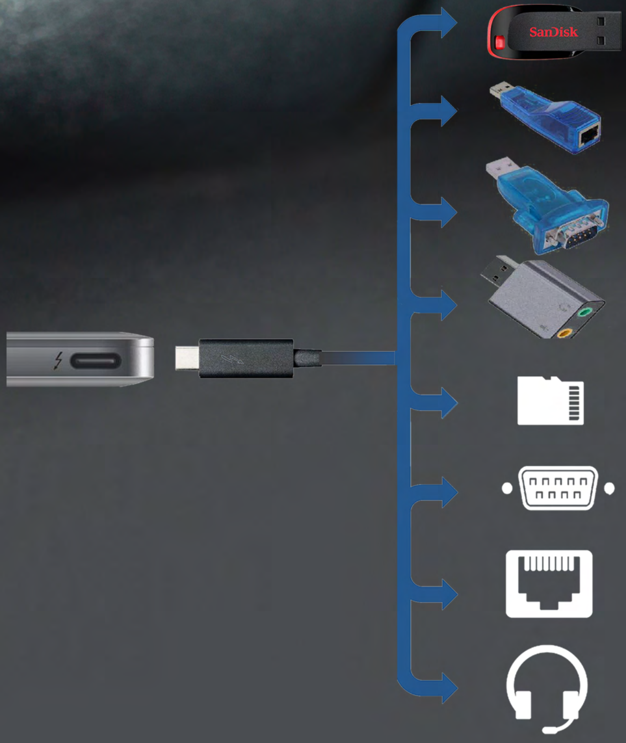
einfochips
An Arrow Company

ARROW
Five Years Out



Guide for Developing USB Composite Device for STM32 Hardware Platform

www.einfochips.com

Contents

1. Overview

The USB interface is extraordinarily complex, multi-level and multi-faceted. Anyone can be easily swayed by the immensity of the information to digest while working on a USB device. The specification of the USB device along with all the respective “Class” devices is itself more than 1000 pages. Preparing the USB device driver from scratch and providing support for some class-specific USB interfaces require a lot of work and time. However, many manufacturers provide a sophisticated ecosystem for their product family which eases out the implementation by providing the framework or library. This document covers the aspects of developing a USB Composite Device for STM32-based hardware using the existing STM32CubeIDE and STM32 USB Device Driver Library.

The ST provides a very sophisticated IDE integration to develop the USB device driver. The STM32CubeIDE tool supports the auto-generation of the code for the USB device for an individual Class. It provides support for many different class devices. But the tool does not support the creation of a composite device. However, once you digest the existing architecture of the USB device driver Library provided by ST electronics along with a few aspects of the USB Class and Descriptor specification, it will be much easier to prepare a Composite Device using the existing code from ST electronics. It will save a lot of time and work to prepare the composite device. Hence, to develop a Composite Device Driver for the STM platform, the audience is requested to at least have a basic understanding of USB device, Composite Device and STM32 USB Library Code architecture.

The document tries to cover the maximum audience who are interested in USB composite device drivers for the STM32 platform. Hence, all the necessary components are described with bare minimum details in the basic details section. The document is divided into three parts:

- Basic Details related to the components of the USB Composite Device Driver Development
- Implementation of the USB Composite Device Driver for the STM32 platform
- Debugging and integrating the USB Composite Device Driver with the Application Layer

All individual aspects of the Composite Device Driver are sufficient to write a full-scale book. The details provided in this document are kept bare minimum but sufficient to keep the audience engaged, active and interested. A working USB composite device of RNDIS + Audio + Virtual Com Port for High-Speed Interface on the STM32H7XX series of processors has been created using these guidelines. The same example is followed throughout the document. However, the audience can perform any combination of class drivers for their required Composite Device using the provided guidelines.

2. Basics of Required Dependencies

2.1 What is a composite USB device?

Usually, one USB device implements only one functionality of any one class type. As an example, a USB device can be either Mass Storage (Thumb Drive), Audio Device (USB Headphone/Microphone), Mouse (CDC), or Serial Com Port (CDC). However, a Composite USB Device incorporates multiple functionalities in a single USB device. As an example, RNDIS + Audio + Virtual Comport + Mass storage devices are accessed through a single USB port of the target hardware.

A composite device-specific detailed information is not available in the USB specification. However, it can be abstracted as a part of the Interface Association Descriptor Specification. There is little information available in the community related to USB composite device development. It becomes exceedingly difficult to figure out what to explore and understand in specifications to develop a healthy and host-compatible composite USB device. The Composite device is no more different than an individual USB class device in terms of the USB specification. The ability to differentiate the functionality and their associated interfaces are the key to developing the USB composite Device. The major class driver can be incorporated into the composite device. However, the user needs to take care of the bandwidth requirements and performance of each functionality while creating the composite device. As an example, a Mass Storage Class in Full

Speed USB device will suffer a lot in throughput if audio and video device class drivers are incorporated into the USB device as a composite device.

2.2 USB specification and descriptors

The USB specification is a noticeably big document. It covers many things for a variety of supported USB class interfaces. However, most of the specifications are implemented readily by the OEM, and the audience is not required to go through all the details.

The following are the important descriptors that need to be understood well to prepare a composite device. The Composite Device needs these descriptors properly defined to get enumerated and work with a host system.

Device descriptor

It describes the device. Its name, manufacturer, serial number, and other device-specific information. Each USB device should have only one Device Descriptor. String data is described by separate string descriptors (String Descriptor).

Configuration descriptor

It describes various supported configurations of the USB device. A device can have one or more configurations. Each configuration determines the speed of communication with the device, a set of interfaces and power settings. So, for example, a USB device can have multiple configurations to support different sampling frequencies. An all-in-one printer may have different configurations to support different speeds and interfaces for communication interfaces.

Interface descriptor

It describes the interface of communication with the device. There can be several interfaces. For example, different functions (MSC, CDC, HID) will implement their interfaces. Some functions (for example, CDC or DFU) implement several interfaces at once for their work. In our case of a composite device, we will need to implement several interfaces from different functions at once and make them have a good relationship with each other.

Endpoint descriptor

It describes the communication channel within a specific interface, sets the packet size and describes the parameters of interrupts. Using endpoints, the actual data will be transmitted or received.

There are various fields and attributes for each Descriptor. Each attribute and field have incredibly significant information to convey. Even a one-bit error in any of the descriptor fields can lead to the failure of the USB device or its functional interface enumeration. Further details of them should be explored in the “Descriptors” Section of the “USBnutshell” Guideline. Every individual interested in a Composite USB device must at least go through the “USBnutshell” Guideline once to understand the concepts and details of important aspects of USB specification more simply.

Interface association interface (IAD)

It describes the association of the interface with functionality. It is a mandatory descriptor for the composite device. For a composite device, each functionality has its IAD descriptor to provide the associated interface number, class, subclass, and device protocol type. A device configuration can have more than one IAD. Each IAD must be located immediately before the interfaces in the interface group that the IAD describes. The following figure from the specification is enough to understand the IAD usage for the composite USB device.

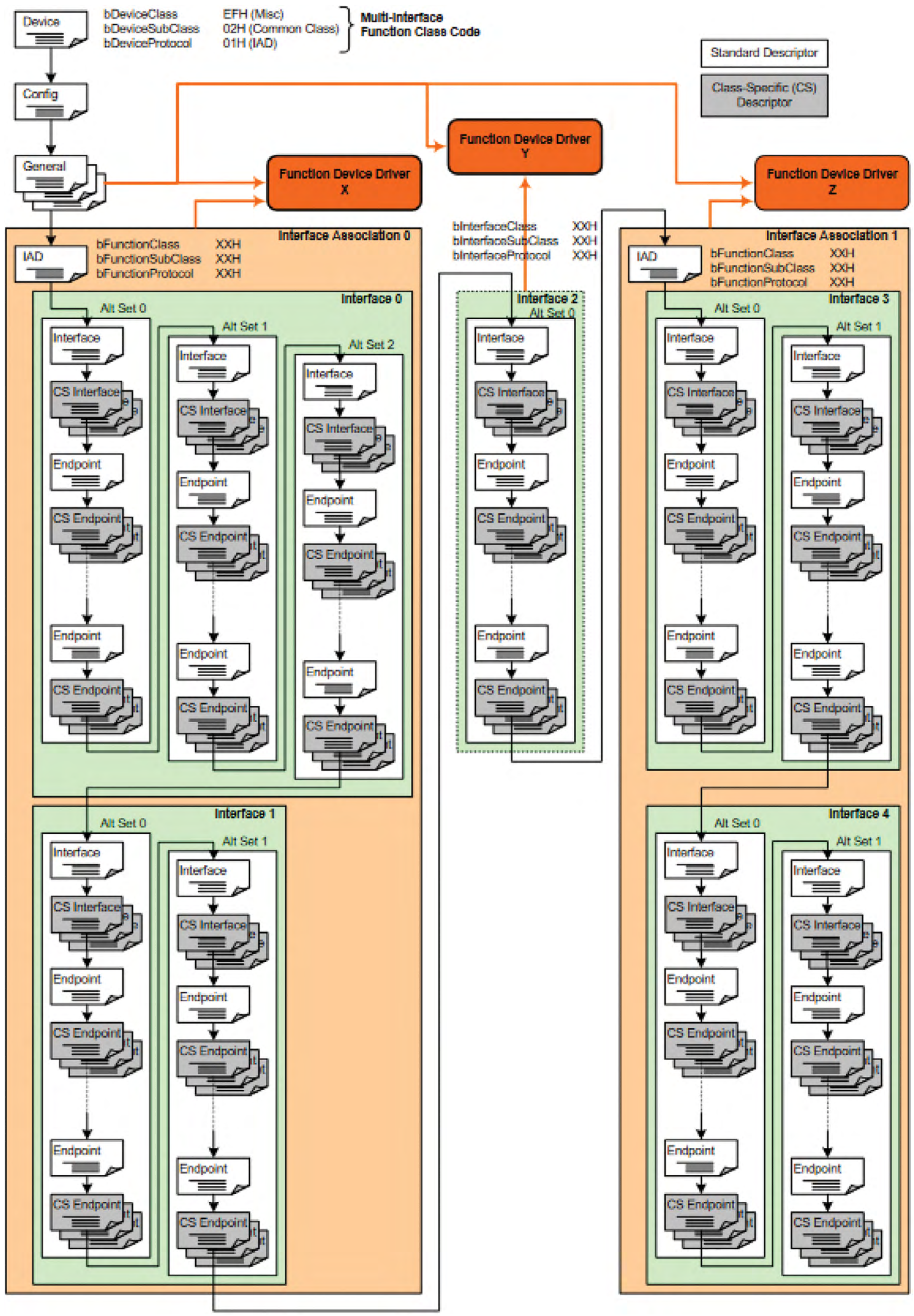


Figure 2.1. Example Device Framework Using Interface Association Descriptors

More information on the Interface Association Descriptor can be accessed here.

The complete USB specification by the OEM can be searched and accessed here.

2.3 USB driver for host system and individual device

The USB interface is host-based. The host initiates the request to access the device. Hence, there is a driver required to initiate and handle the request as per the USB specification in the host system. There is also a driver required on the device side to receive the request from the host system and provide the appropriate response to the request. The functionality on the host side is different than the functionality on the device side. This document only provides information on preparing the USB interface on the device side. The host-side driver is usually available in all modern-day operating systems such as Windows, Mac OS, Linux and others. However, it is observed that each operating system has its compatibility specifications. They also have their own set of rules on top of the USB specification for USB device operations. It has been observed during the composite USB device development that a final and complete composite USB device works well with the Linux host system. However, the Windows system fails to even enumerate either the entire device or a part of its functionality.

The reason to include this topic in this document is to make the audience aware that it is especially important to understand the architecture and support of the host system when a device driver is being developed for the Composite Device. In case of failure to enumerate the work of the developed USB composite device's interface in the intended host system may not necessarily be an issue with the USB specification implementation at the target device driver side. The target device driver implementation can be perfect as per the USB specification and it can still fail to work with the specific host system. This should further be diagnosed and debugged through the tools and techniques available for the targeted host system. It has been observed that Linux is the most versatile and preferable development environment for USB device driver development. The host driver of Linux has better and wider class support. The user also has the flexibility to customize and change it for the development of the targeted device driver. Linux provides a lot of inbuilt and free diagnostic and debugging tools to verify the USB composite device implementation.

When all individual functional interfaces of the respected composite device are working well with the host system, there is no change or update required in the USB host driver or at the host system for the respective composite device with the same interfaces. As an example, a composite device having one Virtual Com Port and One Mass Storage interface does not need anything specific to be implemented or taken care of on the host system side. Such a Composite Device will enumerate on its own without any difficulty and works fine. The system will show two different functionalities under one USB device. Usually, all operating systems support a Mass Storage Device and a Serial Com Port Device individually so the Composite Device of these two interfaces will work without any difficulties in the host system. For the development of a Composite USB device, there is a possibility that the intended host system does not support the class functionality of the device-specific requirements. In such cases, the host driver for the required class should also be developed for the respective operating system.

2.4 STM32CubIDE

STM32CubeIDE is an exceptionally good integrated development tool for the STM-based processor. It allows the users to have a ready initialization of most of the interfaces and functionality. It provides a ready infrastructure for all the on-chip interfaces. Most of the time the user only needs to develop the application-specific interface code. The example Composite Device of the RNDIS + Audio + Virtual Com Port is created through this tool.

It is included in this document for ease of the user to have a readily available user interface file for the individual USB interface of the composite device. The user is requested to generate a USB interface functionality using the tool for all the required interfaces of the composite device.

There are particularly good user video guides available by the OEM for the tool. Visit the following links to know more about the STM32CubeIDE:

https://youtu.be/eumKLXNlM0U

https://www.youtube.com/playlist?list=PLnMKNibPkDnFCosVVv98U5dCulE6T3Iy8

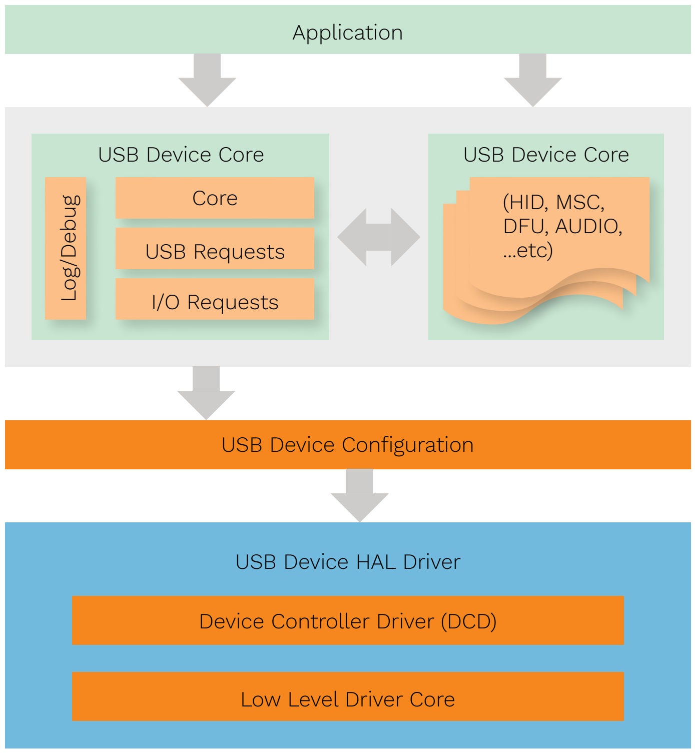
2.5 STM32 USB device library architecture

The USB Device Library from the ST is well-organized and properly layered. The following is the summary of the architecture levels and their respective functionality.

- **Class Driver:** They implement the logic of a specific class of devices - CDC for virtual COM port, MSC for storage devices, HID for keyboards/mice and any specific device with a user interface.
- **USB Core (usbd_core.c, usbd_ctlreq.c, usbd_ioreq.c):** It implements the general logic of operation of all classes of USB devices. It can send the requested descriptors to the host, process requests from the host and configure the USB device. It also redirects data streams from the class driver level to the underlying levels and vice versa.
- **USB HW Driver (usbd_conf.c):** The overlying layers are platform-independent and work the same way for several series of microcontrollers. The code does not have a low-level function that calls for a specific microcontroller. The usbd_conf.c file implements a layer between the USB Core and HAL, a library of low-level drivers for the selected microcontroller. There are simple wrappers that redirect calls from top to bottom and callbacks from bottom to top.
- **HAL (stm32f1xx_hal_pcd.c, stm32f1xx_ll_usb.c):** It is engaged in communication with the microcontroller hardware, operates with registers and responds to interrupts.

The following is a layout summary of the USB library layered architecture:

- **USB Device file** - Init and De-Init functionality
- **Application Layer Interface File** – usbd_class_if.c to be prepared by the User
- **Class Driver** – Individual Class driver available; Composite Class driver to be prepared by the User
- **USB Core** – some code changes are required for the Composite class
- **USB HW Driver** – Update Endpoint FIFO allocation for the Composite class
- **HAL** – No change is required for the Composite class



Reference:
stm32cube-usb-device-library-stmicroelectronics

There are also incredibly good user video guides available by the OEM for the USB device library architecture. Visit the following links if interested to know more about the STM-provided USB Device Library stack.

https://youtu.be/I1HfAkz-brc

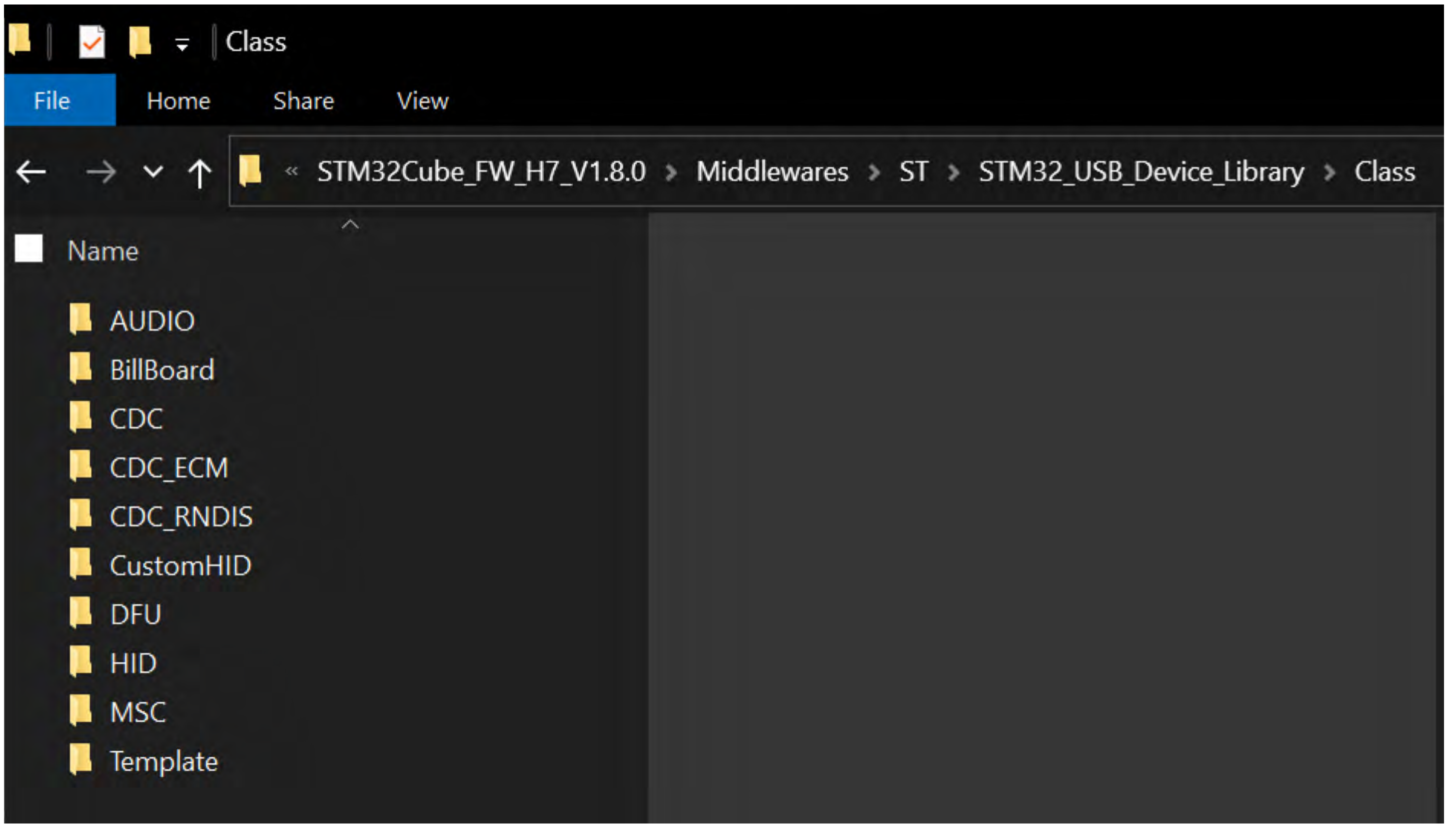
https://www.youtube.com/playlist?list=PLnMKNibPkDnFFRBVD206EfnnHhQZI4Hxa

2.6 STM32 USB device library class drivers

The ST OEM provides support for most of the generic device classes. The respective code can be found in the GIT repository easily. If the STM32CubeIDE is installed and for one of the class USB device codes is generated through the tool, then the entire repository is readily available on the local system at the following locations in the Windows system.

%UserProfile%\STM32Cube\Repository

For the composite device project, the Firmware for the H7 series with revision 1.8.0 was used. It was having the following supported class drivers available locally at "%UserProfile%\STM32Cube\Repository\STM32Cube_FW_H7_V1.8.0\Middlewares\ST\STM32_USB_Device_Library\Class."


Class
File
Home
Share
View
STM32Cube_FW_H7_V1.8.0 › Middlewares › ST › STM32_USB_Device_Library › Class
Name

AUDIO
BillBoard
CDC
CDC_ECM
CDC_RNDIS
CustomHID
DFU
HID
MSC
Template

Each class driver has its configuration descriptor and class-specific interface API pointers to be registered with the Core Library.

The Class Driver File usually has the following coding architecture:

- Class-specific Interface defining structure followed by
- Class-specific configuration descriptor followed by
- Class-specific interface function implementation

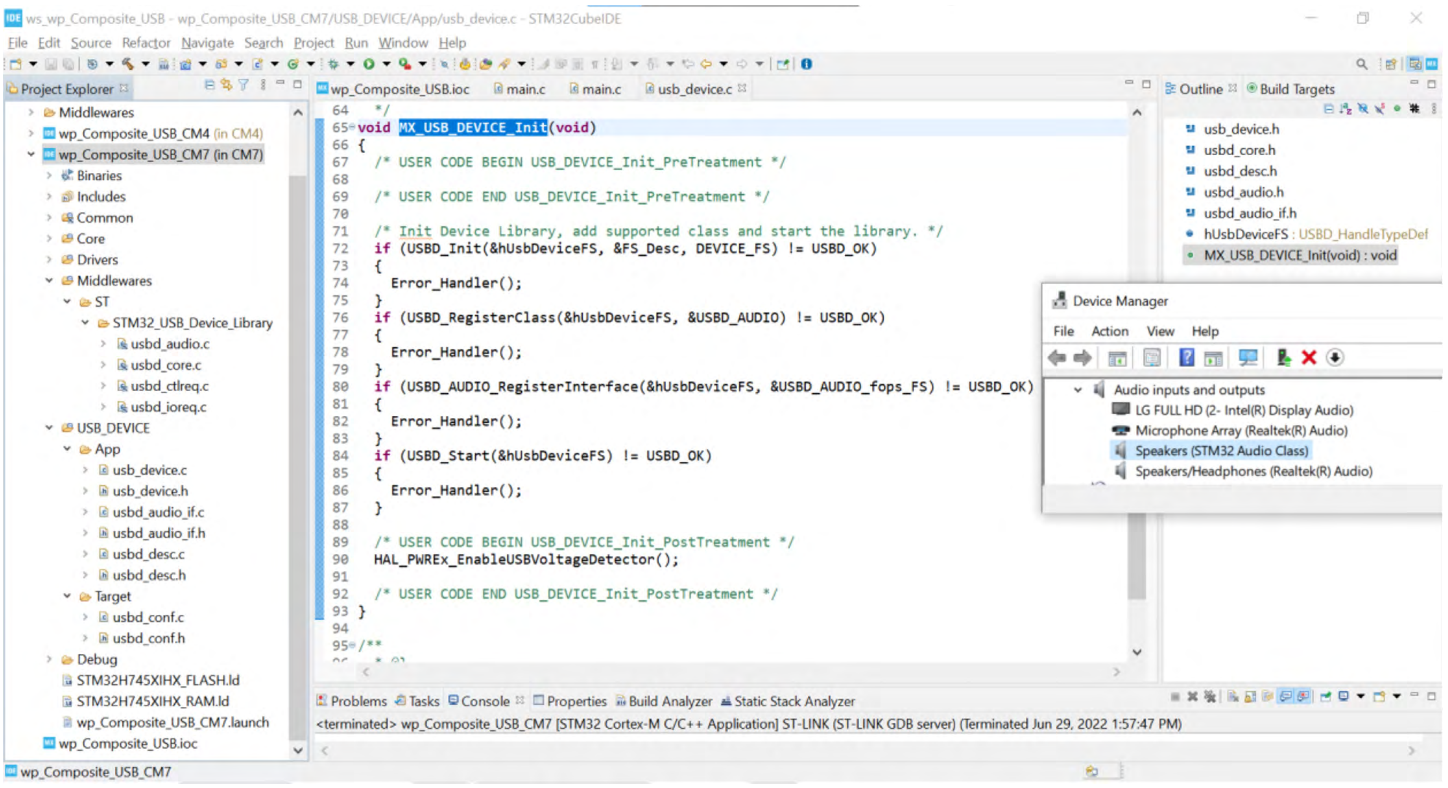
Note: The user can also find example projects of the USB device implementation in the repository. As an example, for the STM32H745 Discovery board, the example code for the DFU and HID standalone USB device can be found at -

"%UserProfile%\STM32Cube\Repository\STM32Cube_FW_H7_V1.8.0\Projects\STM32H745I-DISCO\Applications\USB_Device" location. The example program will provide a base code for the application-level interface integration for the USB device.

3. Implementation of the USB Composite Device Driver for the STM32 Platform

3.1 Architecture of the default USB device with a single class

The STM32CubeIDE auto-generated code for the Audio Class Device for the STM32H745I Disc board shows the following results when the USB device is plugged into the Windows system.



ws_wp_Composite_USB - wp_Composite_USB_CM7/USB_DEVICE/App/usb_device.c - STM32CubeIDE

```
Project Explorer
  > Middlewares
  > wp_Composite_USB_CM4 (in CM4)
  v wp_Composite_USB_CM7 (in CM7)
    > Binaries
    > Includes
    > Common
    > Core
    > Drivers
    v Middlewares
      v ST
        v STM32_USB_Device_Library
          > usbd_audio.c
          > usbd_core.c
          > usbd_ctlreq.c
          > usbd_ioreq.c
    v USB_DEVICE
      v App
        > usb_device.c
        > usb_device.h
        > usbd_audio_if.c
        > usbd_audio_if.h
        > usbd_desc.c
        > usbd_desc.h
      v Target
        > usbd_conf.c
        > usbd_conf.h
    > Debug
    STM32H745XIHX_FLASH.ld
    STM32H745XIHX_RAM.ld
    wp_Composite_USB_CM7.launch
  wp_Composite_USB.ioc
```

```c
  */
void MX_USB_DEVICE_Init(void)
{
  /* USER CODE BEGIN USB_DEVICE_Init_PreTreatment */

  /* USER CODE END USB_DEVICE_Init_PreTreatment */

  /* Init Device Library, add supported class and start the library. */
  if (USBD_Init(&hUsbDeviceFS, &FS_Desc, DEVICE_FS) != USBD_OK)
  {
    Error_Handler();
  }
  if (USBD_RegisterClass(&hUsbDeviceFS, &USBD_AUDIO) != USBD_OK)
  {
    Error_Handler();
  }
  if (USBD_AUDIO_RegisterInterface(&hUsbDeviceFS, &USBD_AUDIO_fops_FS) != USBD_OK)
  {
    Error_Handler();
  }
  if (USBD_Start(&hUsbDeviceFS) != USBD_OK)
  {
    Error_Handler();
  }

  /* USER CODE BEGIN USB_DEVICE_Init_PostTreatment */
  HAL_PWREx_EnableUSBVoltageDetector();

  /* USER CODE END USB_DEVICE_Init_PostTreatment */
}

/**
```

```
Outline
  usb_device.h
  usbd_core.h
  usbd_desc.h
  usbd_audio.h
  usbd_audio_if.h
  hUsbDeviceFS : USBD_HandleTypeDef
  MX_USB_DEVICE_Init(void) : void
```

```
Device Manager
v Audio inputs and outputs
    LG FULL HD (2- Intel(R) Display Audio)
    Microphone Array (Realtek(R) Audio)
    Speakers (STM32 Audio Class)
    Speakers/Headphones (Realtek(R) Audio)
```

```
<terminated> wp_Composite_USB_CM7 [STM32 Cortex-M C/C++ Application] ST-LINK (ST-LINK GDB server) (Terminated Jun 29, 2022 1:57:47 PM)
```

File Edit Source Refactor Navigate Search Project Run Window Help
wp_Composite_USB.ioc
main.c
main.c
usb_device.c
Build Targets
File Action View Help
Problems Tasks Console Properties Build Analyzer Static Stack Analyzer
wp_Composite_USB_CM7

The Audio Driver is ready to enumerate and accept data from the Windows host system. The user needs to consume the data available through the Audio class interfaces as per the application needs of the target system.

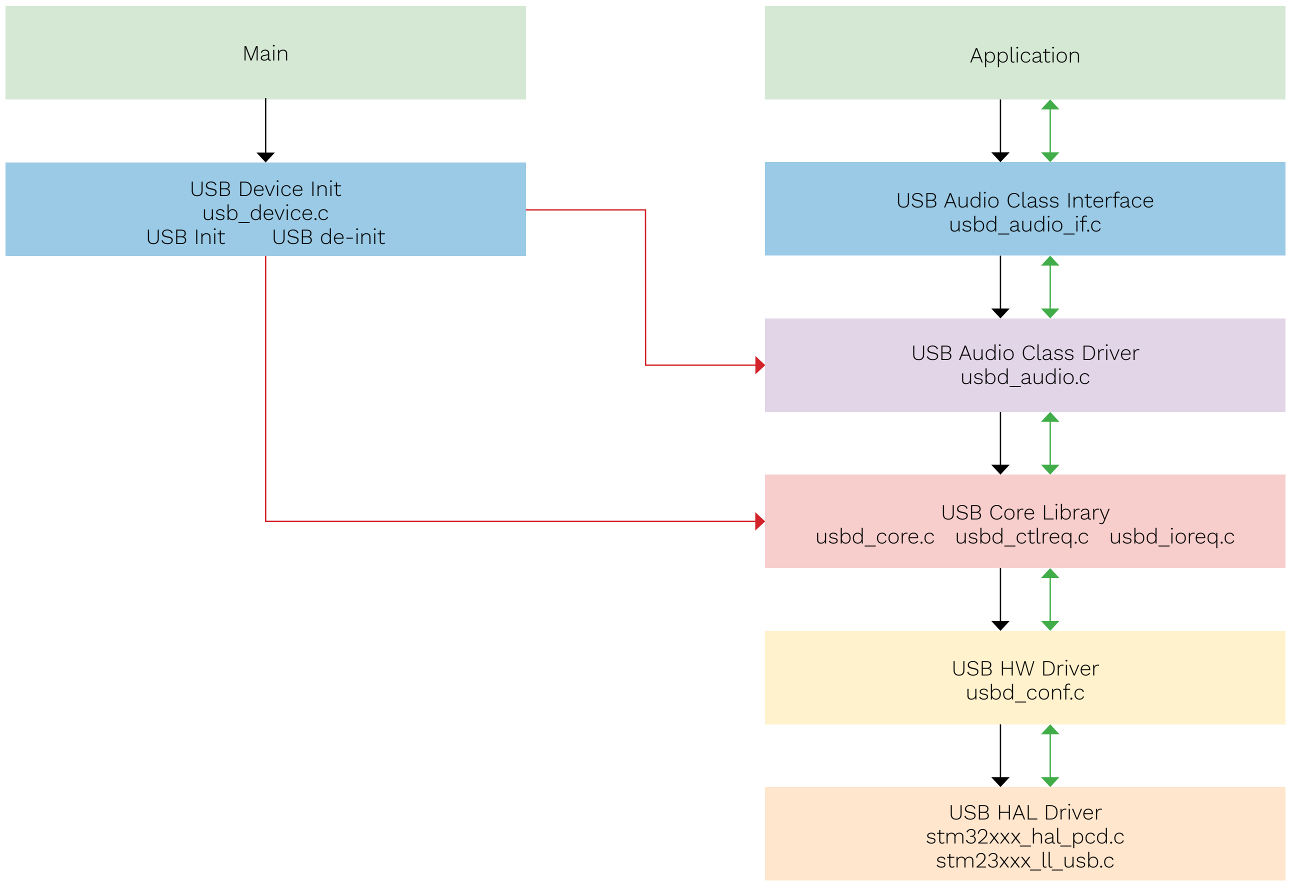
Similarly, the virtual Com Port's respective USB code base can also be easily created and tested out for enumeration.

The Control and code flow for the USB module is as per the following diagram.

Within the main function, the USB device initialization MX_USB_DEVICE_Init from the usb_device.c is called. It performs the basic USB initialization process. Once the USB Device and dependencies are initialized and the USB device is attached to the host system, the enumeration process gets initiated.

The enumeration process follows the host-to-device control flow. Where the host requests information and the device provide it.

The required information during enumeration is available as part of the descriptor and class driver code in usbd_audio.c file.



The Usb_device.c file has the following important components to understand:

```
void MX_USB_DEVICE_Init(void)
{
 /* USER CODE BEGIN USB_DEVICE_Init_PreTreatment */

 /* USER CODE END USB_DEVICE_Init_PreTreatment */

 /* Init Device Library, add supported class and start the library. */
 if (USBD_Init(&hUsbDeviceFS, &FS_Desc, DEVICE_FS) != USBD_OK)
 {
  Error_Handler();
 }
 if (USBD_RegisterClass(&hUsbDeviceFS, &USBD_AUDIO) != USBD_OK)
 {
  Error_Handler();
 }
 if (USBD_AUDIO_RegisterInterface(&hUsbDeviceFS, &USBD_AUDIO_fops_FS) != USBD_OK)
 {
  Error_Handler();
 }
 if (USBD_Start(&hUsbDeviceFS) != USBD_OK)
 {
  Error_Handler();
 }

 /* USER CODE BEGIN USB_DEVICE_Init_PostTreatment */
 HAL_PWREx_EnableUSBVoltageDetector();

 /* USER CODE END USB_DEVICE_Init_PostTreatment */
}
```

USBD_RegisterClass (&hUsbDeviceFS, &USBD_AUDIO)

It is in usb_core.c file which registers the function pointer structure of the Audio Class with the USB Device Structure.

USBD_AUDIO_RegisterInterface (&hUsbDeviceFS, &USBD_AUDIO_fops_FS)

Is in usbd_audio.c file for the audio class driver which registers the structure pointer of the Audio Class User Interface with the USB Device Structure.

Usbd_audio.c file has the following important components and layouts to process all the audio class-related functionalities.

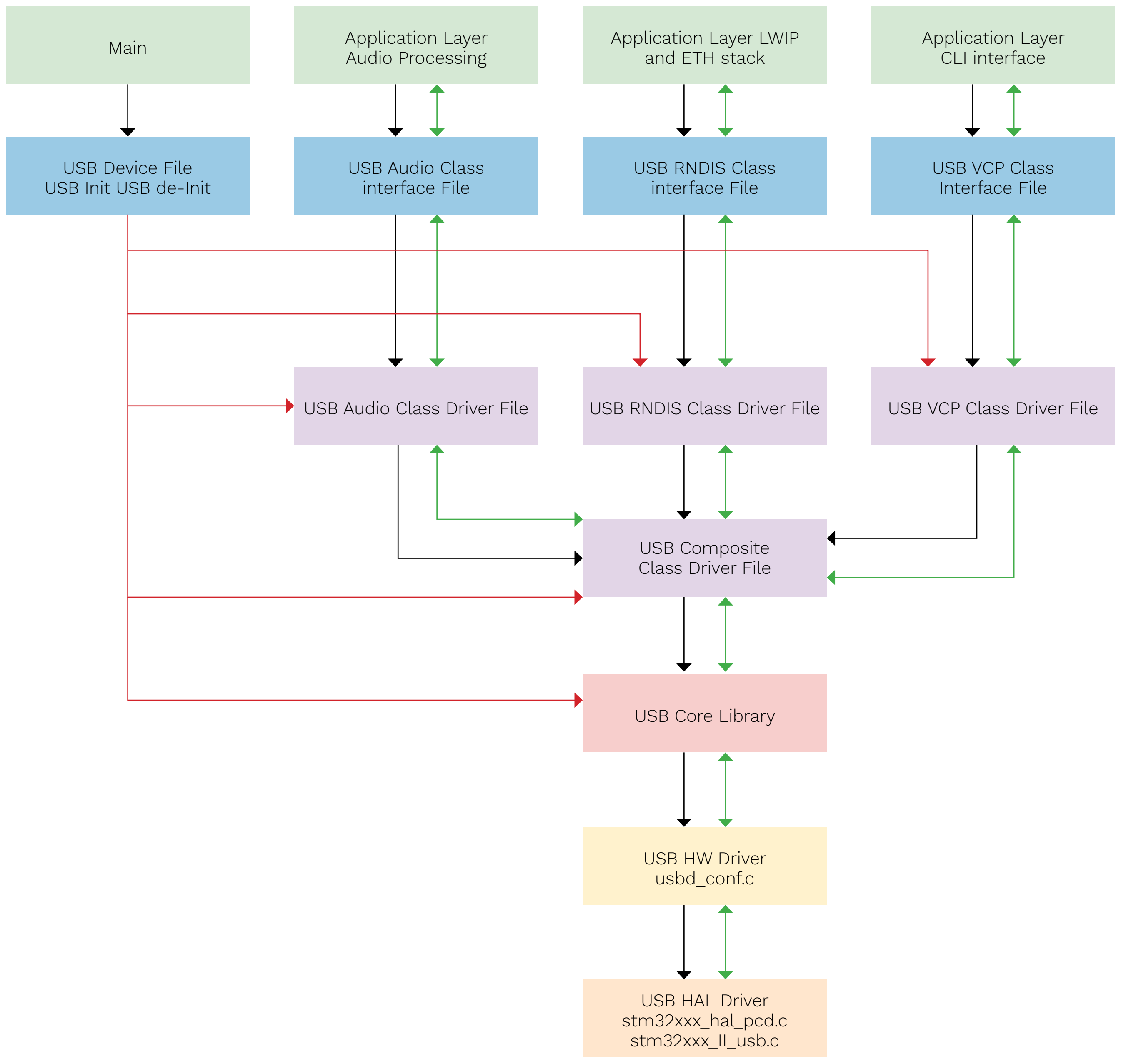
```
USBD_ClassTypeDef USBD_AUDIO =
{
  USBD_AUDIO_Init,
  USBD_AUDIO_DeInit,
  USBD_AUDIO_Setup,
  USBD_AUDIO_EP0_TxReady,
  USBD_AUDIO_EP0_RxReady,
  USBD_AUDIO_DataIn,
  USBD_AUDIO_DataOut,
  USBD_AUDIO_SOF,
  USBD_AUDIO_IsoINIncomplete,
  USBD_AUDIO_IsoOutIncomplete,
  USBD_AUDIO_GetCfgDesc,
  USBD_AUDIO_GetCfgDesc,
  USBD_AUDIO_GetCfgDesc,
  USBD_AUDIO_GetDeviceQualifierDesc,
};

/* USB AUDIO device Configuration Descriptor */
__ALIGN_BEGIN static uint8_t USBD_AUDIO_CfgDesc[USB_AUDIO_CONFIG_DESC_SIZ] __ALIGN_END =
{
 .........
}
 /* USB Standard Device Descriptor */
__ALIGN_BEGIN static uint8_t USBD_AUDIO_DeviceQualifierDesc[USB_LEN_DEV_QUALIFIER_DESC] __ALIGN_END =
{
......

}

function declarion of the Audio class
static uint8_t USBD_AUDIO_Init ( .... )
....
static uint8_t USBD_AUDIO_DeInit ( .... )
....
static uint8_t USBD_AUDIO_Setup ( .... )
....
static uint8_t USBD_AUDIO_EP0_TxReady ( .... )
....
static uint8_t USBD_AUDIO_EP0_RxReady ( .... )
....
static uint8_t USBD_AUDIO_DataIn ( .... )
....
static uint8_t USBD_AUDIO_DataOut ( .... )
....
static uint8_t USBD_AUDIO_SOF ( .... )
....
static uint8_t USBD_AUDIO_IsoINIncomplete ( .... )
....
static uint8_t USBD_AUDIO_IsoOutIncomplete ( .... )
....
static uint8_t USBD_AUDIO_GetCfgDesc ( .... )
....
static uint8_t USBD_AUDIO_GetCfgDesc ( .... )
....
static uint8_t USBD_AUDIO_GetCfgDesc ( .... )
....
static uint8_t USBD_AUDIO_GetDeviceQualifierDesc ( .... )
....
```

3.2 Architecture for the composite USB device with multiple class

The proposed solution for the composite device can be easily visualized as per this architecture diagram. The “application layer” as well as the “application interface layer” remain unchanged. The HAL driver components also remain unchanged.

A new composite class (“SUDO”) driver in between the USB core library and individual class drivers is placed. The idea is to have the new composite class driver as a switching portal. As each Class Functionality will have its associated Interfaces and Endpoints, the Composite driver should be able to redirect the request to the individual class driver based on the Interface Number or Endpoint number in the argument.

To create the Composite USB, all the components connected with the Red Colored control flow arrows in the diagram need to be modified, upgraded, or created. The required changes are explained in detail in the following section.


Main
Application Layer Audio Processing
Application Layer LWIP and ETH stack
Application Layer CLI interface
USB Device File USB Init USB de-Init
USB Audio Class interface File
USB RNDIS Class interface File
USB VCP Class Interface File
USB Audio Class Driver File
USB RNDIS Class Driver File
USB VCP Class Driver File
USB Composite Class Driver File
USB Core Library
USB HW Driver usbd_conf.c
USB HAL Driver stm32xxx_hal_pcd.c stm32xxx_ll_usb.c

Here is the list of things to do in code for the composite USB device development.

- Organize and gather the USB device respective all the files in a single directory with a proper hierarchy and segregation. Rename the files appropriately if needed.
- Create a new set of source files for the Composite class driver from the templates.
- Prepare the USB composite device interface and endpoint footprint for the configuration descriptor.
- Update the USB core library code to hold the multiple “class interface” structures and handle requests based on the new parameters.
- Prepare the new composite class driver file.
  - Prepare the composite class configuration descriptor by copying and appending configuration descriptors from the individual class drivers.
    - Add the IAD descriptor before each class descriptor group.
    - Update the class grouped descriptor for the interface number and Endpoint number.
    - Update the descriptor for the total number of interfaces, Eps, and the total size of the conf descriptor.
  - Prepare the Composite Class Driver APIs to redirect the request from the core library to the individual class driver.
- Update the individual Class Drivers files to support redirecting functionality of the Composite Driver.
  - Remove the structure holding the function pointers of the class driver.
  - Remove the configuration and another descriptor for the class.
  - Make all the class interface functions global.
  - Find and replace the “class interface structure” (pClassData and pUserData)pointer that is renamed as per the core library changes.
- Implement Logic for the EPO_RxReady request to identify the associated individual class as there is no Interface Number or EndPoint Number available in the argument.
  - USBD_CtlPrepareRx Look for the individual class driver – set a global flag that the next EP0 receives a request for this class driver.
- Update the usb_device.c file
  - Register the new composite class driver APIs and all the required “class interface” APIs structure pointers.
- Prepare the usbd_desc.c file -
  - Update the USB standard device descriptor.
- Update the usbd_conf.c file –
  - USBD_LL_Setup_Fifo() function to allocate the proper FIFO for each endpoint as per the new Endpoint Layout.
  - Update the USBD_MAX_NUM_INTERFACES to have the maximum Interfaces in the Composite Device.

3.2.1 Reorganize and gather the USB Device’s respective files

It is better to create a new directory that holds all the files and dependencies related to the composite USB device. All the respective files should be copied over to a new location (Keep the original unchanged for reference. It may help in the future for debugging and verifying).

For the current example: Files for the "USB Composite Device" can be reorganized as the following:

USB composite device

- Applications interface
  - Audio
  - RNDIS
  - VCP (renamed usb_cdc_if.c/.h to usb_cdc_vcp_if.c/.h)
- Class driver
  - Composite class (move all the files of drivers into the common directory) (renamed usbd_cdc.c/.h to usbd_cdc_vcp.c/.h)
- Core library
- USBD_Conf
- USB_Device

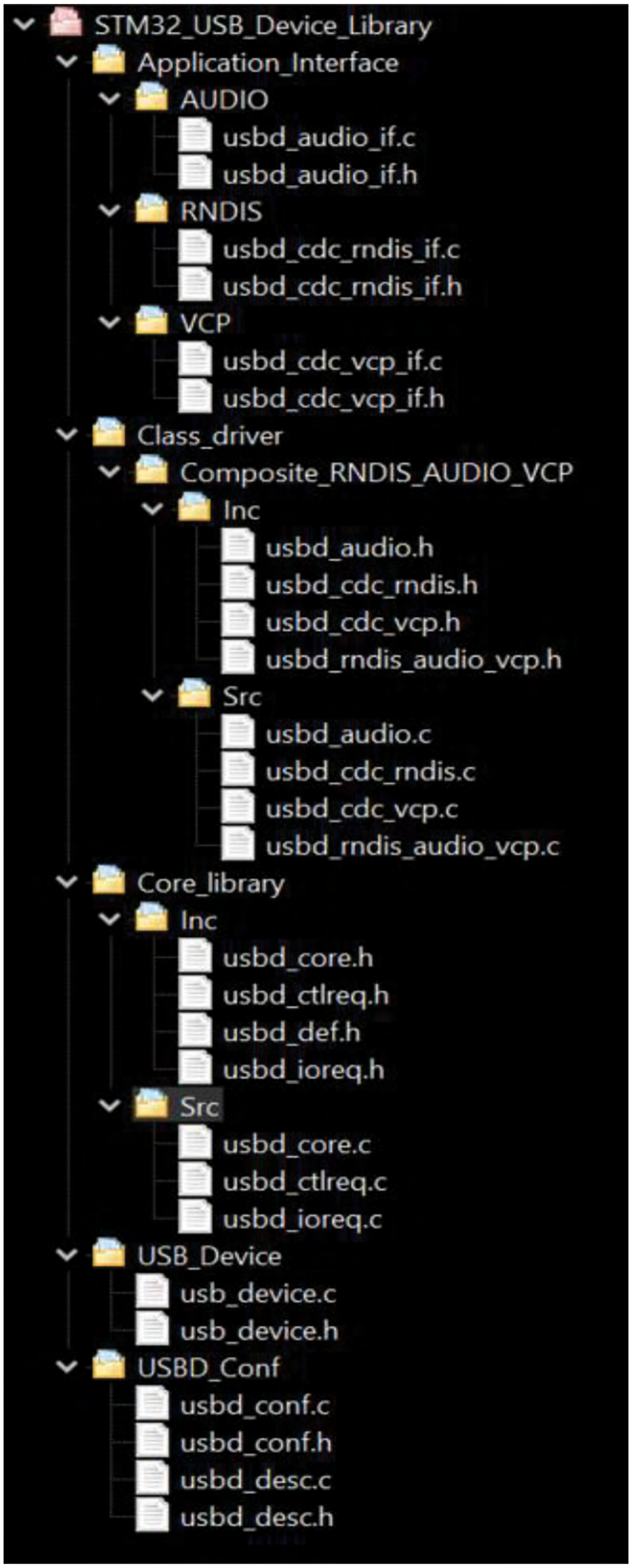
3.2.2 Create a new set of files for the composite class driver

In this example, the composite driver includes RNDIS, AUDIO and VCP. New files are created from the templates class driver files available in the repository. They are renamed as usbd_rndis_audio_vcp.c/.h. It is advisable to create these files from the templates as they incorporate all the curbs and grubs required to be prepared for the class driver.

3.2.3 Interface and endpoint footprint for the USB composite device

By going through the individual class driver, the user can easily find the total number of interfaces required for each class, as well as endpoint requirements for each interface. Simply search for the "bNumInterfaces" keyword in the respective class driver ".c" file which will tell you about the number of required interfaces. Similarly, search for the "Endpoint" keyword, it will tell you about the required endpoints for each interface. For example, in the usbd_cdc_rndis.c file, the user can find the total Interfaces for the class, Interface type, Interface ID and endpoint required in each interface by simply going through the code and comments.

Explore the following code snippet and try to understand the details of the highlighted code:



```
STM32_USB_Device_Library
    Application_Interface
        AUDIO
            usbd_audio_if.c
            usbd_audio_if.h
        RNDIS
            usbd_cdc_rndis_if.c
            usbd_cdc_rndis_if.h
        VCP
            usbd_cdc_vcp_if.c
            usbd_cdc_vcp_if.h
    Class_driver
        Composite_RNDIS_AUDIO_VCP
            Inc
                usbd_audio.h
                usbd_cdc_rndis.h
                usbd_cdc_vcp.h
                usbd_rndis_audio_vcp.h
            Src
                usbd_audio.c
                usbd_cdc_rndis.c
                usbd_cdc_vcp.c
                usbd_rndis_audio_vcp.c
    Core_library
        Inc
            usbd_core.h
            usbd_ctlreq.h
            usbd_def.h
            usbd_ioreq.h
        Src
            usbd_core.c
            usbd_ctlreq.c
            usbd_ioreq.c
    USB_Device
        usb_device.c
        usb_device.h
    USBD_Conf
        usbd_conf.c
        usbd_conf.h
        usbd_desc.c
        usbd_desc.h
```

```
/* USB CDC_RNDIS device Configuration Descriptor */
__ALIGN_BEGIN static uint8_t USBD_CDC_RNDIS_CfgHSDesc[] __ALIGN_END =
{
  /* Configuration Descriptor */
  0x09,                                   /* bLength: Configuration Descriptor size */
  USB_DESC_TYPE_CONFIGURATION,            /* bDescriptorType: Configuration */
  LOBYTE(CDC_RNDIS_CONFIG_DESC_SIZ),      /* wTotalLength: Total size of the Config
descriptor */
  HIBYTE(CDC_RNDIS_CONFIG_DESC_SIZ),
  0x02,                                   /* bNumInterfaces: 2 interface */
  ..........
  ........
  .....
  ...
  ..
  /*---------------------------------------------------------------------------*/
  /* Interface Descriptor */
  0x09,                                   /* bLength: Interface Descriptor size */
  USB_DESC_TYPE_INTERFACE,                /* bDescriptorType: Interface descriptor type */
  CDC_RNDIS_CMD_ITF_NBR,                  /* bInterfaceNumber: Number of Interface */
  0x00,                                   /* bAlternateSetting: Alternate setting */
  0x01,                                   /* bNumEndpoints: One endpoint used */
  ..........
  ........
  .....
  ...
  ..

  /* Notification Endpoint Descriptor */
  0x07,                                   /* bLength: Endpoint Descriptor size */
  USB_DESC_TYPE_ENDPOINT,                 /* bDescriptorType: Endpoint */
  CDC_RNDIS_CMD_EP,                       /* bEndpointAddress */
  0x03,                                   /* bmAttributes: Interrupt */
  LOBYTE(CDC_RNDIS_CMD_PACKET_SIZE),      /* wMaxPacketSize: */
  HIBYTE(CDC_RNDIS_CMD_PACKET_SIZE),
  CDC_RNDIS_HS_BINTERVAL,                 /* bInterval */

  /*---------------------------------------------------------------------------*/
  /* Data class interface descriptor */
  0x09,                                   /* bLength: Endpoint Descriptor size */
  USB_DESC_TYPE_INTERFACE,                /* bDescriptorType: */
  CDC_RNDIS_COM_ITF_NBR,                  /* bInterfaceNumber: Number of Interface */
  0x00,                                   /* bAlternateSetting: Alternate setting */
  0x02,                                   /* bNumEndpoints: Two endpoints used */
  ..........
  ........
  .....
  ...
  ..

  /* Endpoint OUT Descriptor */
  0x07,                                   /* bLength: Endpoint Descriptor size */
  USB_DESC_TYPE_ENDPOINT,                 /* bDescriptorType: Endpoint */
  CDC_RNDIS_OUT_EP,                       /* bEndpointAddress */
  0x02,                                   /* bmAttributes: Bulk */
  LOBYTE(CDC_RNDIS_DATA_HS_MAX_PACKET_SIZE),  /* wMaxPacketSize: */
  HIBYTE(CDC_RNDIS_DATA_HS_MAX_PACKET_SIZE),
  0xFF,                                   /* bInterval: ignore for Bulk transfer */

  /* Endpoint IN Descriptor */
  0x07,                                   /* bLength: Endpoint Descriptor size */
  USB_DESC_TYPE_ENDPOINT,                 /* bDescriptorType: Endpoint */
  CDC_RNDIS_IN_EP,                        /* bEndpointAddress */
  0x02,                                   /* bmAttributes: Bulk */
  LOBYTE(CDC_RNDIS_DATA_HS_MAX_PACKET_SIZE),  /* wMaxPacketSize: */
```

Using this method, the user needs to figure out the following aspects of the composite device class to prepare the configuration descriptor.

- How many total interfaces are required for the composite device class?
- What will be the order of the interface ID and associated device Class?
- For each class what is the interface ID number?
- For each Interface what are the required endpoints?

Based on this, the user can prepare the Interface and associated endpoint footprint required for the composite device class.

Class driver	Required interface	Interface ID	Required endpoint host - client	End point id	Endpoint packet size (bytes)	End point FiFo size (Pkt size/16)
RNDIS	Control interface	0	Out – receive	Not required	NA	NA
			In - transmit	0x81	16	1
RNDIS	Data interface	1	Out - receive	0x02	64	Rx_Fifo
			RNDIS	0x82	64	4
Audio	Control interface	2	Out - receive	Not required	NA	NA
			Audio	Not required	NA	NA
Audio	Data interface	3	Out - receive	0x03	192	Rx_Fifo
			Audio	Not required	NA	1
VCP	Control interface	4	Out – receive	Not required	NA	NA
			VCP	0x84	16	1
VCP	Data interface	5	Out - receive	0x05	64	Rx_Fifo
			In - transmit	0x85	64	4

Table: Footprint of interfaces and endpoints for the composite device

The above table is the first step to prepare the configuration descriptor for the composite device. There might be more than one configuration descriptor in the class driver file. It is observed that each class driver provides a descriptor for High Speed, Full Speed, and another Speed compatibility. Hence, there are three sets of configurations descriptors available. As per the user's needs, the user can prepare the respective configuration map.

3.2.4 Update the USB core library code to hold the multiple "class interface" structures

As per the device initialization code, each interface needs to be registered with the core library. During the registration functionality, the Core library holds the interface-specific details on the "USB device Handle." Refer to the following code and Highlighted line.

```
/**
* @brief  USBD_CDC_RegisterInterface
  * @param  pdev: device instance
  * @param  fops: CD  Interface callback
  * @retval status
  */
uint8_t USBD_CDC_RegisterInterface(USBD_HandleTypeDef *pdev,
                                   USBD_CDC_ItfTypeDef *fops)
{
  if (fops == NULL)
  {
    return (uint8_t)USBD_FAIL;
  }

  pdev->pUserData = fops;

  return (uint8_t)USBD_OK;
}
```

Hence, to support and register the multiple class driver, the user needs to have multiple place holders in the structure. Hence, update the “USBD_HandleTypeDef” structure to hold the required class driver pointers.

```
/* USB Device handle structure */
typedef struct _USBD_HandleTypeDef
{
  uint8_t                 id;
  uint32_t                dev_config;
  ……
 …
 ..

//  void                    *pClassData;
//  void                    *pUserData;
  void                    *pClassRndisData;
  void                    *pUserRndisData;
  void                    *pClassAudioData;
  void                    *pUserAudioData;
  void                    *pClassVcpData;
  void                    *pUserVcpData;
  void                    *pData;
  void                    *pBosDesc;
  void                    *pConfDesc;
} USBD_HandleTypeDef;
```

```
USBD_StatusTypeDef USBD_LL_Reset(USBD_HandleTypeDef *pdev)
{
  /* Upon Reset call user call back */
  pdev->dev_state = USBD_STATE_DEFAULT;
  pdev->ep0_state = USBD_EP0_IDLE;
  pdev->dev_config = 0U;
  pdev->dev_remote_wakeup = 0U;

//  if (pdev->pClassData != NULL )
  if (pdev->pClassRndisData && pdev->pClassAudioData && pdev->pClassVcpData )
  {
    pdev->pClass->DeInit(pdev, (uint8_t)pdev->dev_config);
  }

  . . . .
  . . .
  . .

  return USBD_OK;
}
```

Respectively, rename these pointers throughout the code swiftly. The pClassData and pUserData in the VCP class driver should be replaced with pClassVcpData and pUserVcpData, respectively. Similar changes should also be made for other class drivers of the composite device as well.

3.2.5 Prepare new composite class driver file

The user has all the required info to prepare the Composite Class Driver file. The new file should hold a configuration descriptor and composite class APIs to redirect the request from the core library to the respective class driver.

3.2.5.1 Prepare configuration descriptor

Preparing configuration descriptor for the composite device is considerably easy as each class driver already has its configuration descriptor. The user does not need to write it from scratch. The user needs to copy the configuration descriptor from the respective class and join them together with the help of the Interface Association Descriptor. On top of this, the user also needs to revisit the composite device configuration descriptor to update the total number of interfaces, total size of configuration descriptor, interfaces, and endpoint numbers for each IAD interface.

The user needs to do following three things:

- Add IAD descriptor with appropriate data for each individual class descriptor group in the beginning.
- Update the class grouped descriptor for the interface number and endpoint number.
- Update the descriptor for the total number of interfaces, Eps, and total size.

Note: The descriptor holds extremely sensitive information about the device. The USB host will ask for configuration descriptor and based on the configuration descriptor the host will further initialize the device. Hence, even a single byte error can lead to failure of enumeration for the entire composite device or individual class driver. The user must have necessary knowledge about the fields that require changes for the configuration descriptor.

```
/* USB TEMPLATE device Configuration Descriptor */
__ALIGN_BEGIN static uint8_t USBD_COMPOSITE_RAV_CfgDesc[USB_COMPOSITE_RAV_CONFIG_DESC_SIZ]
__ALIGN_END =
{
  0x09, /* bLength: Configuation Descriptor size */
  USB_DESC_TYPE_CONFIGURATION, /* bDescriptorType: Configuration */
  LOBYTE(USB_COMPOSITE_RAV_CONFIG_DESC_SIZ),           /* wTotalLength: Total size of the Con-
fig descriptor */
  HIBYTE(USB_COMPOSITE_RAV_CONFIG_DESC_SIZ),
  USB_COMPOSITE_RAV_TOTAL_INF_CNT,         /*bNumInterfaces: 6 interface*/
  0x01,         /*bConfigurationValue: Configuration value*/
  0x02,         /*iConfiguration: Index of string descriptor describing the configuration*/
  0xC0,         /*bmAttributes: bus powered and Supports Remote Wakeup */
  0x32,         /*MaxPower 100 mA: this current is used for detecting Vbus*/
  /* 09 */

  /**********  RNDIS CLASS DEVICE DETAILS **************/
  /*---------------------------------------------------------------------------*/
  /* IAD descriptor */
  0x08,                                        /* bLength */
  0x0B,                                        /* bDescriptorType */
  CDC_RNDIS_CMD_ITF_NBR,                       /* bFirstInterface */
  0x02,                                        /* bInterfaceCount */
  0xE0,                                        /* bFunctionClass (Wireless Controller) */
  0x01,                                        /* bFunctionSubClass */
  0x03,                                        /* bFunctionProtocol */
  0x00,                                        /* iFunction */

  /*---------------------------------------------------------------------------*/
  /* Interface Descriptor */
  0x09,                                        /* bLength: Interface Descriptor size */
  USB_DESC_TYPE_INTERFACE,                     /* bDescriptorType: Interface descriptor type */
  CDC_RNDIS_CMD_ITF_NBR,                       /* bInterfaceNumber: Number of Interface */
  0x00,                                        /* bAlternateSetting: Alternate setting */
  0x01,                                        /* bNumEndpoints: One endpoint used */
  0x02,                                        /* bInterfaceClass: Communication Interface
Class */
  0x02,                                        /* bInterfaceSubClass:Abstract Control Model */
  0xFF,                                        /* bInterfaceProtocol: Common AT commands */
  0x00,                                        /* iInterface: */

   . . .

  /**********  AUDIO CLASS DEVICE DETAILS **************/
  /*---------------------------------------------------------------------------*/
  /* IAD descriptor */
  0x08,                                        /* bLength */
  0x0B,                                        /* bDescriptorType */
  AUDIO_CMD_ITF_NBR,                           /* bFirstInterface */
  0x02,                                        /* bInterfaceCount */
  USB_DEVICE_CLASS_AUDIO,                      /* bFunctionClass (Audio) */
  AUDIO_SUBCLASS_AUDIOCONTROL,                 /* bFunctionSubClass */
  AUDIO_PROTOCOL_UNDEFINED,                    /* bFunctionProtocol */
  0x00,                                        /* iFunction */
/* USB Speaker Standard interface descriptor */
  AUDIO_INTERFACE_DESC_SIZE,            /* bLength */
  USB_DESC_TYPE_INTERFACE,              /* bDescriptorType */
  AUDIO_CMD_ITF_NBR,                    /* bInterfaceNumber */
  0x00,                                 /* bAlternateSetting */
  0x00,                                 /* bNumEndpoints */
  USB_DEVICE_CLASS_AUDIO,               /* bInterfaceClass */
  AUDIO_SUBCLASS_AUDIOCONTROL,          /* bInterfaceSubClass */
  AUDIO_PROTOCOL_UNDEFINED,             /* bInterfaceProtocol */
  0x00,                                 /* iInterface */
  /* 09 byte*/

  . . .
  /**********  VCP CLASS DEVICE DETAILS *************/
  /*---------------------------------------------------------------------------*/
  /* IAD descriptor */
  0x08, /* bLength */
  0x0B, /* bDescriptorType */
  CDC_VCP_CMD_ITF_NBR, /* bFirstInterface */
  0x02, /* bInterfaceCount */
  0x02, /* bFunctionClass  */
  0x02, /* bFunctionSubClass */
  0x01, /* bFunctionProtocol */
  0X00, /* iFunction */

  /* Interface Descriptor */
  0x09,                                       /* bLength: Interface Descriptor size */
  USB_DESC_TYPE_INTERFACE,                    /* bDescriptorType: Interface */
  /* Interface descriptor type */
  CDC_VCP_CMD_ITF_NBR,                        /* bInterfaceNumber: Number of Interface */
  0x00,                                       /* bAlternateSetting: Alternate setting */
  0x01,                                       /* bNumEndpoints: One endpoints used */
  0x02,                                       /* bInterfaceClass: Communication Interface Class
*/
  0x02,                                       /* bInterfaceSubClass: Abstract Control Model */
  0x01,                                       /* bInterfaceProtocol: Common AT commands */
  0x00,                                       /* iInterface: */

. . .

};
```

3.2.5.2 Prepare composite class driver function interface

Preparing the composite class driver interfaces are also considerably easy. The user needs to redirect the function call to its respective class driver based on the associated endpoint number or the interface number.

For example, the DataIn, DataOut, IsoIn, and IsoOut interfaces have arguments as endpoint numbers. As per the details from the footprint table, it is already known which corresponds to which class driver.

For example, the setup interface is having USBD_SetupReqTypedef *req as its argument. Through the req->bmRequest and req->wIndex, the user can find the respective class driver. The request can be of interface type or endpoint type. User must filter both out for each class driver.

```
static uint8_t USBD_COMPOSITE_RAV_Init(USBD_HandleTypeDef *pdev, uint8_t cfgidx)
{
    /* CDC RNDIS initialization */
    uint8_t ret = USBD_CDC_RNDIS_Init (pdev, cfgidx);
    if(ret != USBD_OK)
            return ret;
    /* Audio initialization */
    ret = USBD_AUDIO_Init (pdev, cfgidx);
    if(ret != USBD_OK)
            return ret;
    /* CDC VCP initialization */
    ret = USBD_CDC_Init (pdev, cfgidx);
    if(ret != USBD_OK)
            return ret;

  return (uint8_t)USBD_OK;
}

static uint8_t USBD_COMPOSITE_RAV_DeInit(USBD_HandleTypeDef *pdev, uint8_t cfgidx)
{
    /* CDC RNDIS De-initialization */
    USBD_CDC_RNDIS_DeInit(pdev, cfgidx);
    /* Audio 2.0 De-initialization */
    USBD_AUDIO_DeInit(pdev, cfgidx);
    /* CDC VCP De-initialization */
    USBD_CDC_DeInit(pdev, cfgidx);

  return (uint8_t)USBD_OK;
}

static uint8_t USBD_COMPOSITE_RAV_Setup(USBD_HandleTypeDef *pdev,
                                   USBD_SetupReqTypedef *req)
{

  /* Route requests to Audio interface or its endpoints to Audio class implementation */
  if(((req->bmRequest & USB_REQ_RECIPIENT_MASK) == USB_REQ_RECIPIENT_INTERFACE && req->wIndex
== CDC_RNDIS_CMD_ITF_NBR) ||
      ((req->bmRequest & USB_REQ_RECIPIENT_MASK) == USB_REQ_RECIPIENT_ENDPOINT && ((req->wIndex
& 0x7F) == (CDC_RNDIS_CMD_EP & 0x7F))))
  {
      return USBD_CDC_RNDIS_Setup(pdev, req);
  }
  if(((req->bmRequest & USB_REQ_RECIPIENT_MASK) == USB_REQ_RECIPIENT_INTERFACE && req->wIndex
== CDC_VCP_CMD_ITF_NBR) ||
      ((req->bmRequest & USB_REQ_RECIPIENT_MASK) == USB_REQ_RECIPIENT_ENDPOINT && ((req->wIndex
& 0x7F) == (CDC_CMD_EP & 0x7F))))
  {
      return USBD_CDC_Setup(pdev, req);
  }
  return USBD_AUDIO_Setup(pdev, req);
}
```

```
static uint8_t USBD_COMPOSITE_RAV_DataIn(USBD_HandleTypeDef *pdev, uint8_t epnum)
{
    if(epnum == (CDC_RNDIS_CMD_EP & 0X7F) || epnum == (CDC_RNDIS_IN_EP & 0X7F))
        return USBD_CDC_RNDIS_DataIn(pdev, epnum);
    else if(epnum == (CDC_CMD_EP & 0X7F) || epnum == (CDC_IN_EP & 0X7F))
        return USBD_CDC_DataIn(pdev, epnum);
    return USBD_AUDIO_DataIn(pdev, epnum);

}

static uint8_t USBD_COMPOSITE_RAV_DataOut(USBD_HandleTypeDef *pdev, uint8_t epnum)
{
    if(epnum == (CDC_RNDIS_CMD_EP & 0X7F) || epnum == CDC_RNDIS_OUT_EP)
        return USBD_CDC_RNDIS_DataOut(pdev, epnum);
    else if(epnum == (CDC_CMD_EP & 0X7F) || epnum == CDC_OUT_EP)
        return USBD_CDC_DataOut(pdev, epnum);
    return USBD_AUDIO_DataOut(pdev, epnum);
}

static uint8_t USBD_COMPOSITE_RAV_IsoINIncomplete(USBD_HandleTypeDef *pdev, uint8_t
epnum)
{
 . . .
  return (uint8_t)USBD_OK;
}

static uint8_t USBD_COMPOSITE_RAV_IsoOutIncomplete(USBD_HandleTypeDef *pdev, uint8_t
epnum)
{
 . . .
  return (uint8_t)USBD_OK;
}
```

3.2.6 Update existing individual class driver and associated interface files

The individual class driver files require modification to support the composite class driver. There are no logical changes or functional changes required in the individual class driver and its corresponding interface files. However, many cosmetic and compatibility changes are required to be carried out.

The following is the list of changes to be made:

- Remove the structure holding function pointers of the individual class driver.
- Remove the configuration and other descriptor for the individual class.
- Remove the “GetCfgDesc” and “GetDeviceQualifierDescinterfaces” interfaces for the individual class.
- Update all the class interface respective functions from static to global.
- Find and replace the “class interface structure” (pClassData and pUserData) pointer that is renamed as per core library changes in the class driver file.
- Add all class-specific interface declarations into the respective header files.
- Update the interface number and endpoint number respective macros of each class driver as per the new footprint for the composite class.
- Find and replace the “class interface structure” (pClassData and pUserData) pointer that is renamed as per the core library changes in the corresponding interface file (usbd_<class>_if.c/.h file).

Note:

1. Use #if 0 to omit out or remove code from the existing individual class driver file.
2. Use find and replace for (pClassData and pUserData).
3. Use find and replace to change the static scope into global for all the class driver interfaces that are being called from the composite class.

3.2.7 Update for EPO RxReady request

Update existing individual class driver files to determine that the EPO_RxReady is about to be executed. Set a global flag for such an indication. Use the global flag to get an indication of which driver's respective function call should be called. As per the USB specification, all the EP0 respective commands will always be initiated with the request of the class respective Setup command. The individual class driver always prepares the buffer for receiving data through the EP0 afterwards.

The user needs to find the USBD_CtlPrepareRx function at individual class driver and set the respective class specific EP0 request flag. The composite class driver API can use this flag to redirect the EP0 service call to its respective individual driver and reset it afterwards.

3.2.7.1 Update individual class driver to set the EP0 Rx ready functionality

The user needs to find the USBD_CtlPrepareRx function at the individual class driver and set the respective class specific EP0 request flag.

```
bool isEP0CntrlRqst4Audio = 0;

. . .


static void AUDIO_REQ_SetCurrent(USBD_HandleTypeDef *pdev, USBD_SetupReqTypedef *req)
{
  USBD_AUDIO_HandleTypeDef *haudio;
  haudio = (USBD_AUDIO_HandleTypeDef *)pdev->pClassAudioData;

  if (req->wLength != 0U)
  {
        isEP0CntrlRqst4Audio = 1;     /* Composite Device Need to figure out
                                                               EP0 request for
respective Interface
                                                               weather Audio or CDC */

    /* Prepare the reception of the buffer over EP0 */
    (void)USBD_CtlPrepareRx(pdev, haudio->control.data, req->wLength);

    haudio->control.cmd = AUDIO_REQ_SET_CUR;     /* Set the request value */
    haudio->control.len = (uint8_t)req->wLength; /* Set the request data length */
    haudio->control.unit = HIBYTE(req->wIndex);  /* Set the request target unit */
  }
}
```

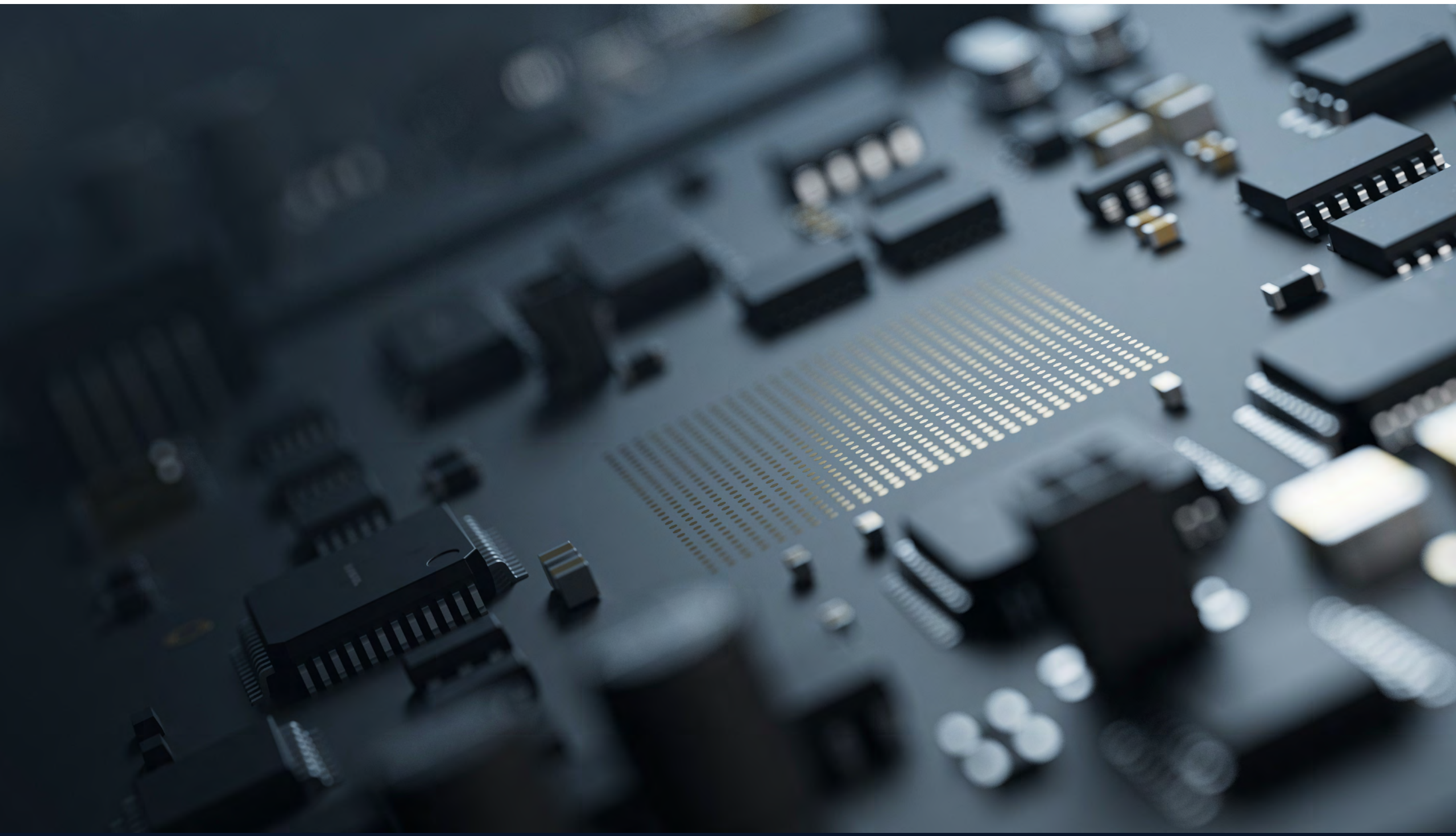
3.2.7.2 Update composite class driver the EPO_RxReady interface to handle the request

The EPO_RxReady interface at the composite driver should access the respective individual class driver's EP0 request flag and based on it call their respective serving interfaces.

```
static uint8_t USBD_COMPOSITE_RAV_EP0_RxReady(USBD_HandleTypeDef *pdev)
{
    if(isEP0CntrlRqst4Audio)
    {
        isEP0CntrlRqst4Audio = 0;
        USBD_AUDIO_EP0_RxReady(pdev);
    }
    else if(isEP0CntrlRqst4RNDIS)
    {
        isEP0CntrlRqst4RNDIS = 0;
        USBD_CDC_RNDIS_EP0_RxReady(pdev);
    }
    else
    {
        USBD_CDC_EP0_RxReady(pdev);
    }
    return (uint8_t)USBD_OK;
}
```

3.2.8 Update usb_device.c file

The usb device.c file registers the function pointer for the class driver and the user space class interface function pointers. As per the newly created composite driver file and updated existing class driver files, the user is supposed to change the call. Refer to the below code snippet that was changed to register the composite USB device class and individual class interfaces for the user space. The user needs to include the class driver and its respective interface files at the top. At the main function, the user requires to make appropriate changes to register all the class interfaces.



```
/* Includes ------------------------------------------------------------------*/

#include "usb_device.h"
#include "usbd_core.h"
#include "usbd_desc.h"
#include "usbd_cdc_vcp.h"
#include "usbd_cdc_vcp_if.h"
#include "usbd_cdc_rndis.h"
#include "usbd_cdc_rndis_if.h"
#include "usbd_audio.h"
#include "usbd_audio_if.h"
#include "usbd_rndis_audio_vcp.h"

/* USER CODE BEGIN Includes */

/* USER CODE END Includes */
 . . .
. .
.
void MX_USB_DEVICE_Init(void)
{
  /* USER CODE BEGIN USB_DEVICE_Init_PreTreatment */

  /* USER CODE END USB_DEVICE_Init_PreTreatment */

  /* Init Device Library, add supported class and start the library. */
  if (USBD_Init(&hUsbDeviceFS, &FS_Desc, DEVICE_FS) != USBD_OK)
  {
    Error_Handler();
  }
  if (USBD_RegisterClass(&hUsbDeviceFS, &USBD_COMPOSITE_RAV_ClassDriver) != USBD_OK)
  {
    Error_Handler();
  }
  if (USBD_CDC_RNDIS_RegisterInterface(&hUsbDeviceFS, &USBD_CDC_RNDIS_fops) != USBD_OK)
  {
    Error_Handler();
  }
  if (USBD_AUDIO_RegisterInterface(&hUsbDeviceFS, &USBD_AUDIO_fops_FS) != USBD_OK)
  {
    Error_Handler();
  }
  if (USBD_CDC_RegisterInterface(&hUsbDeviceFS, &USBD_Interface_fops_FS) != USBD_OK)
  {
    Error_Handler();
  }
  if (USBD_Start(&hUsbDeviceFS) != USBD_OK)
  {
    Error_Handler();
  }

  /* USER CODE BEGIN USB_DEVICE_Init_PostTreatment */
  HAL_PWREx_EnableUSBVoltageDetector();

  /* USER CODE END USB_DEVICE_Init_PostTreatment */
}
```

3.2.9 Update usbd_desc.c file

The device descriptor file is a generic file that can be used across multiple class drivers. The file “usbd_desc.c” holds the USB standard device descriptor. Change the DeviceClass, DeviceSubClass, and DeviceProtocol attributes to 0xFF, 0x02, and 0x01 respectively for the composite device.

```
/** USB standard device descriptor. */
__ALIGN_BEGIN uint8_t USBD_FS_DeviceDesc[USB_LEN_DEV_DESC] __ALIGN_END =
{
  0x12,                       /*bLength */
  USB_DESC_TYPE_DEVICE,       /*bDescriptorType*/
  0x00,                       /*bcdUSB */
  0x02,
  0xEF,                       /*bDeviceClass*/
  0x02,                       /*bDeviceSubClass*/
  0x01,                       /*bDeviceProtocol*/
  USB_MAX_EP0_SIZE,           /*bMaxPacketSize*/
  LOBYTE(USBD_VID),           /*idVendor*/
  HIBYTE(USBD_VID),           /*idVendor*/
  LOBYTE(USBD_PID_FS),        /*idProduct*/
  HIBYTE(USBD_PID_FS),        /*idProduct*/
  0x00,                       /*bcdDevice rel. 2.00*/
  0x02,
  USBD_IDX_MFC_STR,           /*Index of manufacturer  string*/
  USBD_IDX_PRODUCT_STR,       /*Index of product string*/
  USBD_IDX_SERIAL_STR,        /*Index of serial number string*/
  USBD_MAX_NUM_CONFIGURATION  /*bNumConfigurations*/
};

/* USB_DeviceDescriptor */
```

3.2.10 Update usbd_conf.c/.h file

These are wrapper files that play a vital role between the HAL driver and USB composite device architecture. There is no functional change needed at any of the logical implementation for this section except two places.

3.2.10.1 Update the usbd_conf.h file

Change the supported maximum number of interfaces by the driver architecture. It should be set as per the table showing the footprint of Interfaces and Endpoints. For this example, it has been changed to 6.

```
/** @defgroup USBD_CONF_Exported_Defines USBD_CONF_Exported_Defines
  * @brief Defines for configuration of the Usb device.
  * @{
  */

/*---------- -----------*/
#define USBD_MAX_NUM_INTERFACES     6U
/*---------- -----------*/
#define USBD_MAX_NUM_CONFIGURATION     1U
/*---------- -----------*/
#define USBD_MAX_STR_DESC_SIZ     512U
/*---------- -----------*/
#define USBD_DEBUG_LEVEL     0U
/*---------- -----------*/
#define USBD_LPM_ENABLED     0U
/*---------- -----------*/
#define USBD_SELF_POWERED     1U

/****************************************/
/* #define for FS and HS identification */
#define DEVICE_FS         0
#define DEVICE_HS         1
```

3.2.10.2 Update the usbd_conf.c file

Update the USBD_LL_Init function to incorporate the changes needed for the low-level fifo to get aligned with the new EndPoint mapping. Refer to the table showing the footprint of the Interfaces and Endpoints to figure out the FIFO size allocation.

```
USBD_StatusTypeDef USBD_LL_Init(USBD_HandleTypeDef *pdev)
{
  /* Init USB Ip. */
  if (pdev->id == DEVICE_FS) {
  /* Link the driver to the stack. */
  hpcd_USB_OTG_FS.pData = pdev;
  pdev->pData = &hpcd_USB_OTG_FS;
. . . .
. . .
. .
.

#if (USE_HAL_PCD_REGISTER_CALLBACKS == 1U)
  /* Register USB PCD CallBacks */
  HAL_PCD_RegisterCallback(&hpcd_USB_OTG_FS, HAL_PCD_SOF_CB_ID, PCD_SOFCallback);
  HAL_PCD_RegisterCallback(&hpcd_USB_OTG_FS, HAL_PCD_SETUPSTAGE_CB_ID, PCD_SetupStageCallback);
. . .
. .
.
HAL_PCD_RegisterIsoInIncpltCallback(&hpcd_USB_OTG_FS, PCD_ISOINIncompleteCallback);
#endif /* USE_HAL_PCD_REGISTER_CALLBACKS */
//  HAL_PCDEx_SetRxFiFo(&hpcd_USB_OTG_FS, 0x80);
//  HAL_PCDEx_SetTxFiFo(&hpcd_USB_OTG_FS, 0, 0x40);
//  HAL_PCDEx_SetTxFiFo(&hpcd_USB_OTG_FS, 1, 0x80);

  uint16_t tx_fifo_size[6] = {0}; /* TX_FIFO allocation*/
  uint8_t max_tx_ep_num = 0; /* The Max TX_EP_NUMBER*/
  uint16_t tx_fifo_used_size = 0; /* total usage of TX_FIFO*/
  uint16_t rx_fifo_size = 0; /* total usage of RX_FIFO*/

  tx_fifo_size[max_tx_ep_num] = 40; /* Control Intf In EP 0 - packet size(64)/16*/
  tx_fifo_used_size += tx_fifo_size[max_tx_ep_num];
  max_tx_ep_num++;
  tx_fifo_size[max_tx_ep_num] = 10; /* CDC RNDIS CMD In EP 1 - packet size(16)/16*/
  tx_fifo_used_size += tx_fifo_size[max_tx_ep_num];
  max_tx_ep_num++;
  tx_fifo_size[max_tx_ep_num] = 40; /* CDC RNDIS COM In EP 2 - packet size(64)/16*/
  tx_fifo_used_size += tx_fifo_size[max_tx_ep_num];
  max_tx_ep_num++;
  tx_fifo_size[max_tx_ep_num] = 1; /* Audio Intf In EP 3 - packet size(64)/16*/
  tx_fifo_used_size += tx_fifo_size[max_tx_ep_num];
  max_tx_ep_num++;
  tx_fifo_size[max_tx_ep_num] = 10;  /* CDC VCP In EP 4 - packet size(16)/16*/
  tx_fifo_used_size += tx_fifo_size[max_tx_ep_num];
  max_tx_ep_num++;
  tx_fifo_size[max_tx_ep_num] = 40; /* CDC VCP In EP 5 - packet size(64)/16*/
  tx_fifo_used_size += tx_fifo_size[max_tx_ep_num];

  rx_fifo_size = USB_OTG_FIFO_SIZE - tx_fifo_used_size;
  /* initialize and set all the Fifos required by EPs */
  HAL_PCDEx_SetRxFiFo(&hpcd_USB_OTG_FS, rx_fifo_size);
  for(int i = 0; i<= max_tx_ep_num; i++)
  {
    HAL_PCD_SetTxFiFo(&hpcd_USB_OTG_FS, i, tx_fifo_size[i]);
  }
  }
  return USBD_OK;
}
```

4. Debugging and Integrating with Application Layer

4.1 Integrating with application layer

Each individual class driver has its associated user space application interface file. The interface file provides integration with the application layer. The ST OEM provides almost the complete interface file for all the supported class drivers. The user can easily find them alongside the class driver file. The user can also find the enhanced version with the example program for the respective processor family. As per the user requirements, it can be further enhanced and improved. The interfaces registered through the usb_device.c files will be accessed during the device class operation. The integration for each individual class driver for the application layer is usually extremely easy when you understand the call back interfaces functionality required within the call.

Providing details on the implementation for the application layer interface for each class driver can not be covered in this scope. The application layer is always user and requirement dependent. It cannot be generalized. This is the extended part of the composite USB device. Mostly the developer struggles while preparing the composite class and enumeration. However, when a developer has issues integrating the functionality of the class driver with the application layer, it is advisable to refer to the respective available example program from the OEM for that individual class driver.

4.2 Debugging the issues for composite device driver

Debugging such complex implementation and USB enumeration process can be considered one of the most challenging parts during the implementation. As there are many different areas which can lead to failure of the USB composite device enumeration, it is crucial that the user can find out the probable cause of area. Development is preferable on Linux platform or Windows platform.

4.2.1 Debugging on Linux platform

1. Use command prompt for enumeration related issues

a. Use lsusb command to find out the list of the successful USB enumerated interfaces.

Note: It can also be used to read details about the device and configuration descriptor.t

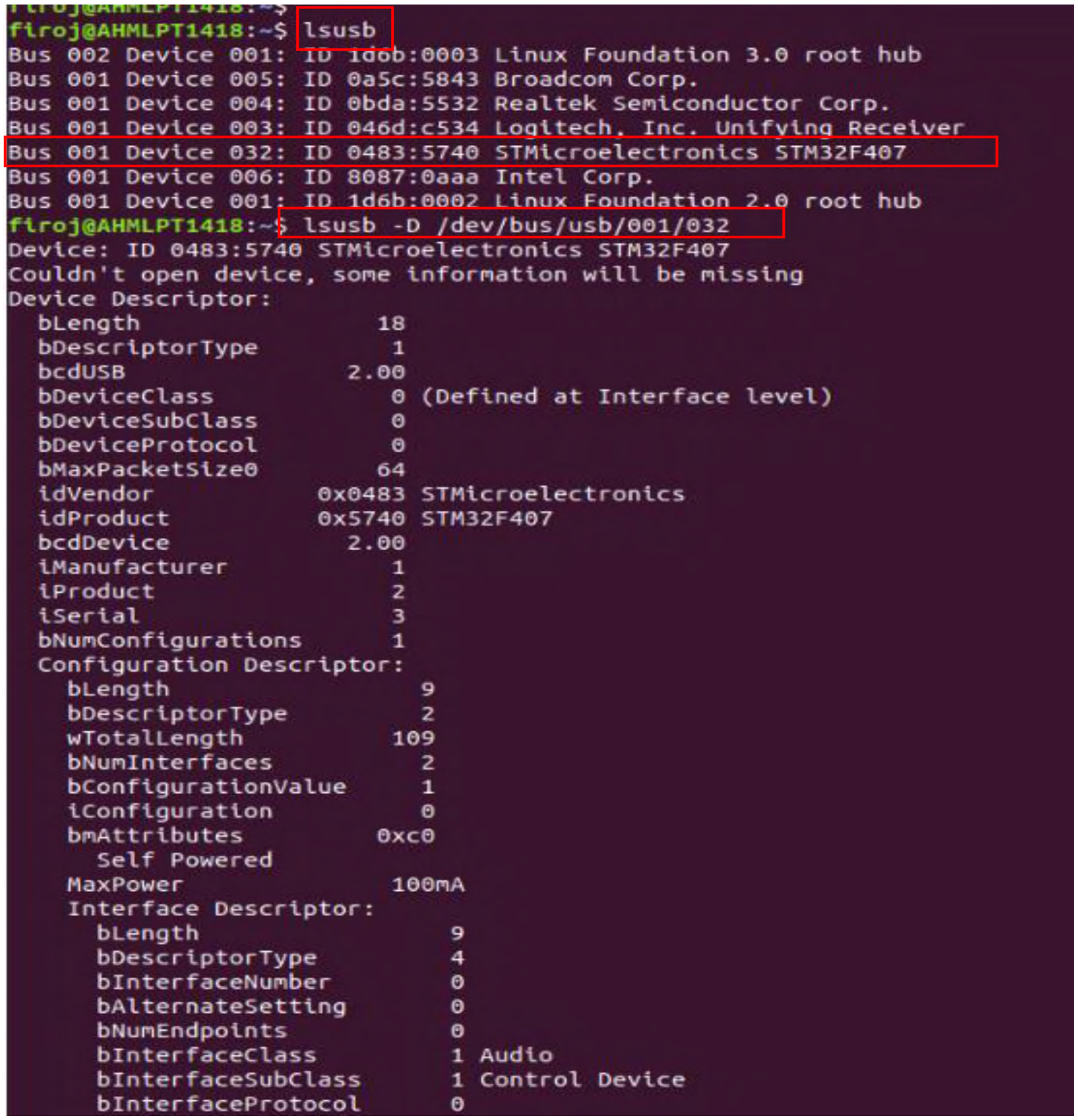
b. Use lsusb command to find out the list of the successful USB enumerated interfaces.

c. Use the dmesg to find out what is happening within the kernel during the enumeration process

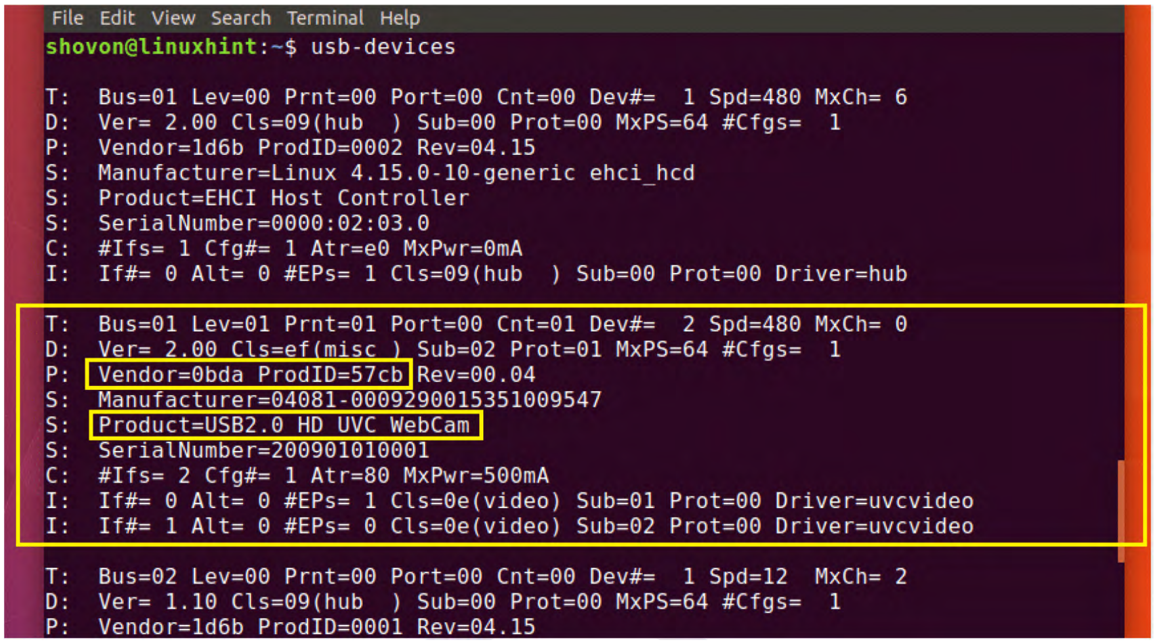
Note: It can help to find out whether the enumeration of each individual class interface is successful or not.

In case of failure during the enumeration of class interface, it can point to possible root cause or issue.

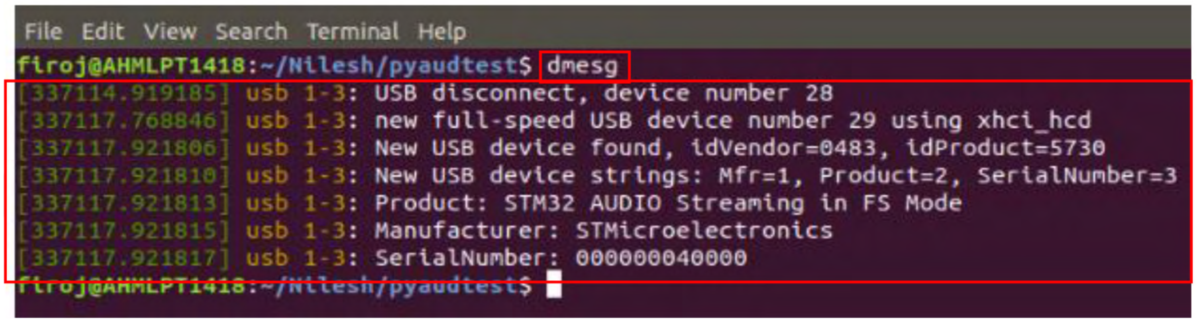
2. Use the wireshark tool to figure out the enumeration issues that the Linux command line is not able to pinpoint. When the developer does not find any helpful information from the Linux command line for the enumeration or USB device failure, wireshark is very robust, efficient, and free tool to debug the issues between the Host and Device. The developer needs to capture the session between the Host and Device. Based on the session data, the user can analyze and figure out the root cause of the issue.



```
firoj@AHMLPT1418:~$ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 005: ID 0a5c:5843 Broadcom Corp.
Bus 001 Device 004: ID 0bda:5532 Realtek Semiconductor Corp.
Bus 001 Device 003: ID 046d:c534 Logitech, Inc. Unifying Receiver
Bus 001 Device 032: ID 0483:5740 STMicroelectronics STM32F407
Bus 001 Device 006: ID 8087:0aaa Intel Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
firoj@AHMLPT1418:~$ lsusb -D /dev/bus/usb/001/032
Device: ID 0483:5740 STMicroelectronics STM32F407
Couldn't open device, some information will be missing
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0        64
  idVendor           0x0483 STMicroelectronics
  idProduct          0x5740 STM32F407
  bcdDevice            2.00
  iManufacturer           1
  iProduct                2
  iSerial                 3
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength          109
    bNumInterfaces          2
    bConfigurationValue     1
    iConfiguration          0
    bmAttributes         0xc0
      Self Powered
    MaxPower              100mA
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        0
      bAlternateSetting       0
      bNumEndpoints           0
      bInterfaceClass         1 Audio
      bInterfaceSubClass      1 Control Device
      bInterfaceProtocol      0
```



```
shovon@linuxhint:~$ usb-devices

T:  Bus=01 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=480 MxCh= 6
D:  Ver= 2.00 Cls=09(hub  ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=1d6b ProdID=0002 Rev=04.15
S:  Manufacturer=Linux 4.15.0-10-generic ehci_hcd
S:  Product=EHCI Host Controller
S:  SerialNumber=0000:02:03.0
C:  #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=0mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub

T:  Bus=01 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=480 MxCh= 0
D:  Ver= 2.00 Cls=ef(misc ) Sub=02 Prot=01 MxPS=64 #Cfgs=  1
P:  Vendor=0bda ProdID=57cb Rev=00.04
S:  Manufacturer=04081-0009290015351009547
S:  Product=USB2.0 HD UVC WebCam
S:  SerialNumber=200901010001
C:  #Ifs= 2 Cfg#= 1 Atr=80 MxPwr=500mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=0e(video) Sub=01 Prot=00 Driver=uvcvideo
I:  If#= 1 Alt= 0 #EPs= 0 Cls=0e(video) Sub=02 Prot=00 Driver=uvcvideo

T:  Bus=02 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12  MxCh= 2
D:  Ver= 1.10 Cls=09(hub  ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=1d6b ProdID=0001 Rev=04.15
```

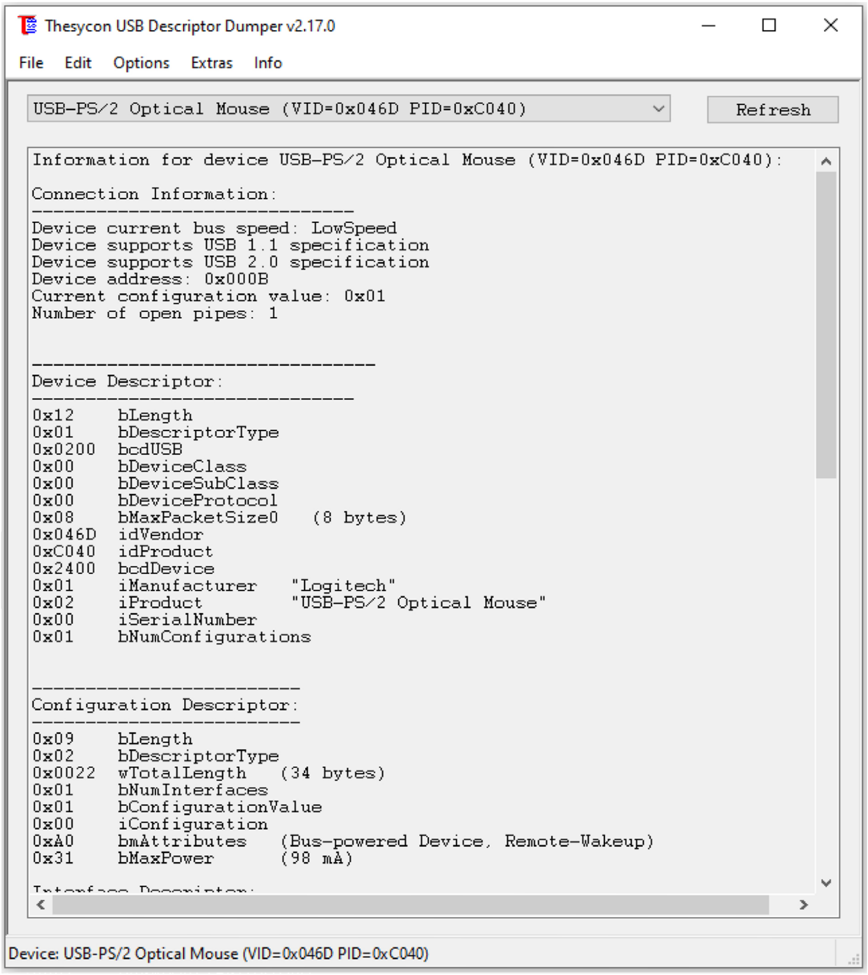


```
firoj@AHMLPT1418:~/Nilesh/pyaudtest$ dmesg
[337114.919185] usb 1-3: USB disconnect, device number 28
[337117.768846] usb 1-3: new full-speed USB device number 29 using xhci_hcd
[337117.921806] usb 1-3: New USB device found, idVendor=0483, idProduct=5730
[337117.921810] usb 1-3: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[337117.921813] usb 1-3: Product: STM32 AUDIO Streaming in FS Mode
[337117.921815] usb 1-3: Manufacturer: STMicroelectronics
[337117.921817] usb 1-3: SerialNumber: 000000040000
firoj@AHMLPT1418:~/Nilesh/pyaudtest$
```

4.2.2 Debugging on Windows platform

There is extremely limited support available on the Windows platform to debug the USB debugging or functionality related issues. Following is the list of points that the developer can try to reach the root cause of the issue.

1. Use tdd.exe file to get the USB device descriptor
   Thesycon's descriptor dumper is a Windows' utility that displays the USB descriptors of any USB device. The dump is in plain text format and can be saved to a file or copy-pasted into an email. This is the most useful tool for developers and technical support personnel.
2. Use Wireshark tool to figure out the enumeration issues
   When a developer does not find any helpful information from the descriptor of the USB device, Wireshark is very robust, efficient, and free tool to debug the issues between the Host and Device. The developer needs to capture the session between the Host and device. Based on the session data, users can analyze and figure out the root cause of the issue.


Thesycon USB Descriptor Dumper v2.17.0
File
Edit
Options
Extras
Info
USB-PS/2 Optical Mouse (VID=0x046D PID=0xC040)
Refresh
Device: USB-PS/2 Optical Mouse (VID=0x046D PID=0xC040)

```
Information for device USB-PS/2 Optical Mouse (VID=0x046D PID=0xC040):

Connection Information:
------------------------------
Device current bus speed: LowSpeed
Device supports USB 1.1 specification
Device supports USB 2.0 specification
Device address: 0x000B
Current configuration value: 0x01
Number of open pipes: 1


--------------------------------
Device Descriptor:
------------------------------
0x12     bLength
0x01     bDescriptorType
0x0200   bcdUSB
0x00     bDeviceClass
0x00     bDeviceSubClass
0x00     bDeviceProtocol
0x08     bMaxPacketSize0   (8 bytes)
0x046D   idVendor
0xC040   idProduct
0x2400   bcdDevice
0x01     iManufacturer   "Logitech"
0x02     iProduct        "USB-PS/2 Optical Mouse"
0x00     iSerialNumber
0x01     bNumConfigurations


-------------------------
Configuration Descriptor:
-------------------------
0x09     bLength
0x02     bDescriptorType
0x0022   wTotalLength   (34 bytes)
0x01     bNumInterfaces
0x01     bConfigurationValue
0x00     iConfiguration
0xA0     bmAttributes   (Bus-powered Device, Remote-Wakeup)
0x31     bMaxPower      (98 mA)
```

About The Author

Vishalkumar Padalia

(Senior Technical Lead, eInfochips Inc.)

Vishalkumar Padalia is working as Senior Technical Lead at eInfochips - an arrow company. He has more than 15 years of experience in Embedded System Design and verification. He has worked for more than 9 years for Airborne Software Development and verification. His experience includes development and verification of software for Display Processing Mission Computer, Flight Cockpit Display System, Flight Mission Computers, Primary Flight Control Computer and so on. Vishalkumar holds a Bachelor of Engineering degree in Electronics and Communication.

References

[1] https://en.wikipedia.org/wiki/USB
[2] https://www.beyondlogic.org/usbnutshell/usb1.shtml
[3] https://www.usb.org/document-library/usb-20-specification
[4] https://www.usb.org/sites/default/files/iadclasscode_r10.pdf
[5] https://wiki.st.com/stm32mcu/wiki/Introduction_to_USB_with_STM32
[6] https://www.st.com/resource/en/user_manual/dm00108129-stm32cube-usb-device-library-stmicroelectronics.pdf#page=1
[7] https://sudonull.com/post/68144-CDC-MSC-USB-Composite-Device-on-STM32-HAL
[8] https://www.programmersought.com/article/18127793400/
[9] https://en.wikipedia.org/wiki/Host_controller_interface_(USB,_Firewire)
[10] https://wiki.ubuntu.com/Kernel/Debugging/USB
[11] https://www.youtube.com/watch?v=C8UKrIKqH78
[12] https://youtu.be/eumKLXNlM0U
[13] https://www.youtube.com/playlist?list=PLnMKNibPkDnFCosVVv98U5dCulE6T3Iy8
[14] https://youtu.be/I1HfAkz-brc
[15] https://www.youtube.com/playlist?list=PLnMKNibPkDnFFRBVD206EfnnHhQZI4Hxa