Application Note

Application Note

Document No.: AN1087

APM32F4xx_OTG Application Note

Version: V1.0

1 Introduction

At present, APM32F4xx series has high-speed and full-speed two USB controllers, both of which support OTG, and the USB high-speed controller has two interfaces. This application note explains the examples based on APM32F4xx_OTG_SDK. This software development kit can be downloaded from APM32F4xx software support on the official website of Geehy.

Contents

2 USB OTG Introduction

OTG in APM32F4xx complies with USB2.0 specification and On-The-Go supplementary standard. In conventional USB, the identity of slave and host is relatively fixed, and the control of data transmission is processed and initiated by the master. OTG allows devices to switch between slave and host, and it can be either a host or a slave. OTG is often used in the interaction between devices, for example, OTG can be used between printer and camera, mobile phone and U disk, for data interaction.

2.1 Role switching principle of OTG between host and slave

The ID line of the USB wiring can be used to distinguish the host from the slave. If the ID line is detected as low level, it means the host, and if the ID line is detected as high level, it means the slave. The ID line of MCU is internally connected with a pull-up resistor. When an external ID line is inserted and grounded, it will be detected as a low level and recognized as the host; when the external ID line is suspended and inserted, it will be detected as a high level and recognized as a slave.

2.1.1 Host Negotiation Protocol (HNP)

The full name of HNP is Host Negotiation Protocol. It is a protocol used for role switching between host and slave. APM32F407 can turn on this function by pulling up the HNPEN bit of the global register GUSBCFG of OTG peripherals.

2.1.2 Session Request Protocol (SRP)

The full name of SRP is Session Request Protocol. This function allows A-device to stop Vbus power supply when the bus is not in use to reduce power consumption. APM32F407 can turn on this function by pulling up the SRPEN bit of the global register GUSBCFG of OTG peripherals

2.2 Characteristics of APM32F407 USB slave

For the current APM32F4 series, the USB OTG module of APM32F407, F405, 417 and 415 series are the same, but the USB module of F4 series may be changed in the future, so F407 is taken as an example here.

When APM32F407 is used as a slave, there are 4 IN endpoints and 4 OUT endpoints available, 8 endpoints in total. The endpoint 0 is special and only serves as a control endpoint. When serving as a slave, it supports full-speed and high-speed controllers. The high-speed controller has two interfaces, including an embedded high-speed PHY, which can reduce the design of peripheral devices.

2.3 Characteristics of APM32F407 USB host

When APM32F407 serves as a host, there are 8 host channels available. Each channel needs to be configured with the corresponding endpoint number, transmission type and other information during the use. As a host, it supports low-speed, full-speed and high-speed controllers. The high-speed controller has two interfaces.

2.4 Common questions and answers for beginners

2.4.1 What is the relationship between USB and OTG?

Answer: USB contains OTG. OTG can be regarded as a special type of USB and it is specially used to design portable devices and realize the role switching between host and slave. OTG can also be used as a normal USB, ignoring the role of ID line, forcing MCU to serve as a host or slave.

2.4.2 What is the difference between A-device, B-device and standard slave?

Answer: A device is an OTG host. Compared with a standard host, A device's Vbus can provide lower current to reduce power consumption, and it can be switched to a slave. B-device is an OTG slave. Its usage is basically the same as that of a standard slave, except that it can be switched to a host through the controller. The standard slave is the USB slave that we usually release. It can only be used as a slave, and the ID line is invalid.

2.4.3 What is the function of Vbus?

Answer: Vbus is the voltage of the bus, and is provided by the USB host to the slave. The slave can be divided into self-powered device and bus-powered device, and the configuration descriptor of USB informs the host. The Vbus line on the slave mainly relies on receiving. The slave controller of APM32F4 has the function of Vbus detection, and can judge the USB insertion after the Vbus pin is pulled up. The slave can also choose to turn off Vbus detection, without Vbus. It is default in the slave of APM32F4 that Vbus is valid, and can use the internal pull-up resistor to control the pull-up resistor to be valid or not through software.

3 Use Example of USB Slave

We can find and download APM32F4xx_OTG_SDK in the APM32F4xx software support on the official website of Geehy. This SDK kit specially provides driver code and related host and slave routines for APM32F4 series USB. The Project\Device_Examples folder in SDK kit contains three routines, among which, the HID is mouse routine and uses HID class; MSC is USB disk routine and uses the MSC high-capacity class; VCP is virtual serial port routine and uses CDC transmission class.

3.1 Initialize USB slave

We can use USBD_Init function to initialize USB, and the structure parameters needing to be configured are as follows:

```
typedef struct
{
  USBD_Descriptor_T *pDeviceDesc;                 //!< Device descriptor
  USBD_Descriptor_T *pConfigurationDesc;          //!< Configuration descriptor
  USBD_Descriptor_T *pStringDesc;                 //!< String descriptor
  USBD_Descriptor_T *pQualifierDesc;              //!< Device qualifier
  USBD_Descriptor_T *pHidReportDesc;              //!< Report descriptor
  USBD_StdReqCallback_T *pStdReqCallback;         //!< Standard request callback function set
  USBD_ReqHandler_T stdReqExceptionHandler;       //!< Standard request error callback function
  USBD_ReqHandler_T classReqHandler;              //!< class request callback function
  USBD_ReqHandler_T vendorReqHandler;             //!< Vendor request callback function
  USBD_CtrlTxStatusHandler_T txStatusHandler;     //!< Control process IN status callback function
  USBD_CtrlRxStatusHandler_T rxStatusHandler;     //!< Control process OUT status callback functio
  USBD_EPHandler_T          outEpHandler;         //!< Callback function of OUT endpoints other than endpoint 0
  USBD_EPHandler_T          inEpHandler;          //!< Callback function of IN endpoints other than endpoint 0
  USBD_ResetHandler_T    resetHandler;            //!< USB reset callback function
  USBD_InterruptHandler_T intHandler;             //!< Interrupt supplementary handling callback function
} USBD_InitParam_T;
```

Figure 1 USBD_InitParam_T Structure

For the meaning of structure members, refer to the comments in the above codes. The function of USB_Init function is to pass the callback function in the formal parameters, and initialize the USB hardware, global register, device register, USB interrupt and other contents.

Users can configure the global interrupt, endpoint interrupt, FIFO size, etc. by configuring the global macro definition of usb_config.h file. The defined macros are briefly explained below:

- **USB_INT_G_SOURCE:** Global interrupt source configuration;
- **USB_INT_EP_OUT_SOURCE:** OUT endpoint interrupt source configuration;
- **USB_INT_EP_IN_SOURCE:** IN endpoint interrupt source configuration;
- **USB_VBUS_SWITCH:** When it is 1, VBUS induction will be turned on during initialization; when it is 0, VBUS induction will be turned off;
- **USB_SOF_OUTPUT_SWITCH:** When it is 1, SOF output will be turned on, and when it is 0, SOF output will be turned off;
- **USBD_CONFIGURATION_NUM:** Number of descriptor configurations, generally 1;
- **USB_EP0_PACKET_SIZE:** The maximum packet size of endpoint 0;
- **USB_FS_RX_FIFO_SIZE:** Size configuration of full-speed receive FIFO
- **USB_FS_TX_FIFO_0_SIZE:** Size configuration of full-speed transmit FIFO 0
- **USB_FS_TX_FIFO_1_SIZE:** Size configuration of full-speed transmit FIFO 1
- **USB_HS_RX_FIFO_SIZE:** Size configuration of high-speed receive FIFO
- **USB_HS_TX_FIFO_0_SIZE:** Size configuration of high-speed transmit FIFO 0
- **USB_HS_TX_FIFO_1_SIZE:** Size configuration of high-speed transmit FIFO 1
- **DELAY_SOURCE:** When its value is USE_DEFAULT, use the default delay function; when its value is USE_USER, the user needs to customize the delay function in usb_user.c file.

3.2 Instructions for interrupt handling

When USB is used as a slave, its communication process is usually handled by interrupt. Refer to the following figure for the interrupt handling process of the slave:

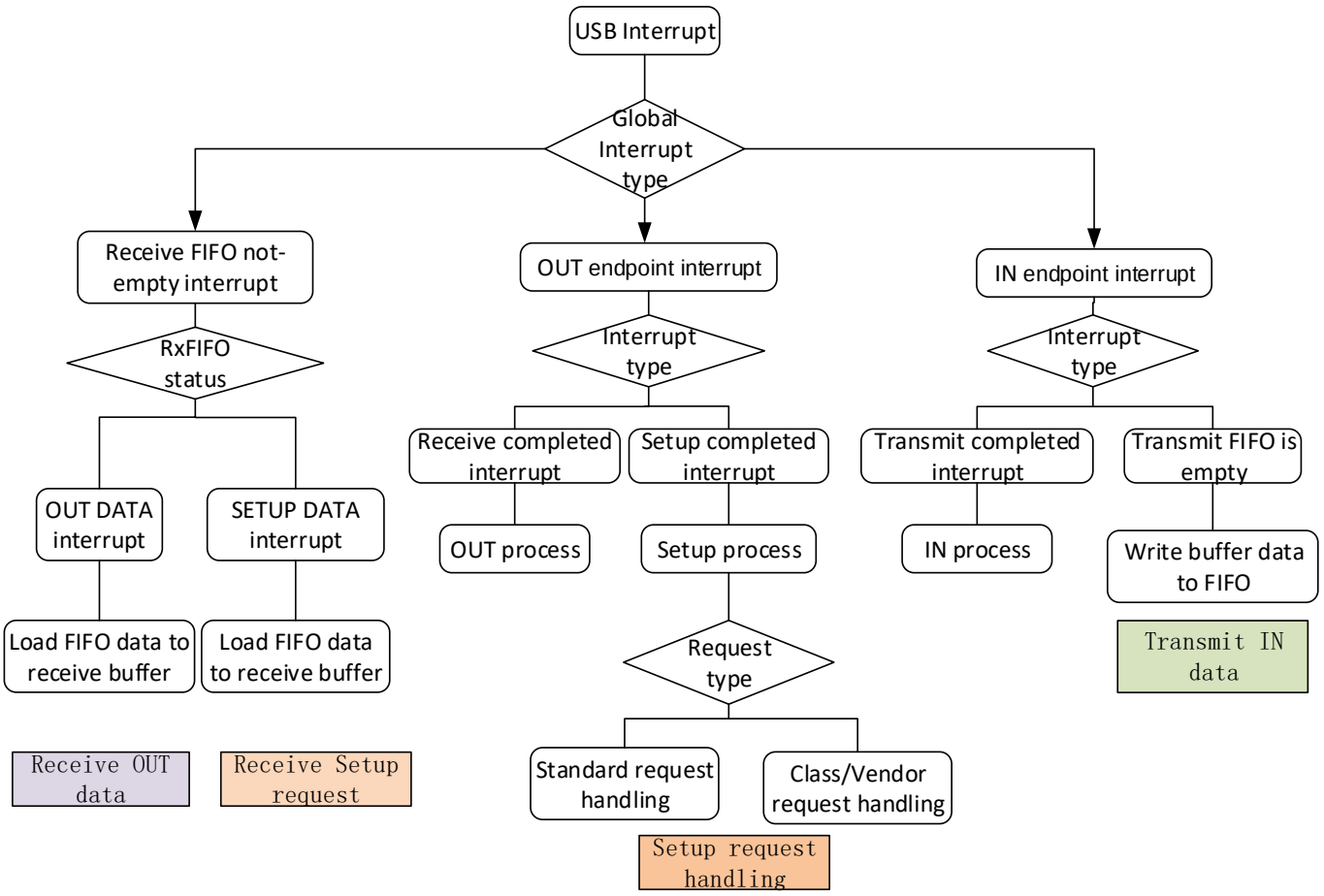

Figure2 Program Flow Chart

In fact, enumeration and communication in USB slave mode basically rely on interrupt. Even if the function of transmitting and receiving USB data is called in polling, the data transmission will finally be processed through interrupt.

3.3 SETUP transaction handling

① **Start Setup package transmission**

Call USB_OTG_ReceiveSetupPacket function in drv_usb_device.h file to start the transmission of Setup package.

② **Receive Setup package in RxFIFO non-empty interrupt**

When USB receives data, RxFIFO non-empty interrupt is triggered, and USBD_RxFifoNoEmptyIsrHandler function (in usbd_interrupt.c file) will be handled. When the RxFIFO pop status indicates that this package is a Setup package, it will be saved to the global variable g_usbDev.reqData.pack.

③ **Handle Setup package**

The Setup package is processed by USBD_SetupProcess function in usbd_core.c. When it is determined that the received Setup package is a standard request, it shall be processed by the USBD_StandardReqeust function in usbd_stdReq.c; the class requests are processed by the global variable callback function g_usbDev.classReqHandler; the vendor requests are processed by the global variable callback function g_usbDev.vendorReqHandler.

The above two callback functions are assigned values when calling the initialization of USBD_Init function, and users can assign the function name as the function pointer value during initialization.

3.4 IN transaction handling

3.4.1 IN transaction of control endpoint

① **Start IN data transmission**

Call USBD_CtrlInData function in usbd_core.c to start IN transaction and configure IN data transmit buffer. For control transmission, at most the maximum packet length of the endpoint 0 can be transmitted at a single startup. If the data length to be transmitted is greater than the maximum packet length, it will be split for multiple transmissions.

When needing to transmit the device status, and transmitting 0-length in packet, the USBD_CtrlTxStatus function in usbd_core.c can be used to transmit the status phase of the device.

If there are multiple IN packets to be transmitted in the IN transaction, the IN data except the first packet is transmitted by USB_OTG_EnableInEpTransfer in drv_usb_device.c to enable transmission.

The difference between the functions of USBD_CtrlInData and USB_OTG_EnableInEpTransfer is that the former configures the transmit buffer, which will be used to transmit data during transmission, and the latter directly operates the register to enable transmission when the buffer is configured, and the former includes the latter

② **Transmit IN data packet in TxFIFO empty interrupt**

OTG SDK uses interrupt to push TxFIFO. When TxFIFO reaches the empty threshold, TxFIFO empty interrupt will be triggered. The threshold is determined by the TXFEL bit in GAHBCFG register. In the interrupt, call USBD_PushDataToTxFIFO function in usbd_core.c to write buffer data to TxFIFO.

③ **IN transmission is completed**

After the IN is transmitted, USBD_CtrlInProcess function in usbd_core.c will be called to process it and determine whether to continue to transmit data or whether to receive OUT status.

3.4.2 IN transaction of other endpoints

① **Start IN data transmission**

Call the USBD_RxData function in usbd_core.c to start OUT transaction.

② **Transmit IN data packet in TxFIFO empty interrupt**

In TxFIFO empty interrupt, call USBD_PushDataToTxFIFO function to write buffer data to TxFIFO.

③ **IN transmission is completed**

After the IN is transmitted, it is processed by the global variable callback function g_usbDev.inEpHandler. This function is assigned a value during initialization.

3.5 Example of user transmitting data

"Transmit" here refers to transmitting data from the slave to the host, which is an IN transaction. Endpoint 0 transmits data by calling USBD_CtrlInData function, and other endpoints transmit data by calling USBD_TxData function.

After we call the transmit function, no matter we are interrupting or polling, it is only "pre-" transmission. The real transmission time is after the transmit FIFO empty interrupt of USB is triggered. When we call the transmit function, the input data pointer and length information will be passed to g_usbDev. inBuf[ep], where ep represents different endpoints. If we need to judge whether the data to be transmitted is successfully executed, we can judge g_usbDev. inBuf[ep]. bufLen, which is because every time data is written to the transmit FIFO, g_usbDev. inBuf[ep]. bufLen will decrease until it becomes 0, indicating that all data have been transmitted.

```
uint8_t data[4] = {1, 2, 3, 4};

/**Endpoint 0 conducts control transmission by default. The demonstration here uses the endpoint 0 to transmit data*/
USBD_CtrlInData(data, 4);

/**The demonstration here uses endpoint 1 to transmit data. This function is applicable to interrupt, batch or synchronous endpoints*/
USBD_TxData(USB_EP_1, data, 4);
```

Figure 3 Code Example of Transmitting Data

3.6 Example of user receiving data

"Receive" here refers to receiving data from the slave to the host, which is an OUT transaction. Endpoint 0 transmits data by calling USBD_CtrlOutData function, and other endpoints transmit data by calling USBD_RxData function.

In the actual USB slave application, what data we can receive is determined by the host, and the receive function can only start receiving once. At this time, when the data comes, we can accurately save the data. Receiving data is often used in the processing of Class requests. When we receive the request from the host, we can know what data the host will transmit or need to receive. At this time, we can call the function for receiving data to start receiving.

```
uint8_t data[4] = {0, 0, 0, 0};

/**Endpoint 0 conducts control transmission by default. The demonstration here uses the endpoint 0 to receive data*/
USBD_CtrlOutData(data, 4);

/**The demonstration here uses endpoint 1 to transmit data. This function is applicable to interrupt, batch or synchronous endpoints*/
USBD_RxData(USB_EP_1, data, 4);
```

Figure 4 Code Example of Receiving Data

4 Use Example of USB Host

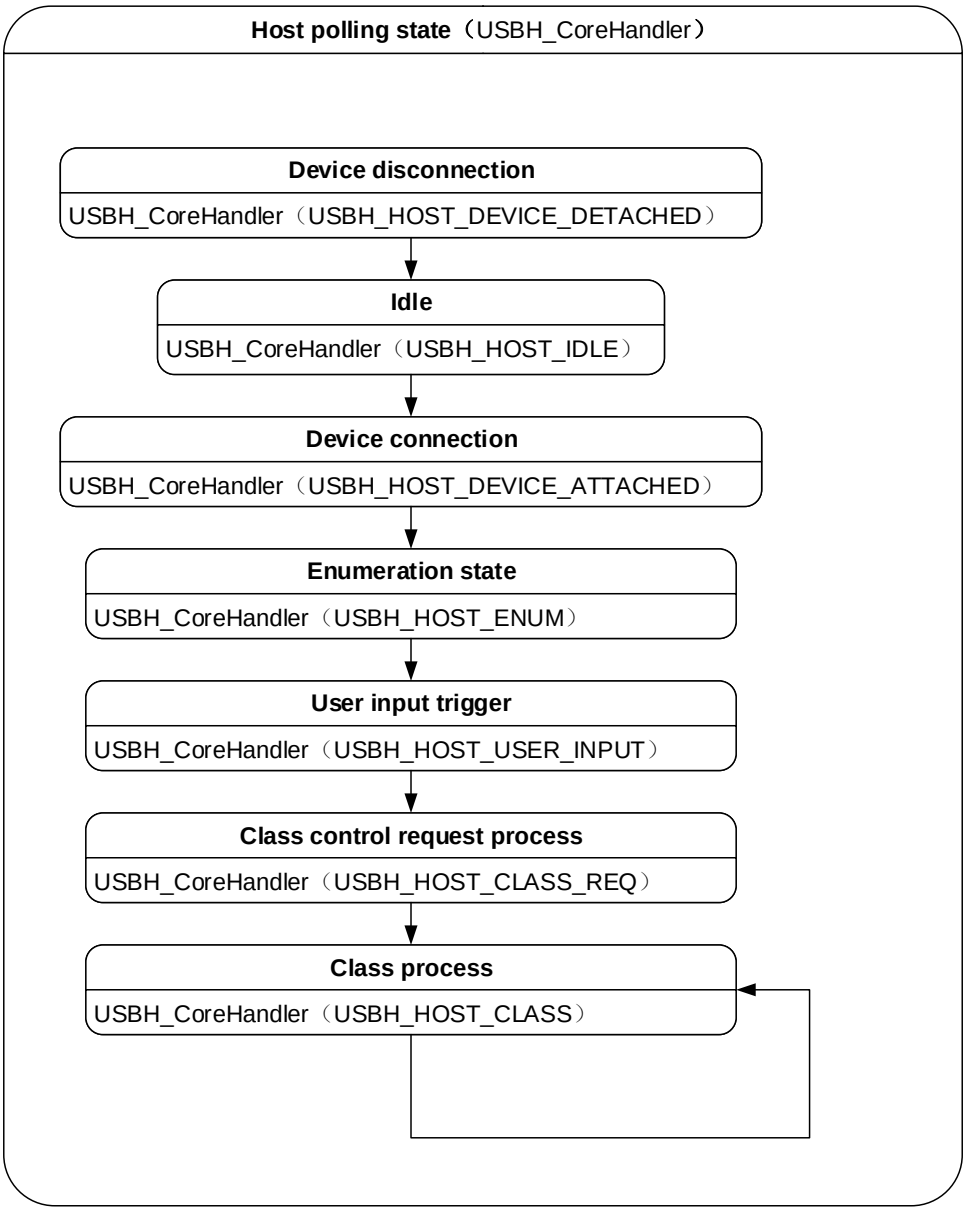

Figure 5 Host Polling Status Machine

The host polling state can be understood as a state machine, which needs to be polled all the time. The state machine processing of the host can be understood as an array of callback functions. It can achieve different states by accessing different arrays. We can use USBH_ConfigHostState function to switch different host states. In the state machine in the above figure, when the program just runs, generally the device is in the state of disconnection. If a device is inserted, it will enter the idle state. The idle state determines whether the device has been inserted. After the device is inserted, the device connection state will be executed. The first three states are relatively fixed, and users do not need to perform additional operations.

4.1 Enumeration state

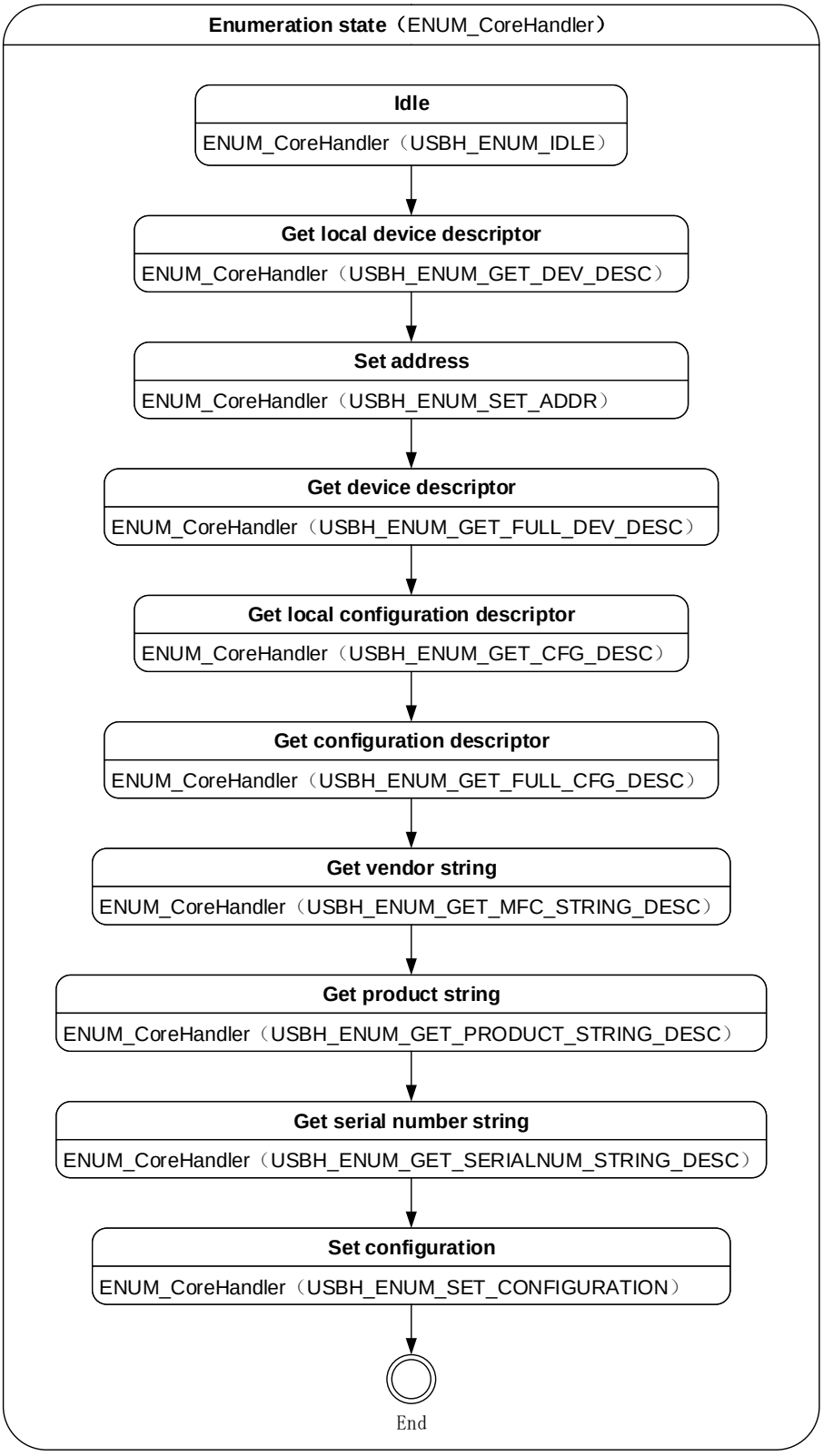

Figure 6 Enumeration State

Enumeration state is an important state for identifying devices after power on. In fact, device enumeration can be realized as long as three steps are implemented in the enumeration phase, namely, setting address, obtaining device descriptor and obtaining configuration descriptor. In practical

application, we may also need to check the string descriptor of the device. The vendor string, product string and serial number string are obtained by default in the driver.

4.2 User input state

```
static void USBH_UserInputHandler(void)
{
    if (g_userCallback.userInputHandler() == USER_OK)
    {
        g_usbHost.classInitHandler();
    }
}
```

Figure 7 User Input State Processing

The function in the above figure is the processing function of user input state, which is in usbd_core.c file. We can see that when the return value of user callback function g_userCallback.userInputHandler is USER_OK, the class initialization function, i.e. g_usbHost.classInitHandler, will be executed.

g_userCallback.userInputHandler callback function is user-defined. For example, it can return USER_OK when the key is pressed down

The class initialization function is used as an input parameter to complete the configuration during USB host initialization (USBH_Init). Generally, we recommend that users change the host state to USBH_HOST_CLASS_REQ or USBH_HOST_CLASS state in the class initialization function.

4.3 Class state

The Class state can be divided into two parts. One part is the control request using endpoint 0 for communication, and the other part is the transmission of endpoints other than endpoint 0. For example, when the HID class obtains the report descriptor, it will input the control request, while the interrupt is transmitted by another class.

The classReqHandler structure member in the USBH_Init initialization function is the callback function required by class for processing, and the other member classCoreHandler is the main callback function processed by class. Users shall build appropriate class callback functions at the initialization stage.

4.4 Suspend, Wake-up, and Error states

The three states are suspend state, wake-up state and error state. The error state will be triggered in some error situations, and the suspend and wake-up states need to be switched by users in characteristic occasions, and the corresponding callback function shall be built when calling USBH_Init initialization function.

4.5 Description of user interface

A g_userCallback global structure variable is built in usb_user.c file, and the structure members are callback function. This global variable can be modified by users as needed.

```
typedef struct
{
   USER_InitHandler_T                  initHandler;
   USER_DeInitHandler_T                deInitHandler;
   USER_ResetDevHandler_T              resetDevHandler;
   USER_DevAttachedHandler_T           devAttachedHandler;
   USER_DevDetachedHandler_T           devDetachedHandler;
    USER_DevSpeedDetectedHandler_T     devSpeedDetectedHandler;
    USER_DevDescHandler_T              devDescHandler;
   USER_CfgDescHandler_T               cfgDescHandler;
    USER_ManufacturerStringHandler_T   manufacturerStringHandler;
    USER_ProductStringHandler_T        productStringHandler;
    USER_SerialNumStringHandler_T      serialNumStringHandler;
   USER_EnumDoneHandler_T              enumDoneHandler;
    USER_UserInputHandler_T            userInputHandler;
   USER_ApplicationHandler_T           applicationHandler;
   USER_DeviceNotSupportedHandler_T    deviceNotSupportedHandler;
   USER_UnrecoveredErrHandler_T        unrecoveredErrHandler;
   USER_DelayCallBack_T                delay;
} USB_UserCallBack_T;
```

For the members of callback function structure:

initHandler is triggered when the USBH_Init function is executed; users can perform initialization;

deInitHandler is triggered when the USB device is disconnected or transmitting errors; users can perform some recovery operations;

resetDevHandler is triggered after the port is reset when the device is connected; users can reset it;

devAttachedHandler is triggered after the device is connected; users can perform some operation after the connection;

devAttachedHandler is triggered after the device is connected; users can perform some operation after the connection;

devSpeedDetectedHandler is triggered after the speed of the device is detected when the device is connected. For the speed of the device, namely, g_usbHost.speed, when its value is 0, it is high speed, 1 means full speed, and 2 means low speed;

devDescHandler is triggered after obtaining the device descriptor; users can extract the device descriptor here;

cfgDescHandler is triggered after obtaining the configuration descriptor; users can extract the configuration descriptor here;

manufacturerStringHandler is triggered after obtaining the manufacturer string; users can extract the manufacturer string here;

productStringHandler is triggered after obtaining the product string; users can extract the product string here;

serialNumStringHandler is triggered after obtaining the serial number string; users can extract the serial number string here;

enumDoneHandler is triggered after the device enumeration is completed; users can perform characteristic operations here;

userInputHandler is triggered in the host polling state machine; when its return value is USER_OK, the classInitHandler callback function configured during initialization will be executed, which means to process the class stage after the enumeration phase;

applicationHandler is actually constructed and placed by the user. It can usually be placed in the polling phase when the application layer is idle;

deviceNotSupportedHandler is a kind of error state, which is triggered when the host does not support such device;

unrecoveredErrHandler belongs to a kind of error state; apparently, this kind of error is an unknown error;

delay is USB delay function; users can use timer delay or variable count delay.

5 Revision History

Table 1 Document Revision History

Date	Version	Change History
June 23, 2022	1.0	New

Statement

This manual is formulated and published by Zhuhai Geehy Semiconductor Co., Ltd. (hereinafter referred to as "Geehy"). The contents in this manual are protected by laws and regulations of trademark, copyright and software copyright. Geehy reserves the right to correct and modify this manual at any time. Please read this manual carefully before using the product. Once you use the product, it means that you (hereinafter referred to as the "users") have known and accepted all the contents of this manual. Users shall use the product in accordance with relevant laws and regulations and the requirements of this manual.

1. Ownership of rights

This manual can only be used in combination with chip products and software products of corresponding models provided by Geehy. Without the prior permission of Geehy, no unit or individual may copy, transcribe, modify, edit or disseminate all or part of the contents of this manual for any reason or in any form.

The "Geehy" or "Geehy" words or graphics with "®" or "TM" in this manual are trademarks of Geehy. Other product or service names displayed on Geehy products are the property of their respective owners.

2. No intellectual property license

Geehy owns all rights, ownership and intellectual property rights involved in this manual.

Geehy shall not be deemed to grant the license or right of any intellectual property to users explicitly or implicitly due to the sale and distribution of Geehy products and this manual.

If any third party’s products, services or intellectual property are involved in this manual, it shall not be deemed that Geehy authorizes users to use the aforesaid third party’s products, services or intellectual property, unless otherwise agreed in sales order or sales contract of Geehy.

3. Version update

Users can obtain the latest manual of the corresponding products when ordering Geehy products.

If the contents in this manual are inconsistent with Geehy products, the agreement in Geehy sales order or sales contract shall prevail.

4. Information reliability

The relevant data in this manual are obtained from batch test by Geehy Laboratory or cooperative third-party testing organization. However, clerical errors in correction or errors caused by differences in testing environment may occur inevitably. Therefore, users should understand that Geehy does not bear any responsibility for such errors that may occur in this manual. The relevant data in this manual are only used to

guide users as performance parameter reference and do not constitute Geehy's guarantee for any product performance.

Users shall select appropriate Geehy products according to their own needs, and effectively verify and test the applicability of Geehy products to confirm that Geehy products meet their own needs, corresponding standards, safety or other reliability requirements. If loses are caused to users due to the user's failure to fully verify and test Geehy products, Geehy will not bear any responsibility.

5. Compliance requirements

Users shall abide by all applicable local laws and regulations when using this manual and the matching Geehy products. Users shall understand that the products may be restricted by the export, re-export or other laws of the countries of the product suppliers, Geehy, Geehy distributors and users. Users (on behalf of itself, subsidiaries and affiliated enterprises) shall agree and promise to abide by all applicable laws and regulations on the export and re-export of Geehy products and/or technologies and direct products.

6. Disclaimer

This manual is provided by Geehy "as is". To the extent permitted by applicable laws, Geehy does not provide any form of express or implied warranty, including without limitation the warranty of product merchantability and applicability of specific purposes.

Geehy will bear no responsibility for any disputes arising from the subsequent design and use of Geehy products by users.

7. Limitation of liability

In any case, unless required by applicable laws or agreed in writing, Geehy and/or any third party providing this manual "as is" shall not be liable for damages, including any general damages, special direct, indirect or collateral damages arising from the use or no use of the information in this manual (including without limitation data loss or inaccuracy, or losses suffered by users or third parties).

8. Scope of application

The information in this manual replaces the information provided in all previous versions of the manual.