HOST_DeviceGetNext function 281
USB_HOST_DeviceIsSuspended function 281
USB_HOST_DeviceResume function 282
USB_HOST_DEVICES_NUMBER macro 271
USB_HOST_DeviceSpeedGet function 283
USB_HOST_DeviceStringDescriptorGet function 283
USB_HOST_DeviceSuspend function 284
USB_HOST_EVENT enumeration 302
USB_HOST_EVENT_HANDLER type 303
USB_HOST_EVENT_RESPONSE enumeration 287
USB_HOST_EventHandlerSet function 285
USB_HOST_HCD structure 303
usb_host_hid_config_template.h 452
USB_HOST_HID_INSTANCES_NUMBER macro 446
USB_HOST_HID_INTERRUPT_IN_ENDPOINTS_NUMBER macro 446
usb_host_hid_mouse.h 452

USB_HOST_HID_MOUSE_BUTTONS_NUMBER macro 446
 USB_HOST_HID_MOUSE_DATA structure 449
 USB_HOST_HID_MOUSE_EVENT enumeration 450
 USB_HOST_HID_MOUSE_EVENT_HANDLER type 450
 USB_HOST_HID_MOUSE_EventHandlerSet function 448
 USB_HOST_HID_MOUSE_HANDLE type 450
 USB_HOST_HID_MOUSE_HANDLE_INVALID macro 451
 USB_HOST_HID_MOUSE_RESULT enumeration 450
 USB_HOST_HID_MOUSE_RESULT_MIN macro 451
 USB_HOST_HID_USAGE_DRIVER_SUPPORT_NUMBER macro 447
 usb_host_hub.h 456
 usb_host_hub_config_template.h 457
 USB_HOST_HUB_INSTANCES_NUMBER macro 454
 USB_HOST_HUB_INTERFACE macro 456
 USB_HOST_HUB_PORTS_NUMBER macro 455
 USB_HOST_HUB_SUPPORT_ENABLE macro 271
 USB_HOST_HUB_TIER_LEVEL macro 272
 USB_HOST_INIT structure 286
 USB_HOST_Initialize function 286
 USB_HOST_IRP structure 156
 USB_HOST_IRP_FLAG enumeration 156
 USB_HOST_IRP_STATUS enumeration 157
 USB_HOST_IRP_STATUS_ABORTED enumeration member 157
 USB_HOST_IRP_STATUS_COMPLETED enumeration member 157
 USB_HOST_IRP_STATUS_COMPLETED_SHORT enumeration member 157
 USB_HOST_IRP_STATUS_ERROR_BUS enumeration member 157
 USB_HOST_IRP_STATUS_ERROR_DATA enumeration member 157
 USB_HOST_IRP_STATUS_ERROR_NAK_TIMEOUT enumeration member 157
 USB_HOST_IRP_STATUS_ERROR_STALL enumeration member 157
 USB_HOST_IRP_STATUS_ERROR_UNKNOWN enumeration member 157
 USB_HOST_IRP_STATUS_IN_PROGRESS enumeration member 157
 USB_HOST_IRP_STATUS_PENDING enumeration member 157
 usb_host_msd.h 468
 usb_host_msd_config_template.h 469
 USB_HOST_MSD_ERROR_CODE enumeration 466
 USB_HOST_MSD_INSTANCES_NUMBER macro 460
 USB_HOST_MSD_INTERFACE macro 465
 USB_HOST_MSD_LUN_HANDLE type 466
 USB_HOST_MSD_LUN_HANDLE_INVALID macro 466
 USB_HOST_MSD_LUNS_NUMBER macro 460
 USB_HOST_MSD_RESULT enumeration 463
 USB_HOST_MSD_Transfer function 462
 USB_HOST_MSD_TRANSFER_CALLBACK type 464
 USB_HOST_MSD_TRANSFER_DIRECTION enumeration 465
 USB_HOST_MSD_TRANSFER_HANDLE type 465
 USB_HOST_MSD_TRANSFER_HANDLE_INVALID macro 466
 USB_HOST_MSD_TransferErrorTasks function 463
 USB_HOST_PIPES_NUMBER macro 272
 USB_HOST_REQUEST_HANDLE type 303
 USB_HOST_REQUEST_HANDLE_INVALID macro 308
 USB_HOST_RESULT enumeration 304
 USB_HOST_RESULT_MIN macro 309
 USB_HOST_Status function 276
 USB_HOST_STRING_REQUEST_COMPLETE_CALLBACK type 305
 USB_HOST_TARGET_PERIPHERAL_LIST_ENTRY enumeration 305
 USB_HOST_Tasks function 276
 USB_HOST_TPL_ENTRY enumeration 306
 USB_HOST_TRANSFERS_NUMBER macro 272
 USB_SPEED enumeration 158
 Using the Library 24, 53, 79, 163, 199, 231, 241, 263, 311, 409, 443, 453, 457
 Generic USB Device Library 241
 USB Audio 1.0 Device Library 24
 USB Audio 2.0 Device Library 53
 USB Audio v1.0 Host Client Driver Library 311
 USB CDC Device Library 163
 USB Device Layer Library 79
 USB HID Device Library 199
 USB HID Host Mouse Driver Library 443
 USB Hub Host Client Driver Library 453
 USB MSD Device Library 231
 USB MSD Host Library 457

V

vid enumeration 307