Creating a USB Audio Device on a PIC32 MCU Using MPLAB Harmony

Introduction

The Universal Serial Bus (USB) is among the most commonly used interfaces for connecting different electronic devices. Along with major PC operating systems, the USB is also supported across various embedded system platforms. The USB Protocol provides native support for transporting digital audio data. This support along with its ease-of-use, makes the USB a popular choice for inter-connecting digital audio devices.

Developing an USB Audio application poses several design challenges, which includes USB protocol complexity, digital audio data synchronization, Codec configuration, and Host operating system compatibility. Therefore, developing a USB Audio application may entail significant development cost and time.

This application note discusses the USB Audio Device Class v1.0 Specification, and provides guidance for implementing USB Audio device solutions on PIC32-based microcontrollers using Microchip's MPLAB® Harmony Integrated Software Framework. The focus of this application note is to discuss development of a USB Headset application, which is included in MPLAB Harmony. The application project is located in the `apps\audio\usb_headset` folder of the MPLAB Harmony installation.

This application note includes the following topics:

- Functional Model: Discusses the functional model of the USB Headset application consisting of an audio subsystem and USB subsystem
- Audio Overview: Describes the essential concepts of digital audio communication pertaining to the audio subsystem
- USB Audio Overview: Describes the essential concepts of USB audio communication pertaining to the USB subsystem
- MPLAB Harmony Overview: Overview Microchip's MPLAB Harmony Integrated Software Framework
- Building a USB Audio Device Using MPLAB Harmony: Provides detailed steps to build the USB Headset application using the MPLAB Harmony Configurator (MHC) and discusses the architecture and usage of MPLAB Harmony audio drivers and the USB Audio Library
- USB Audio Application Considerations: Notes and observation on audio application-specific topics, such as clock tuning and buffering schemes

Table of Contents

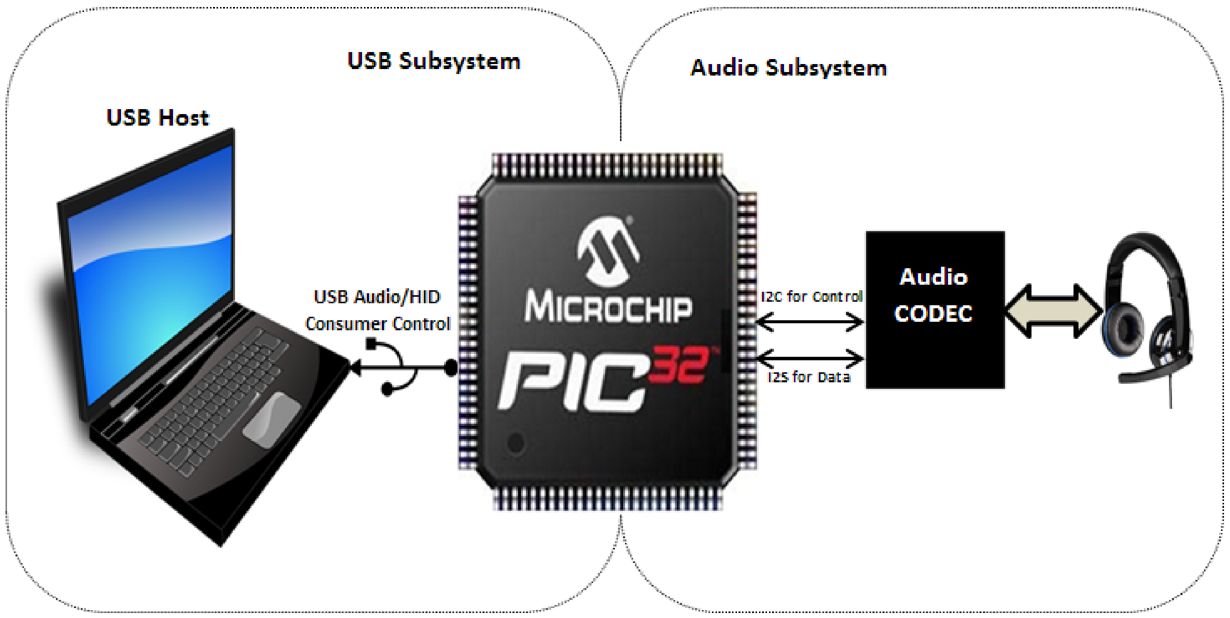
1. Functional Model

The USB subsystem has a host and a device unit. The device unit is connected to the host through a USB cable. A standard PC typically assumes the role of the USB host and an embedded device (PIC32 microcontroller) performs the role of the USB device. The USB host runs the USB host software library and has the necessary USB audio device drivers. The USB device runs the USB device software stack and identifies itself as a USB Audio device to the USB host.

In the USB headset application, the audio subsystem consists of the audio headset, an audio codec and the PIC32 microcontroller. The audio codec translates between analog and digital signal domains and allows the PIC32 microcontroller to send and receive signals from the headset in a digital format. The PIC32 microcontroller interfaces to the audio codec through serial communication. This includes the data and control interfaces. Audio codec parameters, such as volume, mute, or equalization are accessed through the control interface. Additional interfaces, which translate audio controls (mute, volume, etc.,) from the audio subsystem to the USB subsystem, are also part of the USB audio system.

The following figure shows the functional model for the USB headset application. It consists of a USB subsystem and an audio subsystem.

Figure 1-1. USB Audio Device Functional Model for the USB Headset Application

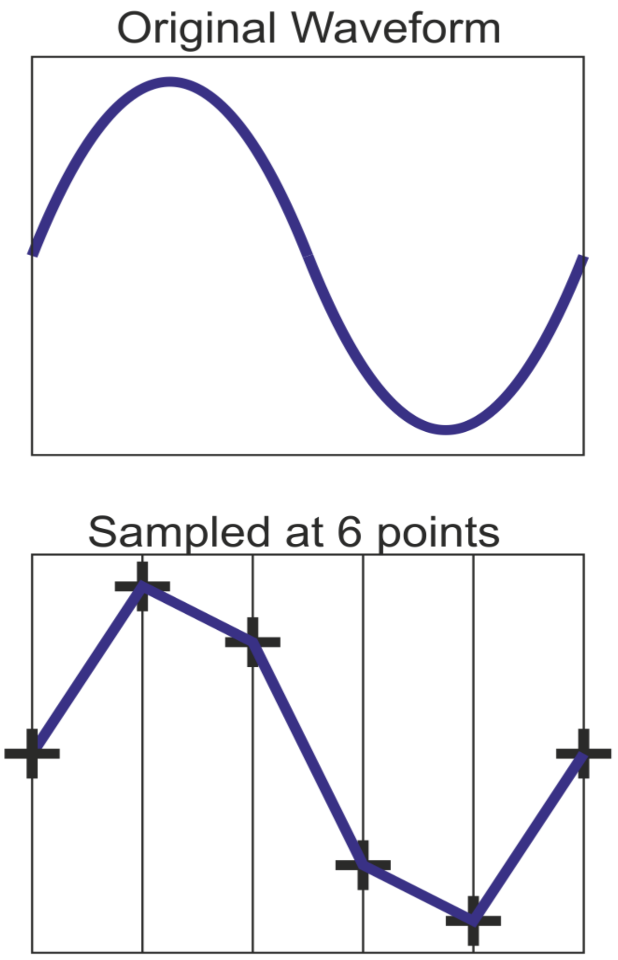


2. Audio Overview

The representation of sound in digital form is a common phenomenon. Audio signals are digitized with the help of converters (Analog-to-Digital (ADC)/Digital-to-Analog Converter (DAC)) and represented and stored in a digital medium. Some of the characteristics and terminologies associated with digital audio are described below.

Sampling Frequency- It is the number of times (per second) the audio signal is sampled and stored. A continuous audio signal is sampled at some rate and these sampled values are converted into a digital representation. The digital samples are discrete values (numbers), which represent the amplitude of the audio signal at different (sampled) points in time. The sampling rate is measured in Hz or kHz. For example, a 44100 Hz sampling rate is used in compact disc (CD) audio (see the following figure). A higher sampling frequency improves the audio reproduction capability of the system, but this also increases the processing requirements.

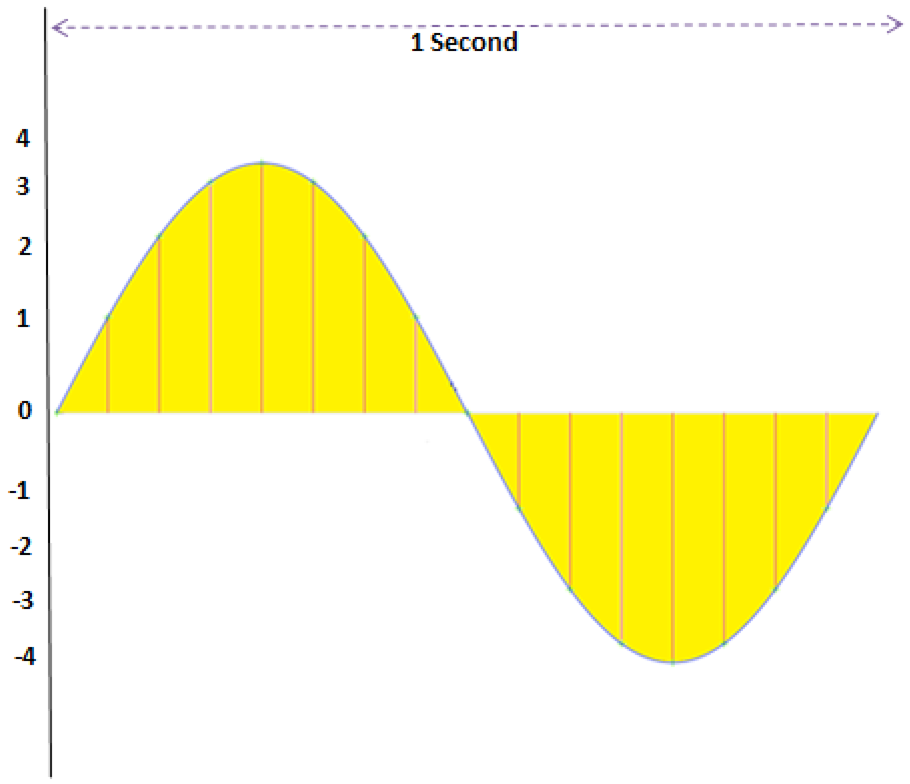
Figure 2-1. Sampling Rate



Bit Depth - Represents the number of bits used to encode the level of an audio sample. This parameter defines the dynamic range of the digital audio system. The dynamic range for a 16-bit audio system (bit depth is 16) is calculated as 2^{16}, giving 65,536 possible levels. The dynamic range for a 24-bit audio system is calculated as 2^{24}, providing over 16 million levels. With every bit, the number of levels is doubled.

In the following figure, there are 14 samples in a second. The largest sample level is four, which can be represented in four bits (3 bits for amplitude and 1 bit for sign). Hence the bit depth of the audio signal is four.

Figure 2-2. Bit Depth



Bit Rate - The number of bits processed per unit of playback time. The bit rate for an uncompressed digital audio file is calculated as: Sample Rate x Bit-Depth x Number of Channels = Bit Rate.

As an example, the bit rate for a CD quality audio: 44,100 x 16 x 2 = 1,411,200 bits per second (or 1411 kbps, or 1.4 Mbps).

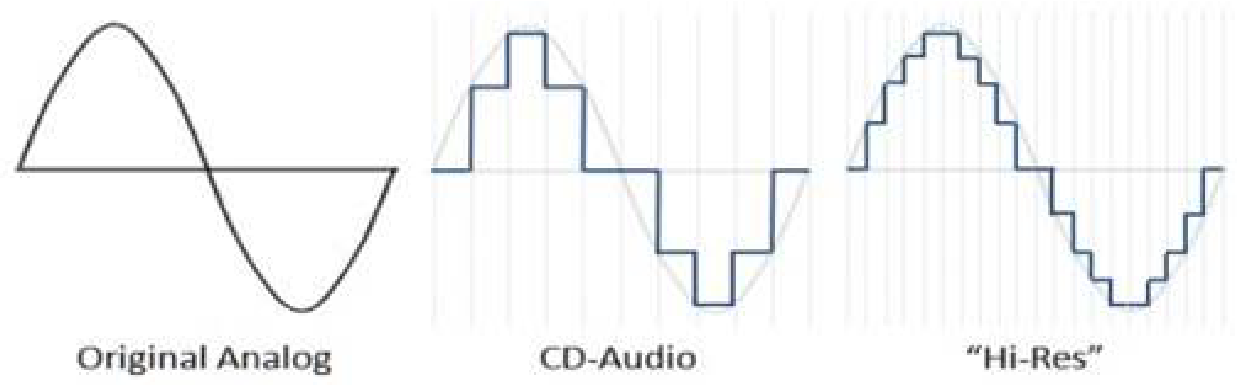
For a compressed digital audio file, the bit rate is the minimum rate at which the compressed data must be transferred for uninterrupted reproduction.

For example, if a CD audio signal (16-bit, 2 channels, 44100 Hz sampling rate, bit rate of 1411 kbps) is encoded "at a bit rate of 128 kbps" using the MP3 algorithm, it would have to be streamed continuously through a link providing a transfer rate of at least 128 kbps.

The **Resolution** - The resolution of a digital audio system is based on three factors: the sample rate, bit depth, and bit rate. For example, a CD audio signal with an audio resolution sampling rate of 44100, with a 16-bit depth at 1411 kbps.

Digital audio signals with a bit rate greater than the CD digital audio signals are called high-resolution audio signals. High resolution or fidelity audio signals are sampled with a frequency greater than 44100 Hz and a bit depth greater than 16 bits. A typical high-resolution audio signals are sampled at 192 kHz with a bit depth of 24 bits. The following figure shows a pictorial representation of a high resolution signal.

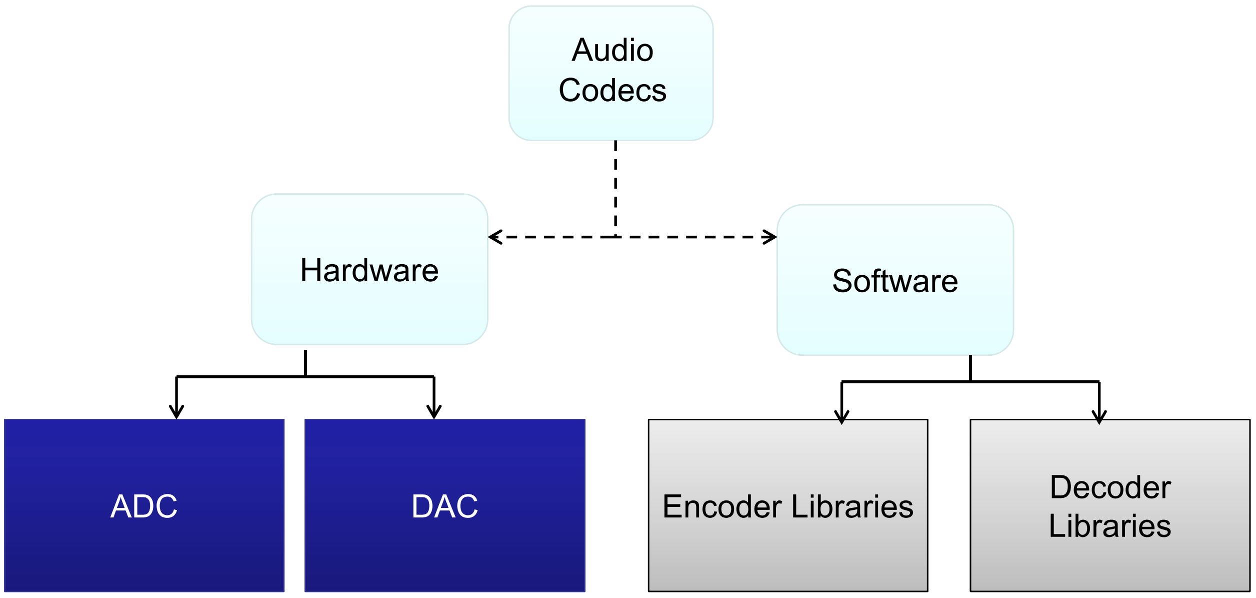
Figure 2-3. Audio Resolution



Audio File Formats- Digital audio data is stored in files in an uncompressed or compressed form. The size of an uncompressed audio file tends to be larger than that of a compressed audio file.

Audio Codecs- An audio codec is a device or a program capable of coding or decoding a digital audio data stream. As shown in the following figure, audio codecs can be classified into software and hardware audio codecs.

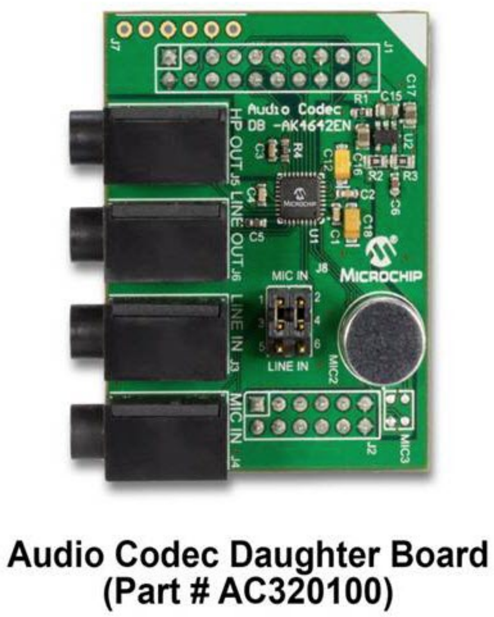
Figure 2-4. Audio Codecs



A software audio codec is a computer program implementing an algorithm that compresses and decompresses digital audio data according to a given audio file or streaming media audio coding format. Software audio codecs are implemented as libraries, for example, MP3, SBC, and AAC codec libraries.

A hardware audio codec refers to a single device that encodes analog audio as digital signals, as ADC, and decodes digital data back into analog signals, as DAC. Sound cards supporting audio inputs and outputs contain a hardware audio codec. The following figure shows an example of the PIC32 Audio Codec AK4642EN Daughter Board.

Figure 2-5. Audio CODEC Board



A typical audio codec device has two interfaces, a control interface and a data interface. The control interface is the medium to configure the control registers of the codec, which typically uses the I^{2}C protocol.

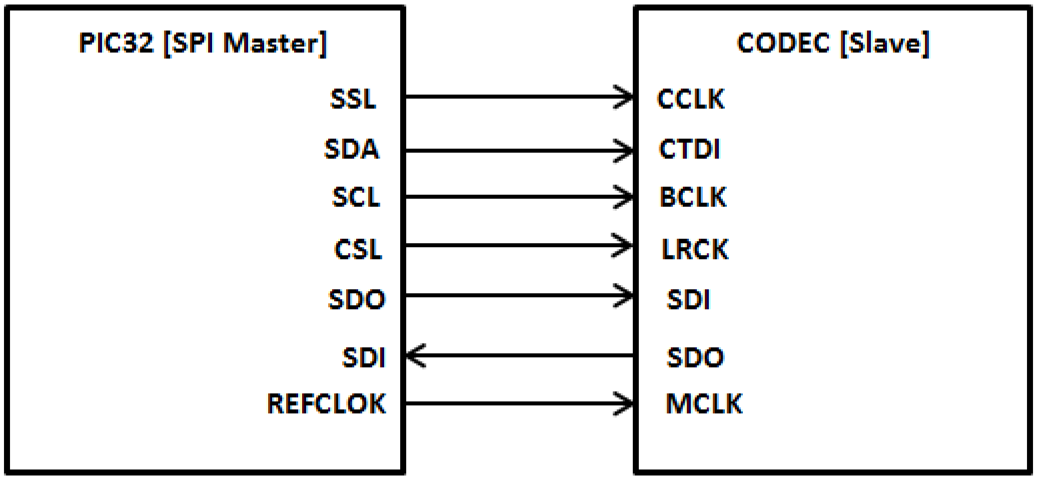
The microcontroller transmits and receives digital audio signals from the codec over the data interface. The data interface typically uses the following signals.

- Serial Data Output (SDO) to transmit audio data to the codec
- Serial Data Input (SDI) to receive audio data from the codec
- Serial Bit Clock (SCK/BCLK) is the required bit clock provided by the serial interface communication master
- Left/Right Clock (LRCK) is the phase clock for stereo data provided by the serial interface communication master

The SPI module on PIC32 devices supports various audio modes for digital audio communication. These modes support various interface formats, bit resolutions, and Master/Slave configurations. The digital audio modes supported include the I^{2}S format, the Left-Justified format, and the Right-Justified format.

The SPI module (in Audio mode) on a PIC32 device serves as the data interface between the codec and the PIC32 microcontroller. When the PIC32 microcontroller is acting as the SPI master, the reference clock (REFCLKO) output from the PIC32 device is the master clock (MCLK) to the slave codec device, as shown in the following figure.

Figure 2-6. Audio Codec Interface



3. USB Audio Overview

3.1 USB Operations

A USB device is connected to a USB host system through a USB port. The host communicates with the device through control transfers over USB Endpoint 0 and retrieves device capability related information. The host then loads the drivers that could operate the device. This process of detecting, identifying and loading drivers for a device is called *Enumeration*.

The USB device reports its attributes and other information, during enumeration, by using descriptors. A USB descriptor is a data structure with a defined format. Each descriptor begins with a field containing the total number of bytes in the descriptor followed by a field identifying the descriptor type.

The following is a list of standard USB descriptors that a USB host will request from a device during enumeration:

- Device descriptor
- Configuration descriptor
- Interface descriptor
- Endpoint descriptors
- String descriptors

The following sequential steps describe the typical enumeration process.

1. **Detecting Device Connection**
   The USB interface consists of four wires: power, ground, data plus (D+) and data minus (D-). When a USB device is connected to the USB host, there is a change on the USB data lines. The host uses this change in the data lines to detect a device connection. The changes in the data lines also identify the speed of the attached device.
2. **Determining Device Type (Device Descriptor)**
   Once the host has established that a USB device is connected, and has identified the communication speed, the host resets the USB device and attempts to read information that identifies the device. The host acquires this information through a *Device Descriptor*. The host loads a driver, which manages the device based on the Vendor ID (VID) and Product ID (PID) contained in the device descriptor.
3. **Determining Device Configuration (Configuration Descriptor)**
   The configuration descriptor provides device specific informations, such as the number of interfaces supported by the device and maximum power the device consumes. A device configuration determines the functions that a USB device will perform while connected to the USB.
4. **Determining Device Interface (Interface Descriptor)**
   The interface descriptor provides informations, such as the associated USB device function and the number of endpoints required for performing this function. The host can load the driver based on the Class, Subclass, and Protocol fields contained in the interface descriptor.

Enumeration related data, such as device descriptors are transferred using Control Transfers. Apart from Control Transfers, the USB supports the following three other transfers: bulk, interrupt, and isochronous transfers.

- Bulk transfers typically deal with transfer of large non-time sensitive data. The USB does not reserve any bandwidth for bulk transfers in that, the scheduling of bulk transfers. Bulk transfers are scheduled when there are no other pending transfers.
- Interrupt transfers send and receive small amounts of data infrequently or in asynchronous periods with bounded latency. A USB audio device may use interrupt transfers to provide audio controls related updates to the host.
- Isochronous transfers deal with constant rate real-time information, such as audio and video data. These transfers involve the flow of continuous data streams. The host guarantees the real-time requirement of data by dedicating a portion of available USB bandwidth. The amount of bandwidth allocated to a particular isochronous endpoint is determined by the information contained in the endpoint's interface descriptor. In isochronous transfers, a new data packet is transferred in every frame. For a full-speed USB device, a frame spans one millisecond. An Isochronous endpoint can be configured to transmit 1 through 1023 bytes per frame. Isochronous transfers do not support error detection and have no hardware-controlled handshaking or error-checking mechanism. A USB audio device uses isochronous transfers.

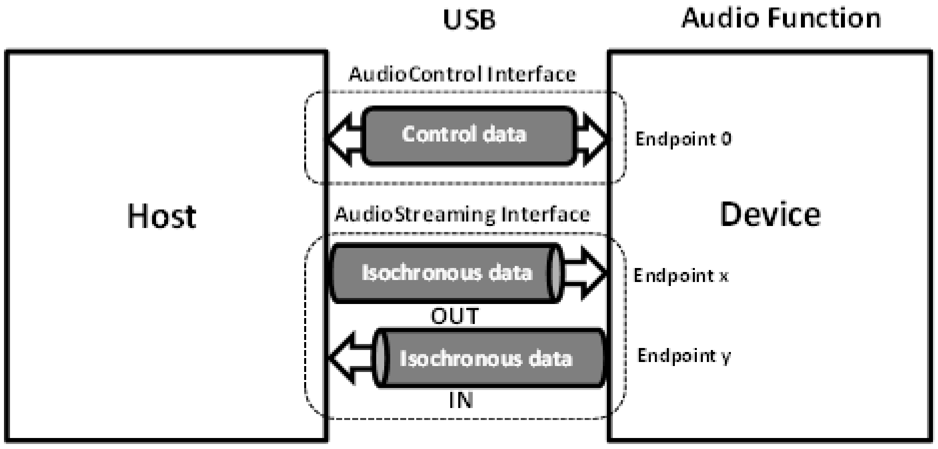
3.2 USB Audio Operational Model

The USB audio device implements audio functionality (headphone, microphone, etc.,) and provides the host with the access to the audio function through the USB interface. The audio function must have an AudioControl (AC) interface and can have multiple AudioStreaming (AS) interfaces, as shown in the following figure.

The AC interface is used to control the audio properties of the audio function, such as volume control and mute control etc.

The AS interface is the transport medium that carries the audio data between the host and the device.

Figure 3-1. USB Audio Operational Model



The AC interface uses Endpoint 0 for control data communication.

The AS interfaces use dedicated isochronous endpoints to transfer the audio data between the host and the device.

3.3 USB Audio Synchronization

A USB Audio device contains the following clock domains:

- **Sampling Clock:** Determines the sampling frequency (rate) of audio samples exchanged between the USB host and device
- **USB Bus Clock:** Runs at 1 kHz frequency on full-speed USB devices and is indicated by the occurrence of Start-of-Frame (SOF) packets on the bus
- **Service Clock:** Rate at which client software runs to service I/O request packets (IRPs) that may have accumulated between executions
- **Codec Clock:** Rate at which audio data is transferred to/from the codec module

For isochronous data to be transferred reliably, the clocks mentioned above must be synchronized. Clock synchronization is necessary to avoid undesirable audible artifacts caused due to clock drift, jitter, etc., Such artifacts are easily audible to the human ear and reduce audio quality.

The USB specification provides three synchronization types to address the clock mismatch issues, as shown in the following table.

Table 3-1. USB Audio Synchronization Type

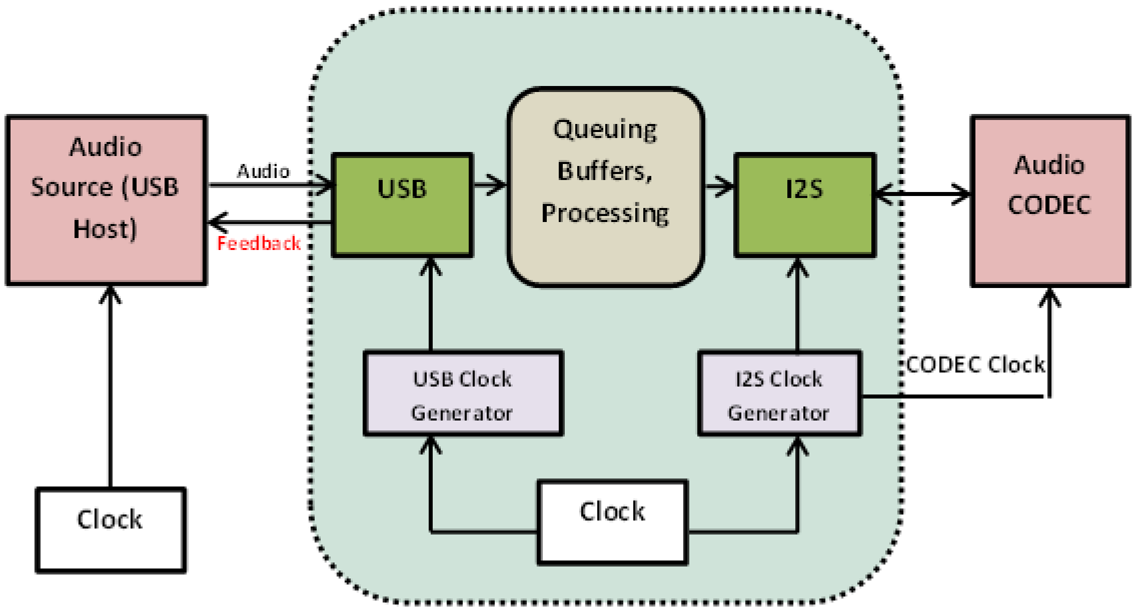
Type	Source	Sink
Asynchronous	Free running clock. Provides implicit feed forward to the sink	Free running clock. Provides explicit feedback to the source
Synchronous	Clock locked to USB SOF Uses implicit feedback	Clock locked to USB SOF Uses implicit feedback
Adaptive	Clock locked to sink Uses explicit feedback	Clock locked to the data flow Uses implicit feedback

3.3.1 Asynchronous

Asynchronous devices are not synchronized with the host by a common clock or the USB SOF signal. In asynchronous devices, a free-running internal clock determines the desired data rate. An asynchronous source endpoint carries its data rate information implicitly in the number of samples it produces per frame.

An asynchronous sink endpoint provides explicit feedback information to the source, as shown in the following figure.

Figure 3-2. USB Asynchronous Type



In a typical asynchronous sink endpoint implementation, the source (host) and sink (device) agree on the operating audio sampling rate (e.g., 48 kHz). The Host starts streaming data with the agreed samples (48) preceded by a SOF every millisecond. The device application maintains a pool of memory buffers to buffer the received audio data. It forwards the data to the codec for playback on the audio device. The device manages the buffering and processing of the data received from the host.

Since the host and device USB operate at different clocks, there are possibilities of a clock mismatch, which could lead to the host sending more buffers than could be managed by the device (buffer overflow) or sending fewer buffers that could lead the device to starve for data (buffer underflow).

In an asynchronous endpoint implementation, the device addresses buffer overflow and underflow by maintaining count or watermark levels for buffer overflow (upper level) and underflow (lower level) conditions. When the device senses that the buffer count is exceeding the watermark levels, it sends a message to the host to speed up or slow down. This message from the device to the host is called the *feedback message.* The feedback message is transferred over a dedicated standard isochronous synchronous endpoint, called the *feedback endpoint*. The explicit feedback from the device contains information such as the amount, by which to speed up or slow down. The host speeds up or slows down by adjusting its data rate corresponding to the feedback value received from the device.

In addition to the USB data synchronization, the clock to the codec needs to be selected such that it can produce the desired audio sampling frequencies. Commonly used audio sampling frequencies are 32 kHz, 44.1 kHz, 48 kHz, 96 kHz, and 192 kHz.

For example, the PIC32MX470F512L device has a flexible reference clock output module. This module can be used to generate the fractional clock frequencies that are required by an audio codec or a DAC to accommodate various sample rates.

Examples:

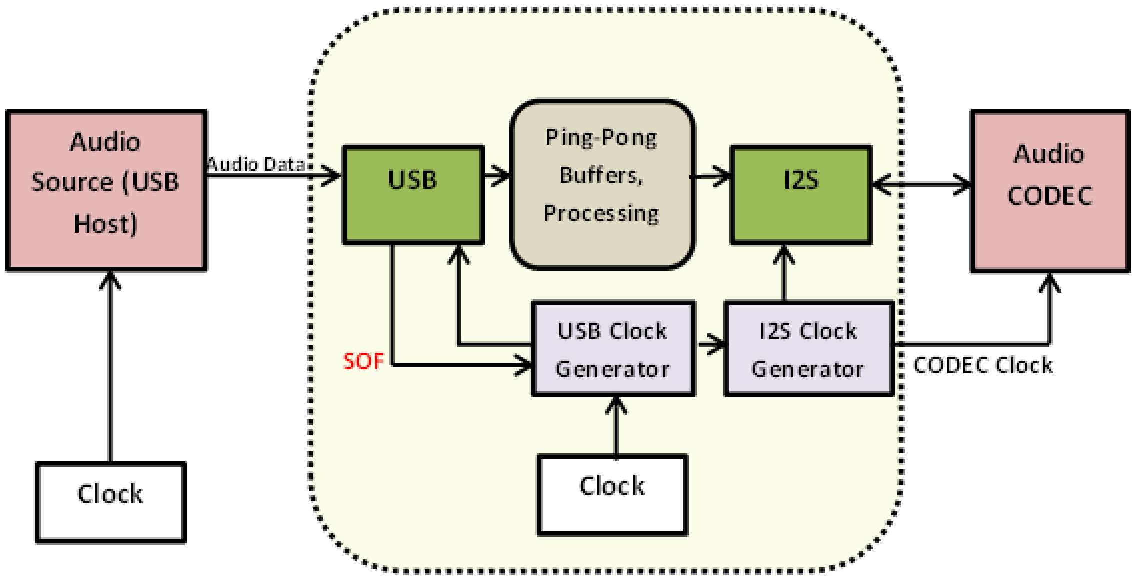
Asynchronous Source - An audio player running on a PC-based USB host that provides its data based on an internal clock.

Asynchronous Sink - A headphone running on its own internal sampling clock.

3.3.2 Synchronous

For the synchronous type, the device synchronizes with the host through the USB SOF signal that precedes each USB frame. The SOF signal is generated by the USB host and occurs at one millisecond intervals. As seen in the following figure, the USB device uses the SOF signal to calibrate an internal oscillator that produces the device sampling clock.

Figure 3-3. USB Synchronous Type



In a typical synchronous sink endpoint implementation, the source (host) and sink (device) agree on an operation audio sampling rate (i.e., 48 kHz). Every one millisecond, the USB host sends the SOF signal followed by the agreed number of samples (48). The SOF signal acts as an input to the USB clock generator (typically a PLL), which in turn causes the USB device clock to synchronize with the USB. The USB clock is further used to generate the master clock to the codec through a PLL block that generates the I^{2}S clock.

In this implementation, the device uses two buffers for data management and processing in a ping-pong buffer mechanism. In a ping-pong buffer mechanism, the data read is stored in a ping buffer while the data in the pong buffer is forwarded to the codec for playback. The ping and pong buffers are swapped and the data read from USB is continuously forwarded to the codec.

The synchronous endpoint implementation greatly simplifies buffer management and processing, since the data packets coming in and going out are based on the same clock. The fact that the USB clock and codec master clock are generated from the same source, helps to mitigate clock mismatch issues for commonly used audio sampling frequencies.

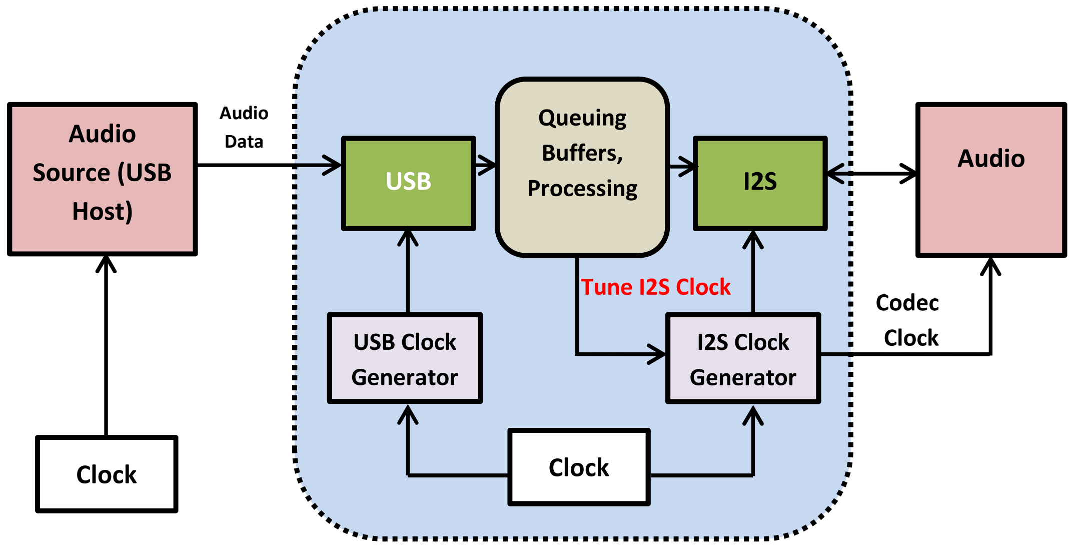
For example, the input to the reference clock output PLL in the PIC32MX470F512L device can be sourced from the USB PLL. The generated reference clock serves as the master clock (MCLK) input to the codec.

An example of a synchronous source is a microphone that synthesizes its sample clock from the SOF and produces a fixed number of audio samples every frame. Likewise, a synchronous sink derives its sample clock from the USB SOF signal and consumes a fixed number of samples every USB frame.

3.3.3 Adaptive

As the word *Adaptive* implies, the source and sink adapt their data rate based on the status from each other, as shown in the following figure.

Figure 3-4. USB Adaptive Type



An adaptive source endpoint produces data at a rate that is controlled by the data sink. The sink provides feedback to the source, which allows the source to know the desired data rate of the sink.

An adaptive sink endpoint embeds the data rate information in the data stream. The average number of samples received in an average time window determines the instantaneous data rate. If this number changes during operation, the data rate is adjusted accordingly.

In a typical adaptive sink endpoint implementation, the source (host) and sink (device) agree on a sampling rate (e.g., 48 kHz). Every one millisecond, the USB host sends the SOF signal followed by the agreed number of samples (48).

The device application maintains a pool of memory buffers to store the received audio data. It forwards the data to the codec for playback on the audio device. The application also maintains the average number of samples received over a defined period. It compares the average number of samples with a set lower and upper watermark level. When the average number of samples received extends beyond the watermark range, the application tunes the clock to the codec by slightly increasing or decreasing the codec master clock. The tuning of the codec master clock prevents buffer underrun and overrun conditions, thereby reducing the audible artifacts in the audio stream.

For example, the PIC32MX470F512L device provides a tunable reference clock output with the USB PLL clock as its source. The reference clock can be tuned on-the-fly in steps. The tuning range, typically between ±0.2%, is such that no audible artifacts are introduced.

For a data stream with a sample frequency (fs) of 48 kHz, the reference clock output would typically be 256 fs, where 256 is the sampling frequency multiplier required by the codec module.

The output reference clock frequency = 256 * 48 = 12,288,000 Hz.

A swing of ±0.2% for this sampling frequency requires the reference clock frequency to change between 12,263,424 Hz and 12,312,576 Hz.

The resulting sampling frequency would be in the range of 47.88 kHz to 48.12 kHz. This capability of the reference clock prevents buffer underrun and overrun situations, while maintaining an acceptable codec sample rate swing range of 0.2% and achieving high-quality audio.

An example of an adaptive source is a CD player that contains a fully adaptive sample rate converter (SRC) so that the output sample frequency no longer needs to be 44.1 kHz, but can be anything within the operating range of the SRC. Adaptive sinks include devices such as high-end digital headphones, headsets, etc.

3.4 Audio Function Topology

The topology of an audio function consists of building blocks that represent the audio function and the interconnection between the various building blocks in the audio signal flow. These building blocks provide the means to manipulate the properties (gain, volume, equalization) of the audio function. The audio function topology uses two types of building blocks: *Units* and *Terminals.*

In the following discussion, an audio channel cluster is a grouping of logical audio channels that share the same characteristics, such as sampling frequency, bit resolution, etc.,

3.4.1 Units

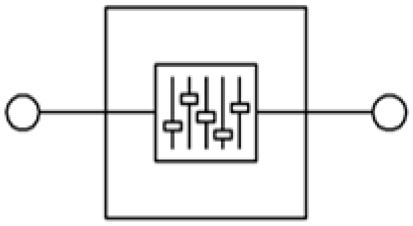
Units provide basic building blocks to fully describe most audio functions. Audio functions are built by connecting together several Units. A unit has many input pins and a single output pin, where each pin represents an audio channel cluster. Units are wired together by connecting their I/O pins according to the required topology. The input pins of a unit are numbered starting from 'one' up to the total number of input pins in the unit. The output pin number is always 'one'. Every unit in the audio function is fully described by its associated USB audio unit descriptor (UD).

The USB Audio v1.0 Specification describes five types of units. These are mixer unit, selector unit, feature unit, processing unit, and extension unit. The feature and the mixer unit types used in the USB headset application are described further. A detailed description of other units is available in the USB Audio v1.0 Specification.

Feature Unit: The feature unit allows manipulation of the incoming audio stream for the following parameters: Volume, Mute, Tone Control (Bass, Mid, Treble), Graphic Equalizer, Automatic Gain Control, Delay, Bass Boost, and Loudness.

The feature unit descriptor reports which controls are present for every channel in the feature unit, including the *master* channel. The feature unit supports multi-channel processing, allowing separate manipulation of each logical channel. The logical channels are numbered from '1' to the total number of channels. The *master* channel (Channel 0) is always virtually present. The following figure represents the feature unit Icon.

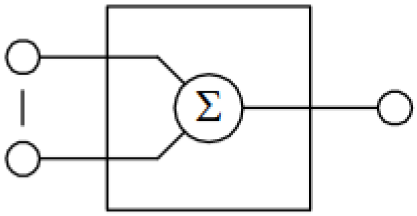
Figure 3-5. Feature Unit Icon



Mixer Unit: The mixer unit transforms a number of logical input channels into a number of logical output channels. The input channels are grouped into one or more audio channel clusters. Each cluster enters the mixer unit through an input pin. The logical output channels are grouped into one audio channel cluster and leave the mixer unit through a single output pin.

The mixer unit descriptor contains information about the number of input pins and the number of logical channels contained in the output pin. The following figure represents a mixer unit icon.

Figure 3-6. Mixer Unit Icon



3.4.2 Terminals

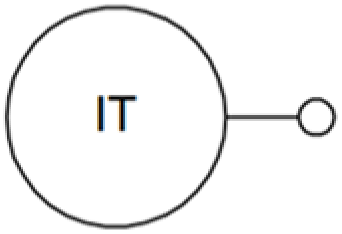
Terminals represent the starting or ending point of an audio stream.

A USB endpoint is a typical example of an input or output terminal. It either provides data streams to the audio function (for example, the media player on a PC-based USB host), or consumes data streams coming from the audio function (for example, USB headphone connected to a PC-based USB host)

Input Terminal (IT): An input terminal is an entity representing the starting point for an audio stream inside the audio function. It serves as a receptacle for audio information flowing into the audio function. Its function is to represent a source of incoming audio data after this data has been extracted from the original audio stream into the separate logical channels that are embedded in this stream (the decoding process). The logical channels are grouped into an audio channel cluster and leave the input terminal through a single output pin.

Typically, the audio stream enters the audio function by means of a USB OUT endpoint. There is a one-to-one relationship between that endpoint and its associated input terminal. The audio class-specific endpoint descriptor contains a field that holds a direct reference to this input terminal. The host needs to use both the endpoint descriptors and the input terminal descriptor to get a full understanding of the characteristics and capabilities of the input terminal. Stream-related parameters are stored in the endpoint descriptors. Control-related parameters are stored in the terminal descriptor. The following figure represents an input terminal icon.

Figure 3-7. Input Terminal Icon

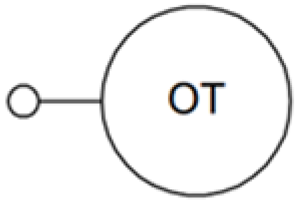


Output Terminal (OT): An output terminal is an entity that represents the ending point of an audio stream inside the audio function. It serves as an outlet for audio information, flowing out of the audio function. Its function is to represent a sink of outgoing audio data before this data is packed from the original separate logical channels into the outgoing audio stream (the encoding process). The audio channel cluster enters the output terminal through a single input pin.

Typically, the audio stream exits the audio function through a USB IN endpoint. There is a one-to-one relationship between that endpoint and its associated output terminal.

The class-specific endpoint descriptor contains a field that holds a direct reference to this output terminal. The host must use both, the endpoint descriptors and the output terminal descriptor to fully understand the characteristics and capabilities of the output terminal. Stream-related parameters are stored in the audio class specific endpoint descriptors. The following figure represents an output terminal icon.

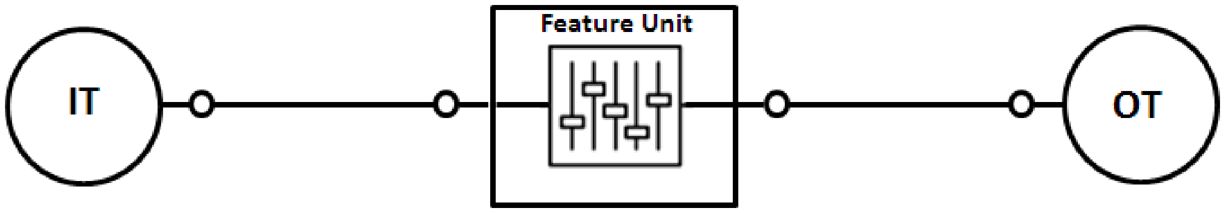
Figure 3-8. Output Terminal Icon



Example:

The basic topology of a USB headphone connected to a PC-based USB host is represented in the following figure.

Figure 3-9. USB Headphone Topology



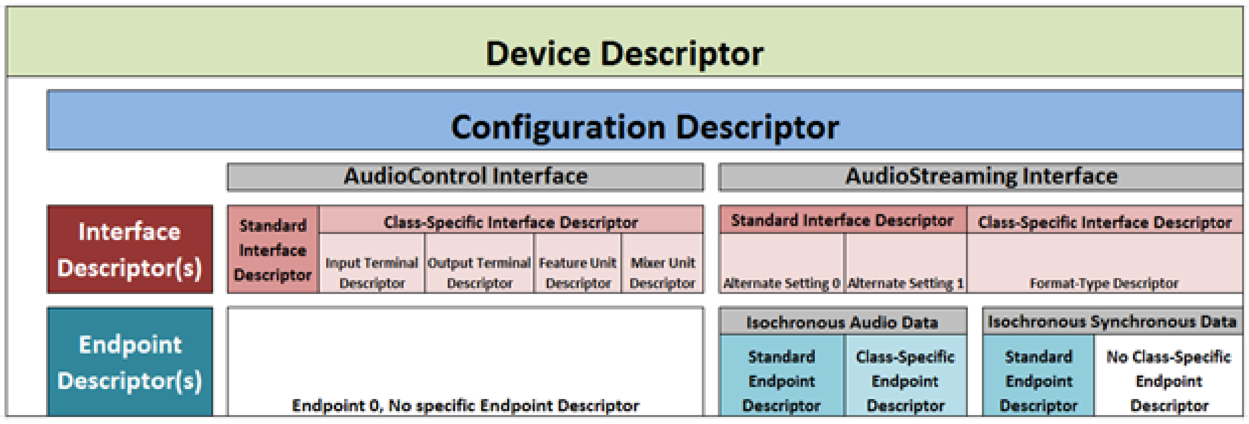
In the previous diagram, the input terminal (IT) represents the audio streaming interface that is used to stream audio from the host to the headphone device. The output pin of the input terminal is connected to the input pin of the feature unit. The output pin of the feature unit is connected to the input pin of the output terminal (OT), which represents the physical headphone unit.

3.5 USB Audio Descriptors

As with standard USB devices, USB audio devices also report their attributes using descriptors.

For audio devices, in addition to the standard descriptors, the USB audio class defines class-specific interface and endpoint descriptors for audio control and stream paths. The following figure shows the descriptors needed for implementation of the USB audio device.

Figure 3-10. USB Audio Descriptors


Device Descriptor
Configuration Descriptor
AudioControl Interface
AudioStreaming Interface
Interface Descriptor(s)
Standard Interface Descriptor
Class-Specific Interface Descriptor
Input Terminal Descriptor
Output Terminal Descriptor
Feature Unit Descriptor
Mixer Unit Descriptor
Standard Interface Descriptor
Class-Specific Interface Descriptor
Alternate Setting 0
Alternate Setting 1
Format-Type Descriptor
Endpoint Descriptor(s)
Endpoint 0, No specific Endpoint Descriptor
Isochronous Audio Data
Standard Endpoint Descriptor
Class-Specific Endpoint Descriptor
Isochronous Synchronous Data
Standard Endpoint Descriptor
No Class-Specific Endpoint Descriptor

Note: The descriptors mentioned in the figure above are covered in detail in the Descriptors section with respect to the USB headset application discussed in this document.

3.6 USB Audio Requests

3.6.1 Standard Requests

The USB audio device class supports the standard requests described in the USB Specification. The audio device class places no specific requirements on the values for the standard requests.

3.6.2 Class-specific Requests

The USB audio device class supports additional class-specific requests to set and get audio related controls. These controls fall into two main groups:

- Controls to manipulate audio functions such as volume, tone, selector position, etc.
- Controls that influence data transfer over an isochronous endpoint, such as the current sampling frequency

Audio Control Requests: In audio control requests, control of an audio function is performed through the manipulation of the attributes of individual controls that are embedded in the entities of the audio function. For example, feature unit.

Audio Streaming Requests: In audio streaming requests, control of the class-specific behavior of an audio streaming interface is performed through manipulation of either interface controls or endpoint controls.

Control Attributes: A typical Set/Get request to an audio control has the following attributes:

- Current setting attribute (SET_CUR)
- Current setting attribute (GET_CUR)
- Minimum setting attribute (SET_MIN)
- Minimum setting attribute (GET_MIN)
- Maximum setting attribute (SET_MAX)
- Maximum setting attribute (GET_MAX)
- Resolution attribute (SET_RES)
- Resolution attribute (GET_RES)

An additional Set/Get request to an entity (terminal, unit, and endpoint) has the following attributes:

- Memory space attribute (SET_MEM)
- Memory space attribute (GET_MEM)

3.7 USB Audio 1.0 Class Features

USB Audio 1.0 Class features include:

- Compliant to USB Full-Speed mode only, allowing a maximum transfer rate of 12 Mbps
- Allows a maximum of 24-bit, 96 kHz audio
- Allows transfer of an audio data frame every one millisecond:
  - The maximum frame size is 1023 bytes. This allows for a 96 kHz sampled and 24-bit wide stereo audio stream. The audio data streams at 576 bytes/millisecond rates.

– A higher sample rate (e.g., 176 kHz) requires 1056 bytes/millisecond, which is in excess of the maximum frame size of 1023 bytes. Therefore, the 24-bit, 176 kHz sampled digital audio option is not supported by this specification.

3.8 Audio Controls Through the Human Interface Devices (HID)

While using USB audio systems (i.e., USB headset), the audio control buttons (Mute, Volume, Next Track, Previous Track, etc.,) are typically present in the audio player application running on the host PC. These controls associate and control the audio functions on the device through building blocks like feature unit. In certain cases, the audio device application may have additional physical controls, such as volume knobs, mute buttons or navigation switches, located on the front panel of the device. These local physical audio controls are represented by a Human Interface Device (HID) class interface on the USB device. This HID class interface coexists with the audio class interface.

To create a binding between the audio function feature unit that deals with the master physical audio controls and the local physical audio controls, the feature unit descriptor can be supplemented by a special associated interface descriptor. This associated interface descriptor holds the link to the HID interface.

The HID class extends the USB specification to provide a standard way of handling devices manipulated by humans. This includes common computer devices, such as keyboard, mouse and joystick, as well as electronic device controllers (for example, VCR remote) and generic controls (for example, knobs, switches).

An HID class device communicates with the HID class driver using an interrupt endpoint (IN).

The device transfers (through the IN) asynchronous data to the host and receives low latency data from the host (through the OUT).

The following table shows the USB HID Usage Name, the Usage ID, and the Usage Page values that the HID report descriptor must use to advertise support for commonly used physical audio controls. Refer to *"HID Usage Tables v1.12"*, which available at www.usb.org/developers/hidpage for more details on these and other HID usage IDs.

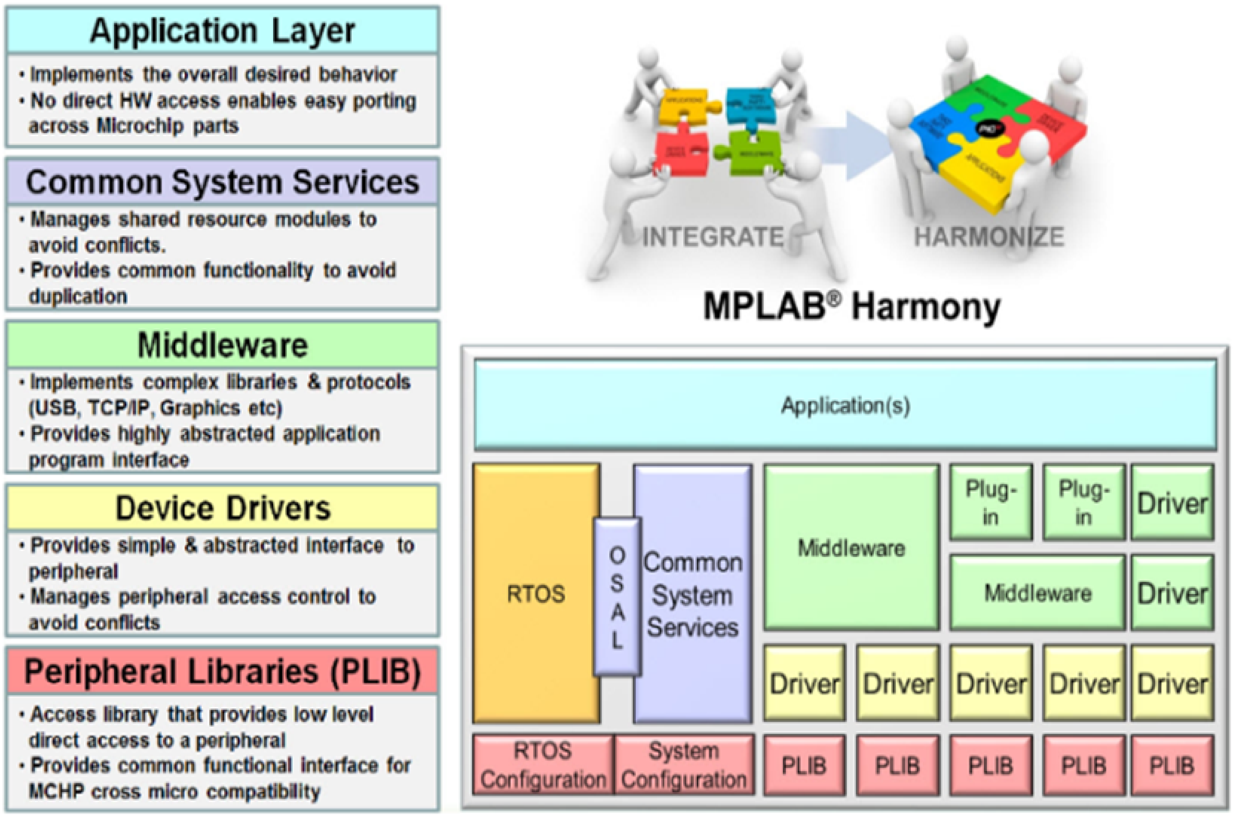
Table 3-2. HID Usage ID and Usage Page for Audio Controls

Usage Name	Usage Page	Usage ID
Volume Up	Consumer (0x0C)	0xE9
Volume Down	Consumer (0x0C)	0x0EA
Phone Mute	Telephony (0x0B)	0x2F
Play/Pause	Consumer (0x0C)	0xCD
Scan Next Track	Consumer (0x0C)	0xB5
Scan Previous Track	Consumer (0x0C)	0xB6
Stop	Consumer (0x0C)	0xB7
Fast Forward	Consumer (0x0C)	0xB3
Rewind	Consumer (0x0C)	0xB4

4. MPLAB Harmony Overview

MPLAB® Harmony is a flexible, abstracted, fully integrated firmware development platform for PIC32 microcontrollers. It takes key elements of modular and object-oriented design, adds the flexibility to use a Real-Time Operating System (RTOS) or work without one, and provides a framework of software modules that are easy to use, configurable to your specific needs, and that work together in complete harmony. The following figure represents the MPLAB Harmony Integrated Software Framework.

Figure 4-1. MPLAB Harmony Software Framework



Application Layer
- Implements the overall desired behavior
- No direct HW access enables easy porting across Microchip parts

Common System Services
- Manages shared resource modules to avoid conflicts.
- Provides common functionality to avoid duplication

Middleware
- Implements complex libraries & protocols (USB, TCP/IP, Graphics etc)
- Provides highly abstracted application program interface

Device Drivers
- Provides simple & abstracted interface to peripheral
- Manages peripheral access control to avoid conflicts

Peripheral Libraries (PLIB)
- Access library that provides low level direct access to a peripheral
- Provides common functional interface for MCHP cross micro compatibility

MPLAB Harmony includes a set of peripheral libraries, drivers, system services, middleware libraries, and tools (MPLAB Harmony Configurator (MHC) and MPLAB Harmony Graphics Composer (MHGC)), which are designed such that the code development format allows maximum reuse and rapid development.

The MHC is a user interface tool plugged into the MPLAB X IDE. MHC makes it easier to use the MPLAB Harmony framework. This tool takes care of the overhead and overall structure of MPLAB Harmony-based projects, allowing you to focus on generating the code for your application. Based on the UI selections of the MHC.

- Adds the MPLAB Harmony source files needed for your project
- Generates some MPLAB Harmony framework source files (based on your MHC selections)
- Generates starter MPLAB Harmony application source files:
  - Creates a starter state machine for your application
  - Adds minimum functions required for a MPLAB Harmony project
  - `#include`s all source files needed
- Generates starter MPLAB Harmony system source files:

  - Initializes the PIC32 (based on your MHC selections for core, clock, and peripherals)
  - Initializes MPLAB Harmony drivers (based on your MHC selections)
  - Creates stub functions for interrupt service routines
- Saves your current MHC selections so they can be imported into another project

5. Building a USB Audio Device Using MPLAB Harmony

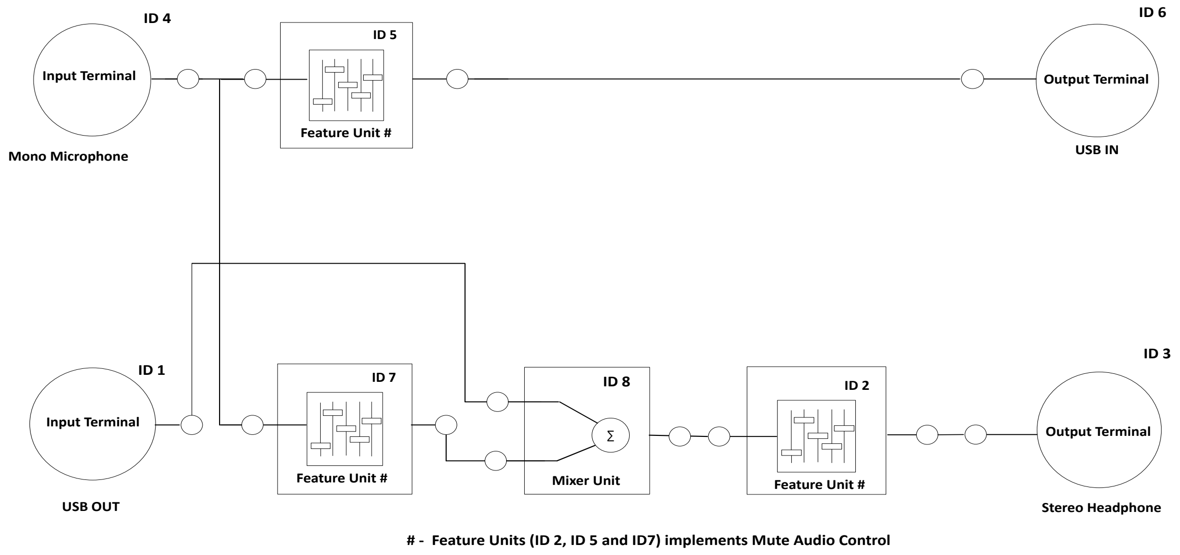
5.1 USB Headset Application

A USB headset device is a combination of a USB microphone and a USB headphone device. In addition, a USB headset also contains a signal path from the microphone input to the headphone output so that the sound picked up by the microphone is also audible through the headset's headphone. The following sections describe the topology and descriptor definitions of a USB headset application.

5.1.1 Topology

The following figure represents the topology of the USB headset application discussed in this application note.

Figure 5-1. USB Headset Application Topology



Input Terminal (ID 4) represents the mono microphone. The output pin of the Input Terminal (ID 4) connects to the input pin of the Feature Unit (ID 5). The output pin of the Feature Unit (ID 5) connects to the input pin of the Output Terminal (ID 6), which represents the audio streaming interface used to stream the microphone data to the host.

Input Terminal (ID 1) represents headphone data streamed from the host to the headset device. The output pin of the Input Terminal (ID 1) connects to the input pin 1 of the Mixer Unit (ID 8). The output pin of the Mixer Unit (ID 8) connects to the input pin of the Feature Unit (ID 2). The output pin of the Feature Unit (ID 2) connects to the input pin of the Output Terminal (ID 3), which represents the physical headphone.

Input Terminal (ID 4) also connects to the input pin of the Feature Unit (ID 7). The output pin of the Feature Unit (ID 7) connects to the input pin 2 of the Mixer Unit (ID 8). This connection creates the side-tone mixing path between the microphone and headphone.

The USB headset application discussed in this application note uses the *bTerminalID* and *bUnitID* fields of the respective descriptors, as shown in the previous figure.

5.1.2 Descriptors

Device Descriptor

The Table below provides details of the device descriptor for the USB headset application.

The *bDeviceClass*, *bDeviceSubClass*, and *bDeviceProtocol* fields are set to 0x00 as these will be defined in the interface descriptor.

Table 5-1. Device Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x12	Size of device descriptor in bytes
1	*bDescriptorType*	1	0x01	DEVICE descriptor
2	*bcdUSB*	2	0x0200	Supports USB 2.0 Specification
4	*bDeviceClass*	1	0x00	Class is specified in the interface descriptor
5	*bDeviceSubClass*	1	0x00	Sub-Class is specified in the interface descriptor
6	*bDeviceProtocol*	1	0x00	Protocol is specified in the interface descriptor
7	*bMaxPacketSize0*	1	0x40	Max packet size for Endpoint 0 is 64 bytes
8	*idVendor*	2	0x04D8	Vendor ID (example Microchip Vendor ID = 04D8h)
10	*idProduct*	2	0x00FF	Product ID
12	*bcdDevice*	2	0x0100	Device version number (example 01.00)
14	*iManufacturer*	1	0x01	Index of Manufacturer string in string descriptors
15	*iProduct*	1	0x02	Index of Product string in string descriptors
16	*iSerialNumber*	1	0x00	Index of device serial number in string descriptors
17	*bNumConfigurations*	1	0x01	Number of configurations is set to 1

Configuration Descriptor

The following table provides details of the configuration descriptor for the USB headset application.

When the device receives a get configuration descriptor request, it will return the configuration descriptor, interface descriptor and endpoint descriptors to the USB host. During enumeration, the USB host will issue a set configuration request to the device, thereby asking the device to set this configuration as active.

Table 5-2. Configuration Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x09	Size of device descriptor in bytes
1	*bDescriptorType*	1	0x02	CONFIGURATION descriptor
2	*wTotalLength*	2	0x00EA	Total length of data returned for this configuration is 232 bytes
4	*bNumInterfaces*	1	0x03	Number of interfaces in this configuration is 0x03

Offset	Field	Size	Value	Description
5	*bConfigurationValue*	1	0x01	Value to use as an argument to the SetConfiguration() request to select this configuration
6	*iConfiguration*	1	0x00	Index of string descriptor describing this configuration
7	*bmAttributes*	1	0xC0	Self-powered
8	*MaxPower*	2	0x32	This device will consume maximum of 100mA (2x MaxPower)

bNumInterfaces is set to a value of three indicating one control interface and two Audio streaming interfaces (for Headphone and Microphone functions each).

wTotalLength represents the total data length of this configuration; this includes the length of the standard and class specific interface and endpoint descriptors mentioned in the following table.

Table 5-3. All Descriptor Length

Descriptor Name	Length
Configuration Descriptor	0x09
Standard AudioControl Interface Descriptor	0x09
Class Specific AudioControl Interface Descriptor	0x0A
Headphone - Input Terminal Descriptor	0x0C
Microphone - Input Terminal Descriptor	0x0C
Feature Unit Descriptor (ID 2)	0x0D
Microphone - Feature Unit Descriptor	0x0B
Microphone (Sidetone) - Feature Unit Descriptor	0x0B
Mixer Unit Descriptor	0x0D
Headphone - Output Terminal Descriptor	0x09
Microphone - Output Terminal Descriptor	0x09
Headphone - Zero Bandwidth Standard AS Interface Descriptor (Alternate Setting 0)	0x09
Headphone – Operational Standard AS Interface Descriptor (Alternate Setting 1)	0x09
Headphone - Class Specific AS General Interface Descriptor	0x07
Headphone - Type 1 Format Type Descriptor	0x11
Headphone - Standard AS Audio Data Endpoint Descriptor	0x09
Headphone - Class Specific AS Audio Data Endpoint Descriptor	0x07
Microphone - Zero Bandwidth Standard AS Interface Descriptor (Alternate Setting 0)	0x09
Microphone - Operational Standard AS Interface Descriptor (Alternate Setting 1)	0x09
Microphone - Class Specific AS General Interface Descriptor	0x07
Microphone - Type 1 Format Type Descriptor	0x11

Descriptor Name	Length
Microphone - Standard AS Audio Data Endpoint Descriptor	0x09
Microphone - Class Specific AS Audio Data Endpoint Descriptor	0x07

Standard Audio Control Interface Descriptor

The following table provides details of the standard audio control interface descriptor for USB headset application.

- *bNumEndpoints* is set to 0x00, which indicates that there are no additional control endpoints other than the control Endpoint 0
- *bNumInterfaces* is set to a value of three indicating one control interface and two Audio streaming interfaces (for Headphone and Microphone functions each).
- *bInterfaceClass* is set to 0x01 corresponding to Audio class
- *bInterfaceSubClass* is set to 0x01 corresponding to Audio control subclass
- *bInterfaceProtocol* is set to 0x000 corresponding to the undefined protocol for audio interface protocol code. These values are specified by USB Audio 1.0. Specification and directs the Host to associate the audio client driver.

Table 5-4. Standard Audio Control Interface Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x09	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x04	INTERFACE descriptor
2	*bInterfaceNumber*	1	0x00	Number of this interface
3	*bAlternateSetting*	1	0x00	Value used to identify the alternate setting for this interface
4	*bNumEndpoints*	1	0x00	Number of endpoints used by this interface (excluding endpoint 0)
5	*bInterfaceClass*	1	0x01	AUDIO - Audio Interface Class code
6	*bInterfaceSubClass*	1	0x01	AUDIOCONTROL - Audio Interface Subclass code
7	*bInterfaceProtocol*	1	0x00	Not used, Set to 0
8	*iInterface*	1	0x00	No string descriptors for this interface

Class-Specific AudioControl Interface Descriptor

The following table provides details of the class-specific audio control interface descriptor for the USB headset application.

Table 5-5. Class Specific AudioControl Interface Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x0A	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x01	HEADER descriptor subtype

Offset	Field	Size	Value	Description
3	*bcdADC*	2	0x0001	Audio Device Class Specification Release Number in Binary-Coded Decimal
5	*wTotalLength*	2	0x0064	Total number of bytes returned for this descriptor.
7	*bInCollection*	1	0x02	Number of Audio Streaming interfaces in this Audio Interface Collection
8	*baInterfaceNr(1)*	1	0x01	Interface number of the first Audio Streaming interface in the Collection
9	*baInterfaceNr(2)*	1	0x02	Interface number of the Second Audio Streaming interface in the Collection

wTotalLength is set to 0x0064. This indicates the combined length of this descriptor header and all its unit and terminal descriptors mentioned in the following table.

Table 5-6. Class-Specific Descriptor Length

Descriptor Name	Length
Class Specific AudioControl Interface Descriptor	0x0A
Headphone - Input Terminal Descriptor	0x0C
Microphone - Input Terminal Descriptor	0x0C
Feature Unit Descriptor (ID 2)	0x0D
Microphone - Feature Unit Descriptor	0x0B
Microphone (Sidetone) -Feature Unit Descriptor	0x0B
Mixer Unit Descriptor	0x0D
Headphone - Output Terminal Descriptor	0x09
Microphone - Output Terminal Descriptor	0x09

bInCollection is set to 0x02. This indicates one streaming interface for the headphone function and one streaming interface for the microphone function.

baInterfaceNr(1) is set to 0x01 representing the interface number of the headphone function.

baInterfaceNr(2) is set to 0x01 representing the interface number of the microphone function.

Headphone - Input Terminal Descriptor

The following table provides details of the input terminal descriptor corresponding to the headphone function for the USB headset application.

bTerminalID is set to 0x01 (refer to the topology in Figure 5-1). The Input Terminal for Headphone function in this application would be referred by this ID.

wTerminalType is set to 0x0101. The terminal type is set to USB streaming, as the terminal is the input point for the USB data streaming out from the PC.

bAssocTerminal is set to 0x00 as this input terminal is not connected directly to an output terminal.

bNrChannels is set to 0x02 as the terminal's output audio channel cluster has two logical channels for the stereo data output (i.e. left channel and right channel).

wChannelConfig is set to 0x0003 corresponding to the Left front (L) and Right Front (R) position of headphone channels fields indicating the spatial location of the terminal's output audio channel cluster.

Table 5-7. Headphone - Input Terminal Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x0C	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x02	INPUT_TERMINAL descriptor subtype
3	*bTerminalID*	1	0x01	Unique ID of this terminal set to 1
4	*wTerminalType*	2	0x0101	Terminal type is USB streaming
6	*bAssocTerminal*	1	0x00	Output Terminal to which this Input Terminal is associated
7	*bNrChannels*	1	0x02	Number of logical output channels in the Terminal's output audio channel cluster
8	*wChannelConfig*	2	0x0003	Bitmap location of the logical channels
10	*iChannelNames*	1	0x00	No non-predefined logical channels, the index is set to 0
11	*iTerminal*	1	0x00	No string descriptors for this interface

Note: The spatial locations present in the audio cluster (bit fields are part of field ***wChannelConfig***) are represented in the following table.

D0	Left Front (L)
D1	Right Front (R)
D2	Center Front (C)
D3	Low Frequency Enhancement (LFE)
D4	Left Surround (LS)
D5	Right Surround (RS)
D6	Left of Center (LC)
D7	Right of Center (RC)
D8	Surround (S)
D9	Side Left (SL)
D10	Side Right (SR)
D11	Top (T)
D15-D12	Reserved

Microphone - Input Terminal Descriptor

The following table provides details of the input terminal descriptor for microphone function for USB headset application.

bTerminalID is 0x04 (refer to the topology in Figure 5-1). The input terminal for microphone function in this application would be referred by this ID.

wTerminalType is set to 0x0201. The terminal type is set to microphone as the terminal is the input point for microphone data.

bAssocTerminal is 0x0 as this input terminal is not connected directly to an output terminal.

bNrChannels is 0x01 as the terminal's output audio channel cluster has one logical channels for the mono data output.

wChannelConfig is 0x0004 corresponding to the Center Front (C) field indicating the spatial location of the terminal's output audio channel cluster.

Table 5-8. Microphone - Input Terminal Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x0C	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x02	INPUT_TERMINAL descriptor subtype
3	*bTerminalID*	1	0x04	Unique ID of this terminal set to 4
4	*wTerminalType*	2	0x0201	Terminal type is Microphone device
6	*bAssocTerminal*	1	0x00	No Output Terminal to which this Input Terminal is associated
7	*bNrChannels*	1	0x01	Number of logical output channels in the Terminal's output audio channel cluster
8	*wChannelConfig*	2	0x0004	Bitmap location of the logical channels
10	*iChannelNames*	1	0x00	No non-predefined logical channels, the index is set to 0
11	*iTerminal*	1	0x00	No string descriptors for this Input Terminal

Mixer - Feature Unit Descriptor

The following table provides details of the feature unit descriptor whose input is the mixer unit for the USB headset application.

bUnitID is 0x02 (refer to the topology in Figure 5-1). The feature unit (with mixer input terminal) in this application would be referred by this ID.

bSourceID is set to 0x08. This is the ID of the mixer unit connected to this feature unit.

bControlSize is 0x02 indicates that the feature unit can support up to 16 (2 bytes – 16 bits) controls for each channel (including master channel).

bmaControls(0) is set to 0x001 indicating that mute audio control is enabled on Channel 0 (Master).

bmaControls(1) is 0x000 indicating that no audio control is enabled on Channel 1 (Left Channel).

bmaControls(2) is 0x000 indicating that no audio control is enabled on Channel 2 (Right Channel).

Table 5-9. Mixer - Feature Unit Descriptor (ID 2)

Offset	Field	Size	Value	Description
0	*bLength*	1	0x0B	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x06	FEATURE_UNIT descriptor subtype
3	*bUnitID*	1	0x02	Unique ID of this Feature Unit set to 2
4	*bSourceID*	1	0x08	This Feature Unit is connected to Input Terminal with ID 8
5	*bControlSize*	1	0x02	Each element of control channels bitmap array is 2 bytes.
6	*bmaControls(0)*	2	0x0001	Mute Audio control enabled in channel 0 (master channel)
8	*bmaControls(1)*	2	0x0000	No audio control enabled in channel 1
10	*bmaControls(2)*	2	0x0000	No audio control enabled in channel 2
12	*iFeature*	1	0x00	No string descriptors for this Feature Unit

Note: The bitmap for audio control channels field *bmaControls(n)* has the bit fields shown in the following table.

Table 5-10. Audio Control Channels Bits

Bit Field	Description
D0	Mute
D1	Volume
D2	Bass
D3	Mid
D4	Treble
D5	Graphic Equalizer
D6	Automatic Gain
D7	Delay
D8	Bass Boost
D9	Loudness

Microphone - Feature Unit Descriptor

The following table details of the feature unit descriptor whose input is the microphone implemented in the audio headset application.

bUnitID is 0x05 (refer to the topology in Figure 5-1). The feature unit (with microphone input terminal) in this application would be referred by this ID.

bSourceID is set to 0x04. This is the ID of the microphone input terminal connected to this feature unit.

bControlSize is 0x02 indicates that the feature unit can support up to 16 (2 bytes – 16 bits) controls for each channel (including master channel).

bmaControls(0) is set to 0x001 indicating that mute audio control is enabled on Channel 0 (Master).

bmaControls(1) is 0x000 indicating that no audio control is enabled on Channel 1.

Table 5-11. Microphone - Feature Unit Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x0B	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x06	FEATURE_UNIT descriptor subtype
3	*bUnitID*	1	0x05	Unique ID of this Feature Unit set to 5
4	*bSourceID*	1	0x04	This Feature Unit is connected to Input Terminal with ID 4
5	*bControlSize*	1	0x02	Each element of control channels bitmap array is 2 bytes.
6	*bmaControls(0)*	2	0x0001	Mute Audio control enabled in master channel
8	*bmaControls(1)*	2	0x0000	No audio control enabled in channel 1
10	*iFeature*	1	0x00	No string descriptors for this Feature Unit

Microphone (Sidetone) - Feature Unit Descriptor

The following table provides details of the feature unit descriptor whose input is the microphone input terminal (Sidetone) for the USB headset application.

bUnitID is 0x07 (refer to the topology in Figure 5-1). The feature unit (with microphone input terminal for sidetone mixing) in this application would be referred to by this ID.

bSourceID is set to 0x04. This is the ID of the microphone input terminal (for sidetone mixing) connected to this feature unit.

bControlSize is 0x02 indicates that the feature unit can support up to 16 (2 bytes – 16 bits) controls for each channel (including the master channel).

bmaControls(0) is set to 0x001 indicating that mute audio control is enabled on Channel 0 (Master).

bmaControls(1) is 0x000 indicating that no audio control is enabled on Channel 1.

Table 5-12. Microphone (Sidetone) - Feature Unit Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x0B	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x06	FEATURE_UNIT descriptor subtype
3	*bUnitID*	1	0x07	Unique ID of this Feature Unit set to 7
4	*bSourceID*	1	0x04	This Feature Unit is connected to Input Terminal with ID 4
5	*bControlSize*	1	0x02	Each element of control channels bitmap array is 2 bytes.
6	*bmaControls(0)*	2	0x0001	Mute Audio control enabled in master channel

Offset	Field	Size	Value	Description
8	*bmaControls(1)*	2	0x0000	No audio control enabled in channel 1.
10	*iFeature*	1	0x00	No string descriptors for this Feature Unit

Mixer Unit Descriptor

The following table provides details of the mixer unit descriptor for USB headset application.

bUnitID is 0x08 (refer to the topology in Figure 5-1). The mixer unit in this application would be referred by this ID.

bNrInPins is 0x02 indicating there are two input pins for this mixer unit.

bSourceID(1) is 0x01 indicating that the Input Terminal with ID 1 (corresponding to the headphone function) is connected to pin 1 of this mixer unit.

bSourceID(2) is 0x07 indicating that the feature unit with ID 7 (corresponding to the microphone with sidetone) is connected to pin 2 of this mixer unit.

bNrChannels is 0x02 as the terminal's output audio channel cluster has two logical channels for the stereo data output (i.e. left channel and right channel).

wChannelConfig is 0x0003 corresponding to the Left Front (L) and Right Front(R) fields indicating the spatial location of the terminal's output audio channel cluster.

bmControls is 0x00 because [(n x m) MOD 8] is 0x0. Where 'n' is the number of input channels (2) and 'm' is the number of output channels (2)

Table 5-13. Mixer Unit Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x0D	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x04	MIXER_UNIT descriptor subtype
3	*bUnitID*	1	0x08	Unique ID of this Mixer Unit set to 8
4	*bNrInPins*	1	0x02	Two input pins for this Mixer Unit
5	*baSourceID(1)*	1	0x01	ID 1 is connected to pin 1 of this unit
6	*baSourceID(2)*	1	0x07	ID 7 is connected to pin 2 of this unit
7	*bNrChannels*	1	0x02	Number of logical output channels in the Terminal's output audio channel cluster
8	*wChannelConfig*	2	0x0003	Bitmap location of the logical channels
10	*iChannelNames*	1	0x00	No non-predefined logical channels, the index is set to 0
11	*bmControls*	1	0x00	No mixing controls are programmable
12	*iMixer*	1	0x00	No string descriptors for this Mixer Unit

Headphone - Output Terminal Descriptor

The following table provides details of the output terminal descriptor corresponding to the headphone function for the USB headset application.

bTerminalID is 0x03 (refer to the topology in Figure 5-1). The output terminal for the headphone function in this application would be referred to by this ID.

wTerminalType is set to 0x0302. The terminal type is set to headphone, as the terminal is the output point for the USB data streaming out from the PC.

bAssocTerminal is 0x00 as this output terminal is not connected directly to an input terminal.

bSource is set to 0x02 indicating the ID of the mixer unit acting as the source for this output terminal.

Table 5-14. Headphone - Output Terminal Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x09	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x03	OUTPUT_TERMINAL descriptor subtype
3	*bTerminalID*	1	0x03	Unique ID of this terminal set to 3
4	*wTerminalType*	2	0x0302	Terminal type is Headphone
6	*bAssocTerminal*	1	0x00	No Input Terminal to which this Output Terminal is associated
7	*bSource*	1	0x02	ID of the Mixer Unit to which this Output Terminal is connected
8	*iTerminal*	1	0x00	No string descriptors for this Output Terminal

Microphone - Output Terminal Descriptor

The following table provides details of the output terminal descriptor corresponding to the headphone function for the USB headset application.

bTerminalID is 0x06 (refer to the topology in Figure 5-1). The output terminal for the microphone function in this application would be referred by this ID.

wTerminalType is set to 0x0101. The terminal type is set to headphone, as the terminal is the output point for the USB data streaming out from the PC.

wTerminalType is set to 0x0101. The terminal type is set to USB streaming, as the terminal is the output point for the USB data streaming in to the PC.

bAssocTerminal is 0x0 as this output terminal is not connected directly to an input terminal.

bSource is set to 0x05 indicating the ID of the feature unit acting as the source for this output terminal.

Table 5-15. Microphone - Output Terminal Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x09	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor

Offset	Field	Size	Value	Description
2	*bDescriptorSubtype*	1	0x03	OUTPUT_TERMINAL descriptor subtype
3	*bTerminalID*	1	0x06	Unique ID of this terminal set to 3
4	*wTerminalType*	2	0x0101	Terminal type is USB Streaming
6	*bAssocTerminal*	1	0x00	No Input Terminal to which this Output Terminal is associated
7	*bSource*	1	0x05	ID of the Feature Unit to which this Output Terminal is connected
8	*iTerminal*	1	0x00	No string descriptors for this Output Terminal

Headphone - Zero Bandwidth Standard AS Interface Descriptor (Alternate Setting 0)
The table below provides details of the standard AS interface descriptor for zero bandwidth (Alternate Setting 0) of the headphone function for the USB headset application.

bInterfaceNumber is set to 0x01. Value identifying this interface in the array of concurrent interfaces under this configuration.

bAlternateSetting is 0x0 indicates a zero-bandwidth setting to relinquish the claimed bandwidth on the bus when the audio device is not in use.

bNumEndpoints is 0x0 as it is a zero-bandwidth interface it does not have an associated endpoint.

Table 5-16. Headphone - Zero Bandwidth Standard AS Interface Descriptor (Alternate Setting 0)

Offset	Field	Size	Value	Description
0	*bLength*	1	0x09	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x04	INTERFACE descriptor
2	*bInterfaceNumber*	1	0x01	Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration
3	*bAlternateSetting*	1	0x00	Alternate setting number is 0 for this interface
4	*bNumEndpoints*	1	0x00	Zero bandwidth interface has no associated endpoints
5	*bInterfaceClass*	1	0x01	Audio Interface Class
6	*bInterfaceSubClass*	1	0x02	Audio streaming interface subclass
7	*bInterfaceProtocol*	1	0x00	Undefined protocol
8	*iInterface*	1	0x00	No string descriptors for this Interface

Headphone - Operational Standard AS Interface Descriptor (Alternate Setting 1)
The following table provides details of the standard AS interface descriptor for operational bandwidth (Alternate Setting 1) of the headphone function for the USB headset application.

bInterfaceNumber is set to 0x01. Value identifying this interface in the array of concurrent interfaces under this configuration.

bAlternateSetting is 0x01 indicates operational setting of this interface. It contains the standard and class-specific interface and endpoint descriptors.

bNumEndpoints is 0x01indicates an endpoint is associated with the operational setting 1 standard interface.

Table 5-17. Headphone - Operational Standard AS Interface Descriptor (Alternate Setting 1)

Offset	Field	Size	Value	Description
0	*bLength*	1	0x09	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x04	INTERFACE descriptor
2	*bInterfaceNumber*	1	0x01	Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration
3	*bAlternateSetting*	1	0x01	Alternate setting number is 1 for this interface
4	*bNumEndpoints*	1	0x01	One endpoint is associated with this interface
5	*bInterfaceClass*	1	0x01	Audio Interface Class
6	*bInterfaceSubClass*	1	0x02	Audio streaming interface subclass
7	*bInterfaceProtocol*	1	0x00	Undefined protocol
8	*iInterface*	1	0x00	No string descriptors for this Interface

Headphone - Class-specific AS General Interface Descriptor

The following table provides details of the class specific AS general interface descriptor of headphone function for the USB headset application.

bTerminalLink is set to 0x01. This is the terminal ID of the input terminal associated with the headphone audio function.

Table 5-18. Headphone - Class-specific AS General Interface Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x07	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x01	AS_GENERAL descriptor subtype
3	*bTerminalLink*	1	0x01	Terminal ID of the Input or Output Terminal to which this interface is connected
4	*bDelay*	1	0x01	Delay of 1 frame transmission is introduced for inter-channel data communication.
5	*wFormatTag*	2	0x0001	PCM data format to be used while communicating with this interface.

Headphone - Type 1 Format Type Descriptor

The following table provides details of the Type 1 format type of headphone function for the USB headset application.

bFormatType is set to 0x01. This indicates a TYPE_I audio format is implemented where audio is constructed on a sample-by-sample conversion scheme, such as PCM.

bNrChannels is 0x02 indicating 2 audio channels, Left and Right.

bSubFrameSize is 0x02 indicating 2 bytes sample. (i.e. left channel has 2 bytes and right channel has 2 bytes0.

bBitResolution is0x10 indicating all 16-bits of a 2 byte sample is effective.

bSamFreqType is 0x03 indicating three sampling frequencies are supported.

Table 5-19. Headphone - Type 1 Format Type Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x11	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x02	FORMAT_TYPE descriptor subtype
3	*bFormatType*	1	0x01	FORMAT_TYPE_I Audio streaming interface
4	*bNrChannels*	1	0x02	Number of channels in the audio stream is 2
5	*bSubFrameSize*	1	0x02	2 bytes per audio sub-frame (sample)
6	*bBitResolution*	1	0x10	16 bite per sample
7	*bSamFreqType*	1	0x03	Three supported frequencies
8	*tSamFreq[1]*	3	0x003E80	16000 Hz sampling rate
11	*tSamFreq[2]*	3	0x007D00	32000 Hz sampling rate
14	*tSamFreq[3]*	3	0x00BB80	48000 Hz sampling rate

Headphone - Standard AS Audio Data Endpoint Descriptor
The following tables provide details of the standard AS audio data endpoint descriptor of headphone function implemented in the audio headset application.

bEndpointAddress is set to 0x01. This indicates that endpoint 1 is selected as an OUT endpoint for headphone data from the PC.

bmAttributes is 0x09 indicating the endpoint is an isochronous adaptive endpoint.

wMaxPacketSize is 0x00C0 indicating the maximum packet size to support sampling rates 16000, 32000 and 48000 Hz computed as follows:

- Number of samples/ms = Sampling rate in Hz/1000
- Size of each sample = (Number of Channels) * (width of each sample (Bit Depth))
- Therefore Bytes per frame (packet) (wMaxPacketSize) = Number of Samples/ms * size of each sample

Sampling Rate	Bit Depth	Number of Channels	wMaxPacket Size
16 kHz	16 bits (2 bytes	2	64
32 kHz	16 bits (2 bytes	2	128
32 kHz	16 bits (2 bytes	2	192

Table 5-20. Headphone - Standard AS Audio Data Endpoint Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x09	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x05	ENDPOINT descriptor type
2	*bEndpointAddress*	1	0x01	Endpoint 1 configured as out
3	*bmAttributes*	1	0x09	Isochronous Adaptive endpoint
4	*wMaxPacketSize*	2	0x00C0	The maximum packet size to support 48000 Hz sampling frequency
6	*bInterval*	1	0x01	Interval for polling endpoint for data transfers expressed in millisecond
7	*bRefresh*	1	0x00	No refresh.
8	*bSyncAddress*	1	0x00	No synchronization endpoint

Headphone - Class-specific AS Audio Data Endpoint Descriptor

The following table provides details of the class specific AS audio data endpoint descriptor of headphone function for USB headset application.

bmAttributes is set to 0x01. Turns on sampling frequency control. The bit D7 is '0' indicating that the endpoint does not necessarily need USB packets of *wMaxPacketSize*, and that it can handle short packets.

Table 5-21. Headphone - Class-specific AS Audio Data Endpoint Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x07	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x25	CS_ENDPOINT descriptor type
2	*bDescriptorSubType*	1	0x01	EP_GENERAL descriptor subtype
3	*bmAttributes*	1	0x01	Turns on sampling frequency control, no pitch control and no packet padding.
4	*bLockDelayUnits*	1	0x00	Indicates the units used for the wLockDelay field
5	*wLockDelay*	1	0x00	time it takes this endpoint to reliably lock its internal clock recovery circuitry

Microphone - Zero Bandwidth Standard AS Interface Descriptor (Alternate Setting 0)

The following table provides details of the standard AS interface descriptor for zero bandwidth (Alternate Setting 0) of the microphone function for the USB headset application.

bInterfaceNumber is set to 0x02. Value identifying this interface in the array of concurrent interfaces under this configuration.

bAlternateSetting is 0x00 indicates a zero-bandwidth setting to relinquish the claimed bandwidth on the bus when the audio device is not in use.

bNumEndpoints is 0x00 as it is a zero-bandwidth interface it doesn't have an associated endpoint.

Table 5-22. Microphone - Zero Bandwidth Standard AS Interface Descriptor (Alternate Setting 0)

Offset	Field	Size	Value	Description
0	*bLength*	1	0x09	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x04	INTERFACE descriptor
2	*bInterfaceNumber*	1	0x02	Interface number. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration
3	*bAlternateSetting*	1	0x00	Alternate setting number is 0 for this interface.
4	*bNumEndpoints*	1	0x00	Zero bandwidth interface has no associated endpoints
5	*bInterfaceClass*	1	0x01	Audio Interface Class
6	*bInterfaceSubClass*	1	0x02	Audio streaming interface subclass
7	*bInterfaceProtocol*	1	0x00	Undefined protocol
8	*iInterface*	1	0x00	No string descriptors for this Interface

Microphone - Operational Standard AS Interface Descriptor (Alternate Setting 1)
The following table provides details of the standard AS interface descriptor for operational bandwidth (Alternate Setting 1) of the microphone function for the USB headset application.

bInterfaceNumber is set to 0x02. Value identifying this interface in the array of concurrent interfaces under this configuration.

bAlternateSetting is set to 0x01. This indicates the operational setting of this interface. It contains the standard and class-specific interface and endpoint descriptors.

bNumEndpoints is set to 0x01. This indicates that an endpoint is associated with the operational setting 1 Standard interface.

Table 5-23. Microphone - Operational Standard AS Interface Descriptor (Alternate Setting 1)

Offset	Field	Size	Value	Description
0	*bLength*	1	0x09	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x04	INTERFACE descriptor
2	*bInterfaceNumber*	1	0x02	Interface number. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration
3	*bAlternateSetting*	1	0x01	Alternate setting number is 1 for this interface
4	*bNumEndpoints*	1	0x01	One endpoint is associated with this interface
5	*bInterfaceClass*	1	0x01	Audio Interface Class
6	*bInterfaceSubClass*	1	0x02	Audio streaming interface subclass

Offset	Field	Size	Value	Description
7	*bInterfaceProtocol*	1	0x00	Undefined protocol
8	*iInterface*	1	0x00	No string descriptors for this Interface

Microphone - Class-specific AS General Interface Descriptor
The following table provides details of the class specific AS general interface descriptor of the microphone function for USB the headset application.

bTerminalLink is set to 0x06. This is the terminal ID of the output terminal associated with the microphone audio function.

Table 5-24. Microphone - Class-specific AS General Interface Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x07	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x01	AS_GENERAL descriptor subtype
3	*bTerminalLink*	1	0x06	Terminal ID of the Input or Output Terminal to which this interface is connected
4	*bDelay*	1	0x00	Delay of 0 frame transmission is introduced for inter-channel data communication.
5	*wFormatTag*	2	0x0001	PCM data format to be used while communicating with this interface.

Microphone - Type 1 Format Type Descriptor
The following table provides details of the Type 1 format type of microphone function for the USB headset application.

bFormatType is set to 0x01. This indicates a TYPE_I audio format is implemented where audio is constructed on a sample-by-sample conversion scheme, such as PCM.

bNrChannels is 0x01 indicating 1 audio channel. Mono audio.

bSubFrameSize is 0x02 indicating 2 byte sample.

bBitResolution is0x10 indicating all 16-bits of 2 byte sample is effective.

bSamFreqType is 0x03 indicating three sampling frequencies are supported.

Table 5-25. Microphone - Type 1 Format Type Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x11	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x24	Class specific INTERFACE descriptor
2	*bDescriptorSubtype*	1	0x02	FORMAT_TYPE descriptor subtype
3	*bFormatType*	1	0x01	FORMAT_TYPE_I Audio streaming interface

Offset	Field	Size	Value	Description
4	*bNrChannels*	1	0x01	Number of channels in the audio stream is 2
5	*bSubFrameSize*	1	0x02	2 bytes per audio sub-frame (sample)
6	*bBitResolution*	1	0x10	16 bite per sample
7	*bSamFreqType*	1	0x03	Three supported frequencies
8	*tSamFreq[1]*	3	0x003E80	16000 Hz sampling rate
11	*tSamFreq[2]*	3	0x007D00	32000 Hz sampling rate
14	*tSamFreq[3]*	3	0x00BB80	48000 Hz sampling rate

Microphone - Standard AS Audio Data Endpoint Descriptor
The following tables provide details of the standard AS audio data endpoint descriptor of the microphone function for the USB headset application.

bEndpointAddress is set to 0x81. This indicates that endpoint 1 is selected as an IN endpoint for microphone data to the PC.

bmAttributes is 0x0D indicating the endpoint is an isochronous synchronous source endpoint.

wMaxPacketSize is 0x0060 indicating the maximum packet size to support sampling rates 16000, 32000 and 48000 Hz for mono audio, computed as follows:

- Number of samples/ms = Sampling rate in Hz/1000
- Size of each sample = (Number of Channels) * (width of each sample (Bit Depth))
- Therefore, Bytes per frame (packet) *(wMaxPacketSize)* = Number of Samples/ms * size of each sample

Sampling Rate	Bit Depth	Number of Channels	wMaxPacket Size
16 kHz	16 bits (2 bytes	2	64
32 kHz	16 bits (2 bytes	2	128
32 kHz	16 bits (2 bytes	2	192

Table 5-26. Microphone - Standard AS Audio Data Endpoint Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x09	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x05	ENDPOINT descriptor type
2	*bEndpointAddress*	1	0x81	Endpoint 1 configured as out
3	*bmAttributes*	1	0x0D	Isochronous synchronous endpoint
4	*wMaxPacketSize*	2	0x0060	The maximum packet size to support 48000 Hz sampling frequency for mono audio
6	*bInterval*	1	0x01	Interval for polling endpoint for data transfers expressed in millisecond

Offset	Field	Size	Value	Description
7	*bRefresh*	1	0x00	No refresh.
8	*bSyncAddress*	1	0x00	No synchronization endpoint

Microphone - Class-specific AS Audio Data Endpoint Descriptor
The following table provides details of the class specific AS audio data endpoint descriptor of the microphone function for the USB headset application.

bmAttributes is set to 0x01. This turns on sampling frequency control. Bit D7 is '0' indicating that the endpoint does not necessarily need USB packets of *wMaxPacketSize* and can handle short packets.

Note: The description and values for the following descriptors are not covered in this section, as the MPLAB Harmony USB headset application discussed in this document does not implement them.

1. The Isochronous Synchronous Data Endpoint Descriptor used to implement the asynchronous synchronization type. This is because the USB headset application configures the headphone data endpoint as adaptive and microphone data endpoint as synchronous.
2. Associated Interface Descriptor for Mixer-Feature Unit, HID Class Interface Descriptor and its Report Descriptor used to implement the HID-compliant consumer control audio commands.

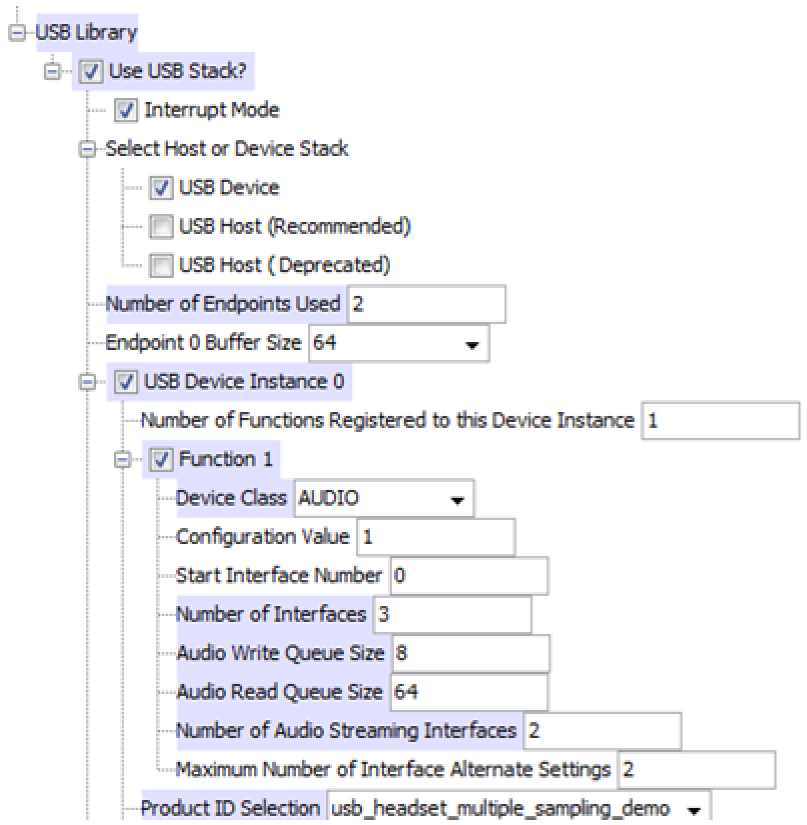
Table 5-27. Microphone - Class-specific AS Audio Data Endpoint Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	0x07	Size of this descriptor in bytes
1	*bDescriptorType*	1	0x25	CS_ENDPOINT descriptor type
2	*bDescriptorSubType*	1	0x01	EP_GENERAL descriptor subtype
3	*bmAttributes*	1	0x01	Turns on sampling frequency control, no pitch control and no packet padding.
4	*bLockDelayUnits*	1	0x00	Indicates the units used for the wLockDelay field
5	*wLockDelay*	1	0x00	Time it takes this endpoint to reliably lock its internal clock recovery circuitry

5.2 Configure the USB Audio 1.0 Library Using the MPLAB Harmony Configurator (MHC)

1. Open MPLAB X IDE and start the MPLAB Harmony Configurator (MHC).
2. Select the MHC **Options** tab, and expand *Harmony Framework Configuration* > *USB Library*.
3. Configure the USB Audio Device Library, as shown in the following figure.

Figure 5-2. MHC - USB Audio Library



- USB Library
  - ☑ Use USB Stack?
    - ☑ Interrupt Mode
    - Select Host or Device Stack
      - ☑ USB Device
      - ☐ USB Host (Recommended)
      - ☐ USB Host (Deprecated)
    - Number of Endpoints Used: 2
    - Endpoint 0 Buffer Size: 64
    - ☑ USB Device Instance 0
      - Number of Functions Registered to this Device Instance: 1
      - ☑ Function 1
        - Device Class: AUDIO
        - Configuration Value: 1
        - Start Interface Number: 0
        - Number of Interfaces: 3
        - Audio Write Queue Size: 8
        - Audio Read Queue Size: 64
        - Number of Audio Streaming Interfaces: 2
        - Maximum Number of Interface Alternate Settings: 2
      - Product ID Selection: usb_headset_multiple_sampling_demo

Note: The following points apply to the USB Audio Library MHC configuration for creating a USB headset application.

- **Number of Endpoints Used** is set to 2. One of the endpoints is the default control endpoint zero, which will be used for the USB device control transfers and the USB audio control transfers. The other endpoint will be used for streaming USB audio data from the host to the device and conversely, from the device to the host.
- **Endpoint 0 Buffer Size** is left to the default value of 64 (maximum buffer size). For full-speed devices, the USB specification allows the endpoint zero buffer to be the size of 8, 16, 32, or 64 bytes.
- **USB Device Instance 0** option is selected and expanded. Since there is only one USB peripheral on the device, there is only one USB instance (selected by default).
- **Number of Functions Registered to this Device Instance** is retained with the default value of 1. This indicates that only one function driver, (i.e., audio device class) is used.
- **Device Class** is set to AUDIO. This is to indicate that the end application would be a USB audio application based on the USB Audio 1.0 Specification.
- **Configuration Value** is set to 1 (default). This represents the configuration to which the function driver (Audio) is tied. There will be only one configuration, and therefore, retains the default Configuration Value of 1.
- Set the **Number of Interfaces** to 3. The first interface will be used for the transfer of audio controls (volume and mute). The second is for transferring audio data from the microphone to the PC, and the third is for transferring audio data from the PC to the headphone.

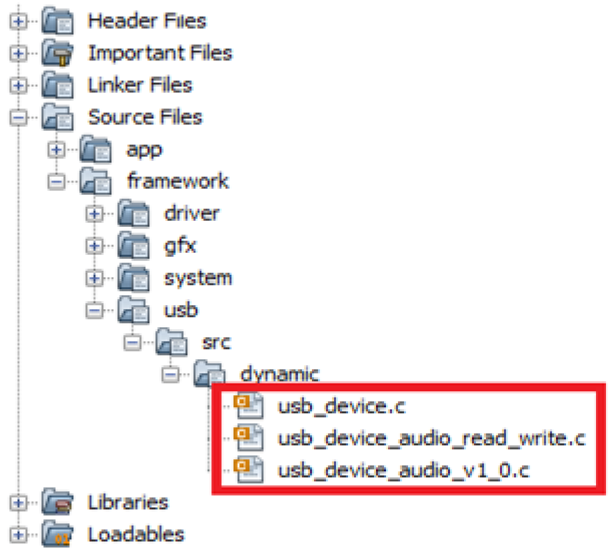
- Set the read and write queue sizes implemented by the function driver. The read and write queues are used by the application for buffering the audio data. The queue sizes are configured as follows:
  - **Audio Write Queue Size** is set to 8. For a continuous transfer of audio stream, and to avoid the loss of audio data, the write queue size is set to eight to allow audio buffer queuing for microphone operations.
  - **Audio Read Queue Size** is set to 64. For a continuous transfer of audio stream, and to avoid the loss of audio data, the read queue size is set to 64 to allow audio buffer queuing for headphone operations.

Note: The Queue Size values assigned above are arrived based on observation/experimentation.

- **Number of Audio Streaming Interfaces** is set to 2, one for the microphone interface and the other for headset interface.
- **Maximum Number of Interface Alternate Settings** is retained with the default value of 2. Two alternate settings are used: one to reclaim the bandwidth when the audio functionality is not being used, and the other when the audio functionality is used.
- Set the **Product ID Selection** as follows:
  - In Product ID Selecting, select **usb_headset_multiple_sampling_demo**. This indicates that the application falls into the category of USB headset. When the usb_headset_multiple_sampling_demo is selected, the following fields are populated, as follows:
  - **Enter Vendor ID**: 0x04D8
  - **Enter Product ID**: 0x00FF
  - **Manufacturer String**: “Microchip Technology Inc.”
  - **Product String**: “Harmony USB Headset Multiple Sampling Rate Example”

Once the USB Audio Library is configured as previously described, the project generated by the MHC adds the necessary USB Audio Library files to the project, as shown in the following figure.

Figure 5-3. USB Audio Library Files



Tip:
To configure and get started with MPLAB Harmony, as shown in the sections Configure the USB Audio 1.0 Library Using the MHC and Configuring Audio Drivers Using the MHC, open the MPLAB X IDE MPLAB Harmony project for the target device (for example, PIC32MX470F512L for the PIC32 Bluetooth Audio Development Kit with AK4642 Codec Daughter Board) and start the MPLAB Harmony Configurator (MHC).

5.3 Configure Audio Drivers Using the MPLAB Harmony Configurator (MHC)

MPLAB Harmony supports various codec driver libraries to be used by the application. For the USB headset application, on the PIC32 Bluetooth Audio Development Kit with the PIC32 Audio Codec Daughter Board - AK4642EN, configure the AK4642 Codec Driver using MHC, as follows.

Note: The AK4642 codec device uses I^{2}C as the control interface and I^{2}S as the data interface medium with the PIC32 microcontroller. The MPLAB Harmony AK4642 Codec Driver Library is built on top of the I^{2}C and I^{2}S driver libraries. The AK4642 Codec Driver also uses the DMA and clock system services; therefore, the following section describes the configuration of these drivers and system services, in addition to the configuration of the AK4642 Codec Driver.

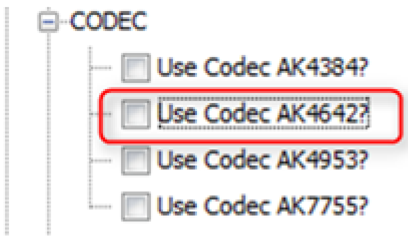
In MHC, select the **Options** tab, and expand *Harmony Framework Configuration*.

5.3.1 Configuring the AK4642 Codec Driver

To configure the AK4642 Codec Driver, perform the following steps:

1. Expand *Drivers* > *CODEC* and select **Use Codec AK4642**.

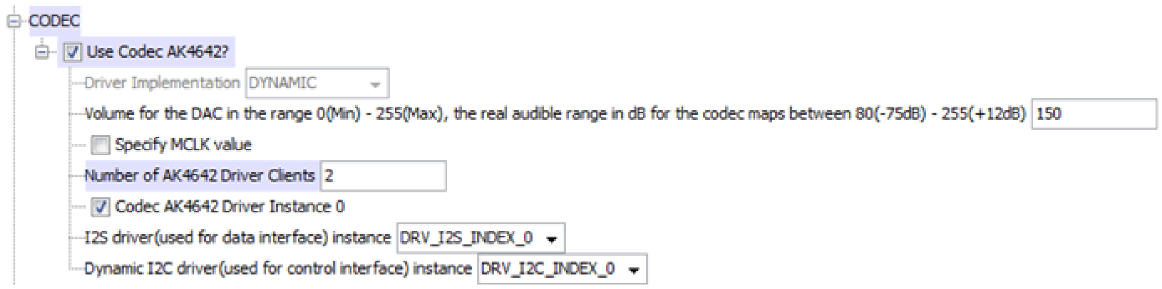
Figure 5-4. MHC - Codec Drivers



CODEC
- [ ] Use Codec AK4384?
- [ ] Use Codec AK4642?
- [ ] Use Codec AK4953?
- [ ] Use Codec AK7755?

2. Configure the AK4642 Codec Driver, as shown in the following figure.

Figure 5-5. MHC - Codec AK4642 Driver



```
CODEC
  [✓] Use Codec AK4642?
      Driver Implementation DYNAMIC
      Volume for the DAC in the range 0(Min) - 255(Max), the real audible range in dB for the codec maps between 80(-75dB) - 255(+12dB) 150
      [ ] Specify MCLK value
      Number of AK4642 Driver Clients 2
      [✓] Codec AK4642 Driver Instance 0
      I2S driver(used for data interface) instance DRV_I2S_INDEX_0
      Dynamic I2C driver(used for control interface) instance DRV_I2C_INDEX_0
```

Note: The following points will apply to the codec driver configuration for creating a USB headset application:

- Number of the AK4642 driver clients is set to 2. One client for the microphone, and another for the headphone.

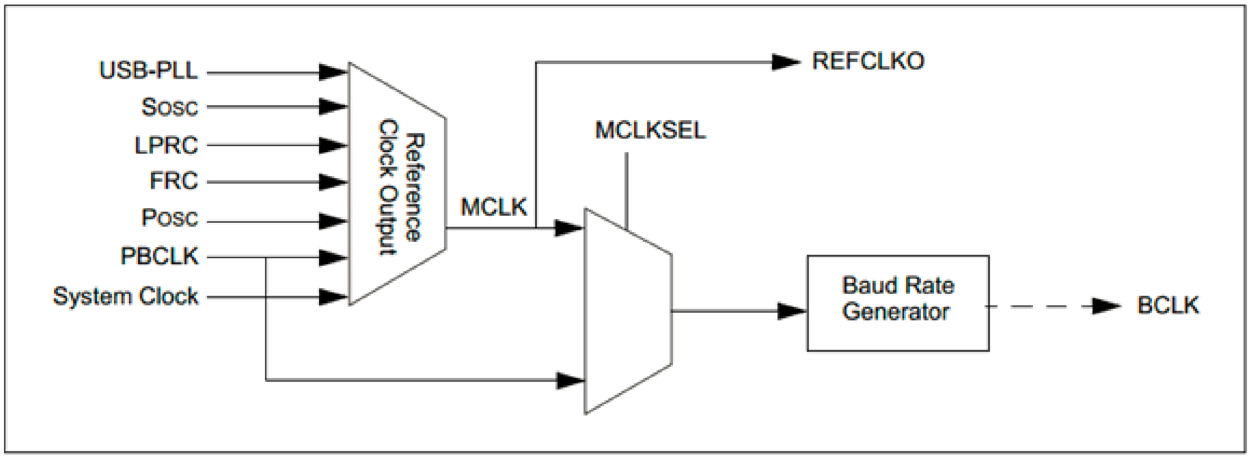
- Specify MCLK value is not selected because a default value of 256 for the MCLK Sampling rate Multiplier is selected under the I^{2}S Driver configuration.

Note: The configuration option *Specify MCLK value* in Codec AK4642 driver and MCLK Sampling rate Multiplier in I^{2}S driver, configure the same parameter, as explained below.

The AK4642 codec device expects an input clock source to generate accurate audio sampling rates. The PIC32MX470F512L device provides a flexible reference clock output (REFCLKO). The reference clock output is used to generate the fractional clock used by the AK4642 codec device to accommodate various sample rates.

The following figure shows the REFCLKO circuit on PIC32MX470F512L used to generate the fractional clock to the AK4642 codec.

Figure 5-6. Codec Master Clock (MCLK) and Bit Clock (BCLK) Generator



The AK4642 requires a Master Clock (MCLK), and its value is the operating sampling frequency multiplied by the MCLK- multiplier (refer to the Specify MCLK value in the Codec AK4642 driver configuration options and MCLK sampling rate multiplier in the I^{2}S driver configuration options).

The following figure shows the MCLK-multiplier values supported by the codec.

Table 5-28. Typical Master Clocks (MCLK) and Bit Clocks (BCLK) Required by the CODEC for Different Sampling Rates

Sample Rate (kHz)	Codec Master Clock/Time Base with Different Oversampling (MHz)							Bit Clock (MHz)	
fs	128 fs	192 fs	256 fs	384 fs	512 fs	768 fs	1152 fs	32 fs	64 fs
32.0	-	-	8.1920	12.2880	16.3840	24.5760	36.8640	1.024	2.0480
44.1	-	-	11.2896	16.9344	22.5792	33.8688	-	1.4112	2.8224
48.0	-	-	12.2880	18.4320	24.5760	36.8640	-	1.536	3.0720
88.2	11.2896	16.9344	22.5792	33.8688	-	-	-	2.8224	5.6448
96.0	12.2880	18.4320	24.5760	36.8640	-	-	-	3.072	6.1440
176.4	22.5792	33.8688	-	-	-	-	-	5.6448	11.2896
192.0	24.5760	36.8640	-	-	-	-	-	6.144	12.2880

For example, with the 256 fs option, the MCLK-multiplier value is 256 and fs is the sampling frequency.

For a sampling rate of 48 kHz and a MCLK-multiplier value of 256, the Master Clock (MCLK) = 256 * 48000 = 12288000 Hz.

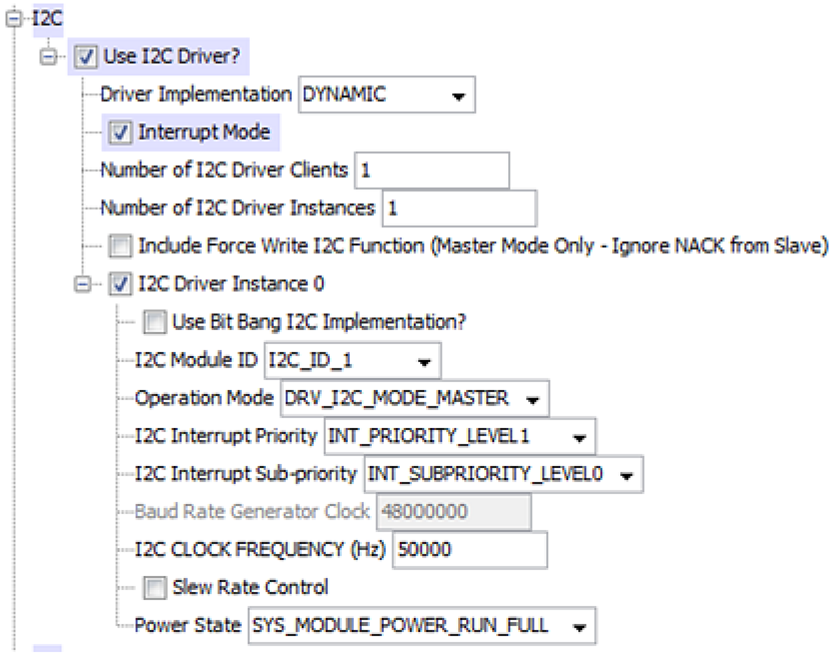
The AK4642 codec driver sets the MCLK-multiplier value to 256 by default. Therefore, the option is not selected. If a different MCLK-multiplier value is desired, the Specify MCLK value option must be selected. The different options available are 128, 192, 256, 384, 512, 768, and 1152.

5.3.2 Configuring the I^{2}C Driver

To configure the I^{2}C Driver, perform the following steps:

1. Expand *Drivers*> *I^{2}C*, select **Use I^{2}C Driver?** and configure the I^{2}C driver, as shown in the following figure.

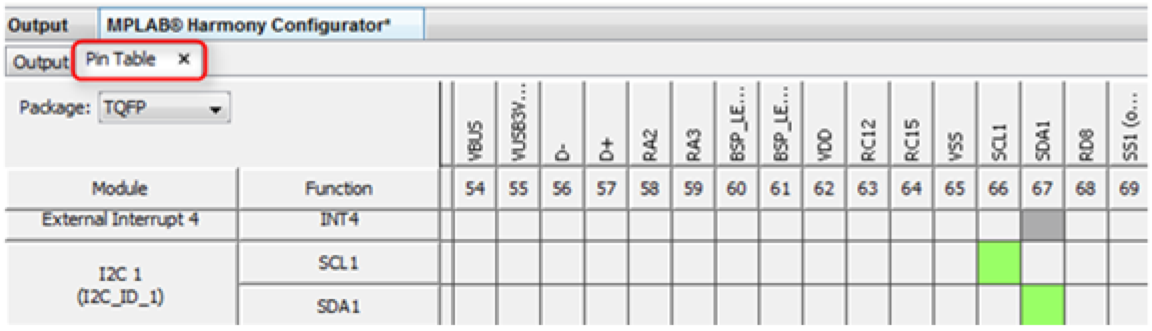
Figure 5-7. MHC – I^{2}C Driver



```
I2C
  [✓] Use I2C Driver?
      Driver Implementation  DYNAMIC
      [✓] Interrupt Mode
      Number of I2C Driver Clients  1
      Number of I2C Driver Instances  1
      [ ] Include Force Write I2C Function (Master Mode Only - Ignore NACK from Slave)
      [✓] I2C Driver Instance 0
          [ ] Use Bit Bang I2C Implementation?
          I2C Module ID  I2C_ID_1
          Operation Mode  DRV_I2C_MODE_MASTER
          I2C Interrupt Priority  INT_PRIORITY_LEVEL1
          I2C Interrupt Sub-priority  INT_SUBPRIORITY_LEVEL0
          Baud Rate Generator Clock  48000000
          I2C CLOCK FREQUENCY (Hz)  50000
          [ ] Slew Rate Control
          Power State  SYS_MODULE_POWER_RUN_FULL
```

The I/O pins used by the I^{2}C module are configured using the Graphical Pin Manager, as shown in the following figure.

Figure 5-8. MHC - Pin Mapping Table - I^{2}C Driver



Module	Function	VBUS	VUSB3V...	D-	D+	RA2	RA3	BSP_LE...	BSP_LE...	VDD	RC12	RC15	VSS	SCL1	SDA1	RD8	SS1 (o...
		54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69
External Interrupt 4	INT4														▨		
I2C 1 (I2C_ID_1)	SCL1													🟩			
	SDA1														🟩		

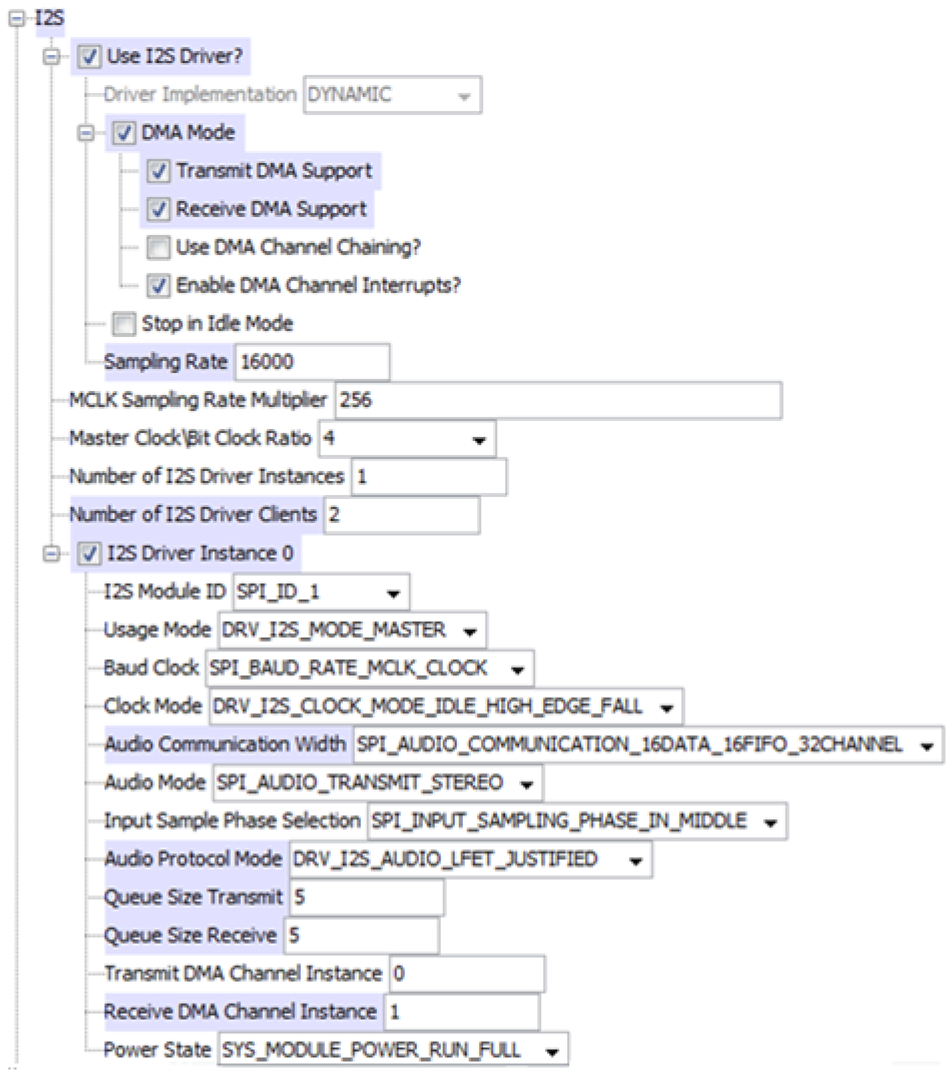
Select the **Pin Diagram** sub-tab, and then select the **Pin Table** tab in the MHC output pane (bottom of IDE window).

5.3.3 Configuring the I^{2}S Driver

To configure the I^{2}S Driver, perform the following steps:

1. Expand *Drivers*> *I^{2}S*. Then, select **Use I^{2}S Driver?** and configure the I^{2}S driver, as shown in the following figure.

MHC – I^{2}S Driver



```
I2S
  [✓] Use I2S Driver?
      Driver Implementation  DYNAMIC
      [✓] DMA Mode
          [✓] Transmit DMA Support
          [✓] Receive DMA Support
          [ ] Use DMA Channel Chaining?
          [✓] Enable DMA Channel Interrupts?
      [ ] Stop in Idle Mode
      Sampling Rate  16000
  MCLK Sampling Rate Multiplier  256
  Master Clock\Bit Clock Ratio  4
  Number of I2S Driver Instances  1
  Number of I2S Driver Clients  2
  [✓] I2S Driver Instance 0
      I2S Module ID  SPI_ID_1
      Usage Mode  DRV_I2S_MODE_MASTER
      Baud Clock  SPI_BAUD_RATE_MCLK_CLOCK
      Clock Mode  DRV_I2S_CLOCK_MODE_IDLE_HIGH_EDGE_FALL
      Audio Communication Width  SPI_AUDIO_COMMUNICATION_16DATA_16FIFO_32CHANNEL
      Audio Mode  SPI_AUDIO_TRANSMIT_STEREO
      Input Sample Phase Selection  SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE
      Audio Protocol Mode  DRV_I2S_AUDIO_LFET_JUSTIFIED
      Queue Size Transmit  5
      Queue Size Receive  5
      Transmit DMA Channel Instance  0
      Receive DMA Channel Instance  1
      Power State  SYS_MODULE_POWER_RUN_FULL
```

Note:
The following points apply for the I^{2}S driver configuration, for creating the USB headset application:

- The DMA Mode of operation is selected. Under DMA Mode, Transmit DMA Support and Receive DMA Support are selected. The Enable DMA Channel Interrupts? option is selected by default. Transmit and Receive DMA Support is enabled to use DMA channels to transfer audio data between memory and I^{2}S peripheral buffers. The DMA channel interrupts are enabled to allow DMA transfer completion notification events.
- The default value of 256 for MCLK Sampling rate Multiplier is retained. Refer to the explanation for the MCLK-multiplier value under the Codec AK4642 driver configuration section above.
- The default value of 4 for the Master Clock/Bit Clock ratio is retained. The Bit Clock (BCLK) provided to the codec by the PIC32 device is derived from the master clock (MCLK). The common

bit clocks (BCLK) that can be generated for the given combination of BCLK-multipliers and sampling rates (for example, 32 fs and 64 fs) would be MCLK/1, MCLK/2, MCLK /4, or MCLK /8.

  – For Example, for a BCLK value of 64 fs, 48000 Hz is the sampling rate.

    The BCLK would be 64 * 48000 = 3072000 Hz.

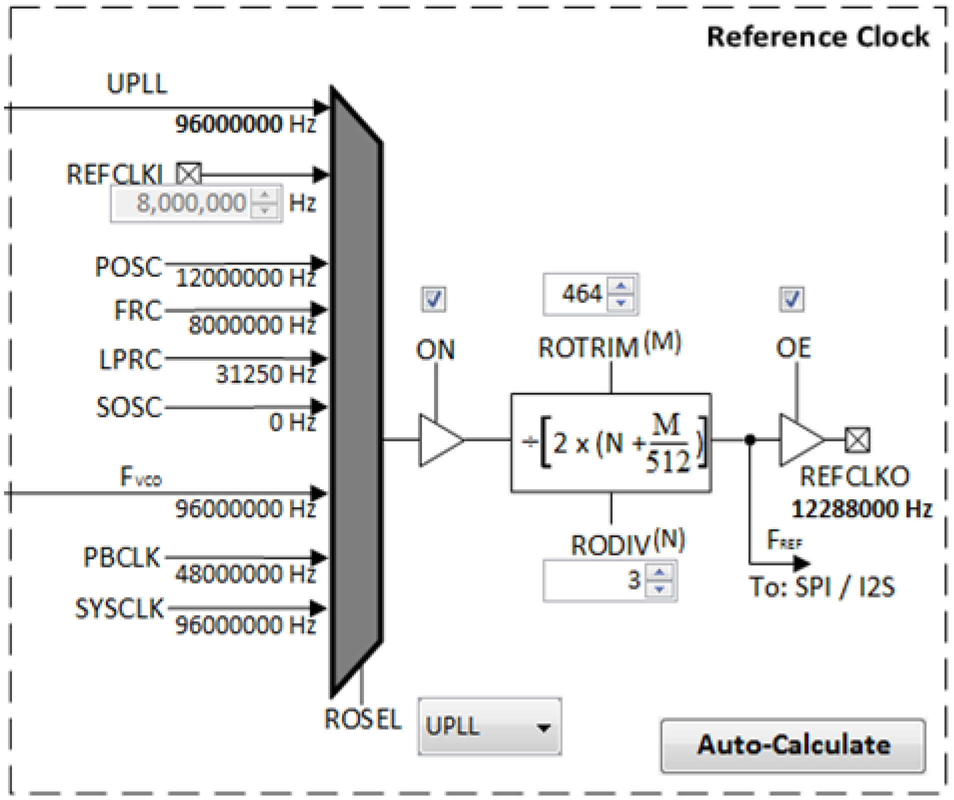
    Therefore, the MCLK/BCLK ratio is 12288000/3072000 = 4.

- The Usage Mode is DRV_I2S_MODE_MASTER. This indicates whether the I^2S instance would act as a Master/Slave. In Master mode, the PIC32 generates the BCLK to the Slave. In Slave mode, the PIC32 receives BCLK from the I^2S Master. In the interface to the AK4642 Codec, the PIC32 I^2S acts as the Master, and generates the BCLK.
- Audio Communication Width is set to SPI_AUDIO_COMMUNICATION_16DATA_16FIFO_32CHANNEL, indicating CD quality Audio playback is supported; therefore, setting 16-bit audio data per channel, with a 32-bit channel width.
- Audio Mode is set as SPI_AUDIO_TRANSMIT_STEREO corresponding to stereo Audio Playback.
- Input Sample Phase Selection is set to SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE corresponding to the default phase for the I^2S protocol.
- Audio Protocol Mode is set to DRV_I2S_AUDIO_LEFT_JUSTIFIED as it is the protocol mode supported by the CODEC AK4642.
- The Queue Size Transmit and Queue Size Receive are set with a value of 5, which is set based on observation/experimentation.

5.3.4 Configuring the Codec Clock

The sampling rate, MCLK-multiplier and MCLK/BCLK ratio provide input to the clock configurator in the MHC to generate the desired codec input master clock through the reference clock circuitry.

1. Expand *System Services* > *Clock* > *Use Clock System Service?*.
2. Configure the reference clock, as shown in the following figure.
   MHC - Reference Clock Generator



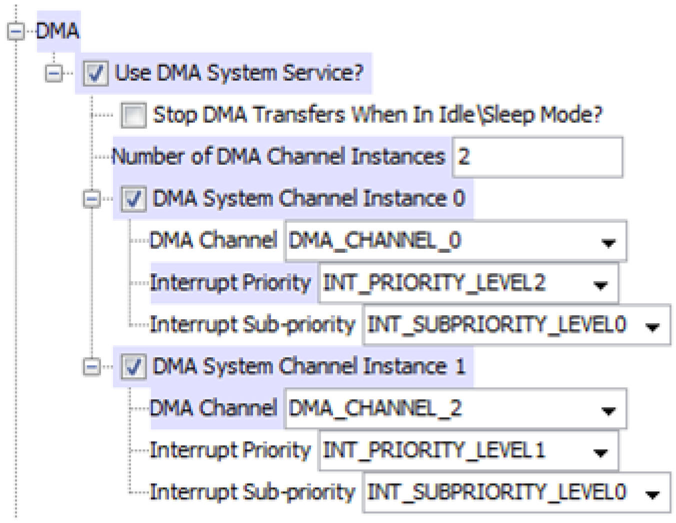
3. Click **Auto-Calculate** and verify that the Reference Clock Divisor and the Trim Auto Calculator window show the desired reference clock and sampling rate values.

5.3.5 Configuring the DMA System Service

The I²S driver is configured to use a DMA channel for transferring and receiving audio data to and from the codec. To configure and verify the DMA System Service, follow these steps:

1. Expand *System Services* > *DMA* and then select **Use Clock System Service?** and configure the DMA system service, as shown in the following figure.

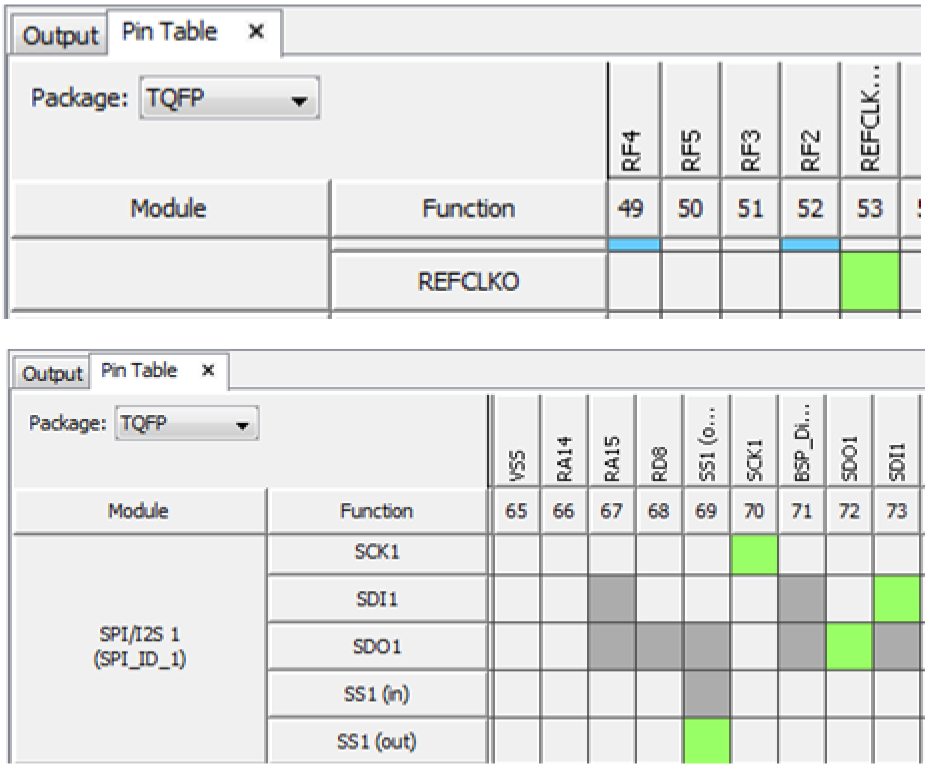
Figure 5-9. MHC - DMA System Service



DMA
- [✓] Use DMA System Service?
  - [ ] Stop DMA Transfers When In Idle\Sleep Mode?
  - Number of DMA Channel Instances: 2
  - [✓] DMA System Channel Instance 0
    - DMA Channel: DMA_CHANNEL_0
    - Interrupt Priority: INT_PRIORITY_LEVEL2
    - Interrupt Sub-priority: INT_SUBPRIORITY_LEVEL0
  - [✓] DMA System Channel Instance 1
    - DMA Channel: DMA_CHANNEL_2
    - Interrupt Priority: INT_PRIORITY_LEVEL1
    - Interrupt Sub-priority: INT_SUBPRIORITY_LEVEL0

Note: The I/O pins used by the I^2S modules are configured using the Graphical Pin Manager, as shown in the following figure.

Figure 5-10. Pin Mapping for Reference Clock and I^2S Driver



Output | Pin Table ×

Package: TQFP

Module	Function	RF4 49	RF5 50	RF3 51	RF2 52	REFCLK... 53
	REFCLKO					🟩

Output | Pin Table ×

Package: TQFP

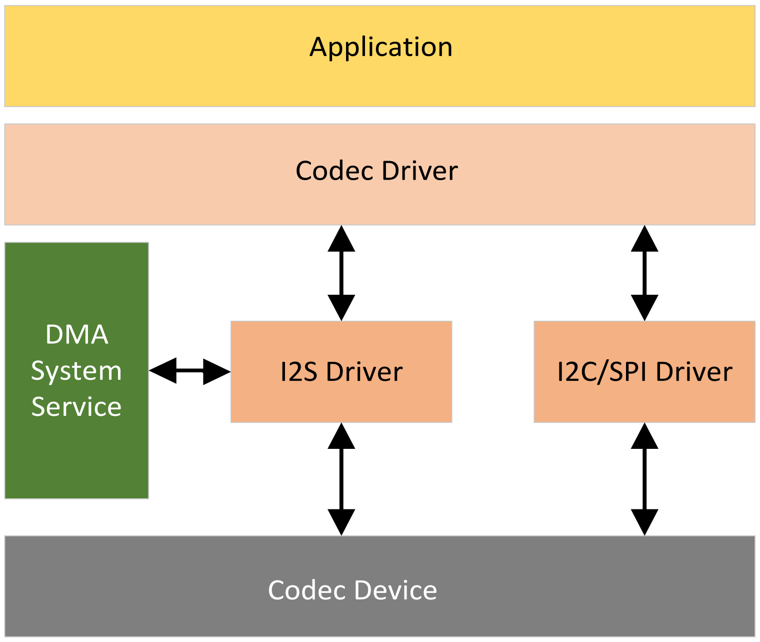
Module	Function	VSS 65	RA14 66	RA15 67	RD8 68	SS1 (o... 69	SCK1 70	BSP_Di... 71	SDO1 72	SDI1 73
SPI/I2S 1 (SPI_ID_1)	SCK1						🟩			
	SDI1			▨				▨		🟩
	SDO1			▨	▨	▨		▨	🟩	▨
	SS1 (in)					▨				
	SS1 (out)					🟩				

5.4 Architecture and Usage of Audio Drivers

5.4.1 Architecture Overview

MPLAB Harmony supports audio development by providing drivers for codec devices installed on the PIC32 development boards. The following figure shows how the codec drivers are positioned in the MPLAB Harmony software framework. The codec drivers use the SPI/ I^2C and I^2S drivers for control and audio data transfers to the codec module. The I^2S driver uses DMA channels (through the DMA System Service) for transmitting and receiving audio data between user buffers and the I^2S peripheral data register.

Figure 5-11. Audio Drivers Architecture



5.4.2 Codec Driver Library API Overview

The Codec Driver Library provides APIs for transferring control commands and digital audio data to the serially interfaced codec module. Codec drivers in MPLAB Harmony implement a common interface, as shown in the following table.

Library Interface Section		Description
System Functions	DRV_CODEC_Initialize DRV_CODEC_Deinitialize DRV_CODEC_Status DRV_CODEC_Tasks	Provides system module interfaces, device initialization, deinitialization, reinitialization, tasks and status functions.
Client Setup Functions	DRV_CODEC_Open DRV_CODEC_Close	Provides open and close functions.
Codec Specific Functions	Example: For DAC AK4384 DRV_AK4384_ZeroDetectEnable	Provides functions that are codec module specific.
Data Transfer Functions	DRV_CODEC_BufferAddWrite DRV_CODEC_BufferAddRead DRV_CODEC_BufferAddWriteRead DRV_CODEC_BufferEventHandlerSet	Provides data transfer functions.
Miscellaneous Functions	DRV_CODEC_SamplingRateSet DRV_CODEC_SamplingRateGet DRV_CODEC_VolumeSet	Provides driver specific miscellaneous functions such as sampling rate setting, control command functions, etc.

Library Interface Section		Description
	DRV_CODEC_VolumeGet DRV_CODEC_MuteOn DRV_CODEC_MuteOff	

Note:
The previous APIs will be mapped to codec-specific APIs in the `system_config.h` file. In the USB headset application, the generic codec APIs are mapped to AK4642 codec-specific APIs, as shown in the following example.

Codec API Mapping

```
#define DRV_CODEC_Initialize
DRV_AK4642_Initialize
#define DRV_CODEC_Deinitialize
DRV_AK4642_Deinitialize
#define DRV_CODEC_Status                                    DRV_AK4642_Status
#define DRV_CODEC_Tasks                                     DRV_AK4642_Tasks
#define DRV_CODEC_Open                                      DRV_AK4642_Open
#define DRV_CODEC_Close                                     DRV_AK4642_Close
#define DRV_CODEC_BufferEventHandlerSet
DRV_AK4642_BufferEventHandlerSet
#define DRV_CODEC_BufferAddWrite
DRV_AK4642_BufferAddWrite
#define DRV_CODEC_BufferAddRead
DRV_AK4642_BufferAddRead
#define DRV_CODEC_BufferAddWriteRead
DRV_AK4642_BufferAddWriteRead
#define DRV_CODEC_SamplingRateSet
DRV_AK4642_SamplingRateSet
#define DRV_CODEC_SamplingRateGet
DRV_AK4642_SamplingRateGet
#define DRV_CODEC_VolumeSet
DRV_AK4642_VolumeSet
#define DRV_CODEC_VolumeGet
DRV_AK4642_VolumeGet
#define DRV_CODEC_MuteOn                                    DRV_AK4642_MuteOn
#define DRV_CODEC_MuteOff                                   DRV_AK4642_MuteOff
#define DRV_CODEC_MicrophoneTypeSet
DRV_AK4642_IntExtMicSet
#define DRV_CODEC_MicrophoneSoundSet
DRV_AK4642_MonoStereoMicSet
#define DRV_CODEC_CommandEventHandlerSet
DRV_AK4642_CommandEventHandlerSet
```

5.4.3 Using the Codec Driver Library

For the USB headset application, the following examples show the data structure for the Codec, I^{2}S, and I^{2}C Drivers and the DMA System Service. These data structures are located in the `system_init.c` file and are generated by the MHC, based on the configuration selected.

Codec Driver Data Initialization

```
/*** CODEC Driver Initialization Data ***/
const DRV_AK4642_INIT drvak4642Codec0InitData =
{
    .moduleInit.value = SYS_MODULE_POWER_RUN_FULL,
    .i2sDriverModuleIndex = DRV_AK4642_I2S_DRIVER_MODULE_INDEX_IDX0,
    .i2cDriverModuleIndex = DRV_AK4642_I2C_DRIVER_MODULE_INDEX_IDX0,
    .volume = DRV_AK4642_VOLUME,
};
```

I2S Driver Data Initialization

```
/*** I2S Driver Initialization Data ***/
const DRV_I2S_INIT drvI2S0InitData =
{
    .moduleInit.value = DRV_I2S_POWER_STATE_IDX0,
    .spiID = DRV_I2S_PERIPHERAL_ID_IDX0,
    .usageMode = DRV_I2S_USAGE_MODE_IDX0,
    .baudClock = SPI_BAUD_RATE_CLK_IDX0,
    .baud = DRV_I2S_BAUD_RATE,
    .clockMode = DRV_I2S_CLK_MODE_IDX0,
    .audioCommWidth = SPI_AUDIO_COMM_WIDTH_IDX0,
    .audioTransmitMode = SPI_AUDIO_TRANSMIT_MODE_IDX0,
    .inputSamplePhase = SPI_INPUT_SAMPLING_PHASE_IDX0,
    .protocolMode = DRV_I2S_AUDIO_PROTOCOL_MODE_IDX0,
    .queueSizeTransmit = QUEUE_SIZE_TX_IDX0,
    .queueSizeReceive = QUEUE_SIZE_RX_IDX0,
    .dmaChannelTransmit = DRV_I2S_TX_DMA_CHANNEL_IDX0,
    .txInterruptSource = DRV_I2S_TX_INT_SRC_IDX0,
    .dmaInterruptTransmitSource = DRV_I2S_TX_DMA_SOURCE_IDX0,
    .dmaChannelReceive = DRV_I2S_RX_DMA_CHANNEL_IDX0,
    .rxInterruptSource = DRV_I2S_RX_INT_SRC_IDX0,
    .dmaInterruptReceiveSource = DRV_I2S_RX_DMA_SOURCE_IDX0,
};
```

I2C Driver Data Initialization

```
/* I2C Driver Initialization Data
*/

const DRV_I2C_INIT drvI2C0InitData =
{
    .i2cId = DRV_I2C_PERIPHERAL_ID_IDX0,
    .i2cMode = DRV_I2C_OPERATION_MODE_IDX0,
    .baudRate = DRV_I2C_BAUD_RATE_IDX0,
    .busspeed = DRV_I2C_SLEW_RATE_CONTROL_IDX0,
    .buslevel = DRV_I2C_SMBus_SPECIFICATION_IDX0,
    .mstrInterruptSource = DRV_I2C_MASTER_INT_SRC_IDX0,
    .errInterruptSource = DRV_I2C_ERR_MX_INT_SRC_IDX0,
};
```

DMA System Service Data Initialization

```
/*** System DMA Initialization Data ***/

const SYS_DMA_INIT sysDmaInit =
{
    .sidl = SYS_DMA_SIDL_DISABLE,

};
```

The AK4642 Codec Driver Library is initialized during system initialization by the SYS_Initialize function from the `system_init.c` file by calling the DRV_AK4642_Initialize function, as shown in the following example.

AK4642 Codec Driver Initialization

```
sysObj.drvak4642Codec0 = DRV_AK4642_Initialize(DRV_AK4642_INDEX_0,
(SYS_MODULE_INIT *)&drvak4642Codec0InitData);
```

The `system_interrupt.c` file contains the interrupt handlers for the configured I^{2}C and DMA System Service, as shown in the following example.

DMA and I^{2}C Interrupt Service Routines

```
//
*****************************************************************************
//
*****************************************************************************
// Section: System Interrupt Vector Functions
//
*****************************************************************************
//
*****************************************************************************

void __ISR(_DMA0_VECTOR, ipl2AUTO) _IntHandlerSysDmaCh0(void)
{
    SYS_DMA_TasksISR(sysObj.sysDma, DMA_CHANNEL_0); (1)
}

void __ISR(_DMA2_VECTOR, ipl1AUTO) _IntHandlerSysDmaCh1(void)
{
    SYS_DMA_TasksISR(sysObj.sysDma, DMA_CHANNEL_2); (2)
}

void __ISR(_I2C_1_VECTOR, ipl1AUTO) _IntHandlerDrvI2CInstance0(void)
{
    DRV_I2C_Tasks(sysObj.drvI2C0); (3)

}
```

The DMA System Service provides the interrupt triggers for the I^{2}S Driver buffer completion events. Two separate DMA channel interrupts (one for transmit and the other for receive) are used. The I^{2}S Driver task, handling the audio data transmission (DRV_I2S_WriteTasks) and reception (DRV_I2S_ReadTasks), is called from the DMA System Service task (SYS_DMA_TasksISR) from the DMA channel ISRs marked as '1' and '2' in the previous example.

The I^{2}C driver task handling function is called from the I2C ISR marked as '3' in the figure above.

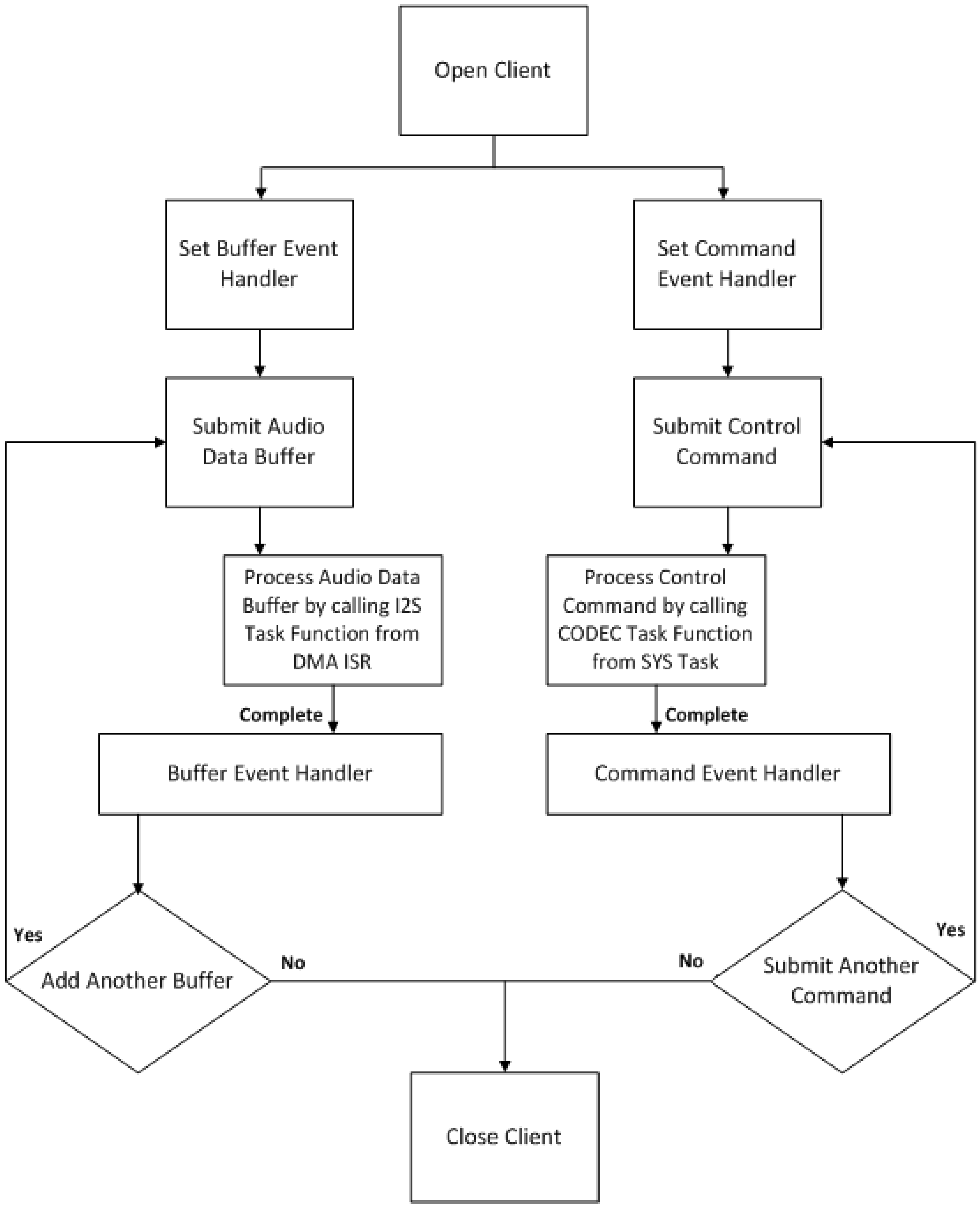
The `system_tasks.c` file contains the SYS_Tasks routine that runs the state machines for codec drivers, as shown in the following example.

AK4642 Codec Tasks Routine System Call

```
void SYS_Tasks ( void )
{
    /* Maintain system services */

    /* Maintain Device Drivers */
    DRV_AK4642_Tasks(sysObj.drvak4642Codec0);

    /* Maintain Middleware & Other Libraries */

     /* USB FS Driver Task Routine */
     DRV_USBFS_Tasks(sysObj.drvUSBObject);

    /* USB Device layer tasks routine */
    USB_DEVICE_Tasks(sysObj.usbDevObject0);

    /* Maintain the application's state machine. */
    APP_Tasks();
}
```

The following diagram shows the flow of codec driver usage by the application.

Figure 5-12. CODEC Driver Usage Flow



The application first opens the AK4642 codec driver twice by calling DRV_CODEC_Open function with the same module index. which obtains two driver handles. Of the two clients opened, one writes audio data (headphone function) and the other reads data (microphone function).

Codec Open

```
appData.codecClientWrite.handle = DRV_CODEC_Open (DRV_CODEC_INDEX_0,
DRV_IO_INTENT_WRITE);

appData.codecClientRead.handle = DRV_CODEC_Open (DRV_CODEC_INDEX_0,
DRV_IO_INTENT_READ);
```

The AK4642 codec driver provides a callback notification function to indicate to the application when the data transfer request has completed. The callback function is registered with the driver by calling DRV_CODEC_BufferEventHandlerSet function.

Codec Buffer Handler Set

```
.codecClientWrite.bufferHandler = (DRV_CODEC_BUFFER_EVENT_HANDLER)
APP_CODECBufferEventHandler,
.
.
.
.codecClientRead.bufferHandler = (DRV_CODEC_BUFFER_EVENT_HANDLER)
APP_CODECBufferEventHandlerRead,
```

The application submits the audio buffer write/read request by calling the DRV_CODEC_BufferAddWrite and DRV_CODEC_BufferAddRead functions, as shown in the following example.

Codec Buffer Write/Read

```
DRV_CODEC_BufferAddWrite(appData.codecClientWrite.handle,
                                   &current->writeHandle,
                                   current->buffer,
                                   appData.codecClientWrite.buffersize);

DRV_CODEC_BufferAddRead(appData.codecClientRead.handle,
                        &appData.codecClientRead.writeBufHandle1,
                        appData.codecClientRead.txbufferObject1,
                        appData.codecClientRead.bufferSize);
```

When the codec driver completes the audio data write/read request, it calls back the registered event handler functions, APP_CODECBufferEventHandler and APP_CODECBufferEventHandlerRead. The application implements logic to handle audio data buffer transmission and reception requests to implement the headphone and microphone functions.

5.5 Architecture and Usage of the USB Audio Library

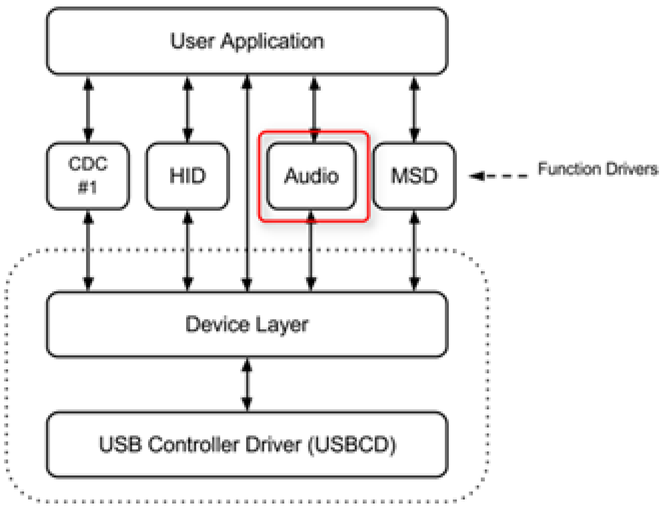
5.5.1 USB Audio Library Architecture Overview

The USB Device Library features a modular and layered framework that allows developers to design and develop a wide variety of USB devices. The USB Device Library facilitates the development of standard USB devices through function drivers that implement standard USB device class specifications. This library consists of the following three major components:

- USB Controller Driver (USBCD)
- Device Layer
- Function Drivers

The audio function driver offers various services to the USB audio device to communicate with the host by abstracting the USB specification details. The audio function driver is used with the USB device layer and USB controller to communicate with the USB host. The following figure shows a block diagram of the MPLAB Harmony USB Device Stack architecture.

Figure 5-13. USB Library Architecture



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 function driver.

5.5.2 USB Audio Library API Overview

The MPLAB Harmony USB Audio Device Library features routines to implement a USB audio device.

The following table lists USB Audio Device Library APIs and their descriptions.

Table 5-29. USB Audio Device Library APIs

Library Interface Section	Description
USB_DEVICE_AUDIO_EventHandlerSet	This function registers an event handler for the specified audio function driver instance.
USB_DEVICE_AUDIO_Read	This function requests a data read from the USB device audio function driver layer.
USB_DEVICE_AUDIO_Write	This function requests a data write to the USB device audio function driver layer.

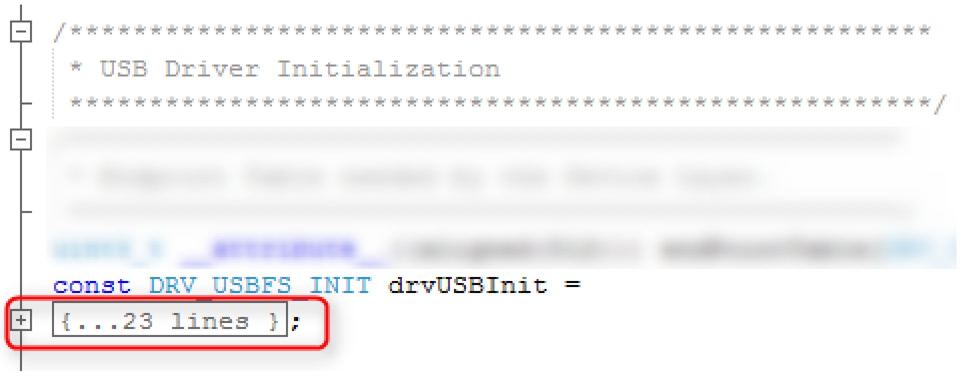
5.5.3 Using the USB Audio Library

Based on the selected configuration options in the MHC, the USB device layer descriptions, as well as the following two data structures are generated by the MHC in the `system_init.c` file for the USB headset application:

- DRV_USBFS_INIT (initialization data structure of the USB driver)
- USB_DEVICE_AUDIO_INIT (initialization data structure of the audio function driver)

The following figure shows the DRV_USBFS_INIT data structure located in `system_init.c`.

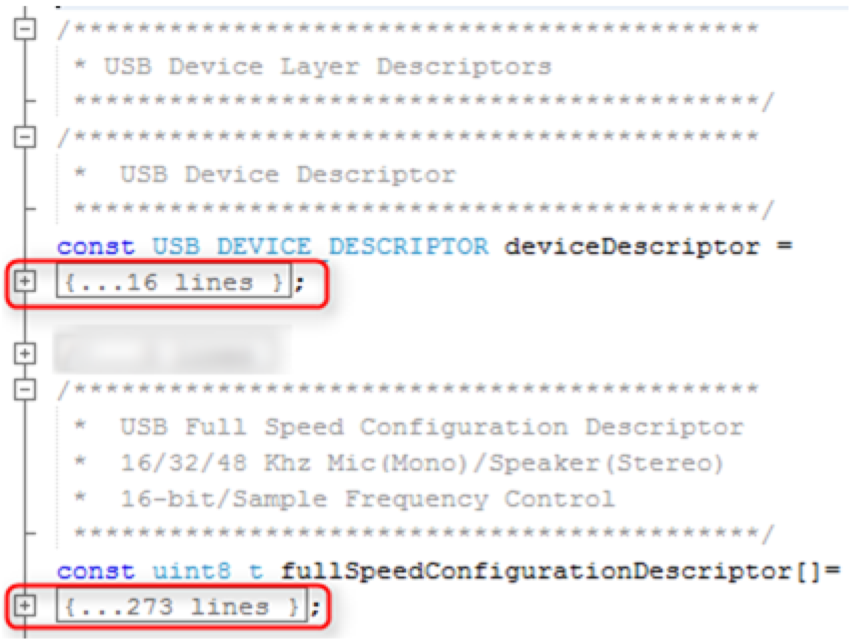
Figure 5-14. USB Driver Data Initialization



```
/*******************************************************
 * USB Driver Initialization
 *******************************************************/
const DRV_USBFS_INIT drvUSBInit =
{...23 lines };
```

The following figure shows the initialized data structures for USB device layer descriptors in `system_init.c`.

Figure 5-15. USB Device Layer Data Initialization



```
/*******************************************
 * USB Device Layer Descriptors
 *******************************************/
/*******************************************
 *  USB Device Descriptor
 *******************************************/
const USB_DEVICE_DESCRIPTOR deviceDescriptor =
{...16 lines };

/*******************************************
 *  USB Full Speed Configuration Descriptor
 *  16/32/48 Khz Mic(Mono)/Speaker(Stereo)
 *  16-bit/Sample Frequency Control
 *******************************************/
const uint8_t fullSpeedConfigurationDescriptor[]=
{...273 lines };
```

The audio function driver is initialized by the device layer when the configuration is set by the host.

The audio function driver initialization data structure, USB_DEVICE_AUDIO_INIT, is initialized with the size of the read and write queues. The funcDriverInit member of the function driver registration table entry of the audio function driver instance points to this initialization data structure. The USB_DEVICE_AUDIO_FUNCTION_DRIVER data structure provides the device layer with an entry point into the audio function driver. The following example shows how the audio function driver is registered with the device layer.

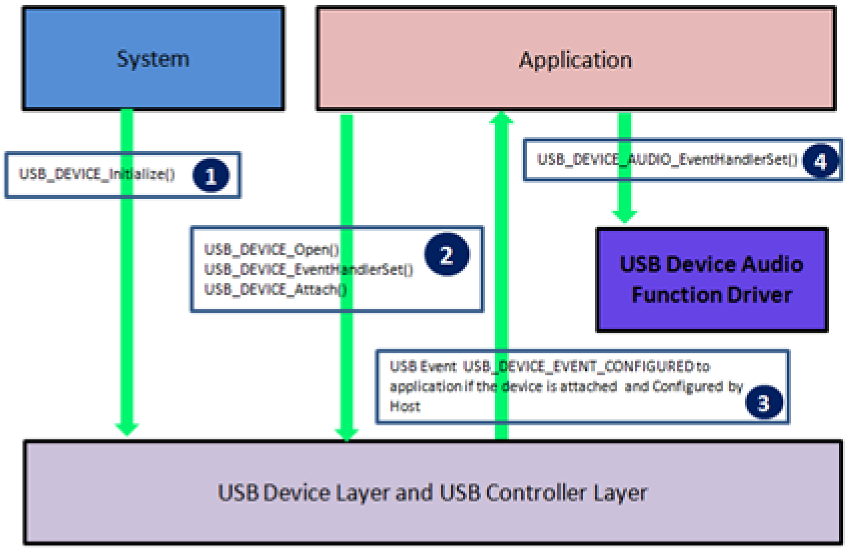
USB Audio Function Driver Data Initialization

```
/*****************************************************
 * USB Device Function Driver Init Data
 *****************************************************/
    const USB_DEVICE_AUDIO_INIT audioInit0 =
    {
        .queueSizeRead = 64,
        .queueSizeWrite = 8
    };
```

```
/**************************************************
 * USB Device Layer Function Driver Registration
 * Table
 **************************************************/
const USB_DEVICE_FUNCTION_REGISTRATION_TABLE funcRegistrationTable[1] =
{
    /* Function 1 */
    {
        .configurationValue = 1,    /* Configuration value */
        .interfaceNumber = 0,       /* First interfaceNumber of this function
*/
        .speed = USB_SPEED_FULL,    /* Function Speed */
        .numberOfInterfaces = 3,    /* Number of interfaces */
        .funcDriverIndex = 0,       /* Index of Audio Function Driver */
        .driver = (void*)USB_DEVICE_AUDIO_FUNCTION_DRIVER,    /* USB Audio
function data exposed to device layer */
        .funcDriverInit = (void*)&audioInit0,    /* Function driver init data*/
    },
};
```

The following figure illustrates the typical sequence followed by the application when using the audio function driver.

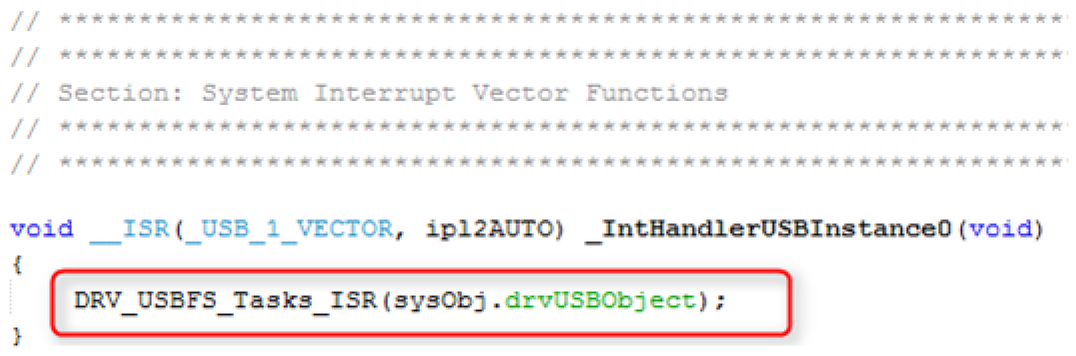
Figure 5-16. USB Audio Device Execution Sequence


System
Application
USB_DEVICE_Initialize()
1
USB_DEVICE_AUDIO_EventHandlerSet()
4
USB_DEVICE_Open()
USB_DEVICE_EventHandlerSet()
USB_DEVICE_Attach()
2
USB Device Audio Function Driver
USB Event USB_DEVICE_EVENT_CONFIGURED to application if the device is attached and Configured by Host
3
USB Device Layer and USB Controller Layer

- The system calls the USB_DEVICE_Initialize function to initialize the USB device layer.
- The device layer allows the application to register a callback function to receive USB device events like attached, powered, configured, and so on. The application uses the USB_DEVICE_EVENT_CONFIGURED event to proceed.
- Once the device layer is configured, the application registers a callback function with the audio function driver to receive audio control transfers, and other audio function driver events. At this point, the application is ready to use the audio function driver APIs to communicate with the USB host.

The `system_interrupt.c` file contains the interrupt handlers for the USB device stack. The USB driver task handling function is called from the USB ISR, as shown in the following figure.

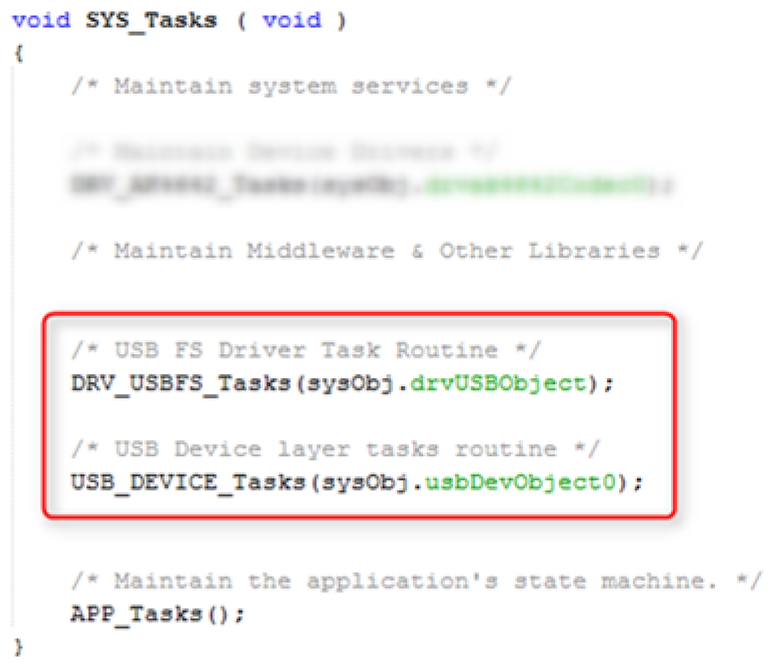
Figure 5-17. USB Interrupt Handler



```
// *****************************************************************
// *****************************************************************
// Section: System Interrupt Vector Functions
// *****************************************************************
// *****************************************************************

void __ISR(_USB_1_VECTOR, ipl2AUTO) _IntHandlerUSBInstance0(void)
{
    DRV_USBFS_Tasks_ISR(sysObj.drvUSBObject);
}
```

The `system_tasks.c` file contains the SYS_Tasks routine that runs the state machines task routines for the USB function driver and device layer, is shown in the following figure.

Figure 5-18. System Calls USB Driver and Device Task



```
void SYS_Tasks ( void )
{
    /* Maintain system services */

    [redacted]

    /* Maintain Middleware & Other Libraries */

    /* USB FS Driver Task Routine */
    DRV_USBFS_Tasks(sysObj.drvUSBObject);

    /* USB Device layer tasks routine */
    USB_DEVICE_Tasks(sysObj.usbDevObject0);

    /* Maintain the application's state machine. */
    APP_Tasks();
}
```

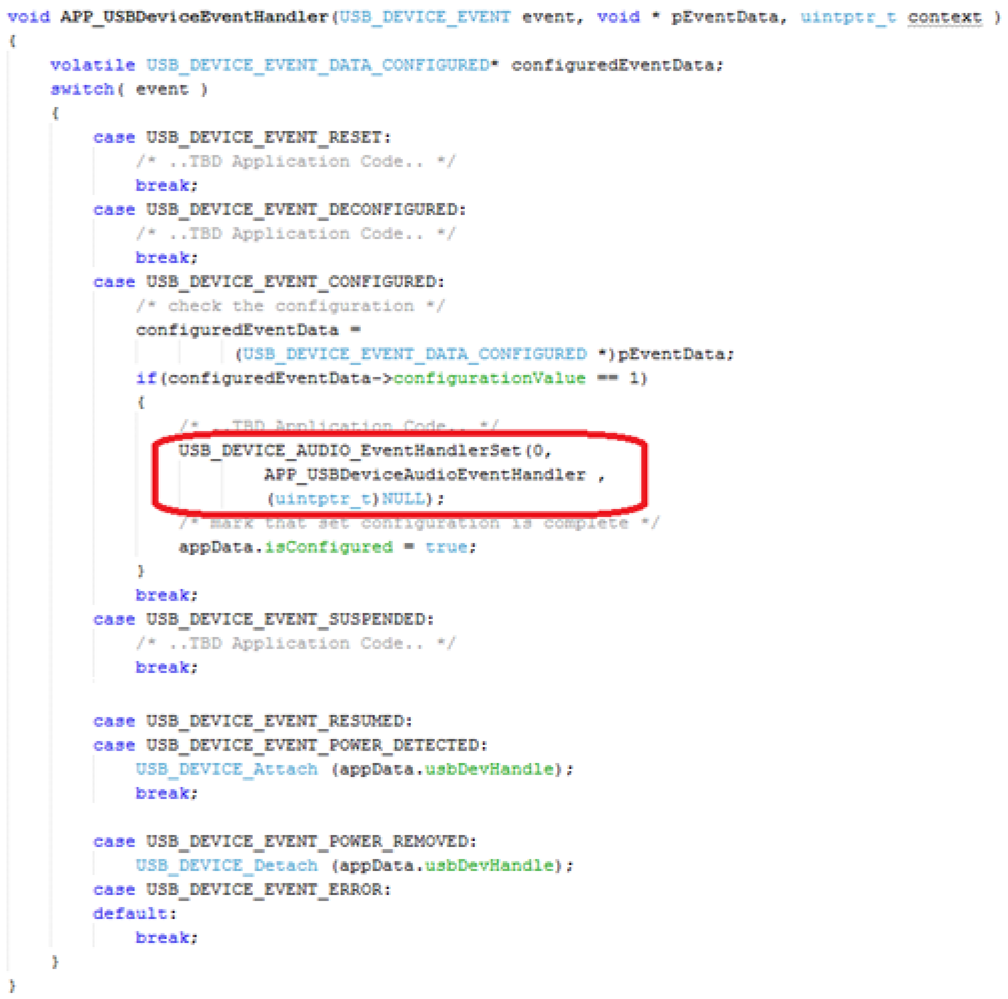
5.5.4 Registering an Audio Function Driver Callback Function

For creating a USB headset application, the application registers the event handler with the audio function driver:

- For receiving audio control requests from the host, such as volume control, mute control, and so on.
- For handling other events from the USB audio device driver (that is, data write complete, or data read complete)

The event handler is registered before the USB device layer acknowledges the SET CONFIGURATION request from the USB host. To ensure this, the callback function is set in the USB_DEVICE_EVENT_CONFIGURED event that is generated by the device layer.

Figure 5-19. USB Device Event Handler Registering Audio Event Handler



```
void APP_USBDeviceEventHandler(USB_DEVICE_EVENT event, void * pEventData, uintptr_t context )
{
    volatile USB_DEVICE_EVENT_DATA_CONFIGURED* configuredEventData;
    switch( event )
    {
        case USB_DEVICE_EVENT_RESET:
            /* ..TBD Application Code.. */
            break;
        case USB_DEVICE_EVENT_DECONFIGURED:
            /* ..TBD Application Code.. */
            break;
        case USB_DEVICE_EVENT_CONFIGURED:
            /* check the configuration */
            configuredEventData =
                        (USB_DEVICE_EVENT_DATA_CONFIGURED *)pEventData;
            if(configuredEventData->configurationValue == 1)
            {
                /* ..TBD Application Code.. */
                USB_DEVICE_AUDIO_EventHandlerSet(0,
                        APP_USBDeviceAudioEventHandler ,
                        (uintptr_t)NULL);
                /* mark that set configuration is complete */
                appData.isConfigured = true;
            }
            break;
        case USB_DEVICE_EVENT_SUSPENDED:
            /* ..TBD Application Code.. */
            break;

        case USB_DEVICE_EVENT_RESUMED:
        case USB_DEVICE_EVENT_POWER_DETECTED:
            USB_DEVICE_Attach (appData.usbDevHandle);
            break;

        case USB_DEVICE_EVENT_POWER_REMOVED:
            USB_DEVICE_Detach (appData.usbDevHandle);
        case USB_DEVICE_EVENT_ERROR:
        default:
            break;
    }
}
```

Note:
The previous figure is a short depiction of the event handler code. It does not represent the complete USB Device event handler as implemented under the USB headset application. Refer to the file `app.c` under the USB headset application to view the source code. The reference to the source code is given in the section, *"USB Headset Application Solution"* below.

- The USB audio device driver provides functions to send and receive data.
- The USB_DEVICE_AUDIO_Read function schedules a data read. When the host transfers data to the device, the audio function driver receives the data and invokes the USB_DEVICE_AUDIO_EVENT_READ_COMPLETE event. This event indicates that audio data is available in the specified application buffer. The audio data is transferred to the codec driver for playback and the USB read buffer queue is made ready to receive the next audio data.
- The USB_DEVICE_AUDIO_Write function schedules a data write. When the host sends a request for the data, the audio function driver transfers the data and invokes the USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE event. The event handler notes the completion of the write to the host and indicates for the arrangement of the next data to be written to the host.

The following example shows the event handler code for USB_DEVICE_AUDIO_EVENT_READ_COMPLETE and USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE events.

USB Device Audio Event Handler Code - Write/Read

```
case USB_DEVICE_AUDIO_EVENT_READ_COMPLETE:
{
    readEventData = (USB_DEVICE_AUDIO_EVENT_DATA_READ_COMPLETE *)pData;

    _APP_SetUSBReadBufferReady(readEventData->handle);
    appPlaybackBuffer.usbReadCompleteBufferLevel++;
    queueEmpty = false;

    if(appData.state == APP_SUBMIT_INITIAL_CODEC_WRITE_REQUEST)
    {
    if(_APP_USBReadAllBufferReady())
    {
        usbReadCompleteFlag = true;
    }
    }
}
    break;

case USB_DEVICE_AUDIO_EVENT_WRITE_COMPLETE:
{
    writeEventData=(USB_DEVICE_AUDIO_EVENT_DATA_WRITE_COMPLETE *)pData;

    if (writeEventData->handle == appData.writeTransferHandle1)
    {
    appData.isUSBWriteComplete1 = true;
    }
    else if(writeEventData->handle == appData.writeTransferHandle2)
    {
    appData.isUSBWriteComplete2 = true;
    }
    else
    {
    }
}
break;
```

When the audio function driver receives an audio class specific control transfer request, it passes this control transfer to the application as an audio function driver event. The USB audio device driver implements the SET and GET events for the current settings of the audio controls. When the host sends the setup packet to set the current audio control, the audio function driver invokes the USB_DEVICE_AUDIO_EVENT_CONTROL_SET_CUR event. The event handler identifies whether the setup packet is directed to the interface or endpoint.

If the recipient of the setup packet is interface, it identifies the entity to which the control command is directed. If the identified entity is one of the three feature units, and the audio control command is to set the mute control, the driver calls the USB_DEVICE_ControlReceive function and gets ready to receive the data packet. When the entity ID is the mixer unit, the driver stalls the request by responding with an error status indicating no audio control is implemented for the mixer unit.

If the recipient of the setup packet is an endpoint, and the audio control command is to set the sampling frequency control, the driver calls the USB_DEVICE_ControlReceive function and gets ready to receive the data packet.

The following example shows the source code for the audio event handler code for the current audio control set event.

USB Device Audio Event Handler Code - Set Current Control

```
case USB_DEVICE_AUDIO_EVENT_CONTROL_SET_CUR:
{
    if(((USB_SETUP_PACKET*)pData)->Recipient ==
         USB_SETUP_REQUEST_RECIPIENT_INTERFACE)
    {
        entityID = ((USB_AUDIO_CONTROL_INTERFACE_REQUEST*)pData)->entityID;
        if ((entityID == APP_ID_FEATURE_UNIT) ||
        (entityID == APP_ID_FEATURE_UNIT_MICROPHONE) ||
        (entityID == APP_ID_FEATURE_UNIT_SIDE_TONING))
        {
        controlSelector = ((USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST*)
                    pData)->controlSelector;
           if (controlSelector == USB_AUDIO_MUTE_CONTROL)
           {
        USB_DEVICE_ControlReceive(appData.usbDevHandle,
                        (void *) &(appData.dacMute),
                        1);
        appData.currentAudioControl = APP_USB_AUDIO_MUTE_CONTROL;
           }

        }
        else if (entityID == APP_ID_MIXER_UNIT)
        {
        USB_DEVICE_ControlStatus(appData.usbDevHandle,
                     USB_DEVICE_CONTROL_STATUS_ERROR);
        }
    }
    else if (((USB_SETUP_PACKET*)pData)->Recipient ==
           USB_SETUP_REQUEST_RECIPIENT_ENDPOINT)
    {
        controlSelector =
        ((USB_AUDIO_ENDPOINT_CONTROL_REQUEST*)
        pData)->controlSelector;

        if (controlSelector == USB_AUDIO_SAMPLING_FREQ_CONTROL)
        {
        if (((USB_AUDIO_ENDPOINT_CONTROL_REQUEST*)
        pData)->endpointNumber == MICROPHONE_EP)
        {
           USB_DEVICE_ControlReceive(appData.usbDevHandle,
                 (void *) &(appData.sampleFreqMic),
                 3);
           appData.currentAudioControl =
               APP_USB_AUDIO_SAMPLING_FREQ_CONTROL_MP;
        }

        else if (((USB_AUDIO_ENDPOINT_CONTROL_REQUEST*)
        pData)->endpointNumber == HEADPHONE_EP)
        {
           USB_DEVICE_ControlReceive(appData.usbDevHandle,
                 (void *) &(appData.sampleFreq),
                 3);
           appData.currentAudioControl =
               APP_USB_AUDIO_SAMPLING_FREQ_CONTROL_HP;
        }
        }
    }
}
break;
```

When the host sends the data packet, the audio function driver invokes the USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_DATA_RECEIVED event. The driver acknowledges the data received and responds with an OK status, indicating that audio control is implemented and the request would be handled. The event handler identifies and handles the control request to change the mute control or change the sampling frequency of the headphone and microphone.

The following example shows the source code for the audio event handler code for the data stage of the current audio control set event.

USB Device Audio Event Handler Code - Data Stage - Set Current Control

```
case USB_DEVICE_AUDIO_EVENT_CONTROL_TRANSFER_DATA_RECEIVED:
{

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;
    }
    if (appData.currentAudioControl == APP_USB_AUDIO_SAMPLING_FREQ_CONTROL_HP)
    {
        if (appData.sampleFreq == SAMPLING_RATE_48000)
        {
            appData.codecClientWrite.bufferSize = 192;
            appData.USBReadBufSize = 192;
        }
        else if (appData.sampleFreq == SAMPLING_RATE_32000)
        {
            appData.codecClientWrite.bufferSize = 128;
            appData.USBReadBufSize = 192;
        }
        else if (appData.sampleFreq == SAMPLING_RATE_24000)
        {
            appData.codecClientWrite.bufferSize = 96;
            appData.USBReadBufSize = 192;
        }
        else if (appData.sampleFreq == SAMPLING_RATE_16000)
        {
            appData.codecClientWrite.bufferSize = 64;
            appData.USBReadBufSize = 192;
        }
        appData.state = APP_SAMPLING_FREQUENCY_CHANGE;
        appData.currentAudioControl = APP_USB_CONTROL_NONE;
    }
    else if (appData.currentAudioControl ==
         APP_USB_AUDIO_SAMPLING_FREQ_CONTROL_MP)
    {
        if (appData.sampleFreqMic == SAMPLING_RATE_48000)
        {
            appData.codecClientRead.bufferSize = 192;
        }
        else if (appData.sampleFreqMic == SAMPLING_RATE_32000)
        {
            appData.codecClientRead.bufferSize = 128;
        }
        else if (appData.sampleFreqMic == SAMPLING_RATE_24000)
        {
            appData.codecClientRead.bufferSize = 96;
        }
        else if (appData.sampleFreqMic == SAMPLING_RATE_16000)
        {
             appData.codecClientRead.bufferSize = 64;
        }
        appData.state = APP_MUTE_AUDIO_PLAYBACK;
        appData.state = APP_SAMPLING_FREQUENCY_CHANGE;
        appData.currentAudioControl = APP_USB_CONTROL_NONE;
    }
}
break;
```

The host sends the setup packet to get the current audio control, the audio function driver invokes the USB_DEVICE_AUDIO_EVENT_CONTROL_GET_CUR event. The event handler identifies whether the setup packet is directed to the interface or endpoint.

If the recipient of the setup packet is interface, it identifies the entity to which the control command is directed. If the identified entity is one of the three feature units, and the audio control command is to get the mute control, the driver calls the USB_DEVICE_ControlSend function and sends the mute control state. When the entity ID is the mixer unit, the driver stalls the requests by responds with an error status indicating no audio control is implemented on the mixer unit.

If the recipient of the setup packet is an endpoint and the audio control command is to get the sampling frequency control, the driver calls the USB_DEVICE_ControlSend function and sends the current sampling frequency value to the host.

The following example shows the source code for the audio event handler code for the current audio control get event.

Device Audio Event Handler Code - Get Current Control

```
case USB_DEVICE_AUDIO_EVENT_CONTROL_GET_CUR:
{
    if (((USB_SETUP_PACKET*)pData)->Recipient ==
         USB_SETUP_REQUEST_RECIPIENT_INTERFACE)
    {
        entityID = ((USB_AUDIO_CONTROL_INTERFACE_REQUEST*) pData)->entityID;
        if ((entityID == APP_ID_FEATURE_UNIT) ||
         (entityID == APP_ID_FEATURE_UNIT_MICROPHONE) ||
         (entityID == APP_ID_FEATURE_UNIT_SIDE_TONING))
        {
           controlSelector = ((USB_AUDIO_FEATURE_UNIT_CONTROL_REQUEST*)
                    pData)->controlSelector;
           if (controlSelector == USB_AUDIO_MUTE_CONTROL)
           {
                USB_DEVICE_ControlSend(appData.usbDevHandle,
                     (void *)&(appData.dacMute),
                     1);
           }
        }

        else if (entityID == APP_ID_MIXER_UNIT)
        {
            USB_DEVICE_ControlStatus (appData.usbDevHandle,
                    USB_DEVICE_CONTROL_STATUS_ERROR);
        }
    }
    else if (((USB_SETUP_PACKET*)pData)->Recipient ==
           USB_SETUP_REQUEST_RECIPIENT_ENDPOINT)
    {
        controlSelector = ((USB_AUDIO_ENDPOINT_CONTROL_REQUEST*)
                            pData)->controlSelector;
        if (controlSelector == USB_AUDIO_SAMPLING_FREQ_CONTROL)
        {
            if (((USB_AUDIO_ENDPOINT_CONTROL_REQUEST*)
                pData)->endpointNumber == MICROPHONE_EP)
            {
               USB_DEVICE_ControlSend(appData.usbDevHandle,
                           (void *)&(appData.sampleFreqMic),
                           3);
            }

            else if (((USB_AUDIO_ENDPOINT_CONTROL_REQUEST*)
                       pData)->endpointNumber == HEADPHONE_EP)
            {
               USB_DEVICE_ControlSend(appData.usbDevHandle,
                       (void *)&(appData.sampleFreq), 3 );
            }
        }
    }
}
break;
```

Note:
The images on the previous pages are depictions of the audio event handler code. They do not represent the complete USB device audio event handler, implemented under USB headset application. Refer to the

file `app.c` under the USB headset application to view the complete source code. The reference to the source code is in the section *USB Headset Application Solution* below.

5.6 USB Headset Application Solution

MPLAB Harmony comes with the pre-developed application *usb_headset* that demonstrates USB headset functionality. A brief description and the location of this application demonstration in MPLAB Harmony is as follows.

Name: `usb_headset`

Description: In this application demonstration, The PIC32 microcontroller-based system interfaces to a USB host (such as a personal computer), which can accommodate a USB audio device class headset. The system enumerates a USB audio device endpoint and enables the system to send playback audio and receive record audio simultaneously from the USB port.

Project Name:usb_headset.X

Location:`<install-dir>/apps/audio/usb_headset/firmware`

Note:
The USB headset application demonstration does not implement HID compliant consumer audio controls.

5.7 Application Considerations

5.7.1 USB and Codec Clock Domain Mismatch Issues

In an USB audio system, audio quality issues may occur due to the presence of multiple clocks; the USB domain clock, sampling rate, service and codec domain clock. Whenever present, the clock mismatch issues need to be addressed to prevent audible noise artifacts. The USB audio application should consider the following guidelines for addressing the clock mismatch issues.

An appropriate synchronization mechanism needs to be selected and implemented. The selection of the mechanism depends on the application requirement, software capability, hardware capability and availability. For example, the synchronous type implementation (discussed in USB Audio Synchronization) is desirable when the hardware allows the codec clock source to synchronize with the USB SOF signal.

The source and the sink synchronization methods need to be compatible. The following table shows the list of synchronization combinations for audio source and sinks that are known to work well.

Table 5-30. USB Audio Synchronization Combinations

Source	Sink	USB Clock is Derived From
Synchronous	Synchronous	SOF
Adaptive	Asynchronous	Sampling Rate
Asynchronous	Asynchronous	Free running internal clock

In the following sections are the important considerations for implementing the USB audio device application.

5.7.2 Audio Data Buffering

The USB audio system poses the typical producer-consumer problem associated with a real-time system. The timing constraints applicable to USB audio systems are as follows.

- Latency - The time needed to process and input an audio sample before it is submitted out
- Sampling Period - The time between two samples
- Processing Time - The time required to process the input sample and produce the output, before the arrival of the next input sample

The USB audio device needs to implement a buffering mechanism to accommodate and address the timing challenges. The following are two possible methods for implementing USB audio data buffering.

Ping-Pong Buffer Method: The ping-pong method uses two buffers. One buffer reads data from the USB endpoint, while the other buffer is submitting the data to the codec for playback. When the submission to the codec is complete, the roles of the two buffers are switched. The role switch is accomplished by modifying the pointers to the buffers.

The ping-pong buffer implementation works in most cases, but there are situations where it may not work as intended:

- The amount of data being exchanged is very large. Large data size requires more processing time to prepare the next set of data. This could lead to situations where a transfer is completed, and the next data to be scheduled is not ready
- The clock sources of the data producer and consumer are different
- The data rate varies
- The data packet size varies

Buffer Queuing Method: The shortcomings of ping-pong buffering are addressed in the buffer queuing method. In this method the application maintains a pool of memory buffers. These buffers store the received audio data. Once a set buffering level is reached, the application starts submitting the read data to the codec on the audio device on a first in first out basis. The reading from the USB endpoint and writing to the codec happens simultaneously through a circular queue implementation of the data buffers.

The USB audio function driver and the codec driver provide support for the application to implement the buffer queuing mechanism. The USB_DEVICE_AUDIO_Read/USB_DEVICE_AUDIO_Write and the DRV_CODEC_BufferAddWrite/DRV_CODEC_BufferAddRead functions are used by the application to implement the buffering mechanisms.

These APIs implement the asynchronous buffer queuing data transfer model. The APIs are non-blocking in nature and can be called more than once to queue up audio data. Therefore, the client does not need to wait for an on-going transfer to complete to submit a new digital audio buffer.

To support the buffer queuing method, the driver (USB and codec) maintains a common pool of buffer objects in an array, whose size is determined by a configuration macro. When a buffer object is needed, the driver logic searches through the array, looking for an unallocated buffer object. The value of the configuration macro affects the amount of static RAM allocated by this driver.

For USB, the configuration constant is USB_DEVICE_AUDIO_QUEUE_DEPTH_COMBINED, and for the codec it is the maximum queue size of the associated I^{2}S driver instanceDRV_I2S_QUEUE_DEPTH_COMBINED. These macros are naturally created when the queue sizes are configured under the MHC driver configurations for USB and I^{2}S.

Note:

The sizes of these macros for a ping-pong implementation would be 2. For a buffer queuing implementation it would be configured appropriately based on the need.

5.7.3 Watermark Levels and Codec Clock Tuning

In USB audio applications, the data synchronization problem is exacerbated by the fact that the producer and consumer are running at different clock rates. In such situations, the application needs to interact with the producer/consumer and instruct the other side to either speed up or slow down.

One of the ways to achieve synchronization is to provide feedback to the peer by maintaining and managing the levels of queued samples. For example, in the implementation of a USB audio device with asynchronous endpoint implementation, the application maintains a count of the data buffer objects to be sent to the codec device. The application then achieves synchronization by sending a feedback message to the USB host through the feedback endpoint (see Asynchronous synchronization discussed in USB Audio Synchronization above).

The codec driver provides assistance to the application in maintaining its buffered samples, by providing API DRV_CODEC_BufferCombinedQueueSizeGet, which expands to the I^{2}S API DRV_I2S_BufferCombinedQueueSizeGet. This API returns the number of data buffer objects currently in the queue.

Clock domain mismatch issues observed in USB audio applications could also be addressed by adjusting (slowing or speeding up) the codec master clock (MCLK). Typically, the application maintains an average number of samples received over a defined period. It compares the average number of samples with an upper and lower water mark levels. When the average number of samples received goes out of the acceptable range, the application tunes the codec MCLK by slightly increasing or decreasing it. The tuning of the codec clock prevents buffer underrun and overrun conditions, thereby reducing the audible artifacts in the audio stream (see Adaptive synchronization discussed in section USB Audio Synchronization above).

The DRV_CODEC_SamplingRateSet codec driver API is used to set the desired sampling rate to speed up or slow down the associated codec clock.

6. Conclusion

Developing a USB audio device application requires an understanding of various standards, protocols, and middleware. Using the MPLAB Harmony USB device stack firmware framework and the accompanying audio firmware and the codec drivers, users can design a solution without having to worry about the underlying standards or protocols. MPLAB Harmony provides users with a flexible, abstracted and fully integrated firmware development platform for PIC32 microcontrollers. This application note introduced the functional model of the USB headset application consisting of audio subsystem and USB subsystem. It provides an overview of the essential concepts of audio communication pertaining to audio and USB subsystems. It also indicates how users can create and configure a USB headset application using the MPLAB Harmony Configurator (MHC) utility. Additionally, it covers observations on specific topics related to the implementation of USB audio application.

Additional examples and demos for various audio solutions are included with the MPLAB Harmony Integrated Software Framework, which is available for download from the Microchip website (see Resources).

7. Resources

Universal Serial Bus Specification 2.0

http://www.usb.org/developers/docs/usb20_docs/usb_20_033017.zip

USB Device Class Definition for Audio Devices

http://www.usb.org/developers/docs/devclass_docs/audio10.pdf

USB Audio Device Class Specification for Basic Audio Devices

http://www.usb.org/developers/docs/devclass_docs/BasicAudioDevice-10.zip

USB Device Class Definition for Human Interface Devices (HID)

http://www.usb.org/developers/hidpage/HID1_11.pdf

AN1422 High-Quality Audio Applications Using the PIC32

http://ww1.microchip.com/downloads/en/AppNotes/01422A.pdf

Atmel AVR32716: AVR UC3 USB Audio Class

http://www.atmel.com/Images/doc32139.pdf

AT91 USB HID Driver Implementation

http://www.atmel.com/Images/doc6273.pdf

“help_harmony.pdf” available with the MPLAB Harmony Software Framework Installer

http://www.microchip.com/mplab/mplab-harmony

http://www.microchip.com/mplab/mplab-harmony

http://microchip.wikidot.com/harmony:overview

MPLAB Harmony Configurator (MHC) Overview

http://microchip.wikidot.com/harmony:mhc-overview

MPLAB® Harmony Graphics Composer (MHGC) Overview

http://microchip.wikidot.com/harmony:mhc-mhgc-overview

8. Revision History

Revision A - 11/2017
This is the initial version of the document.

The Microchip Web Site

Microchip provides online support via our web site at http://www.microchip.com/. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQ), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Customer Change Notification Service

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at http://www.microchip.com/. Under "Support", click on "Customer Change Notification" and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or Field Application Engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.

ISBN: 978-1-5224-2420-8

Quality Management System Certified by DNV

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://www.microchip.com/
support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Austin, TX
Tel: 512-257-3370

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453
Tel: 317-536-2380

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608
Tel: 951-273-7800

Raleigh, NC
Tel: 919-844-7510

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110
Tel: 408-436-4270

Canada - Toronto
Tel: 905-695-1980
Fax: 905-695-2078

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon

Hong Kong
Tel: 852-2943-5100
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Dongguan
Tel: 86-769-8702-9880

China - Guangzhou
Tel: 86-20-8755-8029

China - Hangzhou
Tel: 86-571-8792-8115
Fax: 86-571-8792-8116

China - Hong Kong SAR
Tel: 852-2943-5100
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-3326-8000
Fax: 86-21-3326-8021

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

ASIA/PACIFIC

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-3019-1500

Japan - Osaka
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

Japan - Tokyo
Tel: 81-3-6880- 3770
Fax: 81-3-6880-3771

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-5778-366
Fax: 886-3-5770-955

Taiwan - Kaohsiung
Tel: 886-7-213-7830

Taiwan - Taipei
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

Finland - Espoo
Tel: 358-9-4520-820

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

France - Saint Cloud
Tel: 33-1-30-60-70-00

Germany - Garching
Tel: 49-8931-9700

Germany - Haan
Tel: 49-2129-3766400

Germany - Heilbronn
Tel: 49-7131-67-3636

Germany - Karlsruhe
Tel: 49-721-625370

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Germany - Rosenheim
Tel: 49-8031-354-560

Israel - Ra'anana
Tel: 972-9-744-7705

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Italy - Padova
Tel: 39-049-7625286

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Norway - Trondheim
Tel: 47-7289-7561

Poland - Warsaw
Tel: 48-22-3325737

Romania - Bucharest
Tel: 40-21-407-87-50

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Gothenberg
Tel: 46-31-704-60-40

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820