Freescale Semiconductor
Application Note

Document Number:AN4665
Rev. 0, 01/2013

How to Use Freescale USB Stack to Implement Audio Class Device

by: Wang Hao

Contents

1 Introduction

The USB interface is very well suited for transport of audio ranging from low-fidelity voice connections to high-quality, multichannel audio streams. Many applications from communications, to entertainment, to music recording and playback, can take advantage of audio features of the USB.

The Audio Device Class specification standardizes audio transport mechanisms to keep software drivers as generic as possible. It specifies the standard and class-specific descriptors that must be present in each USB audio function and explains the use of class-specific requests that allow for full audio function control. It also defines addressable entities like Units and Terminals which are used to describe the audio function topology and gives an interface to manipulate the physical properties of an audio function.

To develop audio class device applications from scratch is a big task; however Freescale has provided a bare-metal USB stack which supports many common USB device classes such as personal healthcare device class (PHDC), human interface device (HID), mass storage device (MSD), communications device class (CDC), and audio class. The source code is complimentary, portable, and easy to use and can be downloaded from **freescale.com/medicalUSB**.

2 Audio class device requirements

In many cases, audio functionality does not exist as a standalone device. It is one capability that, together with other functions, constitutes a “composite” device. Audio function is located at the interface level in the device class hierarchy.

2.1 Audio function overview

Each audio function must have a single AudioControl interface and can have zero or more AudioStreaming and zero or more MIDIStreaming interfaces. The AudioControl interface is used to access the Audio Controls of the function such as volume control whereas the AudioStreaming interfaces are used to transport audio streams into and out of the function. The MIDIStreaming interfaces can be used to transport MIDI data streams into and out of the audio function. See the following figure for the global overview of audio function as seen from the USB bus interface.

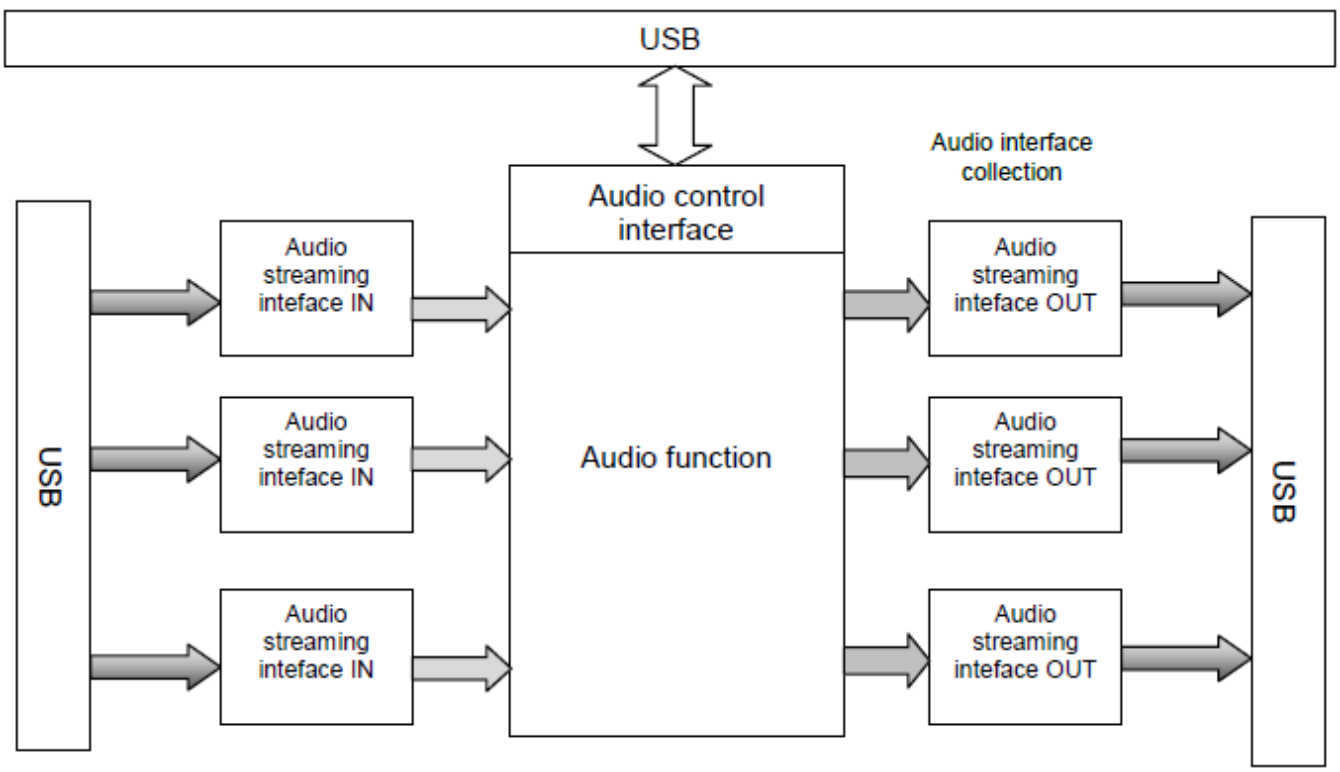

Figure 1. Audio function global overview

The collection of the single AudioControl interface and the AudioStreaming and MIDIStreaming interfaces that belong to the same audio function is called the Audio Interface Collection (AIC). A device can have multiple AICs active at the same time. These collections are used to control multiple independent audio functions located in the same composite device.

2.2 Audio function topology

To easily represent the topology and manipulate the physical properties of an audio function, two types of generic entities are defined and are called Units and Terminals. Also introduced is the audio channel cluster concept where a group of audio channels are put together. The following subsections briefly explain each of these entities.

2.2.1 Units

Units provide the basic building blocks to fully describe most audio functions. Audio functions are built by connecting together several of these Units. A Unit has one or more input pins and a single output pin, where each pin represents a cluster of logical audio channels inside the audio function. Units are wired together by connecting their I/O pins according to the required topology.

Feature Unit is a commonly used multi-channel processing unit that provides the basic manipulation of multiple single-parameter Audio Controls on the incoming logical channels such as mute, volume control, and so on.

The Feature Unit Descriptor reports what controls are present for every channel in the Feature Unit and for the 'master' channel.

2.2.2 Terminals

Two types of Terminals are introduced.

- Input Terminal: This is an entity that represents a starting point for audio channels inside the audio function. The function of the Input Terminal is to represent a source of incoming audio data after this data has been properly extracted from the original audio stream into the separate logical channels that are embedded in this stream.
- Output Terminal: This entity represents an end point for the audio channels. USB endpoint is a typical example of an Input or Output Terminal. The function of the Output Terminal is to represent a sink of outgoing audio data before this data is properly packed from the original separate logical channels into the outgoing audio stream.

2.2.3 Audio channel cluster

An audio channel cluster is a grouping of audio channels that carry tightly related synchronous audio information.

An audio channel cluster is characterized by only two attributes:

- The number of audio channels in the cluster
- The spatial location of each audio channel in the cluster, for example, left and right channel

There are two types of audio channel cluster:

- A logical cluster describes audio data within the audio function where the audio channels are treated as logical concepts.
- A physical cluster describes audio data within an AudioStream interface that handles the actual physical audio channels in the audio stream.

2.3 Descriptors

For a USB device, its descriptors fully describe its capabilities and functions to the host. Since device descriptor and configuration descriptor are similar for any kind of device, only some other descriptors are described in this section.

Following discussion is based on the *audio_speaker* demo provided with the USB stack; the descriptors are defined in *usb_descriptor.c* and comply with the USB Audio Device Class Specification 2.0.

2.3.1 Standard Interface Association Descriptor (IAD)

The standard USB Interface Association mechanism is used to describe the Audio Interface Collection, that is, to bind those interfaces together. The following codelines show that there are two interfaces, one AudioControl and one AudioStream interface.

```
/* Standard Interface Association Descriptor */
0x08,                                   /* bLength(0x08) */
USB_INTERFACE_ASSOCIATION_DESCRIPTOR,   /* bDescriptorType(0x0B) */
0x00,                                   /* bFirstInterface(0x00) */
0x02,                                   /* bInterfaceCount(0x02) */
0x01,                                   /* bFunctionClass(0x01): AUDIO */
0x00,                                   /* bFunctionSubClass(0x00) */
0x20,                    /* bFunctionProtocol(0x2000): 2.0 AF_VERSION_02_00 */
0x00,                                   /* iFunction(0x00) */
```

2.3.2 Standard AudioControl (AC) Interface Descriptor

The Standard AC Interface Descriptor is identical to the standard interface descriptor defined in Chapter 9 in USB 2.0 Specification which can be downloaded from **usb.org**. In the following code, *bNumEndpoints* field is 0 which means that there is only default control pipe and no optional interrupt endpoint for the AudioControl interface.

```
/* AUDIO CONTROL Interface */
/* Standard AC Interface Descriptor(4.7.1) */
0x09,       /* bLength(0x09) */
0x04,       /* bDescriptorType(0x04): INTERFACE */
0x00,       /* bInterfaceNumber(0x00) */
0x00,       /* bAlternateSetting(0x00) */
0x00,       /* bNumEndpoints(0x00) */
0x01,       /* bInterfaceClass(0x01): AUDIO */
0x01,       /* bInterfaceSubClass(0x01): AUDIOCONTROL */
0x20,       /* bInterfaceProtocol(0x20): IP 2.0 IP_VERSION_02_00 */
0x07,       /* iInterface(0x07): Not Requested */
```

2.3.3 Class-specific AC descriptors

The Class-Specific AudioControl Interface Descriptor is a concatenation of all the descriptors that are used to fully describe the audio function, that is, all the Unit and Terminal descriptors.

The total length of the Class-Specific AC Interface Descriptor depends on the number of Units and Terminals in the audio function. Therefore, the descriptor starts with a header that reflects the total length in bytes of the entire class-specific AC interface descriptor in the *wTotalLength* field. See the code given in the following section.

The header descriptor is followed by one or more Unit and/or Terminal descriptors. Each Unit and Terminal within the audio function is assigned a unique identification number, the Unit ID or Terminal ID. Besides uniquely identifying all addressable entities in an audio function, the IDs also serve to describe the topology of the audio function; that is, the *bSourceID* field of a Unit or Terminal descriptor indicates to which other Unit or Terminal this Unit or Terminal is connected.

2.3.3.1 Class-Specific AC Interface Header Descriptor

```
/* Class-Specific AC Interface Header Descriptor(4.7.2) */
0x09,           /* bLength(0x09) */
0x24,           /* bDescriptorType(0x24): CS_INTERFACE */
0x01,           /* bDescriptorSubType(0x01): HEADER */
0x00, 0x02,     /* bcdADC(0x0200): 2.0 */
0x01,           /* bCategory(0x01): DESKTOP_SPEAKER */
//0x40, 0x00,     /* wTotalLength(64): 9 + 8 + 17 + 18 + 12 (2 channels)
0x3C, 0x00,     /* wTotalLength(60): 9 + 8 + 17 + 14 + 12 (1 channel)
```

```
                        Audio Control Interface size */
0x00,           /* bmControls(0b00000000) */
```

2.3.3.2 Clock Source Descriptor

In the following code *bmAttributes* field is 0x01, which means that clock type is internal fixed clock.

```
/* Clock Source Descriptor(4.7.2.1) */
0x08,       /* bLength(0x08) */
0x24,       /* bDescriptorType(0x24): CS_INTERFACE */
0x0A,       /* bDescriptorSubType(0x0A): CLOCK_SOURCE */
0x10,       /* bClockID(0x10): CLOCK_SOURCE_ID */
0x01,       /* bmAttributes(0x01): internal fixed clock */
0x07,       /* bmControls(0x07):
                    clock frequency control: 0b11 - host programmable;
                    clock validity control: 0b01 - host read only */
0x00,       /* bAssocTerminal(0x00) */
0x01,       /* iClockSource(0x01): Not requested */
```

2.3.3.3 Input Terminal Descriptor

The Input Terminal Descriptor (ITD) provides information to the host that is related to the functional aspects of the Input Terminal.

In the following code, *wTerminalType* field is 0x0101, which means that the Input Terminal is dealing with signal carried over an endpoint in an AudioStreaming interface; *bCSourceID* field identifies the clock source for this Input Terminal.

The *bNrChannels*, *bmChannelConfig* and *iChannelNames* fields together constitute the cluster descriptor where the audio channel cluster being a group of audio channels that carry tightly related synchronous audio information.

```
/* Input Terminal Descriptor(4.7.2.4) */
0x11,           /* bLength(0x11): 17 */
0x24,           /* bDescriptorType(0x24): CS_INTERFACE */
0x02,           /* bDescriptorSubType(0x02): INPUT_TERMINAL */
0x20,           /* bTerminalID(0x20): INPUT_TERMINAL_ID */
0x01, 0x01,     /* wTerminalType(0x0101): USB streaming */
0x00,           /* bAssocTerminal(0x00) */
0x10,           /* bCSourceID(0x10): CLOCK_SOURCE_ID */
NB_CHANNELS,    /* bNrChannels(0x01) */
0x00, 0x00, 0x00, 0x00, /* bmChannelConfig(0x00): Mono, no spatial location */
0x00,           /* iChannelNames */
0x00, 0x00,     /* bmControls(0x0000) */
0x02,           /* iTerminal(0x02): not requested */
```

2.3.3.4 Feature Unit Descriptor

In following code, *bSourceID* field is 0x20 , which means that the Feature Unit has the Input Terminal as its source; *bmaControls* field is 0x0000_000F which signifies that the user can change or mute the volume for this audio device with its AudioControl requests.

```
/* Feature Unit Descriptor(4.7.2.8) */
0x0E,           /* bLength(0x0E): 6 + (ch + 1) * 4, 1 channel */
0x24,           /* bDescriptorType(0x24): CS_INTERFACE */
0x06,           /* bDescriptorSubType(0x06): FEATURE_UNIT */
0x30,           /* bUnitID(0x30): FEATURE_UNIT_ID */
0x20,           /* bSourceID(0x20): INPUT_TERMINAL_ID */
0x0F, 0x00, 0x00, 0x00, /* bmaControls(0)(0x0000000F): Master Channel 0
                                0b11: Mute read/write
                                0b11: Volume read/write */
0x00, 0x00, 0x00, 0x00, /* bmaControls(1)(0x00000000): Logical Channel 1
*/
0x00,           /* iFeature(0x00) */
```

2.3.3.5 Output Terminal Descriptor

In the following code, *wTerminalType* field is set to 0x0101 for USB Stream Terminal type and *bSourceID* is set to ID of the Feature Unit, which means that this Output Terminal is connected after the Feature Unit.

```
/* Output Terminal Descriptor(4.7.2.5) */
0x0C,           /* bLength(12) */
0x24,           /* bDescriptorType(0x24): CS_INTERFACE */
0x03,           /* bDescriptorSubType(0x03): OUTPUT_TERMINAL */
0x40,           /* bTerminalID(0x40) */
0x01, 0x01,     /* wTerminalType(0x0101): USB_STREAMING */
0x00,           /* bAssocTerminal(0x00): no association */
0x30,           /* bSourceID(0x30): FEATURE_UNIT_ID */
0x10,           /* bCSourceID(0x10): CLOCK_SOURCE_ID */
0x00, 0x00,     /* bmControls(0x0000) */
0x00,           /* iTerminal(0x00): Not Requested */
```

So based on above descriptors, the internal topology of audio speaker is shown in the figure below.

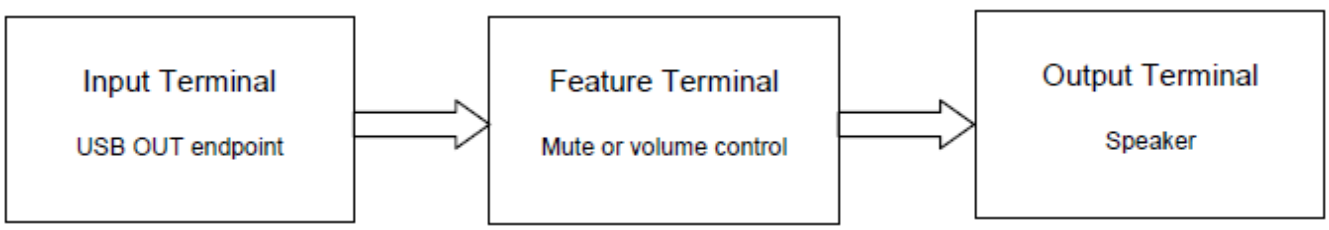

Figure 2. Audio speaker internal topology

2.3.4 Standard AudioStream (AS) Interface Descriptor

In the following code, two alternate settings are defined for this interface.

- Zero-bandwidth setting with its *bNumEndpoints* field set to 0: This setting is used to relinquish the claimed bandwidth on the bus when audio function is not used.
- *bNumEndpoints* field is set to 2: It implies that this interface has both a data endpoint and an explicit feedback endpoint used for synchronization.

```
/* AUDIO STREAMING Interface */
/* Standard AS Interface Descriptor(4.9.1) */
/* Interface 1, Alternate 0 */
/* default alternate setting with 0 bandwidth */
0x09,       /* bLength(9) */
0x04,       /* bDescriptorType(0x04): INTERFACE */
0x01,       /* bInterfaceNumber(0x01) */
0x00,       /* bAlternateSetting(0x00) */
0x00,       /* bNumEndpoints(0x00) */
0x01,       /* bInterfaceClass(0x01): AUDIO */
0x02,       /* bInterfaceSubClass(0x02): AUDIOSTREAMING */
0x20,       /* bInterfaceProtocol(0x20): IP 2.0 */
0x08,       /* iInterface */

/* Standard AS Interface Descriptor(4.9.1) */
/* INterface 1, Alternate 1 */
/* alternate interface for data streaming */
0x09,       /* bLength(9) */
0x04,       /* bDescriptorType(0x04): INTERFACE */
0x01,       /* bInterfaceNumber(0x01) */
0x01,       /* bAlternateSetting(0x01) */
0x02,       /* bNumEndpoints(0x02) */
0x01,       /* bInterfaceClass(0x01): AUDIO */
0x02,       /* bInterfaceSubClass(0x02): AUDIO_STREAMING */
```

```
0x20,       /* bInterfaceProtocol(0x20): IP 2.0 */
0x09,       /* iInterface */
```

2.3.5 Class-Specific AS Interface Descriptor

In the following code, *bTerminalLink* field is set to ID for the Input Terminal, meaning the AudioStreaming interface is connected to the Input Terminal. *bFormatType* and *bmFormats* field together define that the audio data format going through the AudioStreaming interface is commonly used PCM data. The fields *bNrChannels* and *bmChannelConfig* define the physical audio channel cluster in the AS interface, with both left and the right channels.

```
/* Class-Specific AS Interface Descriptor(4.9.2) */
0x10,           /* bLength(16) */
0x24,           /* bDescriptorType(0x024): CS_INTERFACE */
0x01,           /* bDescriptorSubType(0x01): AS_GENERAL */
0x20,           /* bTerminalLink(0x20): INPUT_TERMINAL_ID */
0x00,           /* bmControls(0x00) */
0x01,           /* bFormatType(0x01): FORMAT_TYPE_I */
0x01, 0x00, 0x00, 0x00, /* bmFormats(0x00000001): PCM */
0x02,           /* bNrChannels(0x02): NB_CHANNELS */
0x03, 0x00, 0x00, 0x00, /* bmChannelConfig(0x00000003) */
0x00,           /* iChannelNames(0x00): None */
```

2.3.6 Type I Format Type Descriptor

In following code, it can be seen that the audio samples are in 24 bits, and the sampling rate is 8 kHz.

```
/* Type I Format Type Descriptor(2.3.1.6 - Audio Formats) */
0x06,           /* bLength(6) */
0x24,           /* bDescriptorType(0x24): CS_INTERFACE */
0x02,           /* bDescriptorSubtype(0x02): FORMAT_TYPE */
0x01,           /* bFormatType(0x01): FORMAT_TYPE_I */
0x04,           /* bSubSlotSize(0x01) */
0x18,           /* bBitResolution(0x18): 24 bits per sample */
0x01,                           /* One frequency supported */
0x40,0x1F,0x00,                 /* 8 kHz */
```

2.3.7 Standard AS Isochronous Data EP Descriptor

For the audio speaker demo, it needs an OUT endpoint which receives the audio stream data from the host. In the code, *bmAttributes* is 0x05, which implies that this is an isochronous endpoint, and the synchronization type is asynchronous.

```
/* Standard AS Isochronous Audio Data Endpoint Descriptor(4.10.1.1) */
0x07,       /* bLength(7) */
0x05,       /* bDescriptorType(0x05): ENDPOINT_DESCRIPTOR */
EP01_OUT,   /* bEndpointAddress(0x01) */
0x05,       /* bmAttributes(0x05): iso+asynch+data */
0x08, 0x00, /* wMaxPacketSize(0x0008): 8(8 samples * 1 bytes * 1 channel) */
#if HIGH_SPEED_DEVICE
0x04,       /* bInterval(0x04): 2^x ms */
#else
0x01,       /* bInterval(0x01): 2^x ms */
#endif
```

2.3.8 Class-Specific AS Isochronous Data EP Descriptor

In the followng code, the *bLockDelayUnits* and *wLockDelay* fields are used to indicate to the host how long it takes for the clock recovery circuitry of this endpoint to lock and reliably produce or consume the audio data stream.

```
/* Class-Specific AS IsochronousAudio Data Endpoint Descriptor(4.10.1.2) */
0x08,           /* bLength(8) */
0x25,           /* bDescriptorType(0x25): CS_ENDPOINT */
0x01,           /* bDescriptorSubtype(0x01): EP_GENERAL */
0x00,           /* bmAttributes(0x00): MaxPacketsOnly = FALSE */
0x00,           /* bmControls(0x00) */
0x00,           /* bLockDelayUnits(0x00) */
0x00, 0x00,     /* wLockDelay(0x0000) */
```

2.3.9 Standard AS Isochronous Feedback EP Descriptor

In the following code, the feedback endpoint is an IN endpoint, the *bmAttributes* field is set to 0x11, which means that the transfer type is isochronous and usage type is feedback endpoint.

```
/* Standard AS Isochronous Audio Data Endpoint Descriptor(4.10.1.1) */
 0x07,           /* bLength(7) */
 0x05,           /* bDescriptorType(0x05): ENDPOINT_DESCRIPTOR */
 EP02_IN,        /* bEndpointAddress(0x82) */
 0x11,           /* bmAttributes(0x11): iso+feedback */
 0x04, 0x00,     /* wMaxPacketSize(0x0004) */
#if HIGH_SPEED_DEVICE
 0x04,           /* bInterval(0x04): 2^x ms */
#else
 0x01,           /* bInterval(0x01): 2^x ms */
#endif
```

2.4 Class-specific requests

Class-specific requests are used to set and get audio-related controls. These controls fall into two main groups: those that manipulate the audio function controls such as volume, tone, selector position, etc., and those that influence data transfer over an isochronous endpoint, such as the current sampling frequency.

- AudioControl 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, such as Feature Unit.
- AudioStreaming Requests: Control of the class-specific behavior of an AudioStreaming interface is performed through manipulation of either interface controls or endpoint controls.

2.4.1 Control attributes

Following are the currently defined control attributes for an entity.

- Current setting attribute, to manipulate the current actual setting of a control
- Range attribute, which actually consists of an array of attributes including Minimum, Maximum, and Resolution.

2.4.2 Control request layout

The request layout follows the standard request layout as defined in the USB 2.0 Specification, which can be downloaded from **usb.org**. See the following table.

Table 1. Control request layout

bmRequestType	bRequest	wValue	wIndex	wLength
0010_0001B 1010_0001B	CUR RANGE	Control Selector and Channel Number	Entity ID and interface	Length of parameter block
0010_0010B 1010_0010B			Endpoint	

From Table 1, it can be seen that the request can be directed to either an interface (AudioControl or AudioStreaming) of the audio function, or the isochronous endpoint of an AudioStreaming interface.

The *wValue* field specifies the Control Selector (CS) in the high byte and the Channel Number (CN) in the low byte. The Control Selector indicates which type of control this request is manipulating. The Channel Number (CN) indicates which logical channel of the cluster is to be influenced.

Each different type of Entity or Unit exhibits different type of control requests, for example, for Feature Unit, the user can have the mute control and volume control to the underlying audio stream.

3 Freescale USB Stack

The Freescale USB Stack is divided into several layers to help application developers to concentrate on developing the application instead of being concerned with communications related to low-level USB controller as well as the common framework defined in the USB 2.0 Specification, which can be downloaded from **usb.org**.

Figure 3 is the layered architecture for Freescale USB stack from the point of view of audio class device.

- Application layer: Apart from providing code to implement specific USB function such as audio speaker, the application layer also needs to provide the *usb_descriptor.c* file to inform the host of its capabilities in terms of its descriptors.

- Class layer includes two sublayers, one sublayer for each specific USB device class (such as audio class, CDC class) and includes implementation of class-specific requests; the other sublayer includes what's common among different classes such as registering callback functions for USB events like USB reset, suspend, and resume event.
- The framework layer implements services for the default control pipe and those services comply with Chapter 9 in the USB 2.0 Specification which can be downloaded from **usb.org**.
- The device layer includes code for programming the underlying USB controller for USB communication.

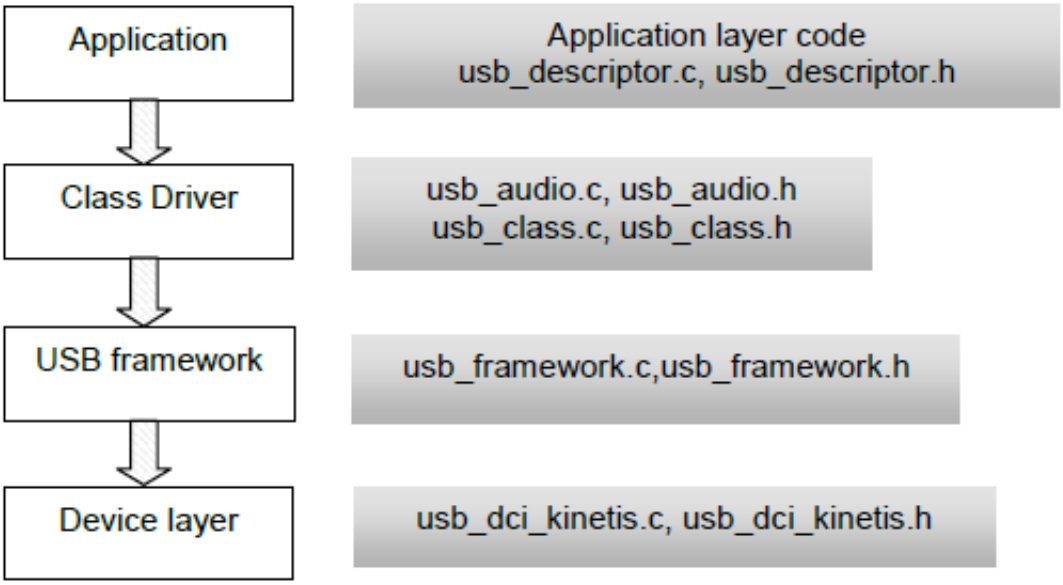

Figure 3. Freescale USB stack architecture

The following table lists the APIs in Freescale USB stack for class layer and USB framework layer.

Table 2. APIs for class layer and framework layer

File name	APIs	Description
usb_audio.c	USB_Class_Audio_Init	• Initialize device layer with *_usb_device_init* • Initialize generic class function with *USB_Class_Init* and register class callback and other request callback function
	USB_Class_Audio_Send_Data	Calls *USB_Class_Send_Data* internally
	USB_Class_Audio_Recv_Data	Calls *_usb_device_recv_data* internally
	USB_Class_Audio_Event	This is the class callback which initializes the audio endpoints when event of completed enumeration is received, then register services for Interrupt pipe and Isochronous pipe.
	USB_Other_Requests	This is the other request callback which handles audio class specific requests such as get and set requests for interface and endpoint.
	USB_Get_Request_Interface USB_Set_Request_Interface	These APIs handle the interface-level request based on the Entity ID and route the request to the appropriate entity. Then, it extracts control selector from the setup packet and call appropriate set or get control function.
	USB_Get_Request_Endpoint USB_Set_Request_Endpoint	These APIs handle end-point level request, extract control selector from the setup packet and call appropriate set or get control function.

Table continues on the next page...

Table 2. APIs for class layer and framework layer (continued)

File name	APIs	Description
usb_class.c	USB_Class_Init	Calls *USB_Framework_Init* internally, register services for USB events such as Bus Reset, SOF, Suspend, Resume and Stall.
	USB_Class_Send_Data	Calls *_usb_device_send_data* internally
usb_framework.c	USB_Framework_Init	Register service for default control pipe.
	USB_Control_Service	Handles standard USB requests or class specific requests.

4 Audio class demos

There are existing demos for audio device class in Freescale USB Stack version 4.0.3, which can be downloaded from **freescale.com**. Those demos can be found under Freescale USB Stack v4.0.3\Source\Device\app, after installing the package. Two demos are provided: audio generator and audio speaker.

4.1 Audio speaker

USB descriptor for audio speaker is already discussed in Audio class device requirements. The main flow for the audio speaker demo is shown in the code below, where FTM0_CH0 is set to output audio data. The user needs to connect a low-pass filter and a microphone externally to hear the audio sent from PC. To setup the demo, follow instruction from Appendix G: USB Audio Demo in the Freescale USB Device Stack Users Guide, available on **freescale.com**.

Then, it calls *USB_Class_Audio_Init* API which initializes the USB controller through device layer API and register callback functions for the application layer; the exact callback is *USB_App_Callback*. This is where the user writes code to respond to different events such as *USB_APP_ENUM_COMPLETE*, *USB_APP_DATA_RECEIVED*, or *USB_APP_SEND_COMPLETE* received from lower layer.

```
void TestApp_Init(void)
{
     sci_init(); //initialize default console for output
     pit1_init(); //initialize PIT timer for 0.1ms timeout
     pwm_init(); //initialize FTM0_CH0 (PTC1) to output the audio signal
     error = USB_Class_Audio_Init(CONTROLLER_ID,USB_App_Callback,
                                  NULL,NULL);
}
```

USB_App_Callback uses the event_type parameter to determine which event occurred and react accordingly. When it gets a *USB_APP_DATA_RECEIVED* event, it will copy the audio data to a local buffer *audio_data_recv* where those data will finally be used to update the PWM duty cycle for FTM0_CH0 in the PIT timer ISR. See the following code.

```
static void USB_App_Callback (
    uint_8 controller_ID, /* [IN] Controller ID */
    uint_8 event_type, /* [IN] value of the event */
    void* val /* [IN] gives the configuration value */
)
{
        if(event_type == USB_APP_BUS_RESET)
         {
             start_app=FALSE;
         }
         else if(event_type == USB_APP_ENUM_COMPLETE)
```

```
        {
            start_app=TRUE;
    #ifdef USE_FEEDBACK_ENDPOINT
           // Send initial rate control feedback (48Khz)
           USB_Class_Audio_Send_Data(controller_ID, AUDIO_FEEDBACK_ENDPOINT,
                          (uint_8_ptr)&feedback_data,
AUDIO_FEEDBACK_ENDPOINT_PACKET_SIZE);
    #endif // USE_FEEDBACK_ENDPOINT
           …
        }
        else if ((event_type == USB_APP_DATA_RECEIVED) && (TRUE == start_app))
        {
            (void)USB_Class_Audio_Recv_Data(controller_ID, AUDIO_ENDPOINT,
                            (uint_8_ptr)g_curr_recv_buf,
AUDIO_ENDPOINT_PACKET_SIZE);
            audio_event = USB_APP_DATA_RECEIVED;
            data_receive = (APP_DATA_STRUCT*)val;
             (void)memcpy(audio_data_recv, data_receive->data_ptr, data_receive->data_size);
        }
    #ifdef USE_FEEDBACK_ENDPOINT
        else if((event_type == USB_APP_SEND_COMPLETE) && (TRUE == start_app))
        {
           feedback_data <<= 14; // 10.14 format
           (void)USB_Class_Audio_Send_Data(controller_ID,
                            AUDIO_FEEDBACK_ENDPOINT,
                                    (uint_8_ptr)&feedback_data,
                                    AUDIO_FEEDBACK_ENDPOINT_PACKET_SIZE);
         }
    #endif
}
```

5 Conclusion

Freescale USB Stack provides a good framework to develop the USB applications. It provides several layers to hide the low level details of USB data communication and many existing demos for common USB device classes where the user can adapt for his own use.

To develop an Audio Class Device, the user needs to know the internal topology of this device, the number of Terminals and Units it has, their interconnection, the number of interfaces the device has, and whether it's a standalone or a composite device. Then the user can write down the USB descriptors for the device following the examples in Freescale USB stack.

Another thing the user needs to do is to write the application callback where it responds to events sent by the lower layer and implement the unique functionality of the audio device.

6 References

The following reference documents are available on **usb.org**.

- USB 2.0 Specification
- USB Device Class Definition for Audio Devices
- USB Device Class Definition for Terminal Types
- USB device Class Definition for Audio Data Formats

The following reference documents are available on **freescale.com**.

- USBUG: USB Stack Users Guide
- USBAPIRM: Freescale USB Stack with PHDC Device API—Reference Manual

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Document Number: AN4665
Rev. 0, 01/2013