MPLAB Harmony USB Stack

MPLAB Harmony USB Stack Help Document

USB Demonstrations Help

This section provides descriptions of the USB demonstrations.

Introduction

USB Library Demonstration Applications Help

Description

This distribution package contains a variety of USB-related firmware projects that demonstrate the capabilities of the MPLAB Harmony USB stack. This section describes the hardware requirement and procedures to run these firmware projects on Microchip demonstration and development boards.

To know more about the MPLAB Harmony USB stack, USB Stack Configuration and the USB Stack APIs, refer to the USB Library Help Section.

USB MSD Host USB Pen Drive Tests

Provides pen drive test specifications.

Description

USB MSD Host USB Pen Drive Tests

The following table lists the commercially available USB pen drives, which have been tested to successfully enumerate with the MSD Host Driver in the MPLAB Harmony USB Host Stack. Note that if the USB pen drive you are using in not included in the table, this indicates that this USB pen drive has not been tested with the MSD Host Driver. However, the USB pen drive could still potentially work with MSD Host Driver. Some USB pen drives in this table did not have their manufacturer or model data available. The USB Pen drives were tested with the msd_basic USB Host demonstration in the latest version of the MPLAB Harmony USB Host Stack.

Manufacturer	Capacity	VID	PID
Verico Tseres	16GB	0x8644	0x8003
Hewlett Packard	16GB	0x03F0	0x5A07
Freescale	1GB	0x2008	0x2018
Imation Corp	16GB	0x0718	0x0704
ITE Tech Inc.	16GB	0x048D	0x0100
Silicon Motion Inc - Taiwan	8GB	0x090C	0x1000
Kingston Technology (Dell)	16GB	0x0951	0x16A7
Verbatim Americas LLC	8GB	0x18A5	0x0245
Apacer Technology	8GB	0x1005	0x0100
Sony Corporation	8GB	0x054C	0x06B0
Sony Coporation	8GB	0x054C	0x0862
Silicon Motion Inc.	4GB	0x090C	0x1000
SanDisk Corporation	16GB	0x0781	0x0127
Etron Technology Inc	16GB	0x1E4E	0x3257
Verbatim Americas LLC	4GB	0x18A5	0x0100
Appotech Limited	4GB	0x1908	0x1320
Decorative Pen Drive	4GB	0xABCD	0x1234
Moser Baer India Ltd.	16GB	0x1EC9	0x0101
Lexar Media Inc	4GB	0x05DC	0x1100
Realtek Semiconductor Corp.	16GB	0x0BDA	0x0109
Silicon Motion Inc. Taiwan	4/8/16GB	0x090C	0x1000

Kingston Technology Company	16GB	0x0951	0x1665
Kingston Data Traveller	4GB	0x0951	0x1643
Lexar Media Inc	32GB	0x05DC	0xA838
SanDisk Corporation	16GB	0x0781	0x5583
SanDisk Corporation	8GB	0x0781	0x5571
Toshiba Corporation	32GB	0x0930	0x6544
Strontium	8GB	0x090C	0x1000
Kingston Technology Company	16GB	0x0951	0x1666
Phison Electronics Corp	32GB	0x13FE	0x6300
Kingston Technology Company	32GB	0x0951	0x16A3
SSP	4GB	0x8644	0x800B
Toshiba Corporation	2GB	0x0930	0x6544

Device Demonstrations

This section describes the USB Device demonstrations.

Description

The section provides a description of all of the USB Device Stack Demonstration Applicaions contained in this package. The description of the each demonstration application contains instructions for compiling the application project, hardware configuration and interaction guidelines and running the demonstration application. Note that a demonstration application may contain more than one MPLAB X project. All projects demonstrate the same application functionality on different hardware platforms and different application configuration.

cdc_com_port_dual

Demonstrates a USB CDC device, emulating dual serial COM ports - one looping back into the other.

Description

This demonstration application creates a USB CDC Device that enumerates as two serial ports on the USB Host PC. This application demonstrates the ability of the MPLAB Harmony USB Device Stack to support multiple instances of the same device class.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/device/cdc_com_port_dual/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a Dual COM Port CDC USB Device application with freeRTOS on a ATSAME70Q21B device.

`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal(non RTOS) Dual COM Port CDC USB Device application on a ATSAME70Q21B device.
`sam_v71_xult_freertos.X`	ATSAMV71Q21B	SAMV71 Xplained Ultra	This project implements a bare-metal(non RTOS) Dual COM Port CDC USB Device application on a ATSAMV71Q21B device.
`sam_v71_xult.X`	ATSAMV71Q21B	SAMV71 Xplained Ultra	This project implements a Dual COM Port CDC USB Device application with freeRTOS on a ATSAMV71Q21B device
`sam_d21_xpro.X`	ATSAMD21J18A	SAM D21 Xplained Pro	This project implements a bare-metal(non RTOS) Dual COM Port CDC USB Device application on a ATSAMD21J18A device.
`sam_e54_xpro.X`	ATSAME54P20A	SAM E54 Xplained Pro	This project implements a bare-metal(non RTOS) Dual COM Port CDC USB Device application on a ATSAME54P20A device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and VBUS (positions 2 and 3).
- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host PC.

SAMV71 Xplained Ultra

- Jumper titled "USB VBUS" must be shorted between PC09 and VBUS (positions 2 and 3)
- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

SAMD21 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

SAME54 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

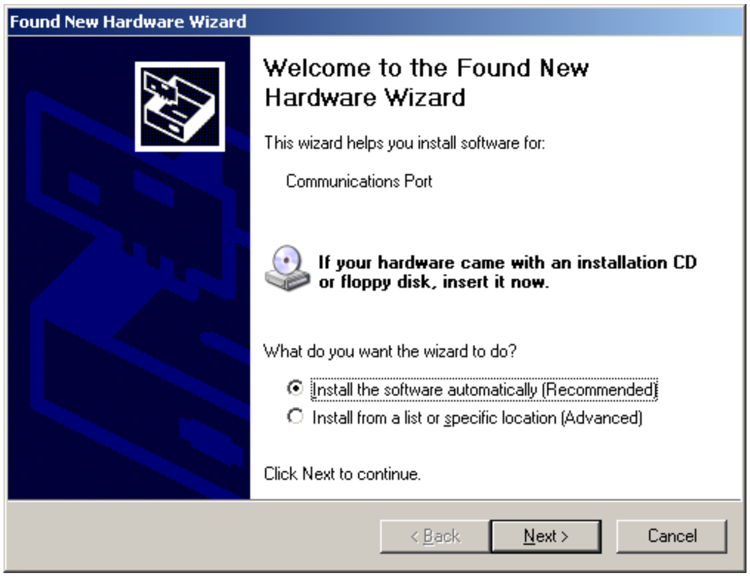
Running the Demonstration

Provides instructions on how to build and run the CDC Dual COM Port demonstration.

Description

This demonstration allows the device to appear like two serial (COM) ports to the host. Do the following to run this demonstration:

1. First compile and program the target device. While compiling, select the appropriate MPLAB X IDE project based on the demonstration board. Refer to Building the Application for details.
2. Attach the device to the host. If the host is a personal computer and this is the first time you have plugged this device into the computer you may be prompted for a `.inf` file.



Welcome to the Found New Hardware Wizard

This wizard helps you install software for:

Communications Port

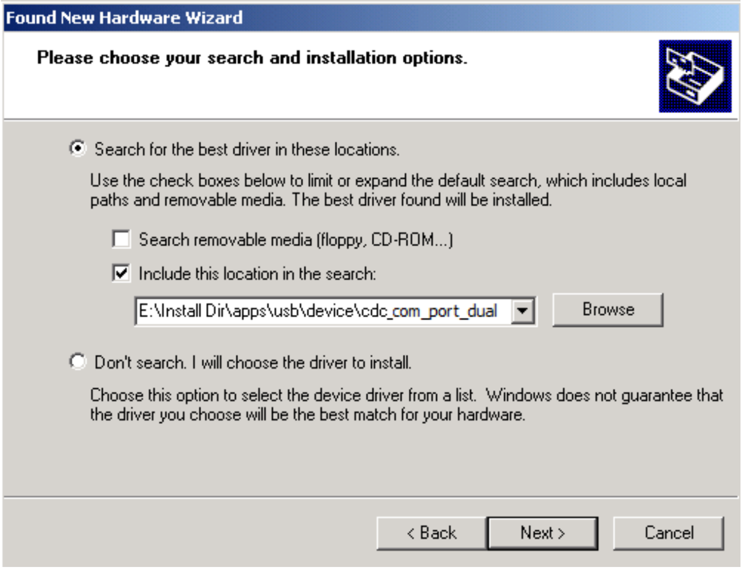
If your hardware came with an installation CD or floppy disk, insert it now.

What do you want the wizard to do?

● Install the software automatically (Recommended)
○ Install from a list or specific location (Advanced)

Click Next to continue.

3. Select the "Install from a list or specific location (Advanced)" option. Specify the `<install-dir>usb/apps/device/cdc_com_port_dual/inf` directory.



Please choose your search and installation options.

● Search for the best driver in these locations.

Use the check boxes below to limit or expand the default search, which includes local paths and removable media. The best driver found will be installed.

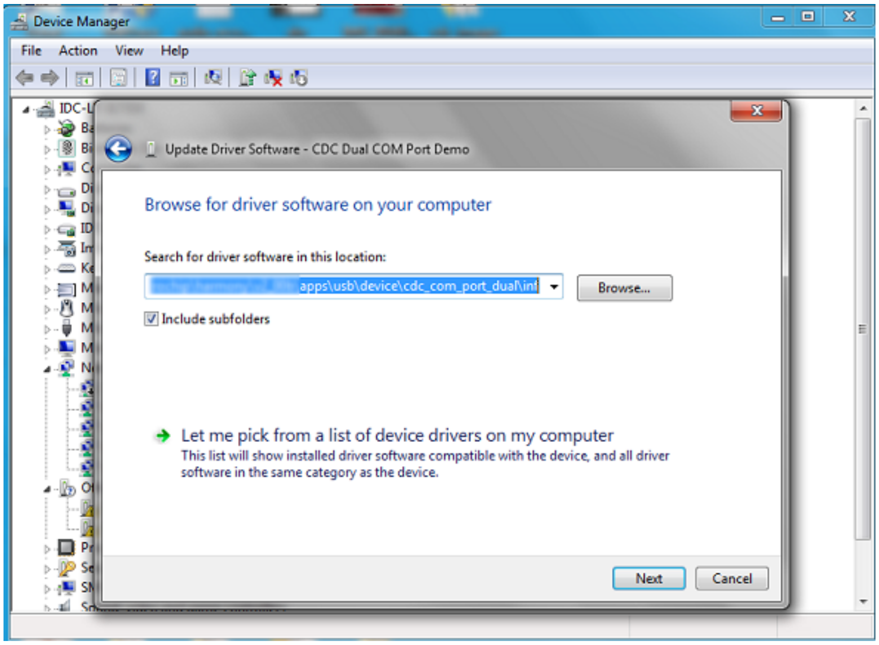
☐ Search removable media (floppy, CD-ROM...)
☑ Include this location in the search:

E:\Install Dir\apps\usb\device\cdc_com_port_dual

○ Don't search. I will choose the driver to install.

Choose this option to select the device driver from a list. Windows does not guarantee that the driver you choose will be the best match for your hardware.

Note: As an option, to specify the driver, you may open the device manager and expand the Ports (COM & LPT) tab, and right click on "Update Driver Software…"



Update Driver Software - CDC Dual COM Port Demo

Browse for driver software on your computer

Search for driver software in this location:

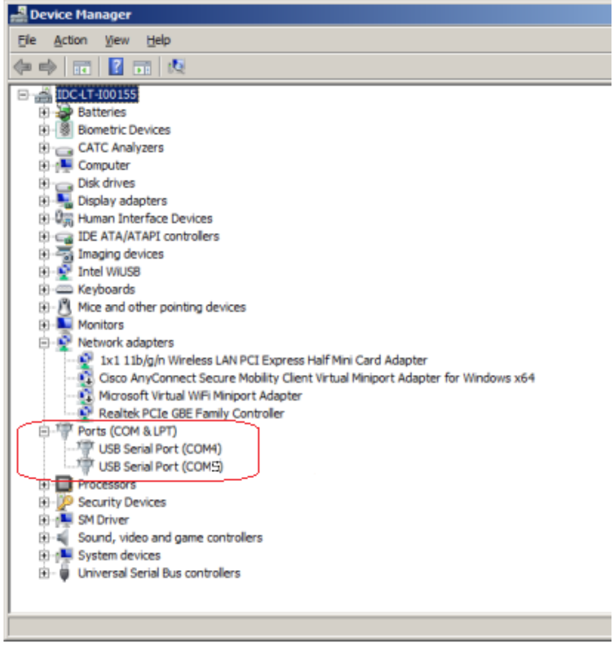
[redacted]apps\usb\device\cdc_com_port_dual\inf

☑ Include subfolders

Let me pick from a list of device drivers on my computer

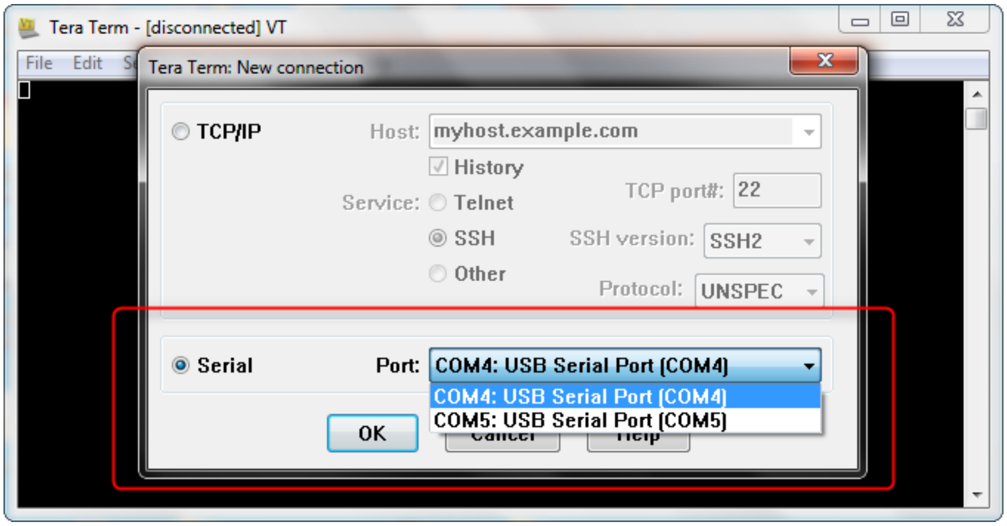
This list will show installed driver software compatible with the device, and all driver software in the same category as the device.

Verify that the enumerated USB device is seen as a virtual USB serial comport in Device Manager.



IDC-LT-I00155
- Batteries
- Biometric Devices
- CATC Analyzers
- Computer
- Disk drives
- Display adapters
- Human Interface Devices
- IDE ATA/ATAPI controllers
- Imaging devices
- Intel WiUSB
- Keyboards
- Mice and other pointing devices
- Monitors
- Network adapters
  - 1x1 11b/g/n Wireless LAN PCI Express Half Mini Card Adapter
  - Cisco AnyConnect Secure Mobility Client Virtual Miniport Adapter for Windows x64
  - Microsoft Virtual WiFi Miniport Adapter
  - Realtek PCIe GBE Family Controller
- Ports (COM & LPT)
  - USB Serial Port (COM4)
  - USB Serial Port (COM5)
- Processors
- Security Devices
- SM Driver
- Sound, video and game controllers
- System devices
- Universal Serial Bus controllers

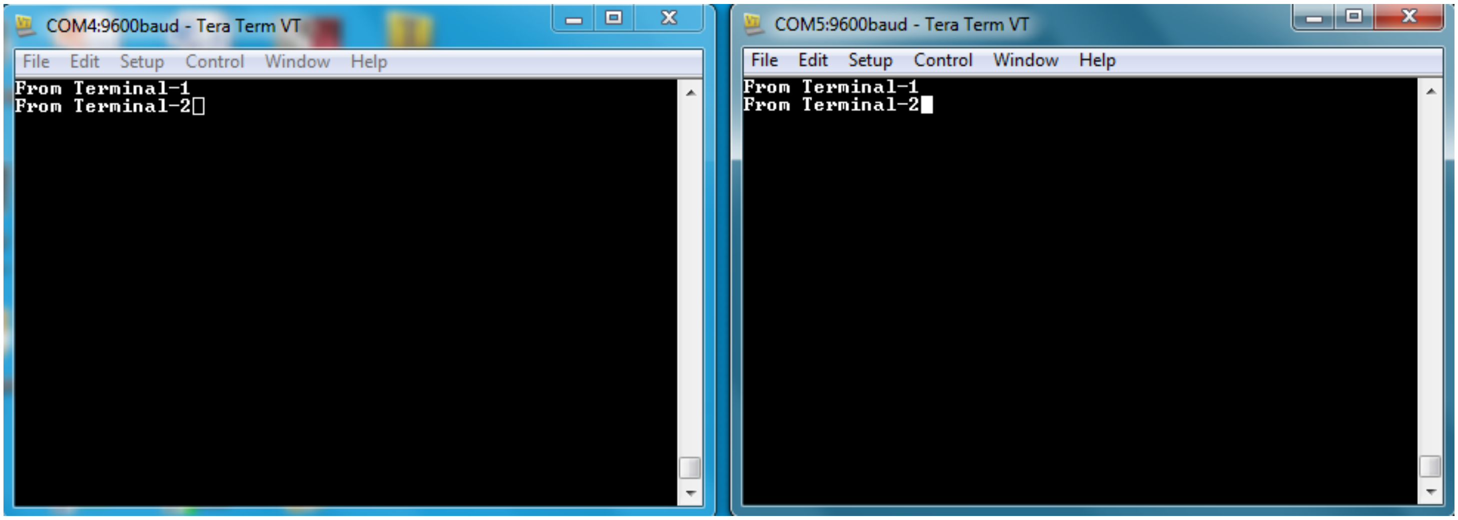
4. Once the device is successfully installed, open up two instances of a terminal program, such as HyperTerminal. Select the appropriate COM port for each of these terminal instances. The following screen shot shows the COM port selection for the Tera Term terminal program.



○ TCP/IP Host: myhost.example.com
☑ History
Service: ○ Telnet TCP port#: 22
● SSH SSH version: SSH2
○ Other Protocol: UNSPEC

● Serial Port: COM4: USB Serial Port (COM4)
COM4: USB Serial Port (COM4)
COM5: USB Serial Port (COM5)

5. The LEDs on the demonstration board will indicate the USB state of the device. Refer to the Configuring the Hardware section for the selected board for more details.
6. To run the demonstration, turn on local echo on both the terminals. For Tera Term terminal application, navigate to Setup->Terminal to turn on local echo. Type a character or string in one terminal window. The same character or string appears on the second terminal window. Similarly, any character typed in the second window appears in the first window. The following screen shot shows two instances of Tera Term.


COM4:9600baud - Tera Term VT
File Edit Setup Control Window Help
COM5:9600baud - Tera Term VT
File Edit Setup Control Window Help

```
From Terminal-1
From Terminal-2
```

```
From Terminal-1
From Terminal-2
```

Note: Some terminal programs, like HyperTerminal, require users to click the disconnect button before removing the device from the computer. Failing to do so may result in having to close and open the program again to reconnect to the device.

cdc_com_port_single

Demonstrates a USB CDC device, emulating a serial COM port.

Description

This demonstration application creates a USB CDC Device that enumerates as a single COM port on the USB host PC. The application demonstrates two-way communication between the USB device and the USB Host PC.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/device/cdc_com_port_single/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a Single COM Port CDC USB Device application with freeRTOS on ATSAME70Q21B device.
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal (no RTOS) Single COM Port CDC USB Device application on the ATSAME70Q21B device.
`sam_d21_xpro.X`	ATSAMD21J18A	SAMD1 Xplained PRO	This project implements a bare-metal (no RTOS) Single COM Port CDC USB Device application on the ATSAMD21J18A device.
`sam_e54_xpro.X`	ATSAME54P20A	SAM E54 Xplained Pro	This project implements a bare-metal(non RTOS) Dual COM Port CDC USB Device application on a ATSAME54P20A device.
`pic32mz_ef_sk.X`	PIC32MZ2048EFH144	PIC32MZ EF Starter Kit	This project implements a bare-metal (no RTOS) Single COM Port CDC USB Device application on the PIC32MZ2048EFH144 device.
`pic32mz_ef_sk_freertos.X`	PIC32MZ2048EFH144	PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit	This project implements a Single COM Port CDC USB Device application with freeRTOS on PIC32MZ2048EFH144 device.
`pic32mz_das_sk.X`	PIC32MZ2064DAS169	PIC32MZ Embedded Graphics with Stacked DRAM (DA) Starter Kit (Crypto)	This project implements a bare-metal (no RTOS) Single COM Port CDC USB Device application on the PIC32MZ2064DAS169 device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and VBUS (positions 2 and 3).
- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Press SW1 to trigger communication from the USB Device to the USB Host.
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host PC.

SAMD21 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Press SW0 to trigger communication from the USB Device to the USB Host.
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

SAME54 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Press SW0 to trigger communication from the USB Device to the USB Host.
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit

- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Press SW1 to trigger communication from the USB Device to the USB Host.
- Use the micro-A/B port J4 (which is located on the bottom side of the board) to connect the USB Device to the the USB Host PC.

PIC32MZ Embedded Graphics with Stacked DRAM (DA) Starter Kit

- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Press SW1 to trigger communication from the USB Device to the USB Host.
- Use the micro-A/B port J6 (which is located on the bottom side of the board) to connect the USB Device to the the USB Host PC.

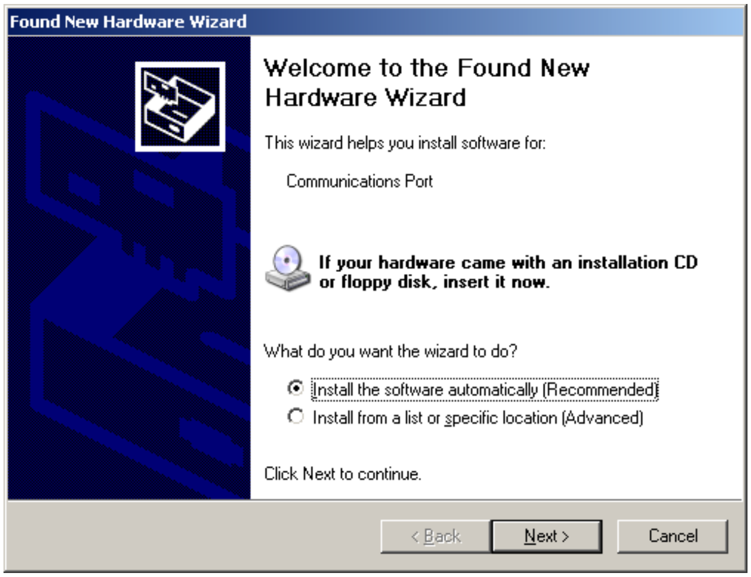
Running the Demonstration

Provides instructions on how to build and run the CDC Single COM Port demonstration.

Description

This demonstration creates a USB device that enumerates as a serial (COM) port on the host. Do the following to run this demonstration:

1. First compile and program the target device. While compiling, select the appropriate MPLAB X IDE project based on the demonstration board. Refer to Building the Application for details.
2. Attach the device to the host. If the host is a personal computer and this is the first time you have plugged this device into the computer, you may be prompted for a `.inf` file.



Welcome to the Found New Hardware Wizard

This wizard helps you install software for:

Communications Port

If your hardware came with an installation CD or floppy disk, insert it now.

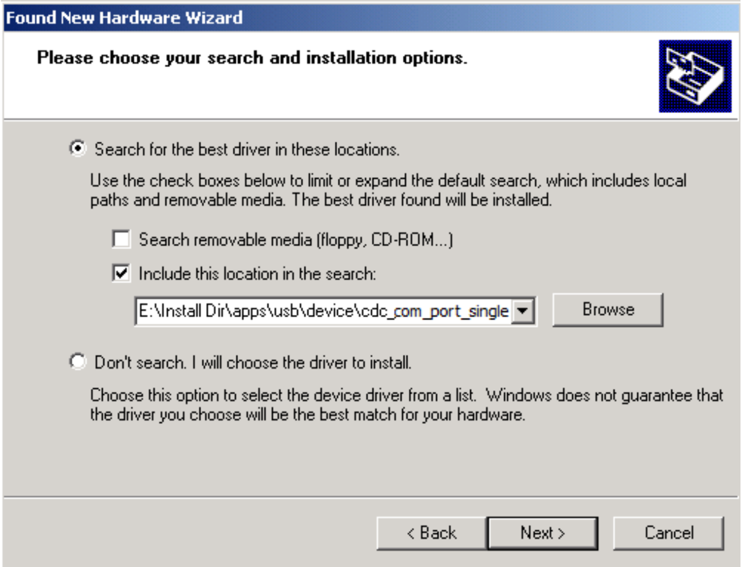
What do you want the wizard to do?

● Install the software automatically (Recommended)

○ Install from a list or specific location (Advanced)

Click Next to continue.

3. Select the "Install from a list or specific location (Advanced)" option. Specify the `<install-dir>/usb/apps/device/cdc_com_port_single/inf` directory.



Please choose your search and installation options.

● Search for the best driver in these locations.

Use the check boxes below to limit or expand the default search, which includes local paths and removable media. The best driver found will be installed.

□ Search removable media (floppy, CD-ROM...)

☑ Include this location in the search:

E:\Install Dir\apps\usb\device\cdc_com_port_single

○ Don't search. I will choose the driver to install.

Choose this option to select the device driver from a list. Windows does not guarantee that the driver you choose will be the best match for your hardware.

4. Once the device is successfully installed, open up a terminal program, such as HyperTerminal and select the appropriate COM port. On most machines this will be COM5 or higher. Set the communication properties to 9600 baud, 1 Stop bit and No parity, with Flow Control set to None.
5. The LEDs on the demonstration board will indicate the USB state of the device, as described in the Configuring the Hardware section.
6. Once connected to the device, there are two ways to run this example project:
   - a) Typing a key in the terminal window will result in the attached device echoing the next letter. Therefore, if the letter 'b' is pressed, the device will echo 'c'.
   - b) If the push button is pressed, the device will echo "PUSH BUTTON PRESSED" to the terminal window. Refer to the Configuring the Hardware section to identify the switch to pressed for the selected target board.

Note: Some terminal programs, like HyperTerminal, require users to click the disconnect button before removing the device from the computer. Failing to do so may result in having to close and open the program again to reconnect to the device.

cdc_msd_basic

Demonstrates a composite USB Device that enumerates as a COM port and as Flash drive simultaneously.

Description

This demonstration application creates a composite USB Device that enumerates as a COM port and as Flash drive simultaneously.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/device/cdc_msd_basic/firmware` folder and provides a description of each project. All projects in listed this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements the CDC Serial Emulator application with freeRTOS on the SAME70Q21B device.
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal (non RTOS) CDC Serial Emulator application on the SAME70Q21B device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and VBUS (positions 2 and 3).
- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host PC.

Running the Demonstration

Provides instructions on how to build and run the cdc_msd_basic demonstration.

Description

This demonstration application creates a composite USB Device that works simultaneously as a CDC and as a MSD device. This application combines the functionality of the cdc_com_port_single and msd_basic demonstration applications into one device.

Refer to Running the Demonstration section of the cdc_com_port_single demonstration and the Running the Demonstration section of the msd_basic demonstration for details on exercising the CDC and MSD device features, respectively.

The LEDs on the demonstration board will indicate the USB state of the device.

cdc_serial_emulator

This application demonstrates the use of the CDC device class to implement a USB-to-Serial Convertor.

Description

This application demonstrates the use of the CDC device class to implemetn a USB-to-Serial Convertor. The application enumerates a COM port on the personal computer. Data received through the CDC USB interface is forwarded to a UART. Data received on the UART is forwarded to the CDC USB interface on the target board. The interface between the UART and the CDC USB Interface on the board demonstrates the USB to UART capability.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/device/cdc_serial_emulator/firmware` folder and provides a description of each project. All projects in listed this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements the CDC Serial Emulator application with freeRTOS on the SAME70Q21B device.
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal (non RTOS) CDC Serial Emulator application on the SAME70Q21B device.
`sam_d21_xpro.X`	ATSAMD21J18A	SAM D21 Xplained Pro	This project implements a bare-metal (non RTOS) CDC Serial Emulator application on the ATSAMD21J18A device.
`sam_e54_xpro.X`	ATSAME54P20A	SAM E54 Xplained Pro	This project implements a bare-metal (non RTOS) CDC Serial Emulator application on the ATSAME54P20A device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and VBUS (positions 2 and 3).
- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- The CDC USB (UART to USB) interface on this board is served by the EDBG connector J300. Connecting this to the PC will create a COM port on the PC Host.
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host PC.

SAMD21 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- The CDC USB (UART to USB) interface on this board is served by the EDBG connector (DEBUG_USB). Connecting this to the PC will create a COM port on the PC Host.
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

SAME54 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- The CDC USB (UART to USB) interface on this board is served by the EDBG connector (DEBUG_USB). Connecting this to the PC will create a COM port on the PC Host.
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

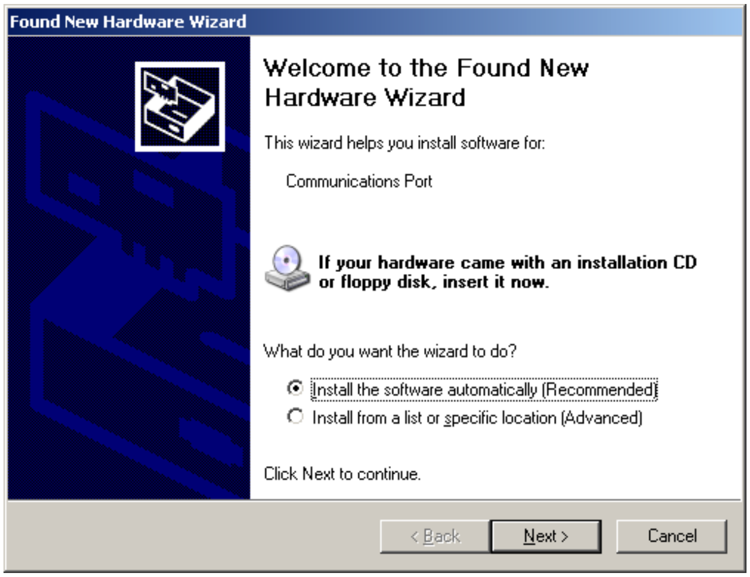
Running the Demonstration

Provides instructions on how to build and run the CDC Serial Emulator Demonstration.

Description

This application demonstrates the use of the CDC Device class in implementing a USB-to-Serial Converter. The application enumerates a COM port on the USB Host PC. Data received through the CDC USB interface is forwarded to a UART. Data received on the UART is forwarded to the CDC USB interface. This emulates a USB-to-Serial Converter.

1. Open the project in MPLAB X IDE and select the desired configuration.
2. Build the code and program the device.
3. Attach the device to the host. If the host is a personal computer and this is the first time you have plugged this device into the computer you may be prompted for a .inf file.



Found New Hardware Wizard

Welcome to the Found New Hardware Wizard

This wizard helps you install software for:

Communications Port

If your hardware came with an installation CD or floppy disk, insert it now.

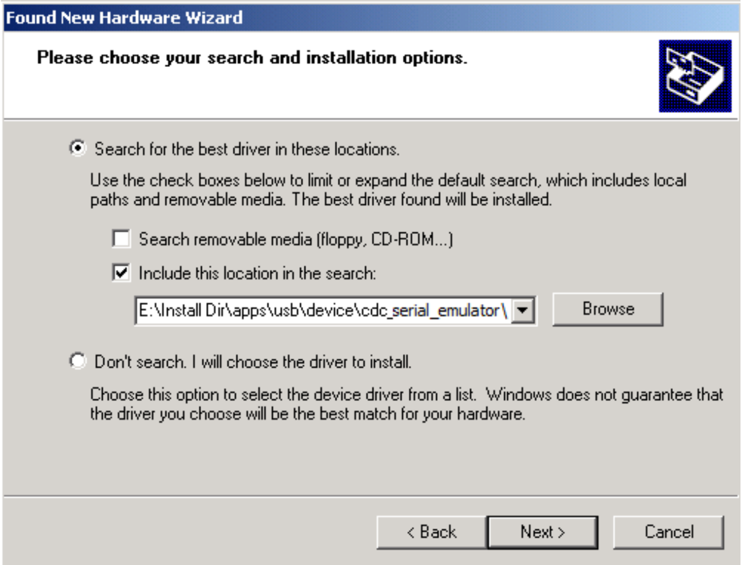
What do you want the wizard to do?

● Install the software automatically (Recommended)

○ Install from a list or specific location (Advanced)

Click Next to continue.

4. Select the "Install from a list or specific location (Advanced)" option. Specify the `<install-dir>/usb/apps/device/cdc_serial_emulator/inf` directory.



Found New Hardware Wizard

Please choose your search and installation options.

● Search for the best driver in these locations.

Use the check boxes below to limit or expand the default search, which includes local paths and removable media. The best driver found will be installed.

□ Search removable media (floppy, CD-ROM...)

☑ Include this location in the search:

E:\Install Dir\apps\usb\device\cdc_serial_emulator\

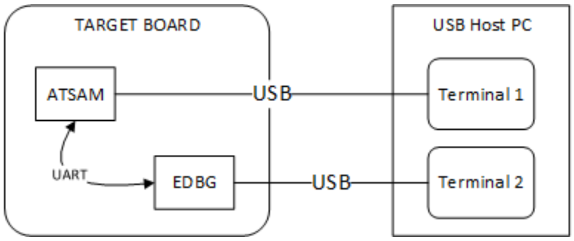
○ Don't search. I will choose the driver to install.

Choose this option to select the device driver from a list. Windows does not guarantee that the driver you choose will be the best match for your hardware.

The LEDs on the demonstration board will indicate the USB state of the device. Refer to the Configuring the Hardware section for the selected target hardware.

5. Open a terminal emulation program of your choice and select the enumerated USB COM port. Set the desired serial baud and other connection related parameters. This is terminal 1.

6. Connect the CDC USB port to the Host PC and ensure that a second COM port is available on the Host PC. Refer to the Configuring the Hardware section for details for the selected hardware. Open a serial terminal program and select the second COM port. The serial settings of this COM port should match the setting made in step 5. This is terminal 2. The setup should resemble the below figure.



7. Text entered into the terminal 1 program will be echoed on terminal 2. Text entered in terminal 2 should be echoed in terminal 1.

hid_basic

This demonstration application creates a custom HID device that can be controlled by a PC-based utility.

Description

This application creates a custom HID device that can be controlled by a PC-based utility. The device allows the USB Host utility to control the LEDs on the board and query the status of a switch.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/device/hid_basic/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a USB HID Device application with freeRTOS on a ATSAME70Q21B device.
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal(non RTOS) USB HID Device application on a ATSAME70Q21B device.
`sam_d21_xpro.X`	ATSAMD21J18A	SAM D21 Xplained Pro	This project implements a bare-metal (no RTOS) Single COM Port CDC USB Device application on the ATSAMD21J18A device.
`sam_e54_xpro.X`	ATSAME54P20A	SAM E54 Xplained Pro	This project implements a bare-metal (no RTOS) Single COM Port CDC USB Device application on the ATSAME54P20A device.
`pic32mz_ef_sk.X`	PIC32MZ2048EFH144	PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit	This project implements a bare-metal (no RTOS) Single COM Port CDC USB Device application on the PIC32MZ2048EFH144 device.
`pic32mz_ef_sk_freertos.X`	PIC32MZ2048EFH144	PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit	This project implements a bare-metal (no RTOS) Single COM Port CDC USB Device application on the PIC32MZ2048EFH144 device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and VBUS (positions 2 and 3).
- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host). Pressing the LED Toggle button on the USB Host PC Utility toggles LED1.
- The USB Host PC utility monitors SW1 switch press on the board.
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host PC.

SAMD21 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host). Pressing the LED Toggle button on the USB Host PC Utility toggles LED0.
- The USB Host PC utility monitors SW0 switch press on the board.
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

SAME54 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host). Pressing the LED Toggle button on the USB Host PC Utility toggles LED1.
- The USB Host PC utility monitors SW0 switch press on the board.
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit

- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host). Pressing the LED Toggle button on the USB Host PC Utility toggles LED1.
- The USB Host PC utility monitors SW1 switch press on the board.
- Use the micro-A/B port J4 (which is located on the bottom side of the board) to connect the USB Device to the the USB Host PC.

Running the Demonstration

Provides instructions on how to build and run the HID Basic demonstration.

Description

This demonstration uses the selected hardware platform as a HID class USB device, but uses the HID class for general purpose I/O operations. While compiling, select the appropriate MPLAB X IDE project based on the demonstration board. Refer to Building the Application for details.

Typically, the HID class is used to implement human interface products, such as mice and keyboards. The HID protocol, is however, quite flexible, and can be adapted and used to send/receive general purpose data to/from a USB device. Using the HID class for general purpose I/O operations is quite advantageous, in that it does not require any kind of custom driver installation process. HID class drivers are already provided by and are distributed with common operating systems. Therefore, upon plugging in a HID class device into a typical computer system, no user installation of drivers is required, the installation is fully automatic.

The LEDs on the demonstration board will indicate the USB state of the device as described in Configuring the Hardware section.

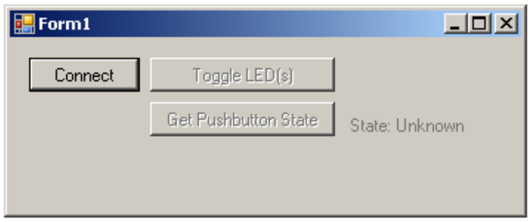
HID devices primarily communicate through one interrupt IN endpoint and one interrupt OUT endpoint. In most applications, this effectively limits the maximum achievable bandwidth for full speed HID devices to 64 kBytes/s of IN traffic, and 64 kBytes/s of OUT traffic (64 kB/s, but effectively "full duplex").

The `GenericHIDSimpleDemo.exe` program, and the associated firmware demonstrate how to use the HID protocol for basic general purpose USB data transfer.

Before you can run the `GenericHIDSimpleDemo.exe` executable, you will need to have the Microsoft® .NET Framework Version 2.0 Redistributable Package (later versions are probably acceptable, but have not been tested) installed on your computer. Programs that were built in the Visual Studio® .NET languages require the .NET redistributable package. The redistributable package can be freely downloaded from Microsoft's website. Users of Windows Vista® operating systems will not need to install the .NET framework, as it comes preinstalled as part of the operating system.

Launching the Application

To launch the application, simply double click the executable `GenericHIDSimpleDemo.exe` in the `<install-dir>\usb\apps\device\hid_basic\bin` directory. A property sheet similar to the following should appear:



Note: If instead of this window, an error message appears while trying to launch the application, it is likely the Microsoft .NET Framework Version 2.0 Redistributable Package has not yet been installed. Please install it and try again.

Send/Receive Packets

To begin sending/receiving packets to the device, you must first find and connect to the device. As configured by default, the application is looking for HID class USB devices with VID = 0x04D8 and PID = 0x003F. The device descriptor in the firmware project meant to be used with this demonstration uses the same VID/PID. If you plug in a USB device programmed with the correct precompiled `.hex` file, and click **Connect**, the other push buttons should become enabled. If clicking **Connect** has no effect, it is likely the USB device is either not connected, or has not been programmed with the correct firmware.

Clicking **Toggle LED(s)** should send a single packet of general purpose generic data to the HID class USB peripheral device. The data will arrive on the interrupt OUT endpoint. The firmware has been configured to receive this generic data packet, parse the packet looking for the Toggle LED(s) command, and should respond appropriately by controlling the LED(s) on the demonstration board.

The Get Pushbutton State option will send one packet of data over the USB to the peripheral device (to the interrupt OUT endpoint) requesting the current push button state. The firmware will process the received Get Pushbutton State command, and will prepare an appropriate response packet depending upon the pushbutton state.

Refer to the Configuring the Hardware section for details on the LED indication and Switch that are relevant to the selected hardware.

hid_joystick

Demonstrates a USB HID device, emulating a Joystick.

Description

This demonstration application creates a custom HID joystick. This application is only intended to demonstrate creation of Joystick HID Report descriptors and may not be a definite end solution. The end application requirements may need the report descriptor to be modified.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/device/hid_joystick/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a USB HID Mouse Device application with freeRTOS on ATSAME70Q21B device.
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal (no RTOS) UBS HID Mouse Device application on the ATSAME70Q21B device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and VBUS (positions 2 and 3).
- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Use Switch SW1 to stop and start the circular motion of the mouse pointer.
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host PC.

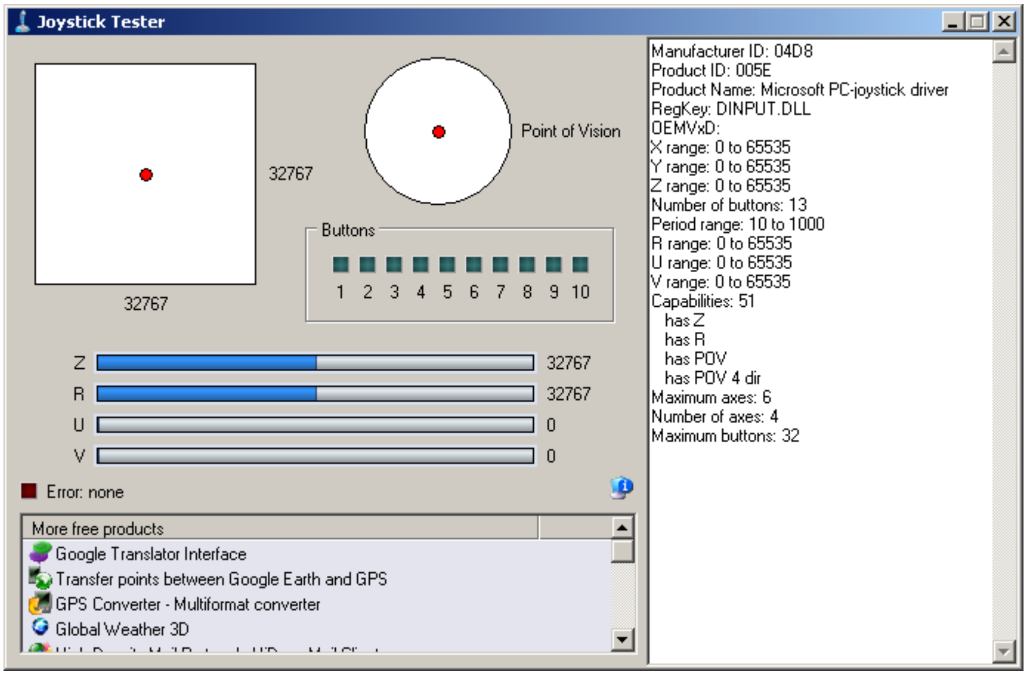
Running the Demonstration

Provides instructions on how to build and run the USB HID Joystick demonstration.

Description

This demonstration uses the selected hardware platform as a USB Joystick. Select the appropriate MPLAB X IDE project configuration based on the demonstration board. Refer to Building the Application for details.

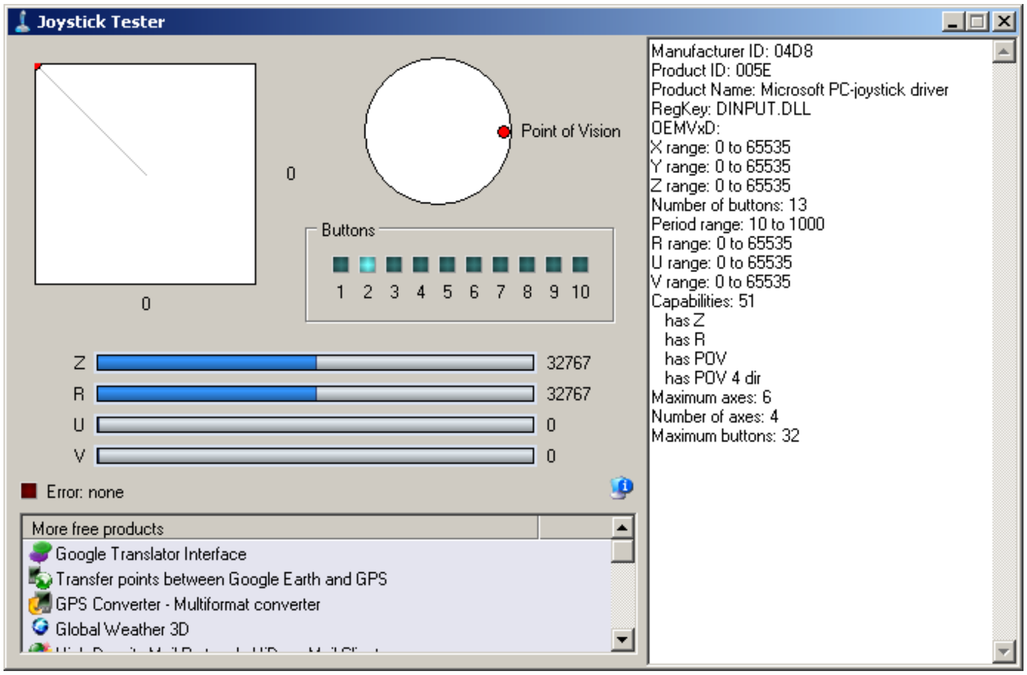
The LED on the demonstration board will indicate the USB state of the device. Refer to the Configuring the Hardware Section for details.



```
Manufacturer ID: 04D8
Product ID: 005E
Product Name: Microsoft PC-joystick driver
RegKey: DINPUT.DLL
OEMVxD:
X range: 0 to 65535
Y range: 0 to 65535
Z range: 0 to 65535
Number of buttons: 13
Period range: 10 to 1000
R range: 0 to 65535
U range: 0 to 65535
V range: 0 to 65535
Capabilities: 51
   has Z
   has R
   has POV
   has POV 4 dir
Maximum axes: 6
Number of axes: 4
Maximum buttons: 32
```

Pressing the button will cause the device to:

- Indicate that the "x" button is pressed, but no others
- Move the hat switch to the "east" position
- Move the X and Y coordinates to their extreme values



```
Manufacturer ID: 04D8
Product ID: 005E
Product Name: Microsoft PC-joystick driver
RegKey: DINPUT.DLL
OEMVxD:
X range: 0 to 65535
Y range: 0 to 65535
Z range: 0 to 65535
Number of buttons: 13
Period range: 10 to 1000
R range: 0 to 65535
U range: 0 to 65535
V range: 0 to 65535
Capabilities: 51
   has Z
   has R
   has POV
   has POV 4 dir
Maximum axes: 6
Number of axes: 4
Maximum buttons: 32
```

hid_keyboard

Demonstrates a USB HID device, emulating a keyboard.

Description

This demonstration application creates a Generic HID keyboard. Pressing a switch on the target board emulates a keyboard key press.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/device/hid_keyboard/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a USB HID Keyboard Device application with freeRTOS on ATSAME70Q21B device.
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal (no RTOS) USB HID Keyboard Device application on the ATSAME70Q21B device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and VBUS (positions 2 and 3).
- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Press Switch SW1 to exercise the keyboard key press function.
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host PC.

Running the Demonstration

Provides instructions on how to build and run the USB HID Keyboard demonstration.

Description

This demonstration uses the selected hardware platform as a USB keyboard. While compiling, select the appropriate MPLAB X IDE project based on the demonstration board. Refer to Building the Application for details.

The LEDs on the demonstration board will indicate the USB state of the device, as described in the Configuring the Hardware section.

Before pressing the button, select a window in which it is safe to type text freely. Pressing the button on the demonstration board will cause the device to print a character on the screen.

hid_mouse

Demonstrates a USB HID device, emulating a mouse pointing device.

Description

This demonstration application creates a USB HID based two-button mouse device. When connected, the device emulates mouse operation by moving the cursor in a circular pattern.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/device/hid_mouse/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a USB HID Mouse Device application with freeRTOS on ATSAME70Q21B device.
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal (no RTOS) UBS HID Mouse Device application on the ATSAME70Q21B device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and VBUS (positions 2 and 3).
- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Use Switch SW1 to stop and start the circular motion of the mouse pointer.
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host PC.

Running the Demonstration

Provides instructions on how to build and run the HID Mouse Demonstration.

Description

This demonstration uses the selected hardware platform as a USB mouse. While compiling, select the appropriate MPLAB X IDE project based on the demonstration board. Refer to Building the Application for details.

The LEDs on the demonstration board will indicate the USB state of the device. Refer to the "Configuring the Hardware" Section for details.

Before connecting the board to the computer through the USB cable please be aware that the device will begin moving the mouse cursor on the computer. There are two ways to stop the device from allowing the cursor to continue to move. The first way is to disconnect the device from the computer. The second is to press the correct button on the hardware platform. Pressing the button again will cause the mouse cursor to start moving in a circle again.

hid_msd_basic

Building the Application

This section identifies the MPLAB X IDE project name and location and lists and describes the available configurations for the this demonstration application.

Description

The following table lists the MPLAB X Projects contained in <install-dir>/usb/apps/device/hid_msd_basic/firmware folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
sam_e70_xult.X	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal (no RTOS) UBS HID plus MSD Device application on the ATSAME70Q21B device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and VBUS (positions 2 and 3).
- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).Pressing the LED Toggle button on the USB Host PC Utility toggles LED1.
- The USB Host PC utility monitors SW1 switch press on the board.
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host PC.

Running the Demonstration

Provides instructions on how to build and run the demonstration.

Description

This demonstration functions as composite USB Device that combines the features of the devices created by the hid_basic and the msd_basic demonstration applications. Refer to Running the Demonstration section of the hid_basic demonstration and Running the Demonstration section of the msd_basic demonstration for details on exercising the HID and MSD functions, respectively.

msd_basic

Demonstrates a USB MSD Device emulating a Flash Drive.

Description

This demonstration application creates a USB Pen drive using the Mass Storage Device Class.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/device/msd_basic/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a Mass Storage Device Basic application with freeRTOS on a ATSAME70Q21B device.
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal(non RTOS) Mass Storage Device basic application on a ATSAME70Q21B device.
`sam_d21_xpro.X`	ATSAMD21J18A	SAM D21 Xplained Pro	This project implements a bare-metal(non RTOS) Mass Storage Device basic application on a ATSAMD21J18A device.
`sam_e54_xpro.X`	ATSAME54P20A	SAM E54 Xplained Pro	This project implements a bare-metal(non RTOS) Mass Storage Device basic application on a ATSAME54P20A device.
`pic32mz_ef_sk.X`	PIC32MZ2048EFH144	PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit	This project implements a bare-metal(non RTOS) Mass Storage Device basic application on a PIC32MZ2048EFH144 device.
`pic32mz_ef_sk_freertos.X`	PIC32MZ2048EFH144	PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit	This project implements a Mass Storage Device basic application with freeRTOS on PIC32MZ2048EFH144 device.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and VBUS (positions 2 and 3).
- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).

- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host PC.

SAMD21 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

SAME54 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit

- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host).
- Use the micro-A/B port J4 (which is located on the bottom side of the board) to connect the USB Device to the the USB Host PC.

Running the Demonstration

Provides instructions on how to build and run the USB MSD Basic demonstration.

Description

This demonstration uses the selected hardware platform as a logical drive on the computer using the internal Flash of the device as the drive storage media. Connect the hardware platform to a computer through a USB cable. The device should appear as a new drive on the computer named "Drive Name". The drive can used to store files.

The LEDs on the demonstration board will indicate the USB state of the device. Refer to the Configuring the Hardware section for details.

Note: Reprogramming the development board will cause any stored files to be erased.

vendor

Demonstrates a custom USB Device created by using the USB Device Layer Endpoint functions.

Description

This demonstration application creates a custom USB device using the USB Device Layer Endpoint functions.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/device/vendor/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal(non RTOS) Vendor USB Device application on a ATSAME70Q21B device.

`sam_v71_xult.X`	ATSAMV71Q21B	SAMV71 Xplained Ultra	This project implements a bare-metal(non RTOS) Vendor USB Device application on a ATSAMV71Q21B device.
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a FreeRTOS Vendor USB Device application on a ATSAME70Q21B device.
`sam_d21_xpro.X`	ATSAMD21J18A	SAM D21 Xplained Pro	This project implements a bare-metal(non RTOS) Vendor USB Device application on a ATSAMD21J18A device.
`sam_e54_xpro.X`	ATSAME54P20A	SAM E54 Xplained Pro	This project implements a bare-metal(non RTOS) Vendor USB Device application on a ATSAME54P20A device.
`pic32mz_ef_sk.X`	PIC32MZ2048EFH144	PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit	This project implements a bare-metal(non RTOS) Vendor USB Device application on a PIC32MZ2048EFH144 device.
`pic32mz_ef_sk_freertos.X`	PIC32MZ2048EFH144	PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit	This project implements a FreeRTOS Vendor USB Device application on a PIC32MZ2048EFH144 device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and VBUS (positions 2 and 3).
- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host). Pressing the Toggle LED button on the PC USB Host application will cause LED1 to toggle.
- The firmware will monitor switch SW1 on the board for switch press and will report this to the PC USB Host application.
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host PC.

SAMV71 Xplained Ultra

- Jumper titled "USB VBUS" must be shorted between PC09 and VBUS (positions 2 and 3)
- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host). Pressing the Toggle LED button on the PC USB Host application will cause LED0 to toggle.
- The firmware will monitor switch SW1 on the board for switch press and will report this to the PC USB Host application.
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

SAMD21 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host). Pressing the Toggle LED button on the PC USB Host application will cause LED0 to toggle.
- The firmware will monitor switch SW0 on the board for switch press and will report this to the PC USB Host application.
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

SAME54 Xplained PRO

- LED0 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host). Pressing the Toggle LED button on the PC USB Host application will cause LED0 to toggle.
- The firmware will monitor switch SW0 on the board for switch press and will report this to the PC USB Host application.
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC.

PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit

- LED1 indicates USB Device Configuration Set Complete event (the USB device functionality has been activated by the USB Host). Pressing the Toggle LED button on the PC USB Host application will cause LED1 to toggle.

- The firmware will monitor switch SW1 on the board for switch press and will report this to the PC USB Host application.
- Use the micro-A/B port J4 (which is located on the bottom side of the board) to connect the USB Device to the the USB Host PC.

Running the Demonstration

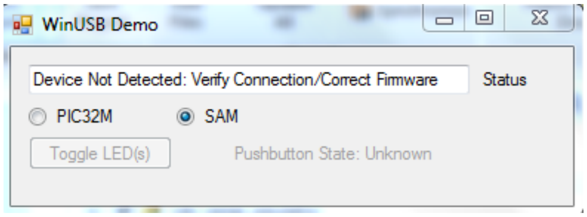
Provides instructions on how to build and run the Vendor USB Device demonstration.

Description

The Vendor device can be exercised by using the WinUSB PnP Demonstration application, which is provided in your installation of MPLAB Harmony.

The LEDs on the demonstration board will indicate the USB state of the device. This application allows the state of the LEDs on the board to be toggled and indicates the state of a switch (pressed/released) on the board. Refer to the Configuring the Hardware section for hardware specific details.

To launch the application, double click `WinUSB PnP Demo.exe` located in `<install dir>/usb/apps/device/vendor/bin`. A dialog box similar to the following should appear:


WinUSB Demo

Device Not Detected: Verify Connection/Correct Firmware

Status

○ PIC32M ● SAM

The appropriate device family that is under testing should be selected in the utility. Pressing the Toggle LED button will cause the LED on the board to toggle. The Pushbutton State field in the application indicates the state of a button on connected USB Device. Pressing the switch on the development board will update the Pressed/Not Pressed status of the Pushbutton State field.

Note: The device family under test should be selected appropriately. An incorrect selection will result in an invalid push button status.

Host Demonstrations

This section describes the USB Host demonstrations.

cdc_basic

This application demonstrates the use of the CDC Host Class Driver to enumerate and operate a CDC Device.

Description

This application demonstrates the use of the CDC Host Class Driver to enumerate and operate a CDC Device. The application uses the USB Host layer and CDC class driver to enumerate a CDC USB device. The demonstration host application then operates and uses the functionality of the attached CDC Device.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/host/cdc_basic/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_d21_xpro.X`	ATSAMD21J18A	SAMD21 Xplained PRO	This project implements a bare-metal(non RTOS) USB CDC Host application on a ATSAMD21J18A device.
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a USB CDC Host application with freeRTOS on a ATSAME70Q21B device.
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal(non RTOS) USB CDC Host application on a ATSAME70Q21B device.
`sam_v71_xult_freertos.X`	ATSAMV71Q21B	SAMV71 Xplained Ultra	This project implements a USB CDC Host application with freeRTOS on a ATSAMV71Q21B device.
`sam_e54_xpro.X`	ATSAME54P20A	SAM E54 Xplained Pro	This project implements a bare-metal(non RTOS) USB CDC Host application on a ATSAME54P20A device.
`pic32mz_ef_sk.X`	PIC32MZ2048EFH144	PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit	This project implements a bare-metal(non RTOS) USB CDC Host application on a PIC32MZ2048EFH144 device.
`pic32mz_ef_sk_freertos.X`	PIC32MZ2048EFH144	PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit	This project implements a USB CDC Host application with freeRTOS on a PIC32MZ2048EFH144 device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and LED 1(positions 1 and 2).
- Use "TARGET USB" J202 connector on the board to connect the USB Device to the the USB Host. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED1 on the board is controlled by the attached USB CDC device.

SAMV71 Xplained Ultra

- Jumper titled "USB VBUS" must be shorted between PC09 and LED1(positions 1 and 2)
- Use "TARGET USB" connector on the board to connect the USB Device to the the USB Host PC. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED0 on the board is controlled by the attached USB CDC device.

SAMD21 Xplained PRO

- Jumper titled "PA03 SELECT" must be shorted between PA03 and USB_ID(positions 2 and 3)
- Use "TARGET USB" connector on the board to connect the USB Device to the the USB Host PC. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED0 on the board is controlled by the attached USB CDC device.

SAME54 Xplained PRO

- Use "TARGET USB" connector on the board to connect the USB Device to the the USB Host PC. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED0 on the board is controlled by the attached USB CDC device.

PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit

- Connect the device to the Type A connector J5, which is located on the top side of the starter kit.
- LED0 on the board is controlled by the attached USB CDC device.

Running the Demonstration

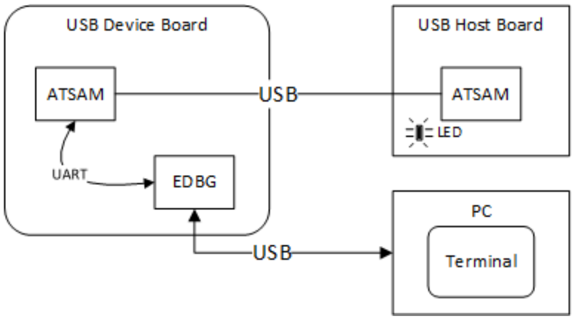
Provides instructions on how to build and run the USB Host CDC Basic Demo.

Description

This application demonstrates the use of the CDC Host Class Driver to enumerate and operate a CDC Device. The application uses the USB Host layer and CDC class driver to enumerate a CDC USB device. The demonstration host application then operates and uses the functionality of the attached CDC Device.

1. Open the MPLAB X IDE project corresponding to the selected board. Refer to the Building the Application section for details.
2. Build the code and program the device.
3. Follow the directions for setting up and running the cdc_serial_emulator USB device demonstration.
4. Connect the USB Device connector of the CDC USB Device board to the USB Host Target board. Refer to the Configuring the Hardware section for any converter requirements.
5. Start a terminal program on the USB Host personal computer and select the Serial-to-USB Dongle as the communication port. Select the baud rate as 9600, no parity, 1 Stop bit and no flow control.
6. A prompt (LED :) will be displayed immediately on the terminal emulation program.
7. Pressing the 1 key on the USB Host keyboard will cause the LED on the Host USB board to switch on. Refer to the Configuring the Hardware section for details on the relevant LED. Pressing any other key at the prompt message will cause the LED to switch off.
8. The prompt will again be displayed on terminal emulation program, and step 7 can be repeated.

The setup should be similar to the following diagram.



The cdc_serial_emulator demonstration emulates a USB-to-Serial Dongle. The CDC Host (running the cdc_basic demonstration application) sends the prompt message to the CDC device. The CDC device forwards the prompt to the UART port from where it is transmitted to the personal computer USB Host through the USB serial interface. A key press on the personal computer USB Host is transmitted to the CDC device, which in turn presents the key press data to the CDC host. The cdc_basic demonstration then analyzes the key press data and switches on the respective LED.

cdc_msd

Demonstrates host support for multiple device classes.

Description

This demonstration application creates a USB Host that can support different device classes in one application.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/host/cdc_msd/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a USB CDC MSD Host application with freeRTOS on a ATSAME70Q21B device.
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal(non RTOS) USB CDC MSD Host application on a ATSAME70Q21B device.
`sam_d21_xpro.X`	ATSAMD21J18A	SAM D21 Xplained Pro	This project implements a bare-metal(non RTOS) USB CDC MSD Host application on a ATSAMD21J18A device.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and LED 1(positions 1 and 2).
- LED2 indicates a Device Connection. (Attached device has been successfully enumerated and configured).
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED1 on the board is controlled by the attached USB CDC device when a CDC device is attached. When a Mass Storage Device is attached, it indicates file write completion.

SAMD21 Xplained PRO

- Jumper titled "PA03 SELECT" must be shorted between PA03 and USB_ID(positions 2 and 3)
- Use "TARGET USB" connector on the board to connect the USB Device to the the USB Host PC. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED0 on the board is controlled by the attached USB CDC device when a CDC device is attached. When a Mass Storage Device is attached, it indicates file write completion.

Running the Demonstration

Provides instructions on how to build and run the USB CDC MSD demonstration.

Description

This demonstration application creates a USB Host application that enumerates a CDC and a MSD device. This application combines the functionality of the Host cdc_basic and msd_basic demonstration applications into one application. If a CDC device is connected, the demonstration application behaves like the cdc_basic host application. If a MSD device is connected, the demonstration application behaves like the msd_basic host application.

Refer to Running the Demonstration section of the host cdc_basic demonstration and the Running the Demonstration section of the host msd_basic demonstration for details on exercising the CDC and MSD host aspects of the demonstration.

hid_basic_keyboard

Demonstrates using the USB HID Host Client driver with the Keyboard Usage driver to facilitate the use of a USB HID Keyboard with a PIC32 USB Host.

Description

This application demonstrates the use of the USB HID Host Client Driver to enumerate and operate a HID keyboard device. The application uses the USB Host layer, HID Client driver and HID Keyboard Usage driver to enumerates a USB keyboard and understand keyboard press release events.

The keyboard events are displayed using a terminal emulator on a personal computer.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/host/hid_basic_keyboard/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a USB HID Host application with freeRTOS on a ATSAME70Q21B device.
`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal(non RTOS) USB HID Host application on a ATSAME70Q21B device.
`sam_d21_xpro.X`	ATSAMD21J18A	SAM D21 Xplained Pro	This project implements a bare-metal(non RTOS) USB HID Host application on a ATSAMD21J18A device.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and LED 1(positions 1 and 2).
- LED2 indicates a Device Connection. (Attached device has been successfully enumerated and configured).
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host. A USB micro AB to type A USB Host receptacle converter will be needed to connect the keyboard.
- A commercially available USB Keyboard will be needed.
- The demonstration uses the EDBG Serial Interface to transfer demonstration application messages on a PC.

SAMD21 Xplained PRO

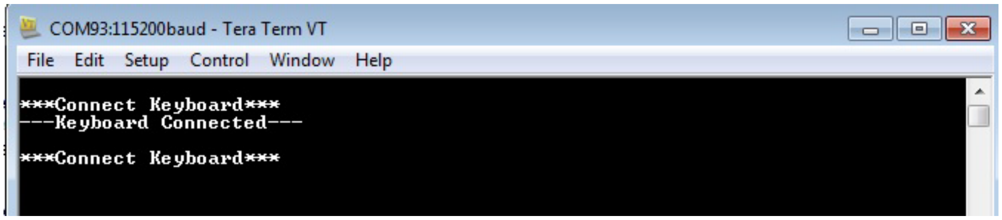
- Jumper titled "PA03 SELECT" must be shorted between PA03 and USB_ID(positions 2 and 3)
- Use "TARGET USB" connector on the board to connect the USB Device to the the USB Host PC. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED0 indicates a Device Connection. (Attached device has been successfully enumerated and configured).
- The demonstration uses the EDBG Serial Interface to transfer demonstration application messages on a PC.

Running the Demonstration

Provides instructions on how to build and run the USB HID Basic Keyboard demonstration.

Description

1. Open the MPLAB X IDE project corresponding to the selected hardware.
2. Build the code and program the device.
3. Connect the serial interface on the board to a PC. On the PC, launch a terminal emulator, such as Tera Term, and select the appropriate COM port and set the serial port settings to 115200-N-1.
4. If a USB keyboard is not connected to the Embedded USB Host, the terminal emulator window will show the *Connect Keyboard* prompt.
5. Attach a USB keyboard to the Host connector of the target hardware. The message, *Keyboard Connected*, will appear in the terminal emulator window.
6. Begin typing on the keyboard and the appropriate keys should be displayed on the serial terminal. Subsequent press and release of modifier keys (i.e., CAPS LOCK, NUM LOCK, etc.) will result in the appropriate keyboard LEDs to turning ON and OFF.
7. Disconnecting the keyboard will result in the message, *Connect Keyboard*.


COM93:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
***Connect Keyboard***
---Keyboard Connected---

***Connect Keyboard***
```

msd_basic

This application demonstrates the use of the MSD Host Class Driver to write a file to USB Flash Drive.

Description

This application demonstrates the use of the MSD Host Class Driver to write a file to a USB Flash drive. The application uses the USB Host layer , MSD class driver and the MPLAB Harmony File System Framework to enumerate a USB Flash drive and to write a file to it.

Building the Application

This section identifies the MPLAB X IDE projects contained in this demo application, the name and location of the projects and lists the target hardware development board for each project.

Description

The following table lists the MPLAB X Projects contained in `<install-dir>/usb/apps/host/msd_basic/firmware` folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
`sam_d21_xpro.X`	ATSAMD21J18A	SAMD21 Xplained PRO	This project implements a bare-metal(non RTOS) USB MSD Host application on a ATSAMD21J18A device.
`sam_e70_xult_freertos.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a USB MSD Host application with freeRTOS on a ATSAME70Q21B device.

`sam_e70_xult.X`	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal(non RTOS) USB MSD Host application on a ATSAME70Q21B device.
`sam_v71_xult_freertos.X`	ATSAMV71Q21B	SAMV71 Xplained Ultra	This project implements a USB MSD Host application with freeRTOS on a ATSAMV71Q21B device.
sam_e54_xpro.X	ATSAME54P20A	SAM E54 Xplained Pro	This project implements a bare-metal(non RTOS) USB MSD Host application on a ATSAME54P20A device.
pic32mz_ef_sk.X	PIC32MZ2048EFH144	PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit	This project implements a bare-metal(non RTOS) USB MSD Host application on a PIC32MZ2048EFH144 device.
pic32mz_das_sk.X	PIC32MZ2064DAS169	PIC32MZ Embedded Graphics with Stacked DRAM (DA) Starter Kit (Crypto)	This project implements a bare-metal(non RTOS) USB MSD Host application on a PIC32MZ2064DAS169 device.

Identify the project for the target device and hardware and open this project in MPLAB X IDE. Build the project using the available Menu or Tool Bar options.

Configuring the Hardware

Describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and LED 1(positions 1 and 2).
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED1 indicates the file write is complete.

SAMV71 Xplained Ultra

- Jumper titled "USB VBUS" must be shorted between PC09 and LED1(positions 1 and 2)
- Use TARGET USB connector on the board to connect the USB Device to the the USB Host PC. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED0 indicates the file write is complete.

SAMD21 Xplained PRO

- Jumper titled "PA03 SELECT" must be shorted between PA03 and USB_ID(positions 2 and 3)
- Use "TARGET USB" connector on the board to connect the USB Device to the the USB Host PC. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED0 indicates the file write is complete.

SAME54 Xplained PRO

- Use "TARGET USB" connector on the board to connect the USB Device to the the USB Host PC. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED0 indicates the file write is complete.

PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit

- Connect the device to the Type A connector J5, which is located on the top side of the starter kit.
- LED1 indicates the file write is complete.

PIC32MZ Embedded Graphics with Stacked DRAM (DA) Starter Kit

- Connect the device to the Type A connector J7, which is located on the top side of the starter kit.
- LED1 indicates the file write is complete.

Running the Demonstration

Provides instructions on how to build and run the USB Host MSD Basic demonstration.

Description

This application demonstrates the use of the MSD Host Class Driver to write a file to USB Flash drive. The application uses the USB Host layer, MSD class driver and the MPLAB Harmony File System Framework to enumerate a USB Flash drive and to write a file to it.

1. Open the MPLAB X IDE project corresponding to the selected hardware. Refer to the Building the Application section for details.
2. Build the code and program the device.
3. With the code running, attach a USB Flash drive to the Host connector on the desired starter kit.
4. The demonstration application will then create a file named `file.txt`. It will then write the text "Hello World" to this file, and then close the file.
5. The LED on the selected hardware will indicate the status of the operation. Refer to the Configuring the Hardware instruction for details.
6. The USB Flash drive can then be attached to a USB Host personal computer to verify the demonstration application operation.
7. Steps 3 through 6 can be repeated.
8. If the USB Flash drive already contains a file with the name `file.txt`, the demonstration application will append the text "Hello World" to the end of the file contents.

hid_basic_mouse_usart

Building the Application

This section does the following:

- Identifies the MPLAB X IDE project name and location.
- Lists and describes the available configurations for the USB HID Basic Mouse USART demonstration.

Description

The following table lists the MPLAB X Projects contained in <install-dir>/usb/apps/host/hid_basic_mouse_usart/firmware folder and provides a description of each project. All projects listed in this table implement the same functionality.

MPLAB X IDE Project

This table lists the name and location of the MPLAB X IDE project folder for the demonstration.

Project Name	Target Device	Target Board	Description
sam_e70_xult_freertos.X	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a USB HID Host Mouse application with freeRTOS on a ATSAME70Q21B device.
sam_e70_xult.X	ATSAME70Q21B	SAME70 Xplained Ultra	This project implements a bare-metal(non RTOS) USB HID Host Mouse application on a ATSAME70Q21B device.
sam_d21_xpro.X	ATSAMD21J18A	SAM D21 Xplained Pro	This project implements a bare-metal(non RTOS) USB HID Host Mouse application on a ATSAMD21J18A device.

Configuring the Hardware

This section describes how to configure the supported hardware.

Description

SAME70 Xplained Ultra

- Jumper J204 must be shorted between PB08 and LED 1(positions 1 and 2).
- LED2 indicates a Device Connection. (Attached device has been successfully enumerated and configured).
- Use TARGET USB J202 connector on the board to connect the USB Device to the the USB Host. A USB micro AB to type A USB Host receptacle converter will be needed to connect the keyboard.
- A commercially available USB Keyboard will be needed.
- The demonstration uses the EDBG Serial Interface to transfer demonstration application messages on a PC.

SAMD21 Xplained PRO

- Jumper titled "PA03 SELECT" must be shorted between PA03 and USB_ID(positions 2 and 3)
- Use "TARGET USB" connector on the board to connect the USB Device to the the USB Host PC. A USB micro AB to type A USB Host receptacle converter will be needed to connect the device.
- LED0 indicates a Device Connection. (Attached device has been successfully enumerated and configured).
- The demonstration uses the EDBG Serial Interface to transfer demonstration application messages on a PC.

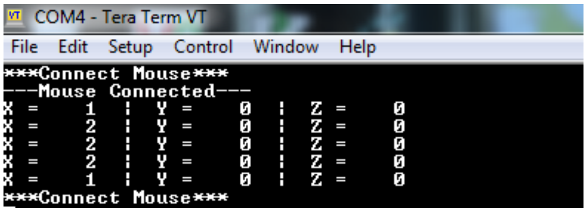
Running the Demonstration

This section provides instructions about how to build and run the USB HID Mouse USART demonstration.

Description

Open the project in MPLAB X IDE and select the project configuration.

- Build the code and program the device.
- Launch a terminal emulator, such as Tera Term. Select the appropriate COM port and set the serial port settings to 115200-N-1.
- If a USB mouse is not connected to the Host connector, the serial terminal emulator window will show the "Connect Mouse" prompt.
- Attach a USB mouse to the Host connector of the target hardware. The message, "Mouse Connected", will display in the serial terminal emulator window.
- Begin moving the mouse and the appropriate relative coordinate changes for X,Y, and Z axes should be displayed in the serial terminal window.
- Click the mouse button to toggle LEDs on the board.
- Disconnecting the mouse will result in the message, "Connect Mouse", to reappear on the serial console.



```
***Connect Mouse***
---Mouse Connected---
X =     1  |  Y =     0  |  Z =     0
X =     2  |  Y =     0  |  Z =     0
X =     2  |  Y =     0  |  Z =     0
X =     2  |  Y =     0  |  Z =     0
X =     1  |  Y =     0  |  Z =     0
***Connect Mouse***
```

USB Libraries Help

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

USB Device Libraries

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 or SAM 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 services 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. The USBCD provides functions to:

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

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 opens the USBCD and registers an 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 USB demonstration application projects is available in the USB Demonstrations section.

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

Application Interaction with Device Layer

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 the 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 recommended approach is to use the available demonstration applications as a starting point for the application.

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 the 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`).

USB Device Layer Library

This section describes the USB Device Layer Library.

Introduction

Introduces the MPLAB Harmony 3 USB Device Layer Library.

Description

The MPLAB Harmony 3 USB Device Layer Library (also referred to as the Device Layer) is part of the MPLAB Harmony 3 USB Device Stack. Within the USB Device Stack, the Device Layer implementation is independent of the USB Controller Hardware. It 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 all other control transfers to function drivers and the application. It provides the application and function drivers with API routines that allow them to respond and complete a control transfer, with a facility to defer the responses to a control transfer. 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. It initializes USB device function drivers which contained in the active configuration. The state machines of these function drivers is maintained by the Device Layer i.e. the Device Layer invokes the function driver tasks routines from within its own task routine. The Device Layer thus treats the 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.
- Hardware independent architecture. API does not change across microcontrollers.
- Supports the SAME7x and SAMD2x families of microcontrollers.
- Supports composite USB devices.
- Supports CDC, HID, MSD, Audio v1.0. Audio v2.0 and Generic/Vendor Devices.
- All functions are non-blocking.
- Operates readily with a RTOS.
- Designed to integrate readily with other MPLAB Harmony middleware.
- Completely interrupt driven.
- Requires minimal application intervention while maintaining the USB Device state.
- Allows implementation of a multi-configuration USB device.
- In case of multiple UBS controllers, allows itself to be instantiated more than once.

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 implemented by USB Device Layer.

Description

The Device Layer in the MPLAB Harmony USB Device Stack handles enumeration requests from the USB Host and provides an abstracted access to Control Transfers and Non-Control Endpoint Management. The Device Layer intercepts all Control transfers issues by the host. Only those control transfers that require application or function driver intervention are forwarded. Standard device control transfers are completely handled by the Device Layer. All access to the bus is routed via the Device Layer.

The following block diagram shows the USB Device Layer interaction with USB Controller Driver, Function Drivers, User

Application and the Harmony System module.

USB Device Layer Interaction Block Diagram

Harmony System Module Interaction

The MPLAB Harmony System Module initializes the Device Layer in the SYS_Initialize() function. It calls the USB Device Layer task routine periodically from the SYS_Tasks() function.

USB Controller Driver Interaction

The Device Layer opens the USB Controller Driver exclusively. It is the only client to the controller driver. The Device Layer manages Endpoint 0 and has exclusive access to this endpoint. It maintains the Control Transfer state machine and seeks intervention from the Application or the Function Drivers where required. The Device Layer provides layered access to the required USB Controller Driver functions.

Function Driver Interaction

The USB Device Layer interaction with the function driver involves the following:

- It initializes the function driver when device is configured by the Host. This happens when the Host issues the standard USB Set Configuration Request to the device. The device layer initializes only those function drivers that are part of 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 issues a Set Configuration request with configuration value set to '0'.
- The Device Layer executes the Function driver task routines from within its own tasks routine (USB_DEVICE_Tasks() function). In an RTOS application, this implies that the function driver task routines run at the same priority and context as the device layer task routine.
- Forwards class/interface specific control transfer requests from host to function drivers for processing. The function drivers can use Device layer API routines to read and write data to Endpoint 0.

The Device Layer initiates all of the above interactions with the function driver independent of function driver type. Each function driver implements a set of common APIs. These common API allow the Device Layer to initialize/deinitialize the function driver, forward control transfers and invoke the function driver's task routine. Function Drivers are registered with the Device Layer as a part of the Device Stack Configuration. This is a compile time step. Function driver registration is explained elsewhere in this help section.

User Application (Client) Interaction

The user application opens the Device Layer and becomes a Device Layer client. It registers an event callback function with the Device layer to get Device Layer event notifications. Other than receiving such notifications, the application client can also interact with the Device Layer to determine device status such as USB speed or initiate a 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 these control transfers.

Library Overview

The Device Layer Library provides API routines which allow the Harmony System module, function driver and the user application to interact with it.

The library interface routines can be classified as shown in the table below

Library Interface Section	Description
System Interaction Functions	These functions allow the Harmony System Module to perform library initialization, deinitialization, reinitialization and task functions.
Client Core Functions	These functions allow the application client to a register an event callback function.
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 (enable, disable and stall) endpoints. These are required in case of a Generic or a Vendor USB Device Implementation.
Device Management Functions	These functions allow the application to manage (attach, detach etc.) the state of the device.
Control Transfer Functions	These functions allow the application to complete control transfers.

How the Library Works

This topic describes the operation of the Device Layer.

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. The Device Layer does not impose any additional requirements on Device descriptors.

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 microcontroller. The contents of this table should not 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. If a device that features multiple configurations, the Function Driver Registration Table will contains 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 microcontroller. The contents of this table should not 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. 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 by the System Module to access this Device Layer context while calling the Device Layer task routine.

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

```
/*************************************************
 * USB Device Layer Initialization.
 **************************************************/

USB_DEVICE_INIT usbDevInitData =
{
    /* 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,

    /* USB Device Speed */
    .deviceSpeed = USB_SPEED_HIGH,

    /* Pointer to the USB Driver Interface */
    .usbDriverInterface = DRV_USBHSV1_DEVICE_INTERFACE
};

/****************************************
 * System Initialization Routine
 ****************************************/
void SYS_Initialize ( void * data )
{

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

    /* Initialize the Application */
    APP_Initialize ( );

}
```

Device Layer Task Routines

Describes the Device Layer task routines.

Description

A call to the USB_DEVICE_Tasks() function should be placed in the 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. 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 with Device Layer.

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 needs a valid Device Layer handle (a handle that is not invalid) 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. It enables USB Device Layer Events and is required for proper functioning of the USB Device Stack. 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.

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 and registering the event handler.

```
/*******************************************************
 * 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 the events generated by Device Layer.

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 (unless otherwise noted). The application must avoid calling computationally intensive functions or blocking functions in the event handler. The application can call interrupt safe 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	The application must check the configuration that was activated and register event handlers with all function drivers that are contained in the activated configuration.
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 specific 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 function 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_SOF	USB_DEVICE_EVENT_DATA_SOF *

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 bus. 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_SOF

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_SOF 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 contained in the active configuration.

Event Specific Data: The pData parameter will point to a USB_DEVICE_EVENT_DATA_CONFIGURED data type that contains the 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 it's 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. The application will recieve this event when the Control Transfer recipient field is set to Other.

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 the operation of USB Device Layer control transfers.

Description

The Device Layer forwards control transfer setup packets to the application, where the Recipient field in the Setup packet is set to "Other". The pData parameter of the event handler will point to the control transfer setup packet. 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 it's control transfer state machine.

                break;

            default:
                break;
        }

        return USB_DEVICE_EVENT_REPONSE_NONE;
    }
```

String Descriptor Table

Describes the Device Layer String Descriptor Table.

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 and the String Descriptor Indexes are continuous. This 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','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
}
sd002 =
{
    sizeof(sd002),
    USB_DESCRIPTOR_STRING,
    {'S','i','m','p','l','e',' ','C','D','C',' ','D','e','v','i','c','e',' ','D','e','m','o' }
};

/*******************************************
 *  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 arrangement may not be optimal. 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 position of the string descriptor in the Advanced String Descriptor Table does not define the String Descriptor Index that the host must to use to identify the string.Instead, 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','c','.'}
};

/*******************************************************
 *  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 about 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 for the BOS descriptor.

Configuring the Library

Describes how to configure the USB Device Library.

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.

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

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

b) Client Core Functions

c) Device Power State Management Functions

d) Device Management Functions

e) Endpoint Management Functions

f) Control Transfer Functions

g) Data Types and Constants

Files

Files

Name	Description
usb_device.h	This is file usb_device.h.
usb_device_config_template.h	This is file usb_device_config_template.h.

Description

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

usb_device.h

This is file usb_device.h.

usb_device_config_template.h

This is file usb_device_config_template.h.

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 is 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.

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.

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
`usb_device_audio_v1_0.h`	This header file must be included in every source file that needs to invoke USB Audio 1.0 Device Driver APIs.

Required File(s)

MHC ***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_v1_0.c`	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.
`/src/dynamic/usb_device_audio_read_write.c`	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

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

b) Data Types and Constants

Files

Files

Name	Description
usb_device_audio_v1_0.h	This is file usb_device_audio_v1_0.h.
usb_device_audio_v1_0_config_template.h	This is file usb_device_audio_v1_0_config_template.h.

Description

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

usb_device_audio_v1_0.h

This is file usb_device_audio_v1_0.h.

usb_device_audio_v1_0_config_template.h

This is file usb_device_audio_v1_0_config_template.h.

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 data structure
        .driver = USB_DEVICE_AUDIO_2_0_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 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 is 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

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.

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

b) Data Types and Constants

Files

Files

Name	Description
usb_device_audio_v2_0.h	This is file usb_device_audio_v2_0.h.
usb_device_audio_v2_0_config_template.h	This is file usb_device_audio_v2_0_config_template.h.

Description

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

usb_device_audio_v2_0.h

This is file usb_device_audio_v2_0.h.

usb_device_audio_v2_0_config_template.h

This is file usb_device_audio_v2_0_config_template.h.

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.

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
SEND_ENCAPSULATED_COMMAND	Required	Implemented by the CDC Function Driver ACM layer. This request is stalled.
GET_ENCAPSULATED_RESPONSE	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.

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.

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 functions
    },
    /* 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 functions
    },
};
```

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 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 */

void APP_USBDeviceEventCallBack ( USB_DEVICE_EVENT event,
     void * eventData, 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 ( *((uint8_t *)eventData) == 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_SUSPENDED:

            break;

        case USB_DEVICE_EVENT_RESUMED:
        case USB_DEVICE_EVENT_ATTACHED:
        case USB_DEVICE_EVENT_DETACHED:
        case USB_DEVICE_EVENT_ERROR:
        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 is 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_DEVICE_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 if 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.

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.

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
`/src/dynamic/usb_device_cdc.c`	This file implements the CDC Data Interface and Communications interface and should be included in the project if the CDC Device function is desired.
`/src/dynamic/usb_device_cdc_acm.c`	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

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

b) Data Types and Constants

Files

Files

Name	Description
usb_device_cdc.h	This is file usb_device_cdc.h.
usb_cdc.h	This is file usb_cdc.h.
usb_device_cdc_config_template.h	This is file usb_device_cdc_config_template.h.

Description

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

usb_device_cdc.h

This is file usb_device_cdc.h.

usb_cdc.h

This is file usb_cdc.h.

usb_device_cdc_config_template.h

This is file usb_device_cdc_config_template.h.

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)
    0x81, 0x00,          // 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 a sample Application Device Layer Event Handler
 * Note how the HID Function Driver event handler APP_USBDeviceHIDEventHandler()
 * 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_USBDeviceHIDEventHandler() function is invoked */

void APP_USBDeviceEventCallBack ( USB_DEVICE_EVENT event,
        void * eventData, uintptr_t context )
{
    uint8_t * configurationValue;
    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 */
            configurationValue = (uint8_t*)eventData;
            if ( *configurationValue == 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_USBDeviceHIDEventHandler, (uintptr_t)&appData);

                /* 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_ATTACHED:
        case USB_DEVICE_EVENT_DETACHED:
        case USB_DEVICE_EVENT_ERROR:
        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_AppHIDEventHandler
   (
      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 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(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_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 handler 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.

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.

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

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

b) Data Types and Constants

Files

Files

Name	Description
usb_device_hid.h	This is file usb_device_hid.h.
usb_device_hid_config_template.h	This is file usb_device_hid_config_template.h.

Description

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

usb_device_hid.h

This is file usb_device_hid.h.

usb_device_hid_config_template.h

This is file usb_device_hid_config_template.h.

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.

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.

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.

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 accepts 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

⚠ **Warning**

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.

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 enabled 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 is not 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 track the transfer. These transfer handles are returned along with the USB_DEVICE_EVENT_ENDPOINT_READ_COMPLETE (when a 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.writeTranferHandle,
        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 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 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.readTranferHandle,
              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 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 msdCBW 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','p'
            },
            {
                '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.

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.

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)

MHC ***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

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

Files

Files

Name	Description
usb_device_msd.h	This is file usb_device_msd.h.
usb_device_msd_config_template.h	This is file usb_device_msd_config_template.h.

Description

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

usb_device_msd.h

This is file usb_device_msd.h.

usb_device_msd_config_template.h

This is file usb_device_msd_config_template.h.

USB Host Libraries

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

Description

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 and SAM microcontrollers. High-Speed devices can be supported with PIC32MZ and SAM 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, USBHS Driver (DRV_USBHS) for PIC32MZ microcontroller or the DRV_USBHS_V1 driver for SAME microcontrollers. 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 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 discusses:

- 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_ACM_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 USBTPList[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 *)USBTPList,
    .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. Sting 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_USBHostSringDescriptorGetCallBack
(
    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_USBHostSringDescriptorGetCallBack, 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.

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.

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

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

b) Data Types and Constants

Files

Files

Name	Description
usb_host.h	This is file usb_host.h.
usb_host_config_template.h	This is file usb_host_config_template.h.

Description

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

usb_host.h

This is file usb_host.h.

usb_host_config_template.h

This is file usb_host_config_template.h.

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 `usb_host_audio_v1_0.c` 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 `usb_host_audio_v1_0.c` 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 USBTPList[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;
            }
        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_FORTMAT 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_FORTMAT 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;
                }
            }
            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_FORTMAT 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_FORTMAT 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.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_FORTMAT 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_FORTMAT 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.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_StreamEnable 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_FORTMAT 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_FORTMAT 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.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_StreamGetNextfunctions. 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_FORTMAT 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_FORTMAT 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.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)
            {
                /* 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_FORTMAT 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_FORTMAT 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.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)
            {
                appData.state = APP_STATE_START_STREAM_DATA;
            }
            break;
       case APP_STATE_START_STREAM_DATA:
            isAudioWriteCompleted = false;
```

```
            appData.state = APP_SATE_WAIT_FOR_WRITE_COMPLETE;
            USB_HOST_AUDIO_V1_0_StreamWrite
            (
                appData.outStreamHandle,
                &transferHandleAudioWrite,
                (void*)&audioSamples,
                192
            );
            break;

        case APP_SATE_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;
    }
}

/*************************************************************/
/* Audio Tasks function */
/************************************************************/
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_BusIsEnabled(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.

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.

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)

MHC ***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

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

b) Audio Stream Access Functions

c) Other Functions

d) Data Types and Constants

Files

Files

Name	Description
usb_host_audio_v1_0.h	This is file usb_host_audio_v1_0.h.
usb_host_cdc.h	This is file usb_host_cdc.h.
usb_host_hub_config_template.h	This is file usb_host_hub_config_template.h.
usb_host_audio_v1_0_config_template.h	This is file usb_host_audio_v1_0_config_template.h.

Description

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

usb_host_audio_v1_0.h

This is file usb_host_audio_v1_0.h.

usb_host_cdc.h

This is file usb_host_cdc.h.

usb_host_hub_config_template.h

This is file usb_host_hub_config_template.h.

usb_host_audio_v1_0_config_template.h

This is file usb_host_audio_v1_0_config_template.h.

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 `usb_host_cdc.c`.
Data Transfer Functions	These functions allow the application client to transfer data to the attached device. These functions are implemented in `usb_host_cdc.c`.
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 `usb_host_cdc_acm.c`.

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 USBTPList[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_PROTOCOLUSB_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_ACM_LineCodingSet
Event	USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE
Event Data Type	USB_HOST_CDC_EVENT_ACM_SET_LINE_CODING_COMPLETE_DATA

Table 2: Get Line Coding

Function	USB_HOST_CDC_ACM_LineCodingGet
Event	USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE
Event Data Type	USB_HOST_CDC_EVENT_ACM_GET_LINE_CODING_COMPLETE_DATA

Table 3: Set Control Line State

Function	USB_HOST_CDC_ACM_ControlLineStateSet
Event	USB_HOST_CDC_EVENT_ACM_CONTROL_LINE_STATE_SET_COMPLETE
Event Data Type	USB_HOST_CDC_EVENT_ACM_CONTROL_LINE_STATE_SET_COMPLETE _DATA

Table 4: Send Break

Function	USB_HOST_CDC_ACM_SendBreak
Event	USB_HOST_CDC_EVENT_ACM_SEND_BREAK_COMPLETE
Event Data Type	USB_HOST_CDC_EVENT_ACM_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_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. */
            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. */
            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_RESPONE_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. */
            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. */
            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.

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.

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

b) Data Transfer Functions

c) CDC Class-specific Functions

d) Data Types and Constants

Files

Files

Name	Description
usb_host_cdc_config_template.h	This is file usb_host_cdc_config_template.h.

Description

usb_host_cdc_config_template.h

This is file usb_host_cdc_config_template.h.

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.

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.

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 `usb_host_hid_mouse.c`.

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 USBTPList[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.

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.

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

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

b) Data Types and Constants

Files

Files

Name	Description
usb_host_hid_mouse.h	This is file usb_host_hid_mouse.h.
usb_host_hid_config_template.h	This is file usb_host_hid_config_template.h.

Description

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

usb_host_hid_mouse.h

This is file usb_host_hid_mouse.h.

usb_host_hid_config_template.h

This is file usb_host_hid_config_template.h.

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.

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.

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.

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 USBTPList[ 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.

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.

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

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

Files

Files

Name	Description
usb_host_hub.h	This is file usb_host_hub.h.

Description

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

usb_host_hub.h

This is file usb_host_hub.h.

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.

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 that media read, write and control operations. If 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.

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.

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 USBTPList[ 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 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;
            }

            break;

        default:
            break;

    }
}
```

Configuring the Library

Describes how to configure the USB Host Layer.

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.

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)

MHC ***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_msd.c`	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

b) Data Types and Constants

Files

Files

Name	Description
usb_host_msd.h	This is file usb_host_msd.h.
usb_host_msd_config_template.h	This is file usb_host_msd_config_template.h.

Description

usb_host_msd.h

This is file usb_host_msd.h.

usb_host_msd_config_template.h

This is file usb_host_msd_config_template.h.

USB Common Driver Interface

This section describes the interface that a USB peripheral driver should implement in order to function with the Harmony USB Stack

Common Interface

Provides information on the USB Driver interface that is common to all PIC32 devices.

Description

The USB Driver Common Interface definition specifies the functions and their behavior that a USB Driver must implement so that the driver can be used by the MPLAB Harmony USB Host and Device Stack.

Note: The MPLAB Harmony USB Driver for PIC32MX and PIC32MZ devices implements the USB Driver Common Interface.

The USB Driver Common Interface contains functions that are grouped as follows:

- *Driver System Functions* - These functions are called by MPLAB Harmony to initialize and maintain the operational state of the USB Driver. The system functions can vary between different PIC32 device USB Drivers. As such, the USB Driver Common Interface does not require these functions to be of the same type. These functions are not called by the USB Host or Device Stack and therefore are allowed to (and can) vary across different PIC32 device USB Drivers. A description of these functions, along with a description of how to initialize the USB Driver for Host, Device or Dual Role operation, is provided in the specific PIC32 device USB Driver help section (see PIC32MX USB Driver and PIC32MZ USB Driver).
- *Driver General Client Functions* -These functions are called by the USB Host or Device Stack to gain access to the driver
- *Driver Host Mode Client Functions* - These functions are called exclusively by the USB Host Stack to operate and access the USB as a Host
- *Driver Device Mode Client Functions* - These functions are called exclusively by the USB Device Stack to operate and access the USB as a Device

The USB Driver Common Interface is defined in the `<install-dir>\framework\driver\usb\drv_usb.h` file. This file contains the data types and structures that define the interface. Specifically, the DRV_USB_HOST_INTERFACE structure, contained in this file, is the common interface for USB Driver Host mode functions. It is a structure of function pointers, pointing to functions that define the Driver Host mode Client functions. The following code example shows this structure and the function pointer it contains.

```
// *****************************************************************************
/* USB Driver Client Functions Interface (For Host mode)

  Summary:
    Group of function pointers to the USB Driver Host mode Client Functions.

  Description:
    This structure is a group of function pointers pointing to the USB Driver
    Host mode Client routines. The USB Driver should export this group of
    functions so that the Host layer can access the driver functionality.

  Remarks:
    None.
*/

typedef struct
{
    /* This is a pointer to the driver Open function. This function may be
     * called twice in a Dual Role application, once by the Host Stack and then
     * by the Device Stack */
    DRV_HANDLE (*open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);

    /* This is pointer to the driver Close function */
    void (*close)(DRV_HANDLE handle);

    /* This is a pointer to the event call back set function */
    void (*eventHandlerSet)(DRV_HANDLE handle, uintptr_t hReferenceData,
           DRV_USB_EVENT_CALLBACK eventHandler);

    /* This is a pointer to the Host IRP submit function */
    USB_ERROR (*hostIRPSubmit)(DRV_USB_HOST_PIPE_HANDLE pipeHandle, USB_HOST_IRP * irp);

    /* This is a pointer to the Host IRP Cancel all function */
    void (*hostIRPCancel)(USB_HOST_IRP * irp);

    /* This is pointer to the Host event disable function */
    bool (*hostEventsDisable)(DRV_HANDLE handle);

    /* This is a pointer to the Host event enable function */
    void (*hostEventsEnable)(DRV_HANDLE handle, bool eventContext);

    /* This is a pointer to the Host pipe setup function */
    DRV_USB_HOST_PIPE_HANDLE (*hostPipeSetup)
    (
        DRV_HANDLE client,
        uint8_t deviceAddress,
        USB_ENDPOINT endpointAndDirection,
        uint8_t hubAddress,
        uint8_t hubPort,
        USB_TRANSFER_TYPE pipeType,
        uint8_t bInterval,
        uint16_t wMaxPacketSize,
        USB_SPEED speed
    );

    /* This is a pointer to the Host Pipe Close function */
    void (*hostPipeClose)(DRV_USB_HOST_PIPE_HANDLE pipeHandle);

    /* This is a pointer to the Host Root Hub functions */
    DRV_USB_ROOT_HUB_INTERFACE rootHubInterface;

} DRV_USB_HOST_INTERFACE;
```

The DRV_USB_DEVICE_INTERFACE structure, contained in this file, is the common interface for USB Driver Device mode functions. It is a structure of function pointers, pointer to functions that define the Driver Device mode Client functions. The following code example shows this structure and the function pointer it contains.

```
// *****************************************************************************
/* USB Driver Client Functions Interface (For Device Mode)

  Summary:
    Group of function pointers to the USB Driver Device Mode Client Functions.

  Description:
    This structure is a group of function pointers pointing to the USB Driver
    Device Mode Client routines. The USB Driver should export this group of
    functions so that the Device Layer can access the driver functionality.

  Remarks:
    None.
*/

typedef struct
{
    /* This is a pointer to the driver Open function */
    DRV_HANDLE (*open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);

    /* This is pointer to the driver Close function */
    void (*close)(DRV_HANDLE handle);

    /* This is a pointer to the event call back set function */
    void (*eventHandlerSet)(DRV_HANDLE handle, uintptr_t hReferenceData,
           DRV_USB_EVENT_CALLBACK eventHandler);

    /* This is a pointer to the device address set function */
    void (*deviceAddressSet)(DRV_HANDLE handle, uint8_t address);

    /* This is a pointer to the device current speed get function */
    USB_SPEED (*deviceCurrentSpeedGet)(DRV_HANDLE handle);

    /* This is a pointer to the SOF Number get function */
    uint16_t (*deviceSOFNumberGet)(DRV_HANDLE handle);

    /* This is a pointer to the device attach function */
    void (*deviceAttach)(DRV_HANDLE handle);

    /* This is a pointer to the device detach function */
    void (*deviceDetach)(DRV_HANDLE handle);

    /* This is a pointer to the device endpoint enable function */
    USB_ERROR (*deviceEndpointEnable)(DRV_HANDLE handle, USB_ENDPOINT endpoint,
                USB_TRANSFER_TYPE transferType, uint16_t endpointSize);

    /* This is a pointer to the device endpoint disable function */
    USB_ERROR (*deviceEndpointDisable)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is a pointer to the device endpoint stall function */
    USB_ERROR (*deviceEndpointStall)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is a pointer to the device endpoint stall clear function */
    USB_ERROR (*deviceEndpointStallClear)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is pointer to the device endpoint enable status query function */
    bool (*deviceEndpointIsEnabled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is pointer to the device endpoint stall status query function */
    bool (*deviceEndpointIsStalled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is a pointer to the device IRP submit function */
    USB_ERROR (*deviceIRPSubmit)(DRV_HANDLE handle, USB_ENDPOINT endpoint,
                USB_DEVICE_IRP * irp);
    /* This is a pointer to the device IRP Cancel all function */
    USB_ERROR (*deviceIRPCancelAll)(DRV_HANDLE handle, USB_ENDPOINT endpoint);

    /* This is a pointer to the device remote wakeup start function */
    void (*deviceRemoteWakeupStart)(DRV_HANDLE handle);

    /* This is a pointer to the device remote wakeup stop function */
    void (*deviceRemoteWakeupStop)(DRV_HANDLE handle);

    /* This is a pointer to the device Test mode enter function */
    USB_ERROR (*deviceTestModeEnter)(DRV_HANDLE handle, USB_TEST_MODE_SELECTORS testMode);

} DRV_USB_DEVICE_INTERFACE;
```

Both of these structures also contain pointers to General Client functions. The specific PIC32 device USB Driver allocates and initializes such a structure. The following code example shows how the PIC32MX USB Host mode Driver allocates and initializes the DRV_USB_HOST_INTERFACE structure. This code is contained in the `<install-dir>\framework\driver\usb\usbhs\src\dynamic\drv_usbfs_host.c` file.

```
/**********************************************************
 * This structure is a set of pointer to the USBFS driver
 * functions. It is provided to the Host layer as the
 * interface to the driver.
 * ******************************************************/

DRV_USB_HOST_INTERFACE gDrvUSBFSHostInterface =
{
    .open = DRV_USBFS_Open,
    .close = DRV_USBFS_Close,
    .eventHandlerSet = DRV_USBFS_ClientEventCallBackSet,
    .hostIRPSubmit = DRV_USBFS_HOST_IRPSubmit,
    .hostIRPCancel = DRV_USBFS_HOST_IRPCancel,
    .hostPipeSetup = DRV_USBFS_HOST_PipeSetup,
    .hostPipeClose = DRV_USBFS_HOST_PipeClose,
    .hostEventsDisable = DRV_USBFS_HOST_EventsDisable,
    .hostEventsEnable = DRV_USBFS_HOST_EventsEnable,
    .rootHubInterface.rootHubPortInterface.hubPortReset = DRV_USBFS_HOST_ROOT_HUB_PortReset,
    .rootHubInterface.rootHubPortInterface.hubPortSpeedGet =
                                                    DRV_USBFS_HOST_ROOT_HUB_PortSpeedGet,
    .rootHubInterface.rootHubPortInterface.hubPortResetIsComplete =
                                                    DRV_USBFS_HOST_ROOT_HUB_PortResetIsComplete,
    .rootHubInterface.rootHubPortInterface.hubPortSuspend = DRV_USBFS_HOST_ROOT_HUB_PortSuspend,
    .rootHubInterface.rootHubPortInterface.hubPortResume = DRV_USBFS_HOST_ROOT_HUB_PortResume,
    .rootHubInterface.rootHubMaxCurrentGet = DRV_USBFS_HOST_ROOT_HUB_MaximumCurrentGet,
    .rootHubInterface.rootHubPortNumbersGet = DRV_USBFS_HOST_ROOT_HUB_PortNumbersGet,
    .rootHubInterface.rootHubSpeedGet = DRV_USBFS_HOST_ROOT_HUB_BusSpeedGet,
    .rootHubInterface.rootHubInitialize = DRV_USBFS_HOST_ROOT_HUB_Initialize,
    .rootHubInterface.rootHubOperationEnable = DRV_USBFS_HOST_ROOT_HUB_OperationEnable,
    .rootHubInterface.rootHubOperationIsEnabled = DRV_USBFS_HOST_ROOT_HUB_OperationIsEnabled,
};
```

Similarly, the PIC32MX USB Device mode Driver allocates and initializes the DRV_USB_DEVICE_INTERFACE structure. This can be reviewed in the `<install-dir>\framework\driver\usb\usbhs\src\dynamic\drv_usbfs_device.c` file.

```
/*****************************************************
 * This structure is a pointer to a set of USB Driver
 * Device mode functions. This set is exported to the
 * Device Layer when the Device Layer must use the
 * PIC32MX USB Controller.
 *****************************************************/

DRV_USB_DEVICE_INTERFACE gDrvUSBFSDeviceInterface =
{
    .open = DRV_USBFS_Open,
    .close = DRV_USBFS_Close,
    .eventHandlerSet = DRV_USBFS_ClientEventCallBackSet,
    .deviceAddressSet = DRV_USBFS_DEVICE_AddressSet,
    .deviceCurrentSpeedGet = DRV_USBFS_DEVICE_CurrentSpeedGet,
    .deviceSOFNumberGet = DRV_USBFS_DEVICE_SOFNumberGet,
    .deviceAttach = DRV_USBFS_DEVICE_Attach,
    .deviceDetach = DRV_USBFS_DEVICE_Detach,
    .deviceEndpointEnable = DRV_USBFS_DEVICE_EndpointEnable,
    .deviceEndpointDisable = DRV_USBFS_DEVICE_EndpointDisable,
    .deviceEndpointStall = DRV_USBFS_DEVICE_EndpointStall,
    .deviceEndpointStallClear = DRV_USBFS_DEVICE_EndpointStallClear,
    .deviceEndpointIsEnabled = DRV_USBFS_DEVICE_EndpointIsEnabled,
    .deviceEndpointIsStalled = DRV_USBFS_DEVICE_EndpointIsStalled,
    .deviceIRPSubmit = DRV_USBFS_DEVICE_IRPSubmit,
    .deviceIRPCancelAll = DRV_USBFS_DEVICE_IRPCancelAll,
    .deviceRemoteWakeupStop = DRV_USBFS_DEVICE_RemoteWakeupStop,
    .deviceRemoteWakeupStart = DRV_USBFS_DEVICE_RemoteWakeupStart,
    .deviceTestModeEnter = NULL


};
```

A pointer to the DRV_USB_HOST_INTERFACE structure is passed to the USB Host Stack as part of USB Host Stack initialization. The following code example shows how this is done.

```
/********************************************************************************
 * This is a table of the USB Host mode drivers that this application will
 * support. Also contained in the driver index. In this example, the
 * application will want to use instance 0 of the PIC32MX USB Full-Speed driver.
 * *****************************************************************************/
const USB_HOST_HCD hcdTable =
{
    .drvIndex = DRV_USBFS_INDEX_0,
    .hcdInterface = DRV_USBFS_HOST_INTERFACE
};

/* Here the pointer to the USB Driver Common Interface is provided to the USB
 * Host Layer via the hostControllerDrivers member of the Host Layer
 * Initialization data structure. */
const USB_HOST_INIT usbHostInitData =
{
    .nTPLEntries = 1 ,
    .tplList = (USB_HOST_TPL_ENTRY *)USBTPList,
    .hostControllerDrivers = (USB_HOST_HCD *)&hcdTable

};
```

A pointer to the DRV_USB_DEVICE_INTERFACE structure is passed to the USB Device Stack as part of the USB Device Stack initialization. The Host Stack and Device Stack then access the driver functions through the function pointers contained in these structures.

The Driver General Client, Host mode and Device mode Client functions are described in this section. Any references to a USB Driver Client in the following sections, implies the client is a USB Host Stack and/or the USB Device Stack.

Driver Host Mode Client Functions

Provides information on the Host mode Client functions for the USB Driver.

Description

The DRV_USB_HOST_INTERFACE structure contains pointers to the USB Driver’s Host mode Client functions. These functions are only applicable when the USB module is operating as a USB Host. Along with the function pointers to the driver’s Host mode specific functions, the DRV_USB_HOST_INTERFACE structure also contains another structure of function pointers of the type DRV_USB_ROOT_HUB_INTERFACE. This structure contains function pointers to the USB Driver’s Root Hub functions. A USB Driver must implement these functions and ensure that the Host Stack can access these functions through the driver’s DRV_USB_HOST_INTERFACE structure. The Driver Host mode Client functions in the DRV_USB_HOST_INTERFACE structure are:

- Driver Host Pipe Setup Function
- Driver Host Pipe Close Function
- Driver Host Events Disable Function
- Driver Host Events Enable Function
- Driver Host IRP Submit Function

- Driver Host IRP Cancel Function

Driver Host Pipe Setup Function

The hostPipeSetup member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host Pipe Setup function. The signature of the Host Pipe Setup function is as follows:

```
DRV_USB_HOST_PIPE_HANDLE (*hostPipeSetup)  ( DRV_HANDLE client,  uint8_t deviceAddress,
                           USB_ENDPOINT endpointAndDirection, uint8_t hubAddress,
                           uint8_t hubPort, USB_TRANSFER_TYPE pipeType, uint8_t bInterval,
                           uint16_t wMaxPacketSize, USB_SPEED speed);
```

The USB Driver Host mode Pipe Setup function must match this signature. The USB Host Stack calls this function to create a communication pipe to the attached device. The function parameters define the property of this communication pipe. The driverHandle parameter is the handle to the driver obtained through the driver Open function. The deviceAddress and the endpointAddress parameters specify the address of the USB device and the endpoint on this device to which this pipe must connect.

If the device is connected to the Host though a hub, hubAddress and hubPort must specify the address of the hub and port to which the device is connected. The USB Driver will use these parameters to schedule split transactions if the target device is a Low-Speed or Full-Speed device and is connected to the Host through a high-speed hub. If the device is connected directly to the Host, these parameters should be set to zero ('0').

The pipeType parameter specifies the type of USB transfers that this pipe would support. The bInterval parameter is interpreted as per the USB 2.0 Specification based on the transfer type and the speed of the pipe. The wMaxPacketSize parameter defines the maximum size of a transaction that the driver should use while transporting a transfer on the pipe. The Host layer will use the information obtained from the USB device descriptors of the attached device to decide the wMaxPacketSize parameter.

The Driver Host Pipe Setup function should be thread-safe, but does not have to be event safe. The Host layer (or the Host Client Drivers) will not, and should not attempt to create a pipe in an interrupt, and therefore, an event context. The function should return DRV_USB_PIPE_HANDLE_INVALID if the driver could not open the pipe. The driver may not be able to open a pipe due to incorrect function parameters or due to lack of resources.

Driver Host Pipe Close Function

The hostPipeClose member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host Pipe Close function. The signature of the Host Pipe Close function is as follows:

```
void (*hostPipeClose)(DRV_USB_HOST_PIPE_HANDLE pipeHandle);
```

The USB Driver Host mode Pipe Close function must match this signature. The USB Host Stack calls this function to close communication pipes. The pipeHandle parameter is the pipe handle obtained from the Pipe Setup function. The Host Client Driver typically closes pipes when a device detach was detected. The Client Driver may also close pipes when a device configuration needs to change or when the Client Driver is being unloaded by the Host. The Pipe Close function has no side effect if the pipe handle is invalid. Closing the pipe will abort all I/O Request Packets (IRP) that are scheduled on the pipe. Any transaction in progress will complete. The IRP callback functions for each IRP scheduled in the pipe will be called with a USB_HOST_IRP_STATUS_ABORTED status.

The USB Driver Pipe Close function must be thread-safe and event-safe. The latter requirement allows the Pipe Close function to be called in the context of the device detach Interrupt Service Routine.

Driver Host Event Disable Function

The hostEventsDisable member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host mode Driver Events Disable function. The signature of the Events Disable function is as follows:

```
bool (*hostEventsDisable)(DRV_HANDLE handle);
```

The USB Driver Host mode Driver Events Disable function must match this signature. The Host Stack will call this function when it wants to execute a section of code that should not be interrupted by the USB Driver. Calling this function should disable USB Driver event generation. The handle parameter is set to the driver handle obtained via the driver Open function. The function will return the present state of the event generation, whether it is enabled or disabled. The Host Stack will pass this value to the USB Driver Host mode Driver Events Enable function when it needs to enable the driver events.

Driver Host Events Enable Function

The hostEventsEnable member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host mode Driver Events Enable function. The signature of the events enable function is as follows:

```
void (*hostEventsEnable)(DRV_HANDLE handle, bool eventContext);
```

The USB Driver Host mode Driver Events Enable function must match this signature. The USB Host Stack calls this function to re-enable the USB Driver Host mode Events (if they were enabled) after it called the USB Driver Host mode Events Disable function to disable driver events. The handle parameter is set to the driver handle obtained via the driver Open function. The `eventContext` parameter is set to the value returned by the Host mode Driver Events Disable function. The USB Driver will use the `eventContext` parameter to restore the event generation status (enabled or disabled) to what it was when the USB Driver Host mode Driver Events Disable function was called.

Driver Host IRP Submit Function

The `hostIRPSubmit` member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host IRP Submit function. The signature of the IRP Submit function is as follows:

```
USB_ERROR (*hostIRPSubmit)(DRV_USB_HOST_PIPE_HANDLE pipeHandle, USB_HOST_IRP * irp);
```

The USB Driver Host IRP Submit function must match this signature. The Host Stack calls this function to submit an IRP to the USB Driver. The USB Driver provides this mechanism to transfer data between the Host Stack and the attached device. The `pipeHandle` parameter should be set to the pipe handle obtained by the Pipe Setup function. The pipe handle specifies the pipe, and therefore, the target device, endpoint, speed and transfer type, on which the I/O must be processed. The `irp` parameter should point to the IRP data structure. The IRP data structure will transport an entire transfer over the pipe. The USB Driver will split up the transfer into transactions based on the parameters specified at the time of pipe creation. This process does not require Host Stack intervention.

The function will return USB_ERROR_HOST_PIPE_INVALID if the pipe handle is not valid. It will return USB_ERROR_OSAL_FUNCTION if an error occurred while performing a RTOS-related operation. It will return USB_ERROR_NONE if the IRP was submitted successfully.

The USB Driver will queue the IRP if there is already an IRP being processed on the pipe. The completion of the IRP processing is indicated by the USB Driver calling the IRP Callback function specified within the IRP. The Host IRP Submit function must be thread-safe and IRP callback-safe. The Host Stack may resubmit the IRP within the IRP Callback function. The IRP Callback function itself executes within an interrupt context. The completion status of the IRP will be available in the `status` member of the IRP when the IRP callback function is invoked.

Driver Host IRP Cancel Function

The `hostIRPCancel` member of the DRV_USB_HOST_INTERFACE structure should point to the USB Driver Host IRP Cancel function. The signature of the IRP Cancel function is as follows

```
void (*hostIRPCancel)(USB_HOST_IRP * irp);
```

The USB Driver Host IRP Cancel function must match this signature. The Host Stack and Host Client Drivers will call this function to cancel an IRP that was submitted. The IRP will be aborted successfully if it is not in progress. If the IRP processing has begun, the on-going transaction will complete and pending transactions in the transfer will be aborted. In either case, the IRP Callback function will be called with the IRP status as USB_HOST_IRP_STATUS_ABORTED.

Driver Host Root Hub Interface

Provides information on the Root Hub interface for the USB Host Driver.

Description

The USB Driver Common Interface requires the USB Driver to be operating in Host mode to provide root hub control functions. If the USB peripheral does not contain root hub features in hardware, these features must be emulated in software by the driver. The USB peripheral on PIC32MX and PIC32MZ devices does not contain root hub features; therefore, the USB Driver for these peripherals emulates the root hub functionality in software. The `rootHubInterface` member of the DRV_USB_HOST_INTERFACE structure is a structure of type DRV_USB_ROOT_HUB_INTERFACE. The members of this structure are function pointers to the root hub control functions of the USB Driver.

Along with other Host mode functions, the USB Driver while operating in Host mode must also ensure that the `rootHubInterface` member of DRV_USB_HOST_INTERFACE is set up correctly so that the USB Host Stack can access the root hub functions. Descriptions of the function pointer types in the DRV_USB_ROOT_HUB_INTERFACE include:

- Driver Host Root Hub Speed Get Function
- Driver Host Root Hub Port Numbers Get Function
- Driver Host Root Hub Maximum Current Get Function
- Driver Host Root Hub Operation Enable Function
- Driver Host Root Hub Operation Enable Status Function
- Driver Host Root Hub Initialize Function

Driver Host Root Hub Speed Get Function

The `rootHubSpeedGet` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub

Speed Get function. The signature of this function is as follows:

```
USB_SPEED (*rootHubSpeedGet)(DRV_HANDLE handle);
```

The USB Driver Root Hub Speed Get function must match this signature. The USB Host Stack calls this function to identify the speed at which the root hub is operating. The `handle` parameter is the handle obtained by calling the USB Driver Open function. The operation speed is configured by the USB Driver initialization and depends on the capability of the USB peripheral. For example, the USB peripheral on PIC32MZ devices supports both Hi-Speed and Full-Speed Host mode operation. It can be configured through initialization to only operate at Full-Speed. The Root Hub Speed Get function must return the USB speed at which the USB peripheral is operating. This should not be confused with the speed of the attached device.

Driver Host Root Hub Port Numbers Get Function

The `rootHubPortNumbersGet` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Numbers Get function. The signature of this function is as follows:

```
USB_SPEED (*rootHubSpeedGet)(DRV_HANDLE handle);
```

The USB Driver Root Hub Speed Get function must match this signature. This function should return the number of ports that the root hub contains. On the USB peripheral for both PIC32MZ and PIC32MX devices, this value is always '1'.

Driver Host Root Hub Maximum Current Get Function

The `rootHubMaxCurrentGet` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Maximum Current Get function. The signature of this function is as follows:

```
uint32_t (*rootHubMaxCurrentGet)(DRV_HANDLE handle);
```

The USB Driver Root Hub Maximum Current Get function must match this signature. This function returns the maximum VBUS current that the root hub can provide. The USB Host Stack calls this function to know the maximum current that the root hub VBUS power supply can provide. This value is then used to determine if the Host can support the current requirements of the attached device. The `handle` parameter is the driver handle obtained by calling the driver Open function.

The PIC32MX and the PIC32MZ USB peripherals cannot supply VBUS. The root hub driver only switches the VBUS supply. The current rating of the VBUS is specified through the USB Driver initialization. The root hub maximum current get function implementation in these drivers returns this value to the Host Stack.

Driver Host Root Hub Operation Enable Function

The `rootHubOperationEnable` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Operation Enable function. The signature of this function is as follows"

```
void (*rootHubOperationEnable)(DRV_HANDLE handle, bool enable);
```

The USB Driver Root Hub Operation Enable function must match this signature. The USB Host Stack calls this function when it ready to receive device attach events from the root hub. Calling this function will cause the USB Driver root hub functionality to enable detection of device attach and detach. The USB Driver will then raise events to the USB Host Stack. The `handle` parameter is the driver handle obtained by calling the driver Open function. Setting the `enable` parameter to true enables the root hub operation. Setting the `enable` parameter to false disables the root hub operation.

Driver Host Root Hub Operation Enable Status Function

The `rootHubOperationIsEnabled` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Operation Enable Status function. The signature of this function is as follows:

```
bool (*rootHubOperationIsEnabled)(DRV_HANDLE handle);
```

The USB Driver Root Hub Operation Enable Status function must match this signature. This USB Host Stack calls this function after calling the operation enable function to check if this has completed. The function returns true if the operation enable function has completed. The USB Host Stack will call this function periodically until it returns true.

Driver Host Root Hub Initialize Function

The `rootHubInitialize` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Initialize function. The signature of this function is as follows:

```
void (*rootHubInitialize)(DRV_HANDLE handle, USB_HOST_DEVICE_OBJ_HANDLE usbHostDeviceInfo);
```

The USB Driver Root Hub Initialize function must match this signature. The USB Host Stack calls this function to assign a device identifier (usbHostDeviceInfo) to the root hub. This function is called before the Host Stack enables the root hub operation. The USB Driver root hub should use this identifier as the parent identifier when it calls the USB_HOST_DeviceEnumerate function to enumerate the attached device. At the time of enumeration, the USB Host Stack will use this parent identifier to identify the parent hub (whether root hub or external hub) of the attached device. The USB Driver root hub should retain the `usbHostDeviceInfo` parameter for the life time of its operation.

Driver Host USB Root Hub Port Interface

Provides information on the Root Hub Port interface of the USB Host Driver.

Description

The `rootHubPortInterface` member of the DRV_USB_ROOT_HUB_INTERFACE structure should point to the USB Driver Root Hub Port functions. The data type of this member is USB_HUB_INTERFACE. This data type is a structure containing function pointers pointing to the port control functions of the root hub. The USB Driver must assign the function pointers in this structure to the root hub port control functions. These same functions are also exported by a Hub Driver to the USB Host Stack, which allow the Host Stack to control a device regardless of whether it is connected to a root hub or an external hub. The port functions are valid only when a device is attached to the port. The behavior of these functions on a port to which no device is connected is not defined. Descriptions of the port control functions are provided, which include:

- Driver Host Hub Port Reset Function
- Driver Host Hub Port Reset Completion Status Function
- Driver Host Hub Port Suspend Function
- Driver Host Hub Port Resume Function
- Driver Host Hub Port Speed Get Function

Driver Host Hub Port Reset Function

The `hubPortReset` member of the USB_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Reset function. The signature of this function is as follows:

```
USB_ERROR (*hubPortReset)(uintptr_t hubAddress, uint8_t port);
```

The USB Driver Root Hub Port Reset function must follow this signature. This function starts reset signaling on the port. If the device is connected to the root hub, the USB Host Stack will set the `hubAddress` parameter to the driver handle obtained through the driver Open function. The USB Host Stack uses the parent identifier provided by the root hub driver when the USB_HOST_DeviceEnumerate function was called to query the driver handle that is linked to this root hub. If the device is connected to an external hub, the `hubAddress` parameter is directly set to the parent identifier.

For the PIC32MX and PIC32MZ USB Drivers, the `port` parameter is ignored. For an external hub, this must be set to the port to which the device is connected. The function returns USB_ERROR_NONE if the function was successful. If the reset signaling is already in progress on the port, calling this function has no effect. The USB Driver will itself time duration of the reset signal. This does not require USB Host Stack intervention. The USB Host Stack will call the port reset completion status function to check if the reset signaling has completed. Calling this function on a port which exists on an external hub will cause the hub driver to issue a control transfer to start the port reset procedure.

Driver Host Hub Port Reset Completion Status Function

The `hubPortResetIsComplete` member of the USB_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Reset Completion Status function. The signature of this function is as follows:

```
bool (*hubPortResetIsComplete)(uintptr_t hubAddress, uint8_t port);
```

The USB Driver Root Hub Port Reset Completion Status function must follow this signature. The USB Host Stack calls this function to check if the port reset sequence that was started on a port has completed. The function returns true if the reset signaling has completed. If the device is connected to the root hub, the USB Host Stack will set the `hubAddress` parameter to the driver handle obtained through the driver Open function. If the device is connected to an external hub, the `hubAddress` parameter is directly set to the parent identifier.

For the PIC32MX and PIC32MZ USB Drivers, the `port` parameter is ignored. For an external hub, this parameter must be set to the port to which the device is connected.

Driver Host Hub Port Suspend Function

The `hubPortSuspend` member of the USB_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Suspend function. The signature of this function is as follows:

```
USB_ERROR(*hubPortSuspend)(uintptr_t hubAddress, uint8_t port);
```

The USB Driver Root Hub Port Suspend function must follow this signature. The USB Host Stack calls this function to suspend the port. If the device is connected to the root hub, the USB Host Stack will set the `hubAddress` parameter to the driver handle obtained through the driver Open function. If the device is connected to an external hub, the `hubAddress` parameter is directly set to the parent identifier.

For the PIC32MX and PIC32MZ USB Drivers, the `port` parameter is ignored. For an external hub, this parameter must be set to the port to which the device is connected. The function returns USB_ERROR_NONE if the request was successful. Calling this function on a suspended port will not have any effect.

Driver Host Hub Port Resume Function

The `hubPortResume` member of the USB_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Resume function. The signature of this function is as follows:

```
USB_ERROR(*hubPortResume)(uintptr_t hubAddress, uint8_t port);
```

The USB Driver Root Hub Port Resume function must follow this signature. The USB Host Stack calls this function to resume a suspended port. If the device is connected to the root hub, the USB Host Stack will set the `hubAddress` parameter to the driver handle obtained through the driver Open function. If the device is connected to an external hub, the `hubAddress` parameter is directly set to the parent identifier.

For the PIC32MX and PIC32MZ USB Drivers, the `port` parameter is ignored. For an external hub, this parameter must be set to the port to which the device is connected. The function returns USB_ERROR_NONE if the request was successful. Calling this function on a port that is not suspended will not have any effect.

Driver Host Hub Port Speed Get Function

The `hubPortSpeedGet` member of the USB_HUB_INTERFACE structure should point to the USB Driver Root Hub Port Speed Get function. The signature of this function is as follows:

```
USB_SPEED(*hubPortSpeedGet)(uintptr_t hubAddress, uint8_t port);
```

The USB Driver Root Hub Port Speed Get function must follow this signature. The USB Host Stack calls this function to obtain the USB speed of the device that is attached to the port. The Host Stack calls this function only after it has completed reset of the port. If the device is connected to the root hub, the USB Host Stack will set the `hubAddress` parameter to the driver handle obtained through the driver Open function. If the device is connected to an external hub, the `hubAddress` parameter is directly set to the parent identifier.

For the PIC32MX and PIC32MZ USB Drivers, the `port` parameter is ignored. For an external hub, this parameter must be set to the port to which the device is connected. The function returns USB_SPEED_ERROR if the request was not successful. It will return the functional USB speed otherwise.

This concludes the section describing the USB Driver Host mode Client Functions. The USB Driver Device Mode Client Functions are discussed in the next section.

Driver Device Mode Client Functions

Provides information on the USB Driver Device mode Client functions.

Description

The DRV_USB_DEVICE_INTERFACE structure contains pointers to the USB Driver's Device mode Client Functions. These functions are only applicable when the USB module is operating as a USB Device. A USB Driver must implement these functions and ensure that the Device Stack can access these functions through the driver's DRV_USB_DEVICE_INTERFACE structure. Descriptions of the Driver Device Mode Client functions in the DRV_USB_DEVICE_INTERFACE structure include:

- Driver Device Address Set Function
- Driver Device Current Speed Get Function
- Driver Device SOF Number Get Function
- Driver Device Attach Function
- Driver Device Detach Function
- Driver Device Endpoint Enable Function
- Driver Device Endpoint Disable Function
- Driver Device Endpoint Stall Function
- Driver Device Endpoint Stall Clear Function
- Driver Device Endpoint Enable Status Function
- Driver Device Endpoint Stall Status Function
- Driver Device IRP Submit Function
- Driver Device IRP Cancel All Function
- Driver Device IRP Cancel Function
- Driver Device Remote Wakeup Start Function
- Driver Device Remote Wakeup Stop Function

- Driver Device Test Mode Enter Function

The PIC32MZ and the PIC32MX USB peripheral drivers implement the Device mode functions and export these functions to the Device Stack though their respective DRV_USB_DEVICE_INTERFACE structure.

Driver Device Address Set Function

The `deviceAddressSet` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device Address Set function. The signature of this function is as follows:

```
void (*deviceAddressSet)(DRV_HANDLE handle, uint8_t address);
```

The USB Driver Device Address Set Function should match this signature. The USB Device Stack will call this function to set the Device USB Address. The function will be called in an interrupt context and hence the function implementation must be interrupt-safe. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `address` parameter is the address provided by the USB Host through the Set Device Address Standard request.

Driver Device Current Speed Get Function

The `deviceCurrentSpeedGet` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Current Speed Get function. The signature of this function is as follows:

```
USB_SPEED (*deviceCurrentSpeedGet)(DRV_HANDLE handle);
```

The USB Driver Device Current Speed Get function should match this signature. The USB Device Stack will call this function to obtain the speed at which the device has connected to the USB. It will call this function after reset signaling has completed. The `handle` parameter is driver handle obtained from calling the driver Open function. This function is called in an interrupt context and should be interrupt-safe.

Driver Device SOF Number Get Function

The `deviceSOFNumberGet` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Start-Of-Frame Number Get function. The signature of this function is as follows:

```
uint16_t (*deviceSOFNumberGet)(DRV_HANDLE handle);
```

The USB Driver SOF Number Get function should match this signature. The USB Device Stack will call this function to obtain the current SOF number. The USB peripheral uses a 16 bit counter to count the number of SOFs that have occurred since USB reset. This value is returned along with the Device Stack Start of Frame event. This function is called from an interrupt context and should be interrupt-safe. The `handle` parameter is the driver handle obtained from calling the driver Open function.

Driver Device Attach Function

The `deviceAttach` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Attach function. The signature of this function is as follows:

```
uint16_t(*deviceAttach)(DRV_HANDLE handle);
```

The USB Driver Attach function should match this signature. The USB Device Stack will call this function when the Device application calls the USB Device Stack Device Attach function. The USB Driver will enable the required signaling resistors for indicate attach to the Host. The application could call this function in response to a VBUS power available event. This function must be interrupt-safe. The `handle` parameter is the driver handle obtained from calling the driver Open function.

Driver Device Detach Function

The deviceDetach member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Detach function. The signature of this function is as follows:

uint16_t(*deviceDetach)(DRV_HANDLE handle);

The USB Driver Detach function should match this signature. The USB Device Stack will call this function when the Device application calls the USB Device Stack Device Detach function. The USB Driver will disable the required signaling resistors to indicate detach to the Host. The application could call this function in response to a VBUS power not available event. This function should be interrupt-safe. The `handle` parameter is driver handle obtained from calling the driver Open function.

Driver Device Endpoint Enable Function

The `deviceEndpointEnable` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Enable function. The signature of this function is as follows:

```
USB_ERROR (*deviceEndpointEnable)(DRV_HANDLE handle, USB_ENDPOINT endpoint,
           USB_TRANSFER_TYPE transferType, uint16_t endpointSize);
```

The USB Driver Endpoint Enable function should match this signature. The USB Device Stack Function Driver will call this function when it is initialized by the USB Device Layer. The Device Layer, on receiving the Set Configuration request from the Host, identifies the function drivers that are required by the configuration and initializes them. The function drivers will call the endpoint enable function to enable the endpoints required for their operation. Enabling the endpoint will cause it reply to transaction requests from the Host and accept transfer requests from the device application.

The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) that should be enabled. The transferType is the type of the USB transfer that this endpoint will handle. The endpointSize is the size of the maximum transaction that the endpoint will handle. This should match the endpoint size communicated to the Host via the device endpoint descriptors.

The function will return USB_ERRROR_NONE if the endpoint was configured successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

The endpoint enable function will be called in an interrupt context and should be interrupt-safe. It is not expected to be thread safe. For standard function drivers, the endpoint enable function will be called in the context of the USB Device Layer Client. For vendor USB devices, the vendor application must call the endpoint enable function in response to and within the context of the device configured event. Again this event itself will execute in the context of the Device Layer.

Driver Device Endpoint Disable Function

The `deviceEndpointDisable` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Disable function. The signature of this function is as follows:

```
USB_ERROR (*deviceEndpointDisable)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Endpoint Disable function should match this signature. The USB Device Stack Function Driver will call this function when it is deinitialized by the USB Device Layer. The Device Layer will deinitialize function drivers when it receives a USB reset event from the driver or on receiving the Set Configuration request from the Host with configuration parameter 0. Disabling the endpoint will cause it NAK transaction request from the Host and not accept transfer requests from the device application.

The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) that should be disabled.

The function will return USB_ERRROR_NONE if the function executed successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

The endpoint disable function will be called in an interrupt context and should be interrupt-safe. It is not expected to be thread safe. For standard function drivers, the endpoint disable function will be called in the context of the USB Device Layer Client. For vendor USB devices, the vendor application must call the endpoint enable function in response to and within the context of the device reset event. Again this event itself will execute in the context of the Device Layer. Disabling the endpoint will not cancel any transfers that have been queued against the endpoint. The function drivers will call the IRP Cancel All function to cancel any pending transfers.

Driver Device Endpoint Stall Function

The `deviceEndpointStall` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Stall function. The signature of this function is as follows:

```
USB_ERROR (*deviceEndpointStall)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Endpoint Stall function should match this signature. The USB Device Stack Function Driver will call this function to stall an endpoint. The Device Layer itself will stall endpoint 0 for several reasons including non-support of the Host request or failure while executing the request. A function driver will also stall an endpoint for protocol specific reasons. The driver will stall both, receive and transmit directions when stalling Endpoint 0. The driver will stall the specified direction while stalling a non-zero endpoint.

This function must be thread safe and interrupt safe. Stalling the endpoint will abort all the transfers queued on the endpoint with the completion status set to USB_DEVICE_IRP_STATUS_ABORTED_ENDPOINT_HALT. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) that should be stalled. The function will return USB_ERRROR_NONE if the function executed successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

Driver Device Endpoint Stall Clear Function

The `deviceEndpointStallClear` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Stall Clear function. The signature of this function is as follows:

```
USB_ERROR (*deviceEndpointStallClear)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Endpoint Stall Clear function should match this signature. The USB Device Stack Function Driver will call this function to clear the stall on a non-zero endpoint. The Device Layer will call this function to clear the stall condition on Endpoint 0. Clearing the stall on a non-zero endpoint will clear all transfers scheduled on the endpoint and transfer completion status will be set to USB_DEVICE_IRP_STATUS_TERMINATED_BY_HOST. When the stall is cleared, the data toggle for non-zero endpoint will be set to DATA0. The data toggle on Endpoint 0 OUT endpoint will be set to DATA1. The USB Driver will clear the Stall condition on an endpoint even if it was not stalled.

This function must be thread safe and interrupt safe. Stalling the endpoint will flush all the transfers queued on the endpoint. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) whose stall condition must be cleared. The function will return USB_ERRROR_NONE if the function executed successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

Driver Device Endpoint Enable Status Function

The `deviceEndpointIsEnabled` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Enable Status function. The signature of this function is as follows:

```
bool (*deviceEndpointIsEnabled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Endpoint Enable Status function should match this signature. The USB Device Stack function will call this function to check if an endpoint has been enabled. The function returns true if the endpoint is enabled. The endpoint is enabled through the USB Driver Endpoint Enable function. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) whose enable status needs to be queried.

Driver Device Endpoint Stall Status Function

The `deviceEndpointIsStalled` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Endpoint Stall Status function. The signature of this function is as follows:

```
bool (*deviceEndpointIsStalled)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Endpoint Stall Status function should match this signature. The USB Device Stack function will call this function to check if an endpoint has been stalled. The function returns true if the endpoint is stalled. The endpoint is stalled through the USB Driver Endpoint Stall function. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) whose stall status needs to be queried.

Driver Device IRP Submit Function

The deviceIRPSubmit member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device IRP Submit function. The signature of the IRP submit function is as follows:

```
USB_ERROR (*deviceIRPSubmit)(DRV_HANDLE handle, USB_ENDPOINT endpoint, USB_DEVICE_IRP * irp);
```

The USB Driver Device IRP Submit function must match this signature. The Device Stack (USB Device calls this function to submit an IRP to the USB Driver. The USB Driver provides this mechanism to transfer data between the device and the Host. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter should set to endpoint through which transfer must be processed. The `irp` parameter should point to the Device IRP data structure. The IRP data structure will transport an entire transfer over the endpoint. The USB Driver will split up the transfer into transactions based on the endpoint size specified at the time of enabling the endpoint. This process does not require Device Stack intervention.

The function will return USB_ERRROR_NONE if the function executed successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid. It will return USB_ERROR_DEVICE_IRP_IN_USE if an in progress IRP is resubmitted. It will return USB_ERROR_ENDPOINT_NOT_CONFIGURED if the IRP is submitted to an endpoint that is not enabled.

The USB Driver will queue the IRP if there is already an IRP being processed on the endpoint. The completion of the IRP processing is indicated by the USB Driver calling the IRP callback function specified within the IRP. The Device IRP Submit function must be thread safe and IRP callback safe. The Device Stack may resubmit the IRP within the IRP callback function. The IRP callback function itself executes within an interrupt context. The completion status of the IRP will be available in the status member of the IRP when the IRP callback function is invoked.

Driver Device IRP Cancel All Function

The `deviceIRPCancelAll` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device IRP Cancel All function. The signature of this is as follows:

```
USB_ERROR (*deviceIRPCancelAll)(DRV_HANDLE handle, USB_ENDPOINT endpoint);
```

The USB Driver Device IRP Cancel All function must match this signature. The USB Device Stack will call this function before disabling the endpoint. Calling this function will call all IRPs that are queued on the endpoint to be canceled. The callback of each IRP will be invoked and the IRP completion status will be set to USB_DEVICE_IRP_STATUS_ABORTED. If an IRP is in progress, an ongoing transaction will be allowed to complete and pending transactions will be canceled. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `endpoint` parameter is the USB endpoint (which indicates the direction along with endpoint number) whose queued IRPs must be canceled.

The function is thread safe and interrupt safe and will return USB_ERRROR_NONE if it executed successfully. The function will return USB_ERROR_DEVICE_ENDPOINT_INVALID if the specified endpoint is not provisioned in the system configuration. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

Driver Device IRP Cancel Function

The `deviceIRPCancel` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device IRP Cancel function. The signature of this is as follows:

```
USB_ERROR (*deviceIRPCancel)(DRV_HANDLE handle, USB_DEVICE_IRP * IRP);
```

The USB Driver Device IRP Cancel function must match this signature. This function is called by the USB Device Stack function driver to cancel a scheduled IRP. If the IRP is in the queue but it's processing has not started, the IRP will removed from the queue and the IRP callback function will be called from within the cancel function. The callback will be invoked with the IRP completion status set to USB_DEVICE_IRP_STATUS_ABORTED. If an IRP is in progress, an ongoing transaction will be allowed to complete and pending transactions will be canceled. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `irp` parameter is the IRP to be canceled.

The function is thread safe and will return USB_ERRROR_NONE if it executed successfully. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid or if the IRP has status indicates that this IRP is not queued or not in progress. The application should not release the data memory associated with IRP unless the callback has been received.

Driver Device Remote Wakeup Start Function

The `deviceRemoteWakeupStart` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device Remote Wakeup Start function. The signature of this function is as follows:

```
void (*deviceRemoteWakeupStart)(DRV_HANDLE handle);
```

The USB Driver Device Remote Wakeup Start function must match this signature. The USB Device Stack will call the function when the device application wants to start remote wakeup signaling. This would happen if the device supports remote wake-up capability and this has been enabled by the Host. The `handle` parameter is the driver handle obtained from calling the driver Open function.

Driver Device Remote Wakeup Stop Function

The `deviceRemoteWakeupStop` member of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device Remote Wakeup Stop function. The signature of this function is as follows:

```
void (*deviceRemoteWakeupStop)(DRV_HANDLE handle);
```

The USB Driver Device Remote Wakeup Stop function must match this signature. The USB Device Stack will call the function when the device application wants to stop remote wakeup signaling. The application would call after calling the remote wakeup start function. The `handle` parameter is the driver handle obtained from calling the driver Open function.

Driver Device Test Mode Enter Function

The `deviceTestModeEnter` parameter of the DRV_USB_DEVICE_INTERFACE structure should point to the USB Driver Device Test Mode Enter function. The signature of this function is as follows:

```
USB_ERROR (*deviceTestModeEnter)(DRV_HANDLE handle, USB_TEST_MODE_SELECTORS testMode);
```

The USB Driver Device Test Mode Enter function should match this signature. The USB Device Stack calls this driver function to place the driver into test mode. This is required when the USB Host (operating at Hi-Speed) send the Set Feature request with the feature selector test set to test mode. This request also specifies which of the test mode signals, the driver should enable. The `handle` parameter is the driver handle obtained from calling the driver Open function. The `testMode` parameter should be set to one of the test modes as defined in table 9-7 of the USB 2.0 specification.

The test mode enter function is only supported by the PIC32MZ USB Driver as the USB peripheral on this controller supports Hi-Speed operation. The function will return USB_ERRROR_NONE if it executed successfully. It will return USB_ERROR_PARAMETER_INVALID if the driver handle is not valid.

This concludes the discussion on the DRV_USB_DEVICE_INTERFACE structure. The following sections describe using the USB Common Driver.

Driver General Client Functions

Provides information on the General Client functions for the USB Driver.

Description

The DRV_USB_HOST_INTERFACE and the DRV_USB_DEVICE_INTERFACE structures contain pointers to the USB Driver's

General Client functions. These functions are not specific to the operation mode (Host, Device, or Dual Role) of the driver. A USB Driver must implement these functions and ensure that the Host or Device Stack can access these functions through the driver's common interface structures. The common interface contains three general client functions:

- Driver Open Function
- Driver Close Function
- Driver Event Handler Set Function

Driver Open Function

The `open` member of the DRV_USB_HOST_INTERFACE and the DRV_USB_DEVICE_INTERFACE structures should point to the USB Driver Open function. The signature of the Open function is as follows:

```
DRV_HANDLE (*open)(const SYS_MODULE_INDEX drvIndex, const DRV_IO_INTENT intent);
```

The USB Driver Open function must match this signature. The Driver Client uses the USB Driver index (drvIndex) to specify the instance of the USB module that Host Stack or the Device Stack should open. The USB Driver should ignore the `intent` parameter. The function should return a driver handle. If the driver is not ready to be opened, it should return an invalid handle (DRV_HANDLE_INVALID). In such a case, the client will continue trying to open the driver by calling the Open function again. The driver may also fail to open for an invalid `index` parameter or if USB module is in an error condition.

When supporting Dual Role operation, both the Host Stack and Device Stack will call the Driver Open function in one application. The USB Driver must support multiple calls to the Open function in the same application. The Open function should be thread-safe.

Driver Close Function

The `close` member of the DRV_USB_HOST_INTERFACE and the DRV_USB_DEVICE_INTERFACE structures should point to the USB Driver Close function. The signature of the Close function is as follows:

```
void (*close)(DRV_HANDLE handle);
```

The USB Driver Close function must match this signature. The Driver Client passes the handle obtained from the Driver Open function as a parameter to the close. The USB Host Stack or USB Device Stack will close the driver only when the stack is deinitialized (which is typically a rare case). The USB Driver should deallocate any client-related resources in the Close function. If the specified driver handle is not valid, the Close function should not have any side effects. The USB Driver expects the Close function to be called from the context of the thread in which the driver was opened; therefore, this function *is not* expected to be thread-safe.

Driver Event Handler Set Function

The `eventHandlerSet` member of the DRV_USB_HOST_INTERFACE and the DRV_USB_DEVICE_INTERFACE structures should point to the USB Driver Event Handler Set function. The signature of the Event Handler Set function is as follows:

```
void (*eventHandlerSet)(DRV_HANDLE handle, uintptr_t hReferenceData, DRV_USB_EVENT_CALLBACK
eventHandler);
```

The USB Driver Event Handler Set function must match this signature. The signature of the Client Event Handling function should match DRV_USB_EVENT_CALLBACK. The USB Driver calls this function when it must communicate USB events to the client. The client can set the `eventHandler` parameter to NULL if it does not want to receive USB Driver events. The client will receive Host mode events if the USB Driver is operating in Host mode. It will receive Device mode events if the USB Driver is operating in Device mode. The DRV_USB_EVENT type enumeration contains all the possible events that the USB Driver would generate. The following code example shows the enumeration.

```
// *****************************************************************************
/* USB Driver Events Enumeration

  Summary:
    Identifies the different events that the USB Driver provides.

  Description:
    Identifies the different events that the USB Driver provides. The USB Driver
    should be able to provide these events.

  Remarks:
    None.
*/

typedef enum
{
    /* Bus error occurred and was reported. This event can be generated in both
     * Host and Device mode. */
    DRV_USB_EVENT_ERROR = 1,
    /* Host has issued a device Reset. This event occurs only in Device mode */
    DRV_USB_EVENT_RESET_DETECT,

    /* Resume detected while USB in suspend mode. This event can be generated in
     * both Host and Device mode. In Host mode, the events occurs when a remote
     * wakeup capable device has generated resume signaling. In Device mode,
     * this event will occur when the Host has issued resume signaling. */
    DRV_USB_EVENT_RESUME_DETECT,

    /* This event is generated in Device mode only. It occurs when the Host
     * suspends the bus and the bus goes idle. */
    DRV_USB_EVENT_IDLE_DETECT,

    /* This event is generated in Host mode and Device mode. In Host mode, this
     * event occurs when the device has stalled the Host. In Device mode, this
     * event occurs when the Host has accessed a stalled endpoint thus
     * triggering the device to send a STALL to the Host. */
    DRV_USB_EVENT_STALL,

    /* This event is generated in Host mode and Device mode. In Device mode,
     * this event occurs when a SOF has been generated by the Host. In Host
     * mode, this event occurs when controller is about to generate an SOF.
     * */
    DRV_USB_EVENT_SOF_DETECT,

    /* This event is generated in Device mode when the VBUS voltage is above
     * VBUS session valid. */
    DRV_USB_EVENT_DEVICE_SESSION_VALID,

    /* This event is generated in Device mode when the VBUS voltage falls
     * below VBUS session valid. */
    DRV_USB_EVENT_DEVICE_SESSION_INVALID,

} DRV_USB_EVENT;
```

This completes the discussion on the Driver General Client Functions.

Opening the Driver

Provides information and examples for opening the driver.

Description

The USB Host Stack and the USB Device Stack must obtain a handle to the USB Driver to access the functionality of the driver. This handle is obtained through the USB Driver Open function. The DRV_USB_DEVICE_INTERFACE structure and DRV_USB_DEVICE_HOST_INTERFACE structure provide access to the USB Driver Open function through the `open` member of these structures. Calling the Open function may not return a valid driver handle the first time the function is called. In fact, the USB Driver will return an invalid driver handle until the driver is ready to be opened. The Host and the Device Stack call the Open function repetitively in a state machine, until the function returns a valid handle.

The USB Host Stack can open the USB Driver but can call its Host mode functions only if the USB Driver was initialized for Host mode or Dual Role operation. The USB Host Stack accesses the driver functions through the DRV_USB_HOST_INTERFACE pointer that was provided to the Host Layer through the Host Stack initialization. The USB Device Stack can open the USB Driver but can call its Device mode functions only if the USB Driver was initialized for Device mode or Dual Role operation. The USB Device Stack accesses the driver functions through the DRV_USB_HOST_INTERFACE pointer that was provided to the Host Layer through the Host Stack initialization

The following code example shows how the USB Host Layer opens the USB Driver.

```
/* This code example shows how the Host Layer open the HCD via the hcdInterface.
 * The driver handle is stored in hcdHandle member of the busObj data structure.
 * The busObj data structure Host Layer local data structure. The Host Layer
 * opens the HCD when the bus is enabled. This operation takes place in the
 * USB_HOST_BUS_STATE_ENABLING state. */

/* Note the Host Layer calls the Open function by accessing the open member of
 * the hcdInterface which is of the type DRV_USB_HOST_INTERFACE. Also note how
 * the function is called repetitively until the Open function returns a valid
 * handle. */

case USB_HOST_BUS_STATE_ENABLING:

    /* The bus is being enabled. Try opening the HCD */
    busObj->hcdHandle = busObj->hcdInterface->open(busObj->hcdIndex, DRV_IO_INTENT_EXCLUSIVE |
            DRV_IO_INTENT_NONBLOCKING | DRV_IO_INTENT_READWRITE );

    /* Validate the Open function status */
    if (DRV_HANDLE_INVALID == busObj->hcdHandle )
    {
        /* The driver may not open the first time. This is okay. We
         * should try opening it again. The state of bus is not
         * changed. */
    }
```

The following code example shows how the USB Device Layer opens the USB Driver.

```
/* This code example shows how the USB Device Layer calls the USBCD open
 * function to open the USBCD. The Device Layer accesses the USBCD Open function
 * through the driverInterface member of the usbDeviceInstanceState object. The
 * driverInterface member is a DRV_USB_DEVICE_INTERFACE type. The
 * usbDeviceInstanceState is a USB Device Layer local object. */

/* The Device Layer attempts to open the USBCD when it is initializing. Note how
 * the Device Layer advances to the next state only when the USBCD returns a
 * valid handle.   */

switch(usbDeviceThisInstance->taskState)
{
    case USB_DEVICE_TASK_STATE_OPENING_USBCD:

        /* Try to open the driver handle. This could fail if the driver is
         * not ready to be opened. */
        usbDeviceThisInstance->usbCDHandle =
           usbDeviceThisInstance->driverInterface->open( usbDeviceThisInstance->driverIndex,
           DRV_IO_INTENT_EXCLUSIVE|DRV_IO_INTENT_NONBLOCKING|DRV_IO_INTENT_READWRITE);

        /* Check if the driver was opened */
        if(usbDeviceThisInstance->usbCDHandle != DRV_HANDLE_INVALID)
        {
            /* Yes the driver could be opened. */

            /* Advance the state to the next state */
            usbDeviceThisInstance->taskState = USB_DEVICE_TASK_STATE_RUNNING;

            /* Update the USB Device Layer state to indicate that it can be
             * opened */
            usbDeviceThisInstance->usbDeviceInstanceState = SYS_STATUS_READY;
        }

        break;
```

USB Driver Host Mode Operation

Provides information on Host mode operation.

Description

The USB Driver operates or can operate in the Host mode when it is initialized for Host mode or Dual Role operation. When operating in Host mode, the USB Driver is also referred to as the Host Controller Driver (HCD). In Dual Role mode, the USB Driver will switch to Host mode when the USB Driver Host Root Hub Operation Enable function is called.

The USB Driver Client must perform these steps to operate the USB Driver in Host mode.

1. Open the USB Driver to obtain the driver handle.
2. Set the event handler.

3. Call the Root Hub Control function to obtain the speed of the root hub, the number of ports that the root hub supports, and the maximum current that the root hub VBUS can supply.
4. Calls the Root Hub Initialize function with an identifier parameter. This `identifier` parameter allows the Host Stack to uniquely identify the root hub when where there are multiple root hubs.
5. The Driver Client will then enable the root hub operation and will wait until the root hub operation is enabled.
6. The Driver Client can now call the USB Driver Host mode functions.

The following sections explain Steps 2 through 6 in more detail.

Handling Host Mode Driver Events

Currently, the HCD does not provide any events to the client. The client can optionally register an event handler through the eventHandlerSet function pointer in the DRV_USB_HOST_INTERFACE structure. Future releases of the USB Driver may contain features that provide events to the Driver Client. Please refer to the following **Root Hub Operation** section for details on how the driver indicates device attach and detach to the client.

Root Hub Operation

A key feature of the HCD is the Root Hub Driver. The Root Hub Driver emulates hub operation in USB Driver software and provides a hub like interface to the USB Host Layer. The USB Host Layer treats the root hub like an external hub. This simplifies the implementation of USB Host Layer while supporting multiple devices through a hub. In that, the USB Host layer does not have to treat a device connected directly to the USB peripheral differently than a device connected to an external hub. The following code example shows how the USB Host Layer calls the root hub function to obtain information about the root hub.

```
/* This code example shows how the USB Host Layer calls the root hub functions to
 * obtain information about the root. The USB Host Layer first opens the HCD and
 * then accesses the root hub functions through the rootHubInterface member of
 * hcdInterface. rootHubInterface is of the type DRV_USB_ROOT_HUB_INTERFACE and
 * the hcdInterface is of the type of DRV_USB_HOST_INTERFACE. */

/* The code example shows how the Host Layer gets to know the root hub operation
 * speed, number of root hub ports and the maximum amount of current that the
 * root can supply. These function can be called only after HCD was opened and a
 * valid driver handle obtained. */

case USB_HOST_BUS_STATE_ENABLING:

    /* The bus is being enabled. Try opening the HCD */
    busObj->hcdHandle = busObj->hcdInterface->open(busObj->hcdIndex, DRV_IO_INTENT_EXCLUSIVE |
            DRV_IO_INTENT_NONBLOCKING | DRV_IO_INTENT_READWRITE );

    /* Validate the Open function status */
    if (DRV_HANDLE_INVALID == busObj->hcdHandle )
    {
        /* The driver may not open the first time. This is okay. We
         * should try opening it again. The state of bus is not
         * changed. */
    }
    else
    {
        /* Update the bus root hub information with the
         * details of the controller. Get the bus speed, number of
         * ports, the maximum current that the HCD can supply,
         * pointer to the root hub port functions. */

        SYS_DEBUG_PRINT(SYS_ERROR_INFO,
             "\r\nUSB Host Layer: Bus %d Root Hub Driver Opened.",hcCount);

        busObj->rootHubInfo.speed =
             busObj->hcdInterface->rootHubInterface.rootHubSpeedGet(busObj->hcdHandle);

        busObj->rootHubInfo.ports =
             busObj->hcdInterface->rootHubInterface.rootHubPortNumbersGet(busObj->hcdHandle);

        busObj->rootHubInfo.power =
             busObj->hcdInterface->rootHubInterface.rootHubMaxCurrentGet(busObj->hcdHandle);

        busObj->rootHubInfo.rootHubPortInterface =
             busObj->hcdInterface->rootHubInterface.rootHubPortInterface;
```

The USB Host Layer must initialize and enable the operation of the root hub. While initializing the Root Hub Driver, the Host layer will assign a unique identifier to the root hub. The root hub will return this value as the parent identifier while calling the USB_HOST_DeviceEnumerate function. The USB Host Layer must then enable the operation of the root hub driver. This will cause the root hub driver to detect device attach and detach. The following code example shows how the USB Host Layer initializes and enables the root hub driver

```
/* The following code example show how the USB Host Layer initializes the root
 * hub and then enables the root hub operation. The
 * rootHubDevice->deviceIdentifier is a unique identifier that allows the USB
 * Host layer to identify this root hub. It is returned by the root hub driver
 * in the USB_HOST_DeviceEnumerate() function as the parent identifier when the
 * device is connected to the root hub. */

/* The hcdHandle is the driver handle. The hcdInterface pointer is of the type
 * DRV_USB_HOST_INTERFACE and points to the HCD interface. */

busObj->hcdInterface->rootHubInterface.rootHubInitialize( busObj->hcdHandle ,
                                                          rootHubDevice->deviceIdentifier );
busObj->hcdInterface->rootHubInterface.rootHubOperationEnable( busObj->hcdHandle , true );
```

When a device is attached, the Root Hub Driver will implement the required settling attach settling delay and will then call the USB Host Layer's USB_HOST_DeviceEnumerate function to enumerate the device. While calling this function, the root hub driver will provide the identifier that was provided to it in its initialize function. The USB_HOST_DeviceEnumerate function will return an identifier which uniquely identifies the attached device. The root hub driver uses this value to identify the device to the Host when the USB_HOST_DeviceDenumerate function is called on device detach. The following code example shows how the Root Hub driver calls the USB_HOST_DeviceEnumerate and the USB_HOST_DeviceDenumerate functions.

```
/* The following code shows how the root hub driver calls the
 * USB_HOST_DeviceEnumerate() function in the device attach interrupt. As seen
 * here, the root hub returns the identifier that the USB Host Layer assigned to
 * it the rootHubInitialize function call. The pUSBDrvObj->usbHostDeviceInfo
 * variable contains this identifier. */

if(PLIB_USB_InterruptFlagGet(usbID, USB_INT_ATTACH))
{
    /* We can treat this as a valid attach. We then clear the
     * detach flag and enable the detach interrupt. We enable
     * the Transaction interrupt */

    PLIB_USB_InterruptFlagClear(usbID, USB_INT_HOST_DETACH);
    PLIB_USB_InterruptEnable(usbID, USB_INT_HOST_DETACH);
    PLIB_USB_InterruptEnable(usbID, USB_INT_TOKEN_DONE);

    /* Ask the Host layer to enumerate this device. While calling
     * this function, the UHD of the parent device which is the
     * root hub in this case.
     * */
    pUSBDrvObj->attachedDeviceObjHandle = USB_HOST_DeviceEnumerate
                                          (pUSBDrvObj->usbHostDeviceInfo, 0);
}

/* The following code example shows how the root hub driver calls the
 * USB_HOST_DeviceDenumerate() function in the device detach interrupt. Note how
 * the attachedDeviceObjHandle that was assigned at the time of device
 * enumeration is returned to the Host Layer to let the Host know which device
 * is being detached. */

if((usbInterrupts & USB_INT_HOST_DETACH) && (enabledUSBInterrupts & USB_INT_HOST_DETACH))
{
    /* Perform other detach related handling */

    /* Ask the Host Layer to de-enumerate this device. */
    USB_HOST_DeviceDenumerate (pUSBDrvObj->attachedDeviceObjHandle);

    /* Disable the LS Direct Connect. It may have been enabled if the last
     attach was for a Low-Speed device. */
    PLIB_USB_EP0LSDirectConnectDisable(pUSBDrvObj->usbID);
    /* Continue to perform detach handling */
}
```

Root Hub Port Operation

The HCD Root Hub Driver exposes a set of port related functions that allow the USB Host Layer to control the port. The most commonly used functions are the function to reset the port and get the port speed. In this case, this is the speed of the attached device. The following code example shows how the USB Host Layer calls the hubPortReset, hubPortResetIsComplete and hubPortSpeedGet port functions.

```
/* The following code shows an example of how the Host Layer called the
 * hubPortReset function to reset the port to which the device is connected.
 * The code proceeds with the port reset if no device on the bus is in an
 * enumeration state. It will then call the hubPortReset function of the parent
 * hub of the device. The parent hub, hubInterface member of deviceObj points to
 * this driver, can be the root hub or an external hub */

if(!busObj->deviceIsEnumerating)
{
    /* Remember which device is enumerating */
    busObj->enumeratingDeviceIdentifier = deviceObj->deviceIdentifier;

    /* Grab the flag */
    busObj->deviceIsEnumerating = true;

    /* Reset the device */
    deviceObj->hubInterface->hubPortReset( deviceObj->hubHandle, deviceObj->devicePort );
}

/* The following code example shows how the Host checks if the port reset
 * operation has completed. If the reset operation has completed, the speed of
 * the attached device can be obtained. The reset settling delay can then be
 * started. */

 case USB_HOST_DEVICE_STATE_WAITING_FOR_RESET_COMPLETE:

    /* Check if the reset has completed */
    if(deviceObj->hubInterface->hubPortResetIsComplete
                 ( deviceObj->hubHandle ,deviceObj->devicePort ))
    {
        /* The reset has completed. We can also obtain the speed of the
         * device. We give a reset recovery delay to the device */

        deviceObj->speed = deviceObj->hubInterface->hubPortSpeedGet
                           (deviceObj->hubHandle, deviceObj->devicePort);

        deviceObj->deviceState = USB_HOST_DEVICE_STATE_START_RESET_SETTLING_DELAY;
    }
```

Opening and Closing a Pipe

The HCD client can open a pipe to the device after resetting the device. The USB Host Layer calls the hostPipeSetup function in the DRV_USB_HOST_INTERFACE structure to open a pipe. The USB Host Layer must open a pipe to communicate to a specific endpoint on a target device. While opening the pipe, the USB Host Layer must specify parameters which specify the address of the target device, the type of the transfer that the pipe must support and the speed of the pipe. If the device is connected to a hub, the address of the hub must be specified. The HCD Pipe Setup function *is not* interrupt-safe. It should not be called in any event handler that executes in an interrupt context.

The Pipe Setup function returns a valid pipe handle if the pipe was opened successfully. Pipe creation may fail if the target device was disconnected or if there are insufficient resources to open the pipe. The pipe handle is then used along with the hostIRPSubmit function to transfer data between the Host and the device. The following code shows example usage of a Pipe Open function.

```
/* The following code example shows how the Host Layer uses the hostPipeSetup
 * function to open a control pipe to the attached device. Most of the
 * parameters that are passed to this function become known when the device is
 * attached. The pipe handle is checked for validity after the hostPipeSetup
 * function call. */
```

```
if(busObj->timerExpired)
{
    busObj->busOperationsTimerHandle = SYS_TMR_HANDLE_INVALID;
    /* Settling delay has completed. Now we can open default address
     * pipe and and get the configuration descriptor */

    SYS_DEBUG_PRINT(SYS_ERROR_INFO,
                    "\r\nUSB Host Layer: Bus %d Device Reset Complete.", busIndex);

    deviceObj->controlPipeHandle =
               deviceObj->hcdInterface->hostPipeSetup( deviceObj->hcdHandle,
               USB_HOST_DEFAULT_ADDRESS , 0 /* Endpoint */,
               deviceObj->hubAddress /* Address of the hub */,
               deviceObj->devicePort /* Address of the port */,
               USB_TRANSFER_TYPE_CONTROL, /* Type of pipe to open */
               0 /* bInterval */, 8 /* Endpoint Size */, deviceObj->speed );

    if(DRV_USB_HOST_PIPE_HANDLE_INVALID == deviceObj->controlPipeHandle)
    {
        /* We need a pipe else we cannot proceed */
        SYS_DEBUG_PRINT(SYS_ERROR_DEBUG,
        "\r\nUSB Host Layer: Bus %d Could not open control pipe. Device not supported.",
busIndex);
    }
}
```

An open pipe consumes computational and memory resources and must therefore must be closed if it will not be used. This is especially true of pipes to a device that is detached. The Host Layer calls the hostPipeClose function in the DRV_USB_HOST_INTERFACE structure to close the pipe. The pipe to be closed is specified by the pipe handle. The Pipe Close function can be called from an event handler. It is interrupt safe. Closing a pipe will cancel all pending transfers on that pipe. The IRP callback for such canceled transfers will be called with the status USB_HOST_IRP_STATUS_ABORTED. The following code example shows an example of closing the pipe.

```
/* The following code example shows an example of how the Host Layer calls the
 * hostPipeClose function to close an open pipe. Pipe should be closed if it
 * will not used. An open pipe consumes memory resources. In this example, the
 * Host Layer closes the pipe if it was not able successfully submit an IRP to
 * this pipe. */

/* Submit the IRP */
if(USB_ERROR_NONE != deviceObj->hcdInterface->hostIRPSubmit
                  ( deviceObj->controlPipeHandle, & (deviceObj->controlTransferObj.controlIRP)))
{
    /* We need to be able to send the IRP. We move the device to
     * an error state. Close the pipe and send an event to the
     * application. The assigned address will be released when
     * the device in unplugged. */

    SYS_DEBUG_PRINT(SYS_ERROR_DEBUG,
        "\r\nUSB Host Layer: Bus %d Set Address IRP failed. Device not supported.", busIndex);

    /* Move the device to error state */
    deviceObj->deviceState = USB_HOST_DEVICE_STATE_ERROR;

    /* Close the pipe as we are about mark this device as unsupported. */
    deviceObj->hcdInterface->hostPipeClose(deviceObj->controlPipeHandle);
}
```

Transferring Data to an Attached Device

The USB Host Layer, the HCD client, needs to transfer data to the attached device to understand the device capabilities and to operate the device. The HCD uses a concept of Input Output Request Packet (IRP) to transfer data to and from the attached device. IRPs are transported over pipes which are setup by calling the USB Driver Pipe Setup function.

A Host IRP is a USB_HOST_IRP type data structure. The IRP is created by the Host layer and submitted to the HCD for processing through the hostIRPSubmit function. At the time of submitting the IRP, the pipe over which the IRP must be transported is specified. The data request in the IRP is transported using the attributes of pipe. When an IRP is submitted to the HCD, it is owned by the HCD and cannot be modified by the Host Layer until the HCD issues an IRP callback. The HCD will issue the IRP callback when it has completed or terminated processing of the IRP.

An IRP does not have its own transfer type. It inherits the properties of the pipe to which it is submitted. Hence an IRP becomes a control transfer IRP it was submitted to a control transfer pipe. A pipe allows multiple IRPs to be queued. This allows the Host Layer to submit IRPs to a pipe even while an IRP is being processed on the pipe. The HCD will process an IRP in the order that it was received. The following code example shows the USB_HOST_IRP data structure.

```
/* The following code example shows the USB_HOST_IRP structure. The Host Layer
 * uses this structure to place data transfer requests on a pipe. */

typedef struct _USB_HOST_IRP
{
    /* Points to the 8 byte setup command packet in case this is a IRP is
     * scheduled on a CONTROL pipe. Should be NULL otherwise */
    void * setup;

    /* Pointer to data buffer */
    void * data;

    /* Size of the data buffer */
    unsigned int size;

    /* Status of the IRP */
    USB_HOST_IRP_STATUS status;

    /* Request specific flags */
    USB_HOST_IRP_FLAG flags;

    /* User data */
    uintptr_t userData;

    /* Pointer to function to be called when IRP is terminated. Can be NULL, in
     * which case the function will not be called. */
    void (*callback)(struct _USB_HOST_IRP * irp);

    /****************************************
     * These members of the IRP should not be
     * modified by client
     ****************************************/
   uintptr_t privateData[7];

} USB_HOST_IRP;
```

The `setup` member of the USB_HOST_IRP structure must point to the 8 byte setup packet for control transfers. The driver will send this 8 byte data in the Setup phase of the control transfer. It can be NULL for non-control transfers. This member is only considered if the IRP is submitted to a control transfer pipe. It is ignored for non-control transfer pipes. The structure of the setup command should match that specified in the USB 2.0 specification.

The `data` member of the USB_HOST_IRP structure points to a data buffer. This data buffer will contain the data that needs to be sent to the device for data stage of a OUT transfer, or it will contain the data that was received from the device during an IN transfer. Any hardware specific cache coherency and address alignment requirements must be considered while allocating this data buffer. The Driver Client should not modify or examine the contents of the IRP after the IRP has been submitted and is being processed. It can be examined after the driver has released the IRP.

The `size` member of the USB_HOST_IRP structure contains the size of the transfer. for Bulk transfers, the size of the transfer can exceed the size of the transaction (which is equal to size of the endpoint reported by the device). The HCD in such a case will split up the transfer into transactions. This process does not require external intervention. For control transfers, the size of the transfer is specified in the setup packet (pointed to by the `setup` member of the USB_HOST_IRP structure). The driver will itself process the Setup, Data (if required) and Handshake stages of control transfer. This process again does not require external intervention. For interrupt and isochronous transfers, the size of transfer specified in the IRP cannot exceed the size of the transaction. If size is specified as 0, then the driver will send a zero length packet. The `size` parameter of the IRP is updated by the driver when IRP processing is completed. This will contain the size of the completed transfer.

The `status` member of the IRP provides the completion status of the IRP and should be checked only when the IRP processing has completed. This is indicated by the driver calling the IRP callback function. The IRP status is a USB_HOST_IRP_STATUS type. The following code example shows the different possible values of the `status` member and an example of submit a control transfer IRP.

```
/* The following code shows an example of how the Host Layer populates
 * the IRP object and then submits it. IRP_Callback function is called when an
```

```
  * IRP has completed processing. The status of the IRP at completion can be
  * checked in the status flag. The size field of the irp will contain the amount
  * of data transferred. */

 void IRP_Callback(USB_HOST_IRP * irp)
 {
     /* irp is pointing to the IRP for which the callback has occurred. In most
      * cases this function will execute in an interrupt context. The application
      * should not perform any hardware access or interrupt unsafe operations in
      * this function. */

     switch(irp->status)
     {
         case USB_HOST_IRP_STATUS_ERROR_UNKNOWN:
             /* IRP was terminated due to an unknown error */
             break;

         case USB_HOST_IRP_STATUS_ABORTED:
             /* IRP was terminated by the application */
             break;

         case USB_HOST_IRP_STATUS_ERROR_BUS:
             /* IRP was terminated due to a bus error */
             break;

         case USB_HOST_IRP_STATUS_ERROR_DATA:
             /* IRP was terminated due to data error */
             break;

         case USB_HOST_IRP_STATUS_ERROR_NAK_TIMEOUT:
             /* IRP was terminated because of a NAK timeout */
             break;

         case USB_HOST_IRP_STATUS_ERROR_STALL:
             /* IRP was terminated because of a device sent a STALL */
             break;

         case USB_HOST_IRP_STATUS_COMPLETED:
             /* IRP has been completed */
             break;

         case USB_HOST_IRP_STATUS_COMPLETED_SHORT:
             /* IRP has been completed but the amount of data processed was less
              * than requested. */
             break;

         default:
             break;
     }
 }

 /* In the following code example the a control transfer IRP is submitted to a
  * control pipe. The setup parameter of the IRP points to the Setup command of
  * the control transfer. The direction of the data stage is specified by the
  * Setup packet. */

 USB_HOST_IRP irp;
 USB_ERROR result;
 USB_HOST_PIPE_HANDLE controlPipe;
 USB_SETUP_PACKET setup;
 uint8_t controlTransferData[32];

 irp.setup = setup;
 irp.data = controlTransferData;
 irp.size = 32;
 irp.flags = USB_HOST_IRP_FLAG_NONE ;
 irp.userData = &someApplicationObject;
```

```
irp.callback = IRP_Callback;

result = DRV_USBFS_HOST_IRPSubmit(controlPipeHandle, &irp);

switch(result)
{
    case USB_ERROR_NONE:
        /* The IRP was submitted successfully */
        break;

    case USB_ERROR_HOST_PIPE_INVALID:
        /* The specified pipe handle is not valid */
        break;

    case USB_ERROR_OSAL_FUNCTION:
        /* An error occurred while trying to grab mutex */
        break;

    default:
        break;
}
```

The `flags` member of the USB_HOST_IRP structure specifies flags which affect the behavior of the IRP. The USB_HOST_IRP_FLAG enumeration specifies the available option. The USB_HOST_IRP_FLAG_SEND_ZLP causes the driver to add a Zero Length Packet (ZLP) to the data stage of the transfer when the transfer size is an exact multiple of the endpoint size. The USB_HOST_IRP_WAIT_FOR_ZLP flag will cause the driver to wait for a ZLP from the device in a case where the size of data received thus far in the transfer is an exact multiple of the endpoint size.

The `callback` member of the USB_HOST_IRP structure points to a function which the driver calls when the IRP processing is completed. The Driver Client must implement this function and assign the pointer to this function to the `callback` member of the IRP. Every IRP can have its own callback function or one common callback function could be used. The callback function will execute in an interrupt context. The Driver Client should not execute interrupt unsafe, blocking, or computationally intensive operations in the callback function. The client can call hostIRPSubmit function in the IRP callback function to submit another IRP or resubmit the same IRP. The client can check the status and size of the IRP in the callback function.

The `userData` member of the USB_HOST_IRP structure can be used by the client to associate a client specific context with the Host. This context can then be used by the client, in the IRP callback function to identify the context in which the IRP was submitted. This member is particularly useful if the client wants to implement one callback function for all IRPs.

The `privateData` member of the IRP is used by the driver and should not be accessed or manipulated by the Driver Client. The following code examples show usage of IRPs to transfer data between the Host and the attached device and along with the different flags.

```
/* The following code shows an example of submitting an IRP to send data
 * to a device. In this example we will request the driver to send a ZLP after
 * sending the last transaction. The driver will send the ZLP only if the size
 * of the transfer is a multiple of the endpoint size. This is not a control
 * transfer IRP. So the setup field of the IRP will be ignored.  */

USB_HOST_IRP irp;
USB_ERROR result;
USB_HOST_PIPE_HANDLE bulkOUTPipeHandle;
uint8_t data[128];

irp.data = data;
irp.size = 128;
irp.flags = USB_HOST_IRP_FLAG_SEND_ZLP ;
irp.userData = &someApplicationObject;
irp.callback = IRP_Callback;

result = DRV_USBFS_HOST_IPRSubmit( bulkOUTPipeHandle, &irp );


/* The following code shows an example of submitting an IRP to receive
 * data to a device. In this example we will request the driver to wait for a
 * ZLP after receiving the last transaction. The driver will wait for the ZLP
 * only if the size of the transfer is a multiple of the endpoint size. This is
 * not a control transfer IRP. So the setup field of the IRP will be ignored.
 * */
```

```
USB_HOST_IRP irp;
USB_ERROR result;
USB_HOST_PIPE_HANDLE bulkINPipeHandle;
uint8_t data[128];

irp.data = data;
irp.size = 128;
irp.flags = USB_HOST_IRP_FLAG_WAIT_FOR_ZLP ;
irp.userData = &someApplicationObject;
irp.callback = IRP_Callback;

result = DRV_USBFS_HOST_IPRSubmit( bulkINPipeHandle, &irp );
```

USB Driver Device Mode Operation

Provides information on Device mode operation.

Description

The USB Driver operates can operate in the Device mode when it is initialized for Device mode or Dual Role operation. When operating in Device mode, the USB Driver is also referred to as the USB Controller Driver (USBCD). In Dual-Role mode, the USB Driver will switch to USBCD mode when the USB Driver Device Attach function is called.

The USB Driver Client must perform these steps to operate the USB Driver in Device mode.

1. Open the USB Driver to obtain the driver handle.
2. Set the event handler.
3. Wait for the application to attach the device to the bus.
4. Enable Endpoint 0 and respond to USB Host Enumeration requests.
5. Allow the application and function drivers to enable other endpoints and communicate with the Host.

The following sections discuss these operations in more detail.

General Device Mode Operations

Provides information on general Device mode operations.

Description

This section describes the USBCD operations such as setting event handlers and attaching and detaching the device.

Handling Device Mode Driver Events

The Device Layer will call the USBCD eventHandlerSet function to register the Device mode event handling function. The USBCD generates various events that indicate different states of the USB. These events are defined by the DRV_USB_EVENT enumeration. The following code example shows how the Device Layer registers the driver event handling function.

```
/* This code example shows the implementation of the USB_DEVICE_Attach and the
 * USB_DEVICE_Detach function. These functions are actually macro that map
 * directly deviceAttach and the deviceDetach function of the driverInterface
 * member of the deviceClient Object (which is the macro parameter) */

#define USB_DEVICE_Attach( x )  ((USB_DEVICE_OBJ *)x)->driverInterface->deviceAttach
                                ( ((USB_DEVICE_OBJ *)(x))->usbCDHandle)
#define USB_DEVICE_Detach( x )  ((USB_DEVICE_OBJ *)x)->driverInterface->deviceDetach
                                ( ((USB_DEVICE_OBJ *)x)->usbCDHandle)
```

If the driver is operating in interrupt mode, the client event handling function will execute in an interrupt context. The client should not call interrupt unsafe, computationally intensive or blocking functions in the event handler. The following code shows a small example of the Device Layer USBCD Event Handler:

```
/* This code example shows a partial implementation of the USB Device Layer
 * event handler. Note how the code type casts the referenceHandle parameter to
 * a USB_DEVICE_OBJ type. This referenceHandle is the same value that the Device
 * Layer passed when the event handler was set. This now easily allows one
 * implementation of the event handling code to be used by multiple Device
 * Layer instances. */

    void _USB_DEVICE_EventHandler
(
    uintptr_t referenceHandle,
    DRV_USB_EVENT eventType,
    void * eventData
)
{
    USB_DEVICE_OBJ* usbDeviceThisInstance;
    USB_DEVICE_MASTER_DESCRIPTOR * ptrMasterDescTable;
    USB_DEVICE_EVENT_DATA_SOF SOFFrameNumber;

    usbDeviceThisInstance = (USB_DEVICE_OBJ *)referenceHandle;

    /* Handle events, only if this instance is in initialized state */
    if( usbDeviceThisInstance->usbDeviceInstanceState <= SYS_STATUS_UNINITIALIZED )
    {
        /* The device should anyway not be attached when the Device Layer is
         * not initialized. If we receive driver event when the Device Layer is
         * not initialized, there is nothing we can do but ignore them. */
        return;
    }

    switch(eventType)
    {
        case DRV_USB_EVENT_RESET_DETECT:

            /* Clear the suspended state */
            usbDeviceThisInstance->usbDeviceStatusStruct.isSuspended = false;

            /* Cancel any IRP already submitted in the RX direction. */
            DRV_USB_DEVICE_IRPCancelAll( usbDeviceThisInstance->usbCDHandle,
                    controlEndpointRx );

            /* Code not shown for the sake of brevity. */

    }
}
```

In the previous code example, the Device Layer (the Driver Client) sets the `hReferenceData` parameter, of the Event Handler Set function, to point to a local object. This pointer is returned to the Device Layer, in the event handler when an event occurs. For multiple instances of USB drivers in one application, this allows the Device Layer to easily associate a Device Layer specific context to the driver instance, thus simplifying implementation of the event handler.

Attaching and Detaching the Device

The USB Device Layer calls the USBCD deviceAttach and deviceDetach functions to attach and detach the device on the USB. The USB Device Layer should be ready to handle events which would occur when the device is attached on the bus. Hence the USB Device Layer should register the USBCD event handler before the attach function is called. The deviceAttach and deviceDetach functions can be called in an interrupt context. These functions are respectively called when the USB Device application detects a valid VBUS voltage and when the VBUS voltage is not valid.

Setting the Device Address

The USB Device Layer will call the USBCD deviceAddressSet function to set the USB address of the device. The Device Layer will do this when it receives the Set Address control request from the Host. The USBCD will reset the device address to '0' when it has received reset signaling from the root hub. The following code example shows how the USB Device Layer calls this function.

```
/* The following code example shows how the USB Device Layer calls the
 * DRV_USB_DEVICE_AddressSet function to set the address. The
 * DRV_USB_DEVICE_AddressSet function is actually a macro that calls the
 * deviceAddressSet function of the driverInterface of usbDeviceThisInstance
 * object. The usbDeviceThisInstance is Device Layer object.
 *
 * As seen in this code, the Device Layer calls the address set function when
 * the it a pending set address control request from the Host has completed. */
```

```
void _USB_DEVICE_Ep0TransmitCompleteCallback(USB_DEVICE_IRP * handle)
{
    USB_DEVICE_IRP * irpHandle = (USB_DEVICE_IRP *)handle;
    USB_DEVICE_OBJ * usbDeviceThisInstance;
    USB_DEVICE_CONTROL_TRANSFER_STRUCT * controlTransfer;

    usbDeviceThisInstance = (USB_DEVICE_OBJ *)irpHandle->userData;
    controlTransfer = &(usbDeviceThisInstance->controlTransfer);

    if(irpHandle->status == USB_DEVICE_IRP_STATUS_ABORTED)
    {
        return;
    }

    if(usbDeviceThisInstance->usbDeviceStatusStruct.setAddressPending)
    {
        DRV_USB_DEVICE_AddressSet(usbDeviceThisInstance->usbCDHandle,
                                  usbDeviceThisInstance->deviceAddress);
        usbDeviceThisInstance->usbDeviceStatusStruct.setAddressPending = false;
    }

    /* Code not shown for the sake of brevity */

}
```

Device Current Speed and SOF Number

The USB Device Layer will call the USBCD deviceCurrentSpeedGet function to know the speed at which the device is attached to the USB. This allows the Device Layer to select the correct endpoint settings at the time of processing the Set Configuration request issued by the Host. The USB Device Layer will call the deviceSOFNumberGet function to return the SOF number at the time of the SOF event.

Device Remote Wake-up

The USB Device Layer will call the USBCD deviceRemoteWakeupStop and deviceRemoteWakeupStart functions to stop and start remote signaling. The Device layer application will call the USB Device Layer Stop and Start Remote Wakeup Signaling functions to remotely let the root hub know that the device is ready to be woken up. The timing of the remote signaling is controlled by the Device Layer. The client should call the remote wakeup function only when the device is suspended by the Host.

Device Endpoint Operations

Provides information on Device Endpoint operations.

Description

The UBSCD Endpoint functions allow the Driver Client to enable, disable, stall and clear the stall condition on an endpoint. The client submits requests to transmit and receive data from the USB Host on an endpoint.

Endpoint Enable and Disable functions

The USBCD client must enable an endpoint it must use the endpoint for communicating with the USB Host. The client will call the USBCD deviceEndpointEnable function to enable the endpoint. While calling this function, the client must specify the endpoint address, the transfer type to be processed on this endpoint and the maximum size of a transaction on this endpoint. This function is thread-safe when called in an RTOS application. The USBCD allows an endpoint to be accessed by one thread only. The USB Device Layer and the device function drivers will enable the endpoint when the Host sets the device configuration. The USBCD deviceEndpointIsEnabled function is available to check if an endpoint is enabled. The following code example shows how the USB Device Layer enables the device endpoint.

```
/* The following code example shows the USB Device Layer enables Endpoint 0 to
 * prepare for the enumeration process after it has received reset signaling
 * from the Host. The Device Layer calls the deviceEndpointEnable function to
 * to enable the endpoint. The driverInterface member of the
 * usbDeviceThisInstance structure points to the USB Device Mode Driver Common
 * Interface. */
```

```
void _USB_DEVICE_EventHandler
(
    uintptr_t referenceHandle,
    DRV_USB_EVENT eventType,
    void * eventData
)
{
    /* Code not shown due to space constraints */

    switch(eventType)
    {
        case DRV_USB_EVENT_RESET_DETECT:

            /* Clear the suspended state */
            usbDeviceThisInstance->usbDeviceStatusStruct.isSuspended = false;

            /* Cancel any IRP already submitted in the RX direction. */
            DRV_USB_DEVICE_IRPCancelAll( usbDeviceThisInstance->usbCDHandle,
                    controlEndpointRx );

            /* Cancel any IRP already submitted in the TX direction. */
            DRV_USB_DEVICE_IRPCancelAll( usbDeviceThisInstance->usbCDHandle,
                    controlEndpointTx );

            /* Deinitialize all function drivers.*/
            _USB_DEVICE_DeInitializeAllFunctionDrivers ( usbDeviceThisInstance );

            /* Disable all endpoints except for EP0.*/
            DRV_USB_DEVICE_EndpointDisableAll(usbDeviceThisInstance->usbCDHandle);

            /* Enable EP0 endpoint in RX direction */
            (void)usbDeviceThisInstance->driverInterface->deviceEndpointEnable
                  (usbDeviceThisInstance->usbCDHandle,
                    controlEndpointTx, USB_TRANSFER_TYPE_CONTROL, USB_DEVICE_EP0_BUFFER_SIZE);

            /* Code not shown due to space constraints */

            break;
    }
}
```

The USB Device Layer and the Function drivers will disable an endpoint when the Host sets a zero-device configuration or when the Host resets the device. The USBCD deviceEndpointDisable function disables an endpoint. When an endpoint is disabled, it does not accept requests for Host communication. Disabling an endpoint does not cancel any communication requests that that have been submitted on the endpoint. These requests must be canceled explicitly.

Device Endpoint Stall and Stall Clear

The USBCD client can call the deviceEndpointStall and deviceEndpointStallClear functions to stall and cleat the stall on an endpoint respectively. The USB Device Layer and function driver may stall endpoint to indicate error or to indicate a protocol state. The endpoint stall condition may be cleared in response to a USB Host Clear Feature request. Stalling or clearing the stall on an endpoint will cause all communication requests on the endpoint to be canceled. The function calls are thread safe and interrupt safe. The deviceEndpointIsStalled function is also available to check if an endpoint is in a stalled state. The following code example shows how the USB Device Layer calls these functions to stall and clear the stall on an endpoint.

```
/* The following code example shows how the USB Device Layer calls the driver
 * endpoint stall function (deviceEndpointStall) to stall an endpoint when the a
 * Host send a Set Feature request with feature selector set to endpoint halt.
 * The endpoint to be halted is identified in the setup packet and is identified
 * in this code example as usbEndpoint. Also shown is how the stall clear
 * (deviceEndpointStallClear) and stall status check (deviceEndpointIsStalled)
 * functions are called. */

/* The driverInterface member of the usbDeviceThisInstance structure is a
 * pointer to the USB Driver Common Interface. */

void _USB_DEVICE_ProcessStandardEndpointRequest
(
    USB_DEVICE_OBJ * usbDeviceThisInstance,
    uint8_t interfaceNumber,
    USB_SETUP_PACKET * setupPkt
)
{
    USB_ENDPOINT usbEndpoint;
    usbEndpoint = setupPkt->bEPID;

    if( setupPkt->bRequest == USB_REQUEST_GET_STATUS )
    {
        usbDeviceThisInstance->getStatusResponse.status = 0x00;
        usbDeviceThisInstance->getStatusResponse.endPointHalt
            =  usbDeviceThisInstance->driverInterface->deviceEndpointIsStalled
               (usbDeviceThisInstance->usbCDHandle, usbEndpoint );

        USB_DEVICE_ControlSend( (USB_DEVICE_HANDLE)usbDeviceThisInstance,
                (uint8_t *)&usbDeviceThisInstance->getStatusResponse, 2 );
    }
    else if( setupPkt->bRequest == USB_REQUEST_CLEAR_FEATURE )
    {
        if( setupPkt->wValue == USB_FEATURE_SELECTOR_ENDPOINT_HALT )
        {
            usbDeviceThisInstance->driverInterface->deviceEndpointStallClear
            (usbDeviceThisInstance->usbCDHandle, usbEndpoint );
            USB_DEVICE_ControlStatus((USB_DEVICE_HANDLE)usbDeviceThisInstance,
                                      USB_DEVICE_CONTROL_STATUS_OK );
        }
    }
    else if (setupPkt->bRequest == USB_REQUEST_SET_FEATURE )
    {
        if( setupPkt->wValue == USB_FEATURE_SELECTOR_ENDPOINT_HALT )
        {
            usbEndpoint = setupPkt->bEPID;
            usbDeviceThisInstance->driverInterface->deviceEndpointStall
                                  (usbDeviceThisInstance->usbCDHandle, usbEndpoint );
            USB_DEVICE_ControlStatus((USB_DEVICE_HANDLE)usbDeviceThisInstance,
                                      USB_DEVICE_CONTROL_STATUS_OK );
        }
    }

    /* Additional code is not shown due to space constraints */
}
```

Transferring Data to the Host

Provides information on transferring data to the Host.

Description

The USB Device Layer, the USBCD client, needs to transfer data to the Host in response to enumeration requests for general operation on the device. The USB uses a concept of Input Output Request Packet (IRP) to transfer data to and from the Host. IRPs are transported over endpoints which are enabled by calling the USBCD Endpoint Enable function.

A Device IRP is a USB_DEVICE_IRP type data structure. The IRP is created by the Device Layer and submitted to the USBCD for processing through the deviceIRPSubmit function. At the time of submitting the IRP, the endpoint over which the IRP must be transported is specified. The data request in the IRP is transported using the attributes of the endpoint. When an IRP is submitted to the USBCD, it is owned by the USBCD and cannot be modified by the Device Layer until the USBCD issues an IRP callback. The USBCD will issue the IRP callback when it has completed or terminated processing of the IRP.

An IRP does not have its own transfer type. It inherits the properties of the endpoint to which it is submitted. Hence an IRP becomes a control transfer IRP it was submitted to a control endpoint. An endpoint allows multiple IRPs to be queued. This allows the Device Layer to submit IRPs to an endpoint even while an IRP is being processed on the endpoint. The USBCD will process an IRP in the order that it was received. The following code example shows the USB_DEVICE_IRP data structure:

```
/* This code example shows the USB_DEVICE_IPR structure. The Device Layer
 * uses such a structure to transfer data through the driver. A structure of
 * this type is allocated by the Device Layer and the other function drivers and
 * passed to the deviceIRPSubmit function. */

typedef struct _USB_DEVICE_IRP
{
    /* Pointer to the data buffer */
    void * data;

    /* Size of the data buffer */
    unsigned int size;

    /* Status of the IRP */
    USB_DEVICE_IRP_STATUS status;

    /* IRP Callback. If this is NULL, then there is no callback generated */
    void (*callback)(struct _USB_DEVICE_IRP * irp);

    /* Request specific flags */
    USB_DEVICE_IRP_FLAG flags;

    /* User data */
    uintptr_t userData;

    /***********************************
     * The following members should not
     * be modified by the client
     ***********************************/
    uint32_t privateData[3];

} USB_DEVICE_IRP;
```

The `data` member of the USB_DEVICE_IRP structure points to a data buffer. This data buffer will contain the data that needs to be sent to the Host for the data stage of an IN transfer. For an OUT transfer, it will contain the data that was received from the Host. Any hardware specific cache coherency and address alignment requirements must be considered while allocating this data buffer. The Driver Client should not modify or examine the contents of the IRP after the IRP has been submitted and is being processed. It can be examined after the driver has released the IRP.

The `size` member of the USB_DEVICE_IRP structure specifies the size of the data buffer. The transfer will end when the device has sent or received size number of bytes. While sending data to the Host, the IRP size can exceed the size of the transaction (which is equal to the size of the endpoint). The USBCD in such a case will split up the transfer into transactions. This process does not require external intervention. The driver uses receive and transmit IRPs to process control transfers. When the driver receives a Setup packet, the IRP completion status would be USB_DEVICE_IRP_STATUS. The Driver Client should then use additional receive and transmit IRPs to complete the control transfer.

For interrupt and isochronous transfers, the size of transfer specified in the IRP cannot exceed the size of the transaction. If size is specified as 0, then the driver will send or expect a zero length packet. The `size` parameter of the IRP is updated by the driver when IRP processing is completed. This will contain the size of the completed transfer.

The `status` member of the IRP provides the completion status of the IRP and should be checked only when the IRP processing has completed. This is indicated by the driver calling the IRP callback function. The IRP status is a USB_DEVICE_IRP_STATUS type. The following code example shows the different possible values of the `status` member and example usage of IRPs to transfer data between the device and the Host.

```
/* The followoing code shows example usage of the device IRP. The submit status
 * of the IRP is available when IRP submit function returns. The completion
 * status of the IRP is available when the IRP has terminated and the IRP
 * callback function is invoked. The IRP callback
 * function shown in this example shows the possible complete status of the IRP.
 * The end application may or may not handle all the cases. Multiple IRPs can be
 * queued on an endpoint. */

void IRP_Callback(USB_DEVICE_IRP * irp)
{
    /* irp is pointing to the IRP for which the callback has occurred. In most
     * cases this function will execute in an interrupt context. The application
     * should not perform any hardware access or interrupt unsafe operations in
     * this function. */

    switch(irp->status)
    {
        case USB_DEVICE_IRP_STATUS_TERMINATED_BY_HOST:
            /* The IRP was aborted because the Host cleared the stall on the
             * endpoint */
            break;

        case USB_DEVICE_IRP_STATUS_ABORTED_ENDPOINT_HALT:
            /* IRP was aborted because the endpoint halted */
            break;


        case USB_DEVICE_IRP_STATUS_ABORTED:
            /* USB Device IRP was aborted by the function driver */
            break;

        case USB_DEVICE_IRP_STATUS_ERROR:
            /* An error occurred on the bus when the IRP was being processed */
            break;

        case USB_DEVICE_IRP_STATUS_COMPLETED:
            /* The IRP was completed */
            break;

        case USB_DEVICE_IRP_STATUS_COMPLETED_SHORT:
            /* The IRP was completed but the amount of data received was less
             * than the requested size */
            break;

        default:
            break;

    }
}

/* In the following example, the IRP is submitted to Endpoint 0x84. This is
 * interpreted as an IN direction endpoint (MSB of 0x84 is 1) and Endpoint 4.
 * The data contained in source will be sent to the USB Host. Assuming
 * the endpoint size is 64, the 130 bytes of data in this case will be sent to
 * the Host in three transaction of 64, 64 and 2 bytes. A transaction completes
 * when the Host polls (sends an IN token) the device.  The callback function
 * will then called indicating the completion status of the IRP. The application
 * should not modify the privateData field of the IRP. If the IRP was submitted
 * successfully, the buffer will be owned by the driver until the IRP callback
 * function has been called. Because the size of the transfer is not a multiple
 * of the endpoint size, the IRP flag must be set
 * USB_DEVICE_IRP_FLAG_DATA_COMPLETE. This directs the driver to not perform any
 * explicit signaling to the Host to indicate end of transfer. The last packet
 * in this case is a short packet and this signals the end of the transfer. */

USB_DEVICE_IRP irp;
USB_ERROR result;
uint8_t source[130];

irp.data = source;
irp.size = 130;
irp.called = IRP_Callback;
flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
userData = &someApplicationObject;

result = DRV_USBFS_DEVICE_IRPSubmit(driverHandle, 0x84, &irp);

switch(result)
{
    case USB_ERROR_PARAMETER_INVALID:
        /* This can happen if the driverHandle is invalid */
        break;

    case USB_ERROR_DEVICE_IRP_IN_USE:
        /* This can happen if the IRP is being resubmitted while it is still in
         * process (it was submitted before but processing has not completed */
        break;

    case USB_ERROR_DEVICE_ENDPOINT_INVALID;
        /* The endpoint to which this IRP is being submitted is not provisioned
         * in the system. This is controller by DRV_USBFS_ENDPOINTS_NUMBER
         * configuration parameter. */
        break;

    case USB_ERROR_ENDPOINT_NOT_CONFIGURED:
        /* The endpoint to which this IRP is being submitted is not enabled. It
         * must be enabled by calling the DRV_USBFS_DEVICE_EndpointEnable()
         * function. */
        break;

    case USB_ERROR_PARAMETER_INVALID:
        /* The USB_DEVICE_IRP_FLAG_DATA_PENDING flag was specified but the
         * transfer size is not a multiple of the endpoint size. If the IRP was
         * submitted to a receive endpoint, this error can occur if the size is
         * not a multiple of the endpoint size. */
        break;

    case USB_ERROR_OSAL_FUNCTION:
        /* An error occurred while trying to grab a mutex. This is applicable
         * when the driver is running with a RTOS. */
        break;

    case USB_ERROR_NONE:
        /* The IRP was submitted successfully. */
        break;

    default:
        break;
}

/* The following code example shows how an IRP is submitted to an OUT endpoint.
 * In this case data will be pointing to a buffer where the received data will
 * be stored. Note that the size of the IRP should be a multiple of the endpoint
 * size. The flags parameter is ignored in the data receive case. The IRP
 * terminates when the specified size of bytes has been received (the Host sends
 * OUT packets) or when a short packet has been received. */

USB_DEVICE_IRP irp;
USB_ERROR result;
uint8_t destination[128];

irp.data = destination;
irp.size = 128;
irp.called = IRP_Callback;
userData = &someApplicationObject;

result = DRV_USBFS_DEVICE_IRPSubmit(driverHandle, 0x04, &irp);
```

For IRPs submitted to an Interrupt or Isochronous endpoints, the driver will always send either the less than or equal to the maximum endpoint packet size worth of bytes in a transaction. The application could either submit an IRP per Interrupt/Isochronous polling interval or it could submit one IRP for multiple polling intervals.

The `flags` member of the USB_DEVICE_IRP structure specifies flags which affect the behavior of the IRP. The USB_DEVICE_IRP_FLAG enumeration specifies the available option. The USB_DEVICE_IRP_FLAG_DATA_COMPLETE causes the driver to add a Zero Length Packet (ZLP) to the data stage of the IN transfer when the transfer size is an exact multiple of the endpoint size. If the transfer size is not a multiple of the endpoint size, no ZLP will be sent. The USB_DEVICE_IRP_FLAG_PENDING flag will cause the driver to not send a ZLP in a case where the size of the IN transfer is an exact multiple of the endpoint size. The following code example demonstrates this.

```
/* In the following code example, the IRP is submitted to an IN endpoint whose size
 * is 64. The transfer size is 128, which is an exact multiple of the endpoint
 * size. The flag is set to USB_DEVICE_IRP_FLAG_DATA_COMPLETE. The driver
 * will send two transactions of 64 bytes each and will then automatically send a
 * Zero Length Packet (ZLP), thus completing the transfer. The IRP callback will
 * be invoked when the ZLP transaction has completed. */

USB_DEVICE_IRP irp;
USB_ERROR result;
uint8_t source[128];

irp.data = source;
irp.size = 128;
irp.called = IRP_Callback;
flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
userData = &someApplicationObject;

result = DRV_USBFS_DEVICE_IRPSubmit(driverHandle, 0x84, &irp);

/* In the following code example, the IRP is submitted to an IN endpoint whose size
 * is 64. The transfer size is 128, which is an exact multiple of the endpoint
 * size. The flag is set to to USB_DEVICE_IRP_FLAG_DATA_PENDING. The driver will
 * send two transactions of 64 bytes each but will not send a ZLP. The USB Host
 * can then consider that there is more data pending in the transfer. The IRP
 * callback will be invoked when the two transactions have completed. */

USB_DEVICE_IRP irp;
USB_ERROR result;
uint8_t source[128];

irp.data = source;
irp.size = 128;
irp.called = IRP_Callback;
flags = USB_DEVICE_IRP_FLAG_DATA_COMPLETE;
userData = &someApplicationObject;

result = DRV_USBFS_DEVICE_IRPSubmit(driverHandle, 0x84, &irp);
```

The `callback` member of the USB_DEVICE_IRP structure points to a function which the driver calls when the IRP processing is completed. The Driver Client must implement this function and assign the pointer to this function to the `callback` member of the IRP. Every IRP can have its own callback function or one common callback function could be used. The callback function will execute in an interrupt context. The Driver Client should not execute interrupt unsafe, blocking or computationally intensive operations in the callback function. The client can call deviceIRPSubmit function in the IRP callback function to submit another IRP or resubmit the same IRP. The client can check the status and size of the IRP in the callback function.

The `userData` member of the USB_DEVICE_IRP structure can be used by the client to associate a client specific context with the Host. This context can then be used by the client, in the IRP callback function to identify the context in which the IRP was submitted. This member is particularly useful if the client wants to implement one callback function for all IRPs.

The `privateData` member of the IRP is used by the driver and should not be accessed or manipulated by the Driver Client.

Driver Signing and Windows 8 (for v1.08)

This section provides information on driver signing with Window 8.

Description

This section provides information related to USB driver signatures, the types of signatures needed for the different versions of Windows operating system, and how to get a signed driver package.

What are "Signed" Drivers?

Provides a description of signed drivers.

Description

Most USB drivers operate in what is known as "Kernel mode" on Windows-based personal computers (PC). Kernel mode drivers have low-level access to the PC and its resources. This low-level access to the PC is normally necessary to implement the kind of functionality that the driver is intended to provide to top-level applications.

Malicious software developers would want their software to operate, since it provides the greatest control and access to the PC. Therefore, in the interest of protecting Windows security, Windows operating systems place restrictions on what code is allowed to be operated in Kernel mode.

Windows "trusts" drivers and executable programs that have been signed, more so than software that is unsigned. Signing a driver package is analogous to placing an embossed wax seal on an envelope. The signature/wax seal does not effect or alter the contents of the package, but it provides proof that the contents have not been modified or tampered with, since the time that the signature/wax seal was first applied.

There are three types of USB driver signatures:

- Embedded digital signatures
- Full driver package (digital signature - Microsoft Authenticode)
- Full driver package (digital signature - WHQL

Embedded Digital Signatures

This type of signature resides inside of driver `.sys` files (Kernel mode driver binary files). No additional/external files are associated with this type of signature. These types of signatures only protect against tampering with the `.sys` file itself, and do not include other files that may be a part of the driver package (i.e., `.inf` and `.dll` files). All driver `.sys` files provided by Microsoft operating systems, as well as most third-party Kernel mode drivers will contain at least this level of signature.

"Full Driver Package" Digital Signature – Microsoft Authenticode

This type of signature can be thought of as a "wrapper" over the entire driver package content files. A driver package can be as simple as a single `.inf` file (a plain text installation instruction file that Windows uses when installing new drivers), or may encompass additional files (such as `.dll` and/or `.sys` files). The full driver package signature comes in the form of a properly created security catalog file (`.cat`), which will be part of the driver package distribution. A driver package signed with an Authenticode signature is relatively easy to create; however, it less trustworthy to that of a WHQL digital signature.

“Full Driver Package” Digital Signature – WHQL

This type of signature is the most trusted by Windows, and is very similar to the full driver package Microsoft Authenticode signature, but is more expensive and harder to obtain. To obtain a Windows Hardware Quality Labs (WHQL) signature, a driver package must undergo extensive testing, and passing log files and submission fees must be supplied to Microsoft. If a driver package has already previously been tested and WHQL-certified, but has since been modified, in some cases it is possible to get the driver recertified through a simpler and less expensive "Driver Update Acceptable" process with Microsoft.

Any modifications to a driver package once the signature has been applied, including adding or deleting a single character of whitespace in the driver `.inf` file, will invalidate a full driver package signature. However, a driver package can have two simultaneous signatures, one covering the full driver package, and one embedded inside the driver binary file(s). Modifications to a `.inf` file do not invalidate an embedded digital signature inside of a driver binary file.

Once a signature has been invalidated, Windows will no longer trust the driver package as much, and will place restrictions on its installation (or outright prevent its installation on some operating systems). The driver package can however be resigned, to restore the trustworthiness of the driver to Windows.

Minimum Driver Signature Requirements

Provides information on minimum driver signature requirements.

Description

Full driver package WHQL signatures are the best and most trusted by all versions of Windows. Windows allows the installation of properly WHQL signed drivers, without producing a prompt warning the user about the driver’s trustworthiness.

However, current Windows versions do not require WHQL signatures to allow installation. Lesser signatures (or no signatures in some cases) are allowed, but will generate user dialogs/warnings during the installation process.

Operating System	Minimum Signature to Allow Installation
Windows 2000	None.
Windows XP 32-bit	None.
Windows XP 64-bit	None.
Windows Vista 32-bit	None.

Windows Vista 64-bit	Embedded.
Windows 7 32-bit	Embedded.
Windows 7 64-bit	Embedded.
Windows 8 32-bit	Embedded.
Windows 8 64-bit	Embedded plus full package Authenticode.
Windows RT (ARM)	Third -party drivers and driver packages are not currently allowed. All USB devices for this operating system must use Microsoft supplied drivers.

Using Older Drivers With Windows 8

Provides information on using older drivers with Windows 8.

Description

In general, USB driver packages that are designed for Windows 7 and prior operating system versions will also work in Windows 8, but there is one important exception to this. Starting with Windows 8 64-bit, all drivers must contain a proper "full driver package" digital signature (prior operating systems only required an embedded signature in the `.sys` file, rather than the entire driver package including the `.inf` file). The driver package signature exists as a `.cat` file that comes with the driver package, and needs to be correctly referenced from within the `.inf` file. If either the `.cat` file is entirely missing, or it is not being correctly referenced from the `.inf` file, Windows 8 will generate the following error message when the user attempts to install the driver: *The third party INF does not contain digital signature information*.

If the `.cat` file is present and is correctly referenced, but something in the driver package was modified since the signature was applied, a slightly different error message will occur: *The hash for the file is not present in the specified catalog file. The file is likely corrupt or the victim of tampering.*

In both cases, Windows 8 64-bit will not allow the driver package to be installed, even though it may technically be capable of functioning correctly. To correct this, the driver package must be properly signed with a full package signature. This signature may be either a WHQL signature (which is the best kind of signature), or a Microsoft Authenticode signature.

The MPLAB Harmony USB CDC and WinUSB Application driver packages include a WHQL signature and can be installed successfully on Windows 8 32-bit and 64-bit (as well as prior operating systems). When the firmware is using the same VID/PID as the default value from the demonstration, the latest driver package from MPLAB Harmony should install directly.

When the application uses a customized `.inf` file (e.g., VID/PID and/or strings are different), it will not be possible to directly use the driver package from MPLAB Harmony. The reason for this is that any time anyone makes any changes to the driver package (including adding or deleting one character of whitespace in the `.inf` file), this action will break and invalidate the driver package signature. Therefore, even if the `.cat` file is present, the signature will be invalid (and will not install correctly).

Therefore, if an application needs to use a custom-modified driver package, the only practical solution is to make the modifications, and then resign the driver package. A driver package can be signed with an Authenticode signature using the procedure described in Using a Code Signing Certificate to Sign Driver Packages. A package signed with the Microsoft Authenticode signature will install successfully on Windows 8, but will still produce a user prompt asking if they would like to trust the company that signed the driver package. This user dialog can be suppressed if the driver package instead contains a WHQL signature.

Although not very suitable for end-consumers, Windows 8 does have a feature that allows driver package signing enforcement to be temporarily disabled. This is particularly useful for development and testing purposes. The feature is hidden under several layers of menus and requires the following steps to enable:

1. From the desktop, move the mouse to the lower right hand corner of the screen, to launch the charm bar.
2. Click the Settings “gear” icon.
3. Click the “Change PC Settings” option.
4. In the PC Settings menu on the left, select the “General” option.
5. In the right hand pane, scroll down to the bottom of the options list. Under the “Advanced startup” section, click the “Restart now” button. This does not directly reboot the computer, but launches a page that provides additional restart options.
6. In the “Choose an option” page, select the “Troubleshoot” option.
7. From the Troubleshoot menu, click “Advanced options”.
8. In the “Advanced options” dialog, click the “Startup Settings” option.
9. From the “Startup Settings” dialog, click the “Restart” button.

10. The computer should now begin a reboot cycle. During the boot up sequence, a special “Startup Settings” dialog screen should appear.
11. On the “Startup Settings” dialog, press the “F7” key, to select the “Disable driver signature enforcement” option.
12. Allow Windows 8 to finish booting.

Once driver signing enforcement is disabled, unsigned driver packages can then be installed. After rebooting the PC, driver signing enforcement will be reenabled, but Windows 8 will continue to allow the installed unsigned driver(s) to be loaded for the hardware, without requiring the system to be repeatedly rebooted into the driver signing enforcement disabled mode.

USB Application Project Driver Signatures

Provides information on driver signatures for MPLAB Harmony USB application projects.

Description

Projects Based on WinUSB

WinUSB is a Microsoft created/supplied driver. All Microsoft-supplied drivers contain an embedded signature from Microsoft.

In operating systems prior to Windows 8, WinUSB-based devices require the user to install a driver package for the hardware. However, starting with Windows 8, it is possible to make WinUSB-based devices that are fully "plug and play", and do not require any user-supplied driver package. Windows 8 allows for automatic installation of the WinUSB driver, when the device firmware implements the correct Microsoft specific "OS" and related USB descriptors. These special descriptors are optional, but when implemented, allow for automatic driver installation using the WinUSB driver that is distributed with the operating system installation.

Projects Based on CDC

When used with Windows, the CDC projects in MPLAB Harmony use the Microsoft created/supplied `usbser.sys` driver. This driver contains an embedded signature from Microsoft.

Projects Based on HID, MSD, Audio Class

These USB device classes/projects rely on Microsoft-supplied drivers that are distributed with the operating system, and do not require any user-supplied driver packages or `.inf` files. Therefore, driver package signing is usually not relevant for these types of applications, as the drivers are normally installed automatically when the hardware is attached to the PC.

Obtaining a Microsoft Authenticode Code Signing Certificate

Provides information on obtaining a Microsoft Authenticode code signing certificate.

Description

There are several Certificate Authority (CA) companies that can sell your organization a signing certificate, which will allow you to sign your own driver packages. However, when submitting a driver package to Microsoft for WHQL certification, either as a new device/driver, or by reusing a previous submission through the Driver Update Acceptable (DUA) process, Microsoft currently requires that the submitted files be signed with an Authenticode signing certificate issued by VeriSign.

Therefore, it is generally preferred to obtain the Microsoft Authenticode code signing certificate from VeriSign (now a part of Symantec Corporation). Before purchasing the certificate, it is recommended to search for possible promotional/discounted rates. Historically, Microsoft has run a program providing for discounted prices for first-time purchasers of VeriSign certificates.

Authenticode code signing certificates are usually sold on an annual or multi-year basis. Once purchased, the signing certificate can normally be used to sign an unlimited number of driver package security catalog files (e.g., `.cat` files), along with other types of files (e.g., `.exe` executable programs). The certificate itself (i.e., typically a `.pvk` file, though other extensions are possible) needs to be kept physically secure, and should never be distributed publicly.

Code Signing Certificates (Other Uses)

Provides information on other uses of code signing certificates.

Description

In addition to signing driver packages, a Microsoft Authenticode signing certificate can be used to sign certain other types of files, such as executable (`.exe`) programs. Windows, especially Windows 8, does not trust unsigned executables as much as signed executables. In Windows 8, an unsigned executable that has "no history" and has no reputation established with Microsoft will be treated as relatively untrustworthy, and is blocked from execution, unless the user manually overrides the operating system behavior, through an advanced options dialogue that is typically hard for new users to find.

Additionally, some virus scanning applications also rely on executable signatures, to help establish relative trustworthiness. In some cases, unsigned executables, free of malware/viruses, can still be blocked from execution by the virus scanning software, until a history/reputation is built up establishing the executable as trustworthy. Signing the executable with a Microsoft Authenticode signing certificate will generally make the executable more trustworthy and less likely to be (incorrectly) flagged as malware.

Using a Code Signing Certificate to Sign Driver Packages

Provides information on using a code signing certificate to sign driver packages.

Description

If you make modifications to a driver package and need to resign the package, the easiest method is to sign it with a Microsoft Authenticode code singing certificate. This can be done using the following procedure:

1. Start from a known working driver package `.inf` file from the latest MPLAB Harmony release.
2. Modify the `.inf` file as desired. The `.inf` file is a plain text file (i.e., editable with a text editor, such as Notepad) that contains installation instruction/information that tells the operating system what driver needs to be used for the hardware, and anything else that may need to occur during the driver installation process. When changing the `.inf` file device list sections, please remove all existing Microchip VID/PIDs, before replacing them with your own. The manufacturer and product strings should also be updated as applicable for your device.
3. Delete the security catalog (`.cat`) file that is already supplied with the package. After modifying the `.inf` file, the security catalog file will no longer be valid and you will need to create a new one.
4. Download the latest version of the Windows Driver Kit (WDK) from Microsoft by visiting: http://msdn.microsoft.com/en-us/library/windows/hardware/gg487428.aspx Version 8.0 or later is needed (prior versions don’t have awareness of Windows 8 specifics).
5. Use the Inf2Cat utility in the WDK to regenerate a new `.cat` file from the modified `.inf` file.
   - Inf2Cat is a command line utility. Open a command prompt, navigate to the directory of the inf2cat tool, and then run it at the command line to get a small help/explanation of usage syntax. The program is typically located in the following location: `C:\Program Files\Windows Kits\8.0\bin\x64` (or `\x86` folder for 32-bit)
   - Typical usage syntax would be similar to the following (all on one line): `inf2cat /driver:C:\[path to dir with .inf file] /os:XP_X86,XP_X64,Vista_X86,Vista_X64,7_X86,7_X64,8_X86,8_X64,Server2003_X86,Server2003_X64,Server2008_X86,Server2008_X64,Server2008R2_X64,Server8_X64`.
     Assuming the Inf2Cat utility runs successfully, it will generate a raw `.cat` file. The `.cat` file will still need to be signed to be useful.
6. If your organization does not already have one, purchase a code signing certificate from a Certificate Authority (CA) such as VeriSign (now Symantec Corporation). See Obtaining a Microsoft Authenticode Code Signing Certificate for more details.
7. Use the `signtool.exe` utility, along with the signing certificate purchased from the CA, to sign the `.cat` file. The signtool utility is small Microsoft program that is distributed in the Windows SDK (and/or in older versions of the WDK, prior to v8.0). The Windows SDK can currently be obtained by visiting: http://msdn.microsoft.com/en-us/windows/desktop/hh852363.aspx
8. Typical syntax when using the signtool would be as follows, wwhen executed in the directory of the .cat file, assuming the directory to the signtool is in the path, and the certificate has a .pfx extension without a password, and that the certificate resides on "E:", like a typical USB flash drive: `signtool sign /v /f "E:\[path to certificate]\[certificate file name].pfx" /t http://timestamp.verisign.com/scripts/timestamp.dll [FileNameToSign.cat]`
9. Verify that the signature has been properly applied using the verify command line option: `signtool verify /a /pa [FileNameToSign.cat]`. The verify step should report success.

The driver package should now be correctly signed with a Microsoft Authenticode signature. Test it on all target operating systems. Distribute both the `.inf` file and `.cat` file together to the end-consumer (along with any other driver package files that may be necessary, which may include `.dll` files, particularly in the case of the WinUSB driver package). Never distribute the signing certificate that you purchased from the CA, this should be kept in a safe place, out of the hands of the public (the certificate can be reused to sign any number of driver packages, as well as `.exe` files, which will have some benefits).

Support

This section provides support information for MPLAB Harmony.

Using the Help

This topic contains general information that is useful to know to maximize using the MPLAB Harmony help.

Description

Help Formats

MPLAB Harmony Help is provided in three formats:

- Stand-alone HyperText Markup Language (HTML)
- Microsoft Compiled HTML Help (CHM)
- Adobe® Portable Document Format (PDF)

TIP! When using the MPLAB Harmony Help PDF, be sure to open the "bookmarks" if they are not already visible to assist in document navigation. See Using the Help for additional information.

Help File Locations

Each of these help files are included in the installation of MPLAB Harmony in the following locations:

- HTML - `<install-dir>/doc/html/index.html`
- CHM - `<install-dir>/doc/help_harmony.chm`
- PDF - `<install-dir>/doc/help_harmony.pdf`

Refer to Help Features for more information on using each output format.

Where to Begin With the Help

The help documentation provides a comprehensive source of information on how to use and understand MPLAB Harmony. However, it is not required to read the entire document before starting to work with MPLAB Harmony.

Prior to using MPLAB Harmony, it is recommended to review the Release Notes for any known issues. A PDF copy of the release notes is provided in the `<install-dir>/doc` folder of your installation.

New Users

For new users to MPLAB Harmony, it is best to follow the Guided Tour provided in *Volume I: Getting Started With MPLAB Harmony*.

Experienced Users

For experienced users already somewhat familiar with the MPLAB Harmony installation and online resources and, looking to jump right into a specific topic, follow the links provided in the table in *Volume 1: Getting Started With MPLAB Harmony Libraries and Applications > Guided Tour*.

Trademarks

Provides information on trademarks used in this documentation.

Description

Trademarks

The Microchip name and logo, the Microchip logo, AnyRate, AVR, AVR logo, AVR Freaks, BeaconThings, BitCloud, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, Heldo, JukeBlox, KeeLoq, KeeLoq logo, Kleer, LANCheck, LINK MD, maXStylus, maXTouch, MediaLB, megaAVR, MOST, MOST logo, MPLAB, OptoLyzer, PIC, picoPower, PICSTART, PIC32 logo, Prochip Designer, QTouch, RightTouch, SAM-BA, SpyNIC, SST, SST Logo, SuperFlash, tinyAVR, UNI/O, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

ClockWorks, The Embedded Control Solutions Company, EtherSynch, Hyper Speed Control, HyperLight Load, IntelliMOS, mTouch, Precision Edge, and Quiet-Wire are registered trademarks of Microchip Technology Incorporated in the U.S.A.

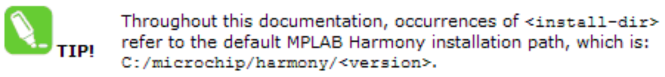
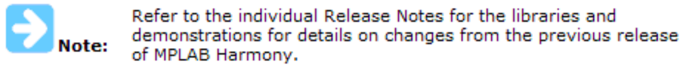
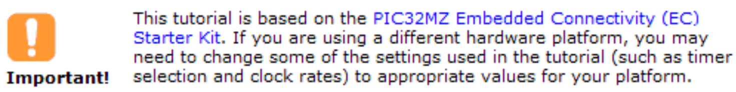
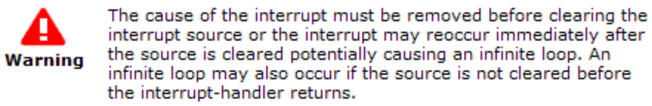
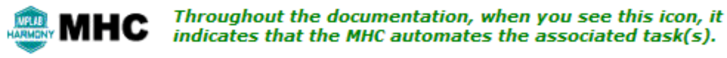
Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BodyCom, chipKIT, chipKIT logo, CodeGuard, CryptoAuthentication, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, EtherGREEN, In-Circuit Serial Programming, ICSP, Inter-Chip Connectivity, JitterBlocker, KleerNet, KleerNet logo,

Typographic Conventions

This topic describes the typographic conventions used in the MPLAB Harmony Help.

Description

The MPLAB Harmony Help uses the following typographic conventions:

Convention	Represents	Example
TIP!	Provides helpful information to assist the user.	 **TIP!** Throughout this documentation, occurrences of `<install-dir>` refer to the default MPLAB Harmony installation path, which is: `C:/microchip/harmony/<version>`.
Note:	Provides useful information to the user.	 **Note:** Refer to the individual Release Notes for the libraries and demonstrations for details on changes from the previous release of MPLAB Harmony.
Important!	Provides important information to the user.	 **Important!** This tutorial is based on the PIC32MZ Embedded Connectivity (EC) Starter Kit. If you are using a different hardware platform, you may need to change some of the settings used in the tutorial (such as timer selection and clock rates) to appropriate values for your platform.
Warning	Warns the user of a potentially harmful issue.	 **Warning** The cause of the interrupt must be removed before clearing the interrupt source or the interrupt may reoccur immediately after the source is cleared potentially causing an infinite loop. An infinite loop may also occur if the source is not cleared before the interrupt-handler returns.
MHC Followed by ***Green italicized*** text	Indicates a process step that is automated by the MPLAB Harmony Configurator (MHC).	 **MHC** ***Throughout the documentation, when you see this icon, it indicates that the MHC automates the associated task(s).***
Italic Characters	Referenced documentation and emphasized text.	• *MPLAB X IDE User's Guide* • ...is the *only* option.
Initial Capitalization	• A window • A dialog • A menu selection	• the Output window • the SaveAs dialog • the Enable Programmer menu
Quotation Marks	A field name in a window or dialog.	"Save project before build"
Italic text with right angle bracket	A menu path.	*File > Save*
Bold Characters	• Topic headings • A dialog button or user action, such as clicking an icon or selecting an option	• **Prerequisites** • Click **OK**

Courier New text enclosed in angle brackets	A key on the keyboard.	Press `<Ctrl><V>`.
Courier New text	• Sample source code • File names • File paths	• `#define START` • `system_config.h` • `<install-dir>/apps/examples`
Square Brackets	Optional arguments.	`command [options] file [options]`
Curly Braces and Pipe Character	Choice of mutually exclusive arguments; an OR selection.	`errorlevel {0|1}`

Recommended Reading

The following Microchip documents are available and recommended as supplemental reference resources.

Description

Device Data Sheets

Refer to the appropriate device data sheet for device-specific information and specifications.

Reference information found in these data sheets includes:

- Device memory maps
- Device pin out and packaging details
- Device electrical specifications
- List of peripherals included on the devices

To access this documentation, please visit, http://www.microchip.com/pic32/ and click **Documentation**. Then, expand **Data Sheets** to see the list of available documents.

MPLAB® XC32 C/C++ Compiler User’s Guide (DS50001686)

This document details the use of Microchip’s MPLAB XC32 Compiler for 32-bit microcontrollers to develop 32-bit applications. Please visit the Microchip website to access the latest version of this document.

MPLAB® X IDE User’s Guide (DS50002027)

Consult this document for more information pertaining to the installation and implementation of the MPLAB X IDE software. Please visit the Microchip website to access the latest version of this document.

Documentation Feedback

This topic includes information on how to provide feedback on this documentation.

Description

Your valuable feedback can be provided to Microchip in several ways. Regardless of the method you use to provide feedback, please include the following information whenever possible:

- The Help platform you are viewing:
  - Adobe® Portable Document Format (PDF)
  - Windows® Compiled Help (CHM)
  - HyperText Markup Language (HTML)
- The title of the topic and the section in which it resides
- A clear description of the issue or improvement

How To Send Your Feedback

It is preferred that you use one of the following two methods to provide your feedback:

- Through the Documentation Feedback link, which is available in the header and footer of each topic when viewing compiled

Help (CHM) or HTML Help

- By email at: docerrors@microchip.com

If either of the two previous methods are inconvenient, you may also provide your feedback by:

- Contacting your local Field Applications Engineer
- Contacting Customer Support at: http://support.microchip.com

Help Features

Describes the features available in the Help files provided in MPLAB Harmony.

CHM Help Features

Provides detailed information on the features available in CHM Help files.

Description

The MPLAB Harmony CHM files are located in the ./doc subfolder of the package it documents. For example, documentation on the MPLAB Harmony 3 Configurator is found at ./mhc/doc/help_mhc.chm and documentation on the MPLAB Harmony Graphics Library is found at ./gfx/doc/help_harmony_gfx.chm.

Help Icons

Several icons are provided in the interface of the Help, which aid in accessing the Help content.

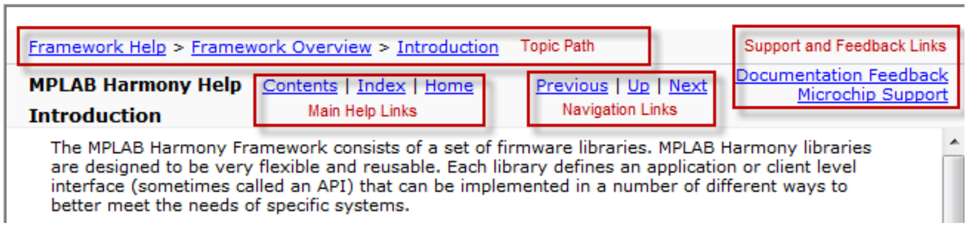
Table 1: Help Icon Features

Help Icon	Description
Hide Show	Use the Hide icon to turn off the left Help pane. Once the Hide icon is selected, it is replaced with the Show icon. Clicking the Show icon restores the left Help pane.
Locate	Use the Locate icon to visually locate the Help topic you are viewing in the Contents. Clicking the Locate icon causes the current topic to be highlighted in blue in the Contents pane.
Back	Use the Back icon to move back through the previously viewed topics in the order in which they were viewed.
Forward	Use the Forward icon to move forward through the previously viewed topics in the order in which they were viewed.
Home	Use the Home icon to return to the first topic in the Help.
Print	Use the Print icon to print the current topic or the selected heading and all subtopics.
Options	Use the Options icon to: • Hide tabs • Locate a topic • Go Back, Forward, and Home • Stop • Refresh • Set Internet Explorer options • Print topics • Turn Search Highlight Off and On

Topic Window

The Topic Window displays the current topic. In addition to the Help content, special links are provided in the upper portion of the window, as shown in Figure 2. Table 2 lists and describes the different links by their category

Figure 2: Help Links



The MPLAB Harmony Framework consists of a set of firmware libraries. MPLAB Harmony libraries are designed to be very flexible and reusable. Each library defines an application or client level interface (sometimes called an API) that can be implemented in a number of different ways to better meet the needs of specific systems.

Table 2: Help Links

Link Category	Description
Topic Path	The full path of the current topic is provided at the top and bottom of each topic, beginning with the top-level section name.
Support and Feedback Links: • Documentation Feedback • Microchip Support	 Click this link to send feedback in the form of an email (see **Note 1**). Click this link to open the Microchip Support Web page.
Main Help Links: • Contents • Index • Home	 Click this link to open the Contents in the left pane. Click this link to open the Index in the left pane (see **Note 2**). Click this link to go to the initial Help topic (see **Note 2**).
Navigation Links: • Previous • Up • Next	 Click this link to go back to the previously viewed topic. Click this link to go to the parent section of the topic. Click this link to go to the next topic.

Notes:

1. To use the *Documentation Feedback* link, you must have an email system, such as Outlook configured. Clicking the link automatically opens a new email window and populates the recipient and subject lines.
2. The *Home* and *Index* links do not appear initially. Once you begin traversing the topics, they dynamically appear.

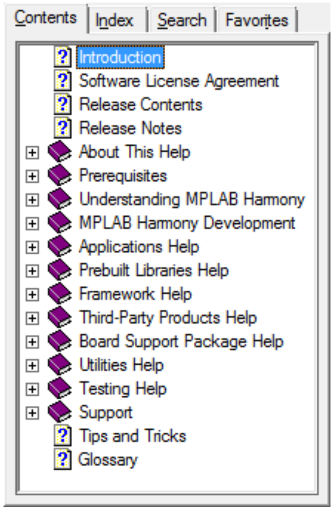
Tabs

The CHM Help provides four Tabbed windows: *Contents*, *Index*, *Search*, and *Favorites*.

Contents

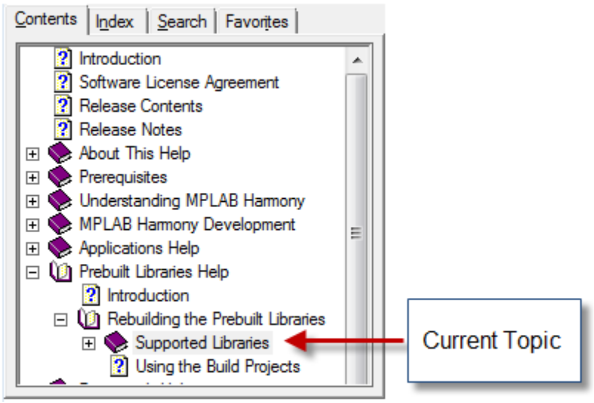
The Contents tab displays the top-level topics/sections. Figure 3 shows the initial view when the CHM Help is first opened.

Figure 3: Initial Contents Tab View



As topics are explored, the information in the Contents tab dynamically updates. For example, by clicking **Prebuilt Libraries Help** and using the Next link in the current topic to traverse through this section, the collapsed section automatically expands and the current topic is highlighted in light gray, as shown in Figure 4.

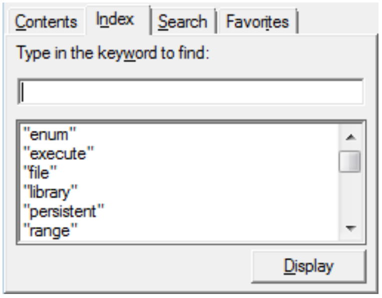
Figure 4: Current Topic Highlighting



Index

Clicking the Index tab results in an alphabetic list of all Help index entries. Figure 5 shows the default Index interface.

Figure 5: Default Index Interface

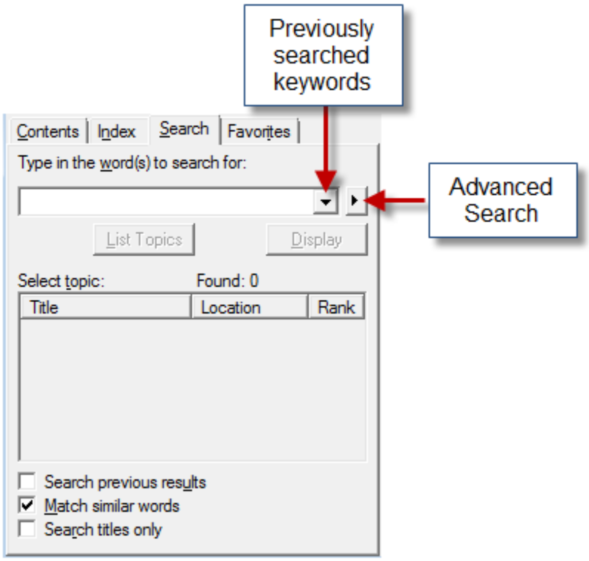


- To locate a specific entry, enter the keyword in the *Type in the keyword to find:* box. As you type, the index list dynamically updates.
- To display the desired item in the list, select the item and click **Display**, or double-click the desired item. The related content appears in the Help window.

Search

Clicking the Search tab provides an efficient way to find specific information. Figure 6 shows the default Search interface.

Figure 6: Default Search Interface

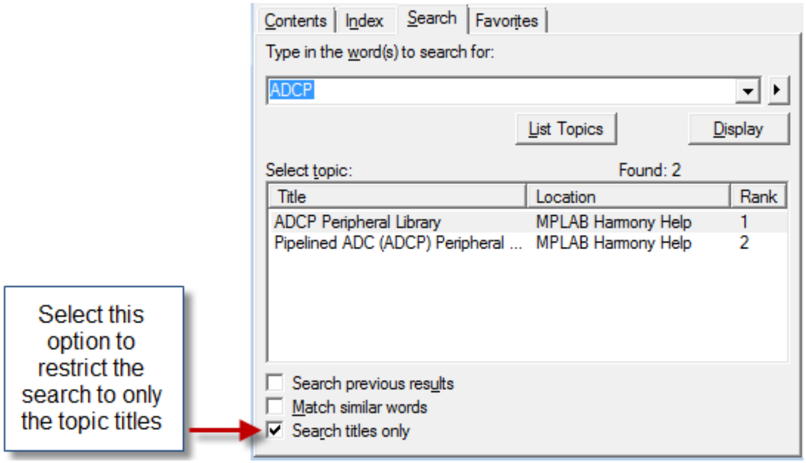


☐ Search previous results
☑ Match similar words
☐ Search titles only

- Enter the specific word or words in the *Type in the word(s) to search for:* box
- Clicking the drop-down arrow provides the list of previously searched words
- The right arrow provides Advanced Search options: AND, OR, NEAR, and NOT
- Located at the bottom left of the Search window, three options are provided to narrow-down your search. By default, *Match similar words is selected*. To reduce the number of returned words, clear this box and select *Search titles only*, which restricts

the search to only the topic titles in the Help, as shown in Figure 7.

Figure 7: Search Titles Only



Type in the word(s) to search for:

ADCP

Select topic: Found: 2

Title	Location	Rank
ADCP Peripheral Library	MPLAB Harmony Help	1
Pipelined ADC (ADCP) Peripheral ...	MPLAB Harmony Help	2

☐ Search previous results

☐ Match similar words

☑ Search titles only

Select this option to restrict the search to only the topic titles

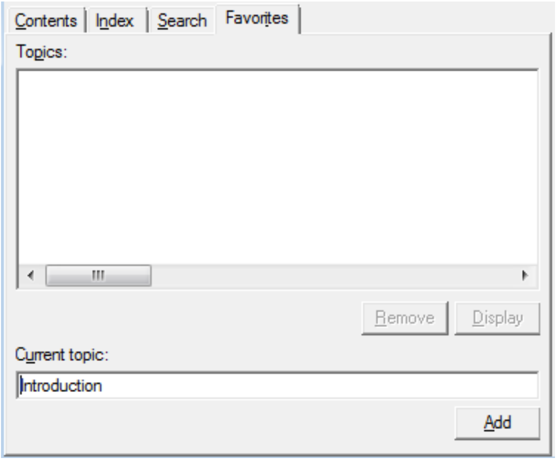
- The *Title* column provides the list of related topics
- The *Location* column lists in which Help system the topic was found (see **Note**)
- The *Rank* column determines to search result that most closely matches the specified word

Note: The *Location* column is automatically included in the CHM Help when the Advanced Search features are implemented and cannot be excluded. Its purpose is to provide the name of the Help system in which the topic is located for Help output that is generated from multiple sources. Since the MPLAB Harmony Help is contained within a single Help system, this information is the same for all searches. Do not confuse this column to mean the actual topic location.

Favorites

Use the Favorites tab to create a custom list of topics that you may want to repeatedly access. Figure 8 shows the default Favorites interface.

Figure 8: Default Favorites Interface

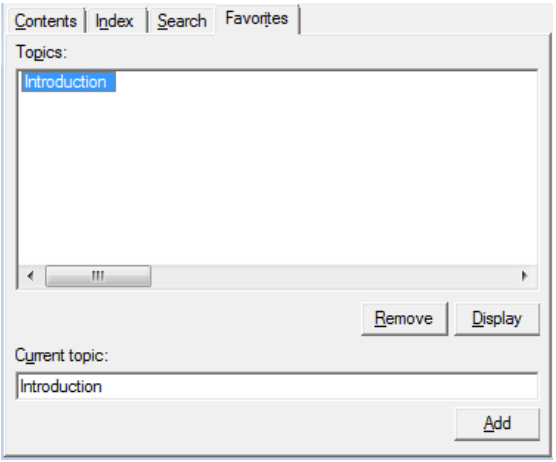


Current topic:

Introduction

- The title of the current topic is shown in the *Current topic:* box.
- Click **Add** to add the topic to the *Topics:* list, as shown in Figure 9.
- Click **Display** to view the selected topic.
- Click **Remove** to remove the selected topic from the list of favorites.

Figure 9: Adding a Favorite Topic



HTML Help Features

Provides detailed information on the features available in the stand-alone HTML Help.

Description

The HTML Help output for MPLAB Harmony has two purposes. First, it can be used as "stand-alone" Help. Second, the HTML files are used by the MPLAB Harmony Configurator (MHC) when using MHC in MPLAB X IDE.

Stand-alone HTML Help

The MPLAB Harmony index.html file that is the root for all HTML help is located in the ./doc subfolder of the package it documents. For example, documentation on the MPLAB Harmony 3 Configurator is found at ./mhc/doc/index.html and documentation on the MPLAB Harmony Graphics Library is found at ./gfx/doc/index.html.

To use the HTML Help in a "stand-alone" manner, open the file `index.html` in your browser of choice. Click **Allow blocked content** if a message appears regarding ActiveX controls.

The following links are provided:

- *Table of Contents* - Located at the top left, clicking this link opens the Table of Contents in the right frame
- *Topic Path* - At the top and bottom of each topic, the full path to the current topic is listed
- *Microchip Logo* - Clicking this image opens a new browser tab and displays the Microchip website (www.microchip.com)
- *Contents* - The Contents topic is a static file, which displays and lists the major sections available in the Help in the left frame. Due to a restriction with the Help browser used by the MHC, a dynamic Contents topic cannot be used.
- *Home* - This link returns to the Introduction topic (see **Note 1**)
- *Previous* and *Next* navigation links - Use these links to traverse through the Help topics
- *Documentation Feedback* - Use this link to provide feedback in the form of an email (see **Note 2**)
- *Microchip Support* - Use this link to open the Support page of the Microchip website

Notes:

1. The *Home* link does not appear initially. Once you begin traversing the topics, it dynamically appears.
2. To use the *Documentation Feedback* link, you must have an email system such as Outlook configured. Clicking the link automatically opens a new email message and populates the recipient and subject lines.

PDF Help Features

Provides detailed information on the features available in the PDF version of the Help.

Description

The MPLAB Harmony Help provided in Portable Document Format (PDF) provides many useful features. By default, PDF bookmarks should be visible when opening the file. If PDF bookmarks are not visible, click the PDF Bookmark icon, which is located near the top of the left navigation pane or by selecting *View* > *Show/Hide* > *Navigation Panes* > *Bookmarks.*

The MPLAB Harmony PDF files are located in the ./doc subfolder of the package it documents. For example, documentation on the MPLAB Harmony 3 Configurator is found at ./mhc/doc/help_mhc.pdf and documentation on the MPLAB Harmony Graphics Library is found at ./gfx/doc/help_harmony_gfx.pdf.

To make full use of the PDF features, it is recommended that Adobe products be used to view the documentation (see **Note**).

Help on how to use the PDF features is available through your copy of Acrobat (or Acrobat Reader) by clicking **Help** in the main menu.

Note: The MPLAB Harmony Help PDF files can be viewed using a PDF viewer or reader that is compatible with Adobe PDF Version 7.0 or later.

Microchip Website

This topic provides general information on the Microchip website.

Description

The Microchip website can be accessed online at: http://www.microchip.com

Accessible by most Internet browsers, the following information is available:

Product Support

- Data sheets
- Silicon errata
- Application notes and sample programs
- Design resources
- User’s guides
- Hardware support documents
- Latest software releases and archived software

General Technical Support

- Frequently Asked Questions (FAQs)
- Technical support requests
- Online discussion groups
- Microchip consultant program member listings

Business of Microchip

- Product selector and ordering guides
- Latest Microchip press releases
- Listings of seminars and events
- Listings of Microchip sales offices, distributors, and factory representatives

Microchip Forums

This topic provides information on the Microchip Web Forums.

Description

The Microchip Web Forums can be accessed online at: http://www.microchip.com/forums

Microchip provides additional online support via our web forums.

The Development Tools Forum is where the MPLAB Harmony forum discussion group is located. This forum handles questions and discussions concerning the MPLAB Harmony Integrated Software Framework and all associated libraries and components.

Additional Development Tool discussion groups include, but are not limited to:

- MPLAB X IDE - This forum handles questions and discussions concerning the released versions of the MPLAB X Integrated Development Environment (IDE)
- MPLAB XC32 - This forum handles questions and discussions concerning Microchip’s 32-bit compilers, assemblers, linkers, and related tools for PIC32 microcontrollers
- Tips and Tricks - This forum provides shortcuts and quick workarounds for Microchip's development tools
- FAQs - This forum includes Frequently Asked Questions

Additional forums are also available.

Customer Support

This topic provides information for obtaining support from Microchip.

Description

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office (see Contact Microchip Technology to locate your local sales office)
- Field Application Engineer (FAE)
- Technical Support (http://support.microchip.com)

Contact Microchip Technology

Worldwide sales and service contact information for Microchip Technology Inc.

Description

Please visit the following Microchip Web page for contact information: http://www.microchip.com/about-us/contact-us

Glossary

This topic contains a glossary of general MPLAB Harmony terms.

Description

Glossary of MPLAB Harmony Terms

Term	Definition
Application	One or more application modules define the overall behavior of a MPLAB Harmony system. Applications are either demonstrations or examples provided with the installation or are implemented by you, using MPLAB Harmony libraries to accomplish a desired task.
Client	A client module is any module that uses the services (calls the interface functions) of another module.
Configuration	A MPLAB Harmony configuration consists of static definitions (C language `#define` statements), executable source code, and other definitions in a set of files that are necessary to create a working MPLAB Harmony system. (See the System Configurations section for additional information.)
Configuration Options	Configuration options are the specific set of `#define` statements that are required by any specific MPLAB Harmony library to specify certain parameters (such as buffer sizes, minimum and maximum values, etc.) build that library. Configuration options are defined in the `system_config.h` system-wide configuration header.
Driver	A "driver" (or device driver) is a MPLAB Harmony software module designed to control and provide access to a specific peripheral, either built into or external to the microcontroller.
Driver Index	Dynamic MPLAB Harmony drivers (and other dynamic modules) can manage the more than one instance of the peripheral (and other resources) that they control. The "driver index" is a static index number (0, 1, 2,...) that identifies which instance of the driver is to be used. **Note:** The driver index is not necessarily identical to the peripheral index. The association between these two is made when the driver is initialized.
Driver Instance	An instance of a driver (or other module) consists of a complete set of the memory (and other resources) controlled by the driver's code. Selection of which set of resources to control is made using a driver index. **Note:** Even though there may be multiple instances of the resources managed by a dynamic driver, there is only ever one instance of the actual object code. However, static drivers always maintain a 1:1 relationship between resource and code instances.
Framework	The MPLAB Harmony framework consists of a set of libraries (and the rules and conventions used to create those libraries) that can be used to create MPLAB Harmony systems.

Handle	A handle is a value that allows one software module to "hold" onto a specific instance of some object owned by another software module (analogous to the way a valet holds the handle of a suitcase), creating a link between the two software modules. A handle is an "opaque" value, meaning that the "client" module (the module that receives and holds the handle) must not attempt to interpret the contents or meaning of the handle value. The value of the handle is only meaningful to the "server" module (the module that provides the handle). Internal to the server module, the handle may represent a memory address or it may represent a zero-based index or any other value, as required by the "server" module to identify the "object" to which the client is linked by the handle.
Initialization Overrides	Initialization overrides are configuration options that can be defined to statically override (at build time) parameters that are normally passed into the "Initialize" function of a driver or other MPLAB Harmony module. This mechanism allows you to statically initialize a module, instead of dynamically initializing the module.
Interface	The interface to a module is the set of functions, data types, and other definitions that must be used to interact with that module.
Middleware	The term "middleware" is used to describe any software that fits between the application and the device drivers within a MPLAB Harmony system. This term is used to describe libraries that use drivers to access a peripheral, and then implement communication protocols (such as TCP/IP, USB protocols, and graphics image processing), as well as other more complex processing, which is required to use certain peripherals, but is not actually part of controlling the peripheral itself.
Module	A MPLAB Harmony software module is a closely related group of functions controlling a related set of resources (memory and registers) that can be initialized and maintained by the system. Most MPLAB Harmony modules provide an interface for client interaction. However, "headless" modules with no interface are possible.
Peripheral Index	A peripheral index is a static label (usually an C language "enum" value) that is used to identify a specific instance of a peripheral. **Note:** Unlike a driver index, which always starts at '0', a peripheral index may be internally represented as any number, letter, or even a base address and the user should not rely on the value itself, but only the label.
Peripheral Instance	An instance of a peripheral is a complete set of the registers (and internal physical resources) necessary to provide the core functionality of a given type of peripheral (either built into or external to the microcontroller). **Note:** A specific peripheral instance is identified using a peripheral index.
System	A MPLAB Harmony system is a complete set of libraries, applications, and configuration items loaded and executing on a specific hardware platform (microcontroller, board, and external peripherals) or the source items necessary to build such a system. **Note:** Since a system can multiple configurations, one MPLAB Harmony project may support multiple systems through multiple supported configurations. See the demonstration applications included in the installation for examples.
System Service	A system service is a MPLAB Harmony module that provides access to and control of common system resources (memory and registers) with which other modules (drivers, middleware, libraries and application) may interact. **Note:** System services, much like drivers, manage sharing of resources so as to avoid conflicts between modules that would otherwise occur if each module attempted to manage the share resource itself. But, unlike drivers, system services do not normally need to be "opened" to use them.

Interface

This section contains information about the data types in the USB Common Driver Interface.

Files

The following table lists files in this documentation.

Files

Name	Description
drv_usbfs_config_template.h	USB Full Speed (USBFS) Driver Configuration Template.

drv_usbfs_config_template.h

USB Full Speed (USBFS) Driver Configuration Template.

Description

USB Full Speed Driver Configuration Template.

This file lists all the configurations constants that affect the operation of the USBFS Driver.

File Name

drv_usbfs_config_template.h

Company

Microchip Technology Inc.

Index