AN0097
Application Note

AT32 MCU USB Device Library Application Note

Introduction

This application note mainly describes the AT32F4xx MCU USB device library architecture and application method, so that developers can use the library to quickly develop USB-related applications, and also introduces the corresponding USB routines in AT32 BSP.

The Universal Serial Bus (USB) is widely used for interconnection of devices like mouse, keyboard, headset, printer, etc. Developers can use the AT32 USB device library to implement USB device applications for most common device classes (HID, AUDIO, CDC, MSC, etc.), and modify these device classes according to application requirements. The AT32F4xx_Firmware_Library_V2.x.x is applicable.

Applicable products:

Part number	AT32F4xx

Contents

List of Tables

List of Figures

1 AT32 USB device protocol library

This section mainly introduces AT32 USB device library architecture and application method. The AT32 USB is based on USB Specification 2.0, and it supports full-speed mode and does not support high-speed mode. The USB device library is used to manage USB peripherals and implement USB basic protocol, so that developers can get started with development faster.

USB Device library is composed of the following modules, as shown in Figure 1.

- User application

  This module is developed by developers according to specific needs.

- USB Core Driver and USB Device Class

  USB Core Driver: This module implements USB device standard protocol stack, standard request and other interfaces.

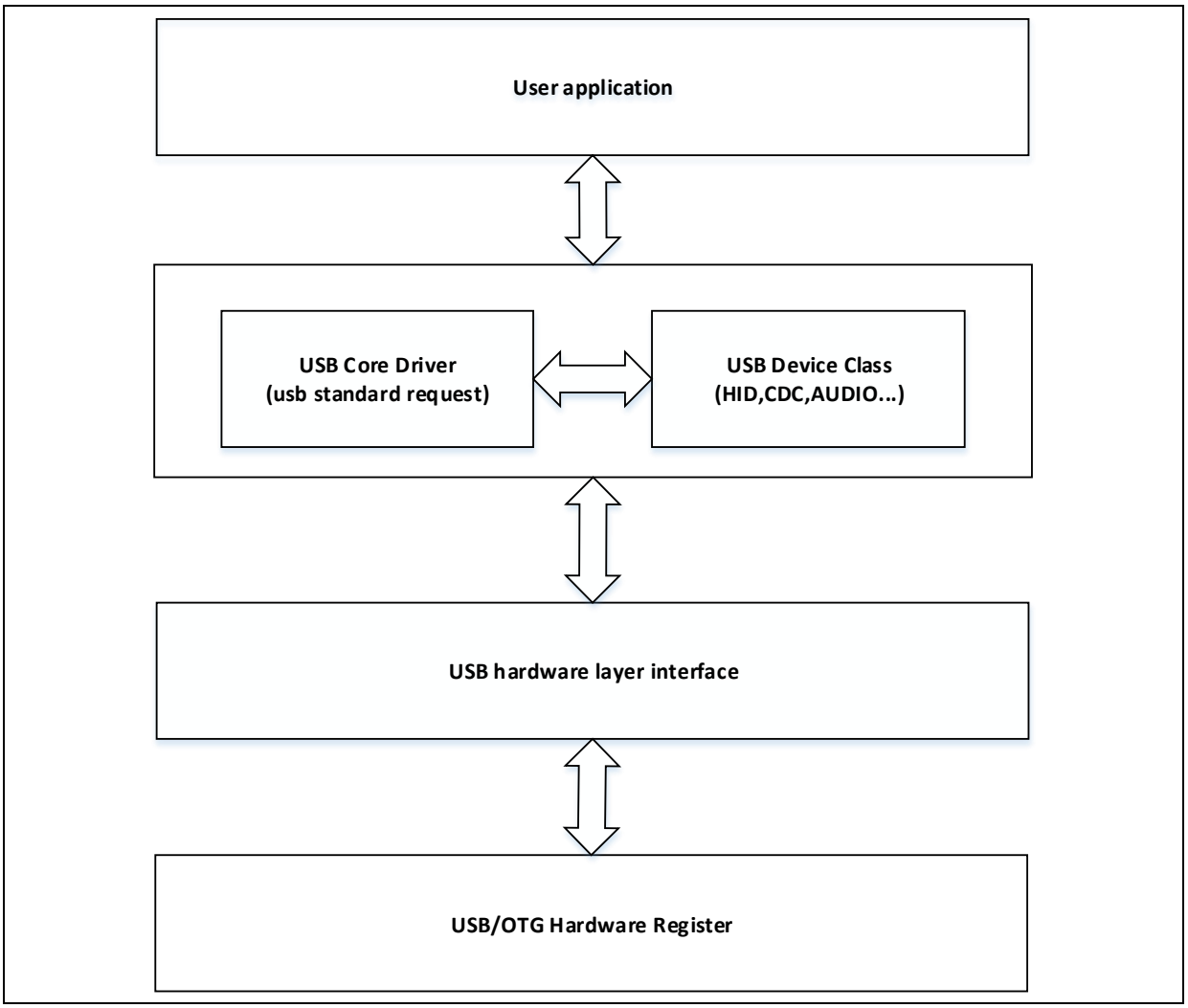
  USB Device Class: This module implements the description and device request for a specific USB device.

- USB hardware layer interface

  This module implements the hardware register abstract interface.

- USB/OTG peripherals

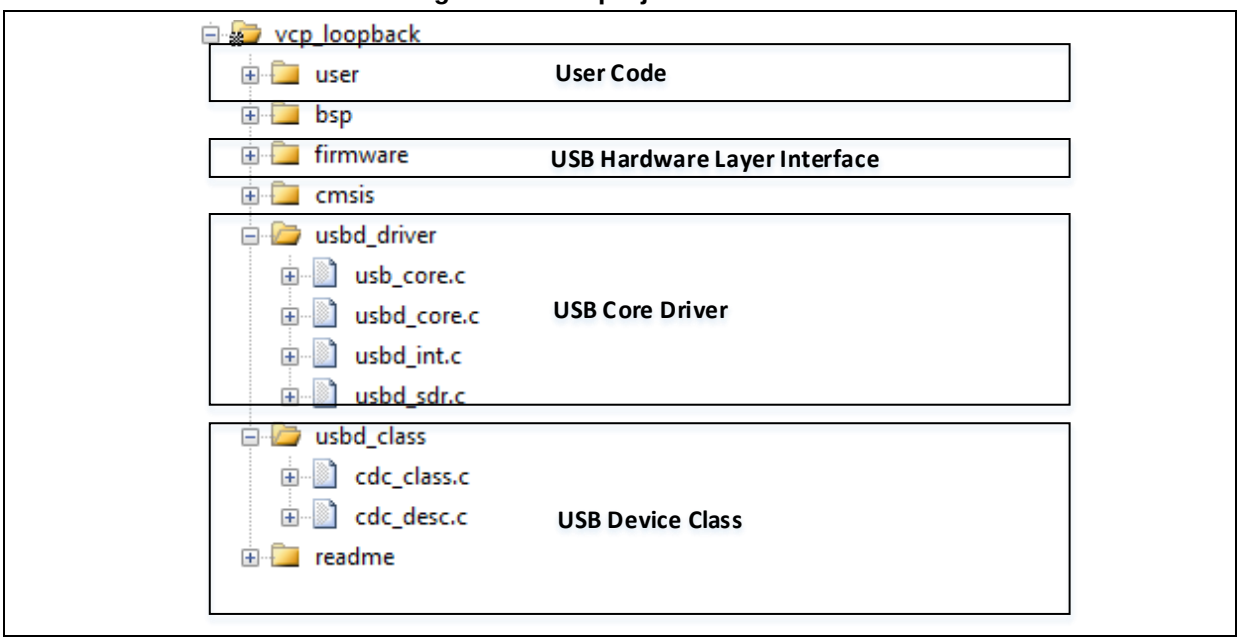
Figure 1. USB library architecture



1.1 AT32 USB library file

The AT32 USB application project structure is shown in the figure below.

Figure 2. AT32 project structure



Core Driver path: OTGFS -->middlewares\usb_drivers

USBFS -->middlewares\usbd_drivers

Device Class path: middlewares\usbd_class

The USB library files are listed below.

Table 1. USB library files

File name	Content
usb_core.c/h	USB Core
usbd_core.c/h	USB Device Core Library
usbd_int.c/h	USB interrupt processing
usbd_sdr.c/h	USB standard request
usbd_std.h	USB standard header
at32f4xx_usb.c/h	USB hardware register interface

Table 2. USB device class files

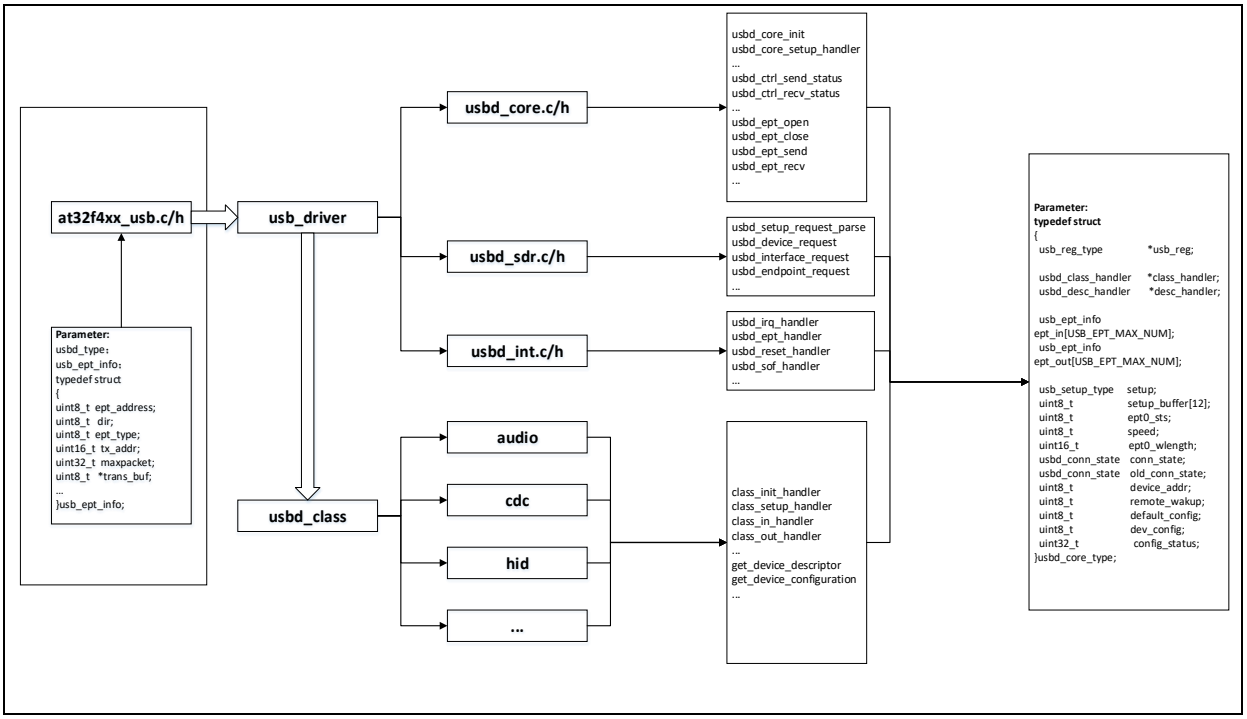
File name	Content
audio_class.c/h	audio device data processing interface
audio_desc.c/h	audio device description
cdc_class.c/h	cdc device data processing interface
cdc_desc.c/h	cdc device description
cdc_keyboard_class.c/h	cdc and keyboard composite device data processing interface
cdc_keyboard_desc.c/h	cdc and keyboard composite device description
custom_hid_class.c/h	custom hid data processing interface
custom_hid_desc.c/h	custom hid device description
hid_iap_class.c/h	hid iap data processing interface

File name	Content
hid_iap_desc.c/h	hid iap device description
keyboard_class.c/h	keyboard data processing interface
keyboard_desc.c/h	keyboard device description
mouse_class.c/h	mouse data processing interface
mouse_desc.c/h	mouse device description
msc_class.c/h	mass storage data processing interface
msc_bot_scsi.c/h	bulk-only transfer and scsi command processing interface
msc_desc.c/h	mass storage device description
printer_class.c/h	printer data processing interface
printer_desc.c/h	printer device description
audio_hid_class.c/h	audio and hid composite device data processing interface
audio_hid_desc.c/h	audio and hid composite device description

2 USB device library files

USB library implements USB device standard requests and meanwhile implements function interfaces for four transfer types (control, interrupt, bulk, isochronous).

Figure 3 USB library file structure



2.1 USB device file function interfaces

2.1.1 usbd_int.c

The usbd_int.c is mainly used for low level interrupt processing, and it may be modified depending on different USB peripherals. The USB peripherals of AT32 series chips are USBFS and OTGFS, and function interfaces are basically the same.

Table 3 usbd_int function interfaces

Interface	Description
usbd_irq_handler	USB interrupt entry function
usbd_ept_handler	USB endpoint interrupt function
usbd_reset_handler	USB reset interrupt function
usbd_sof_handler	USB SOF interrupt function
usbd_suspend_handler	USB suspend interrupt function
usbd_wakeup_handler	USB wakeup interrupt function
…	…

2.1.2 usbd_core.c

The usbd_core.c encapsulates different USB interfaces for calls, including some receive and send functions.

Table 4 usbd_core function interfaces

Interface	Description
usbd_core_init	usb device initialization interface function
usbd_core_in_handler	usb device IN transaction processing interface function
usbd_core_out_handler	usb device OUT transaction processing interface function
usbd_core_setup_handler	usb device setup transaction processing interface function
usbd_ctrl_unsupport	Unsupported transfer interface function
usbd_ctrl_send	Control transfer send interface function
usbd_ctrl_recv	Control transfer start receive interface function
usbd_ctrl_send_status	Control transfer IN handshake status interface function
usbd_ctrl_recv_status	Control transfer OUT handshake status interface function
usbd_set_stall	Set endpoint status STALL interface function
usbd_clear_stall	Clear endpoint status STALL interface function
usbd_ept_open	Open endpoint interface function
usbd_ept_close	Close endpoint interface function
usbd_ept_send	Endpoint send data interface function
usbd_ept_recv	Endpoint start receive interface function
usbd_connect	USB connect interface function
usbd_disconnect	USB disconnect interface function
usbd_set_device_addr	Set USB device address
usbd_get_recv_len	Get the current receive data length
usbd_connect_state_get	Get the current USB state
…	…

2.1.3 usbd_sdr.c

The usbd_sdr.c is used to process some USB standard requests.

Table 5 usbd_sdr function interfaces

Interface	Description
usbd_device_request	Device request interface function
usbd_interface_request	Interface request interface function

Interface	Description
usbd_endpoint_request	Endpoint request interface function
…	…

The supported device requests are listed below.

Table 6 Standard device requests

Request	Description
GET_STATUS	Get status
CLEAR_FEATURE	Clear feature
SET_FEATURE	Set feature
SET_ADDRESS	Set device address
GET_DESCRIPTOR	Get device description
GET_CONFIGURATION	Get device configuration
SET_CONFIGURATION	Set device configuration
GET_INTERFACE	Get alternate setting
SET_INTERFACE	Set alternate setting

2.1.4 usbd_xx_class.c

The usbd_xx_class.c is used for data processing of specific device classes by using structure functions. Developers can implement the following functions for specific device classes to realize different applications.
The function structure is shown below.

Table 7 Device class function structure

```
typedef struct
{
  usb_sts_type (*init_handler)(void *udev);                                  /*!< usb class init handler */
  usb_sts_type (*clear_handler)(void *udev);                                 /*!< usb class clear handler */
  usb_sts_type (*setup_handler)(void *udev, usb_setup_type *setup);   /*!< usb class setup handler */
  usb_sts_type (*ept0_tx_handler)(void *udev);                       /*!< usb class endpoint 0 tx complete handler */
  usb_sts_type (*ept0_rx_handler)(void *udev);                       /*!< usb class endpoint 0 rx complete handler */
  usb_sts_type (*in_handler)(void *udev, uint8_t ept_num);          /*!< usb class in transfer complete handler */
  usb_sts_type (*out_handler)(void *udev, uint8_t ept_num);         /*!< usb class out transfer complete handler */
  usb_sts_type (*sof_handler)(void *udev);                           /*!< usb class sof handler */
  usb_sts_type (*event_handler)(void *udev, usbd_event_type event);      /*!< usb class event handler */
  void              *pdata;                                                  /*!< usb class data pointer */
}usbd_class_handler;
```

Table 8 Device class function interfaces

Interface function	Description
init_handler	Class initialization interface
clear_handler	Class clear interface
setup_handler	Device class setup interface function
ept0_tx_handler	Endpoint 0 handshake stage send complete interface function

Interface function	Description
ept0_rx_handler	Endpoint 0 handshake stage receive complete interface function
in_handler	IN endpoint transfer complete interface function
out_handler	OUT endpoint receive complete interface function
sof_handler	SOF interrupt interface function
event_handler	USB event interface function

2.1.5 usbd_xx_desc.c

The usbd_xx_desc.c is a device description file, and device descriptions are sent back to the host through function interfaces in this file.

Table 9 Device description function structure

```
typedef struct
{
  usbd_desc_t *(*get_device_descriptor)(void);                  /*!< get device descriptor callback */
  usbd_desc_t *(*get_device_qualifier)(void);                   /*!< get device qualifier callback */
  usbd_desc_t *(*get_device_configuration)(void);               /*!< get device configuration callback */
  usbd_desc_t *(*get_device_other_speed)(void);                 /*!< get device other speed callback */
  usbd_desc_t *(*get_device_lang_id)(void);                     /*!< get device lang id callback */
  usbd_desc_t *(*get_device_manufacturer_string)(void);         /*!< get device manufacturer callback */
  usbd_desc_t *(*get_device_product_string)(void);              /*!< get device product callback */
  usbd_desc_t *(*get_device_serial_string)(void);               /*!< get device serial callback */
  usbd_desc_t *(*get_device_interface_string)(void);            /*!< get device interface string callback */
  usbd_desc_t *(*get_device_config_string)(void);               /*!< get device config callback */
}usbd_desc_handler;
```

Table 10 Device description interface functions

Interface function	Description
get_device_descriptor	Get device description
get_device_qualifier	Get device qualifier description
get_device_configuration	Get device configuration
get_device_other_speed	Get device other speed configuration
get_device_lang_id	Get device lang id
get_device_manufacturer_string	Get device manufacturer information
get_device_product_string	Get device product description
get_device_serial_string	Get device serial number
get_device_interface_string	Get interface information
get_device_config_string	Get configuration description

2.1.6 Other parameters

The parameter structure is shown below, and the usbd_core_type is used for parameter transfer in the USB device library.

Figure 4 Global structure

```
typedef struct
{
  usb_reg_type              *usb_reg;            /*!< usb register pointer */
  usbd_class_handler        *class_handler;      /*!< usb device class handler pointer */
  usbd_desc_handler         *desc_handler;       /*!< usb device descriptor handler pointer */
  usb_ept_info               ept_in[USB_EPT_MAX_NUM];    /*!< usb in endpoint infomation struct */
  usb_ept_info               ept_out[USB_EPT_MAX_NUM];   /*!< usb out endpoint infomation struct */
  usb_setup_type             setup;                      /*!< usb setup type struct */
  uint8_t                    setup_buffer[12];           /*!< usb setup request buffer */
  uint8_t                    ept0_sts;                   /*!< usb control endpoint 0 state */
  uint8_t                    speed;                      /*!< usb speed */
  uint16_t                   ept0_wlength;               /*!< usb endpoint 0 transfer length */
  usbd_conn_state            conn_state;                 /*!< usb current connect state */
  usbd_conn_state            old_conn_state;             /*!< usb save the previous connect state */
  uint8_t                    device_addr;                /*!< device address */
  uint8_t                    remote_wakup;               /*!< remote wakeup state */
  uint8_t                    default_config;             /*!< usb default config state */
  uint8_t                    dev_config;                 /*!< usb device config state */
  uint32_t                   config_status;              /*!< usb configure status */
}usbd_core_type;
```

USB device connect state is shown in the figure below.

The connect states include:

- Default
- Address
- Configured
- Suspend

Developers can use the usbd_connect_state_get function query the current connect state of USB device.

Figure 5 USB device connect state

```
typedef enum
{
  USB_CONN_STATE_DEFAULT                =1, /*!< usb device connect state default */
  USB_CONN_STATE_ADDRESSED,                 /*!< usb device connect state address */
  USB_CONN_STATE_CONFIGURED,                /*!< usb device connect state configured */
  USB_CONN_STATE_SUSPENDED                  /*!< usb device connect state suspend */
}usbd_conn_state;
```

USB devices return values, and USB function interface uses the following return values.

Figure 6 Function return values

```
typedef enum
{
  USB_OK,                /*!< usb status ok */
  USB_FAIL,              /*!< usb status fail */
```

```
  USB_WAIT,                /*!< usb status wait */
  USB_NOT_SUPPORT,   /*!< usb status not support */
  USB_ERROR,               /*!< usb status error */
}usb_sts_type;
```

2.2 Endpoint FIFO allocation

In order for USB to send and receive data normally, it is necessary to allocate a send/receive FIFO for each endpoint during initialization. The FIFO size can be confirmed according to the maximum packet length transmitted on the endpoint. Note that the sum of FIFO sizes allocated to all endpoints cannot exceed the maximum length that the system allocated to the USB buffer. Refer to Reference Manual for specific USB buffer size.

Developers can refer to the usb_conf.h routine to customize FIFO allocation for each endpoint. For USBFS and OTGFS, the endpoint FIFO allocation in usb_conf.h is slightly different.

2.2.1 USBFS endpoint FIFO allocation

USBFS endpoint allocation methods include automatic allocation and custom allocation.

- Automatic allocation:
  1. Enable automatic allocation by opening the USB_EPT_AUTO_MALLOC_BUFFER macro in usb_conf.h;
  2. When calling the open endpoint function (usbd_ept_open), the FIFO is allocated automatically according to the incoming maximum packet length;
  3. Double buffer mode (synchronous endpoint, double buffer bulk): call (usbd_ept_dbuffer_enable) before opening the endpoint to use double buffer mode; refer to the audio routine;
  4. The audio routine automatic allocation is configured as follows.

     usb_conf.h

```
/**
* @brief auto malloc usb endpoint buffer
*/
#define USB_EPT_AUTO_MALLOC_BUFFER    /*!< usb auto malloc endpoint tx and rx buffer */
```

     audio_class.c endpoint open:

```
static usb_sts_type class_init_handler(void *udev)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  usb_audio_type *paudio = (usb_audio_type *)pudev->class_handler->pdata;
  /* enable microphone in endpoint double buffer mode */
  usbd_ept_dbuffer_enable(pudev, USBD_AUDIO_MIC_IN_EPT);

  /* open microphone in endpoint */
  usbd_ept_open(pudev, USBD_AUDIO_MIC_IN_EPT, EPT_ISO_TYPE,
  AUDIO_MIC_IN_MAXPACKET_SIZE);

  /* enable speaker out endpoint double buffer mode */
```

```
  usbd_ept_dbuffer_enable(pudev, USBD_AUDIO_SPK_OUT_EPT);

  /* open speaker out endpoint */
  usbd_ept_open(pudev, USBD_AUDIO_SPK_OUT_EPT, EPT_ISO_TYPE,
  AUDIO_SPK_OUT_MAXPACKET_SIZE);

  /* enable speaker feedback endpoint double buffer mode */
  usbd_ept_dbuffer_enable(pudev, USBD_AUDIO_FEEDBACK_EPT);

  /* open speaker feedback endpoint */
  usbd_ept_open(pudev, USBD_AUDIO_FEEDBACK_EPT, EPT_ISO_TYPE,
  AUDIO_FEEDBACK_MAXPACKET_SIZE);

  /* start receive speaker out data */
  usbd_ept_recv(pudev, USBD_AUDIO_SPK_OUT_EPT, paudio->audio_spk_data,
  AUDIO_SPK_OUT_MAXPACKET_SIZE);

  return status;
}
```

- Custom allocation:
  1. Enable custom allocation by closing the USB_EPT_AUTO_MALLOC_BUFFER macro in usb_conf.h;
  2. When calling the open endpoint function (usbd_ept_open), call the usbd_ept_buf_custom_define function to customize FIFO allocation for endpoint; refer to the vcp_loopback routine;
  3. The vcp_loopback routine custom allocation is configured as follows.
     usb_conf.h:

```
/**
* @brief auto malloc usb endpoint buffer
*/
//#define USB_EPT_AUTO_MALLOC_BUFFER    /*!< usb auto malloc endpoint tx and rx buffer */

#ifndef USB_EPT_AUTO_MALLOC_BUFFER
      /**
* @brief user custom endpoint buffer
*            EPTn_TX_ADDR, EPTn_RX_ADDR must less than usb buffer size
*/
/* ept0 tx start address 0x40, size 0x40, so rx start address is 0x40 + 0x40 = 0x80 */
#define EPT0_TX_ADDR                    0x40      /*!< usb endpoint 0 tx buffer address offset */
#define EPT0_RX_ADDR                    0x80      /*!< usb endpoint 0 rx buffer address offset */

#define EPT1_TX_ADDR                    0xC0      /*!< usb endpoint 1 tx buffer address offset */
#define EPT1_RX_ADDR                    0x100     /*!< usb endpoint 1 rx buffer address offset */

#define EPT2_TX_ADDR                    0x140     /*!< usb endpoint 2 tx buffer address offset */
```

```
#define EPT2_RX_ADDR                0x180    /*!< usb endpoint 2 rx buffer address offset */

#define EPT3_TX_ADDR                0x00     /*!< usb endpoint 3 tx buffer address offset */
#define EPT3_RX_ADDR                0x00     /*!< usb endpoint 3 rx buffer address offset */

#define EPT4_TX_ADDR                0x00     /*!< usb endpoint 4 tx buffer address offset */
#define EPT4_RX_ADDR                0x00     /*!< usb endpoint 4 rx buffer address offset */

#define EPT5_TX_ADDR                0x00     /*!< usb endpoint 5 tx buffer address offset */
#define EPT5_RX_ADDR                0x00     /*!< usb endpoint 5 rx buffer address offset */

#define EPT6_TX_ADDR                0x00     /*!< usb endpoint 6 tx buffer address offset */
#define EPT6_RX_ADDR                0x00     /*!< usb endpoint 6 rx buffer address offset */

#define EPT7_TX_ADDR                0x00     /*!< usb endpoint 7 tx buffer address offset */
#define EPT7_RX_ADDR                0x00     /*!< usb endpoint 7 rx buffer address offset */
```

cdc_class.c endpoint open:

```
static usb_sts_type class_init_handler(void *udev)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  cdc_struct_type *pcdc = (cdc_struct_type *)pudev->class_handler->pdata;

#ifndef USB_EPT_AUTO_MALLOC_BUFFER
  /* use user define buffer address */
  usbd_ept_buf_custom_define(pudev, USBD_CDC_INT_EPT, EPT2_TX_ADDR);
  usbd_ept_buf_custom_define(pudev, USBD_CDC_BULK_IN_EPT, EPT1_TX_ADDR);
  usbd_ept_buf_custom_define(pudev, USBD_CDC_BULK_OUT_EPT, EPT1_RX_ADDR);
#endif

  /* open in endpoint */
  usbd_ept_open(pudev, USBD_CDC_INT_EPT, EPT_INT_TYPE,
  USBD_CDC_CMD_MAXPACKET_SIZE);

  /* open in endpoint */
  usbd_ept_open(pudev, USBD_CDC_BULK_IN_EPT, EPT_BULK_TYPE,
  USBD_CDC_IN_MAXPACKET_SIZE);

  /* open out endpoint */
  usbd_ept_open(pudev, USBD_CDC_BULK_OUT_EPT, EPT_BULK_TYPE,
  USBD_CDC_OUT_MAXPACKET_SIZE);

  /* set out endpoint to receive status */
  usbd_ept_recv(pudev, USBD_CDC_BULK_OUT_EPT, pcdc->g_rx_buff,
  USBD_CDC_OUT_MAXPACKET_SIZE);
```

```
    cdc_struct_init(pcdc);

    return status;
}
```

2.2.2 OTGFS endpoint FIFO allocation

OTGFS is shared by endpoint receive buffers; therefore, only one receive FIFO needs to be allocated for all OUT endpoints. For send buffers, it is necessary to allocate a dedicated FIFO to each send endpoint. Refer to the Reference Manual of the corresponding series for the available endpoints. Developers need to allocate OTGFS endpoint FIFO according to the maximum available packet length. Note that the FIFO size allocated to endpoints in usb_conf.h is in word (Byte).
Take the vcp_loopback routine as an example:

```
#define USBD_RX_SIZE            128   /* Shared receive FIFO, 128*4 Byte */
#define USBD_EP0_TX_SIZE         24   /* Endpoint 0 send FIFO, 24*4 Byte */
#define USBD_EP1_TX_SIZE         20   /* Endpoint 1 send FIFO, 20*4 Byte */
#define USBD_EP2_TX_SIZE         20   /* Endpoint 2 send FIFO, 20*4 Byte */
#define USBD_EP3_TX_SIZE         20   /* Endpoint 3 send FIFO, 20*4 Byte */
#define USBD_EP4_TX_SIZE         20   /* Endpoint 4 send FIFO, 20*4 Byte */
#define USBD_EP5_TX_SIZE         20   /* Endpoint 5 send FIFO, 20*4 Byte */
#define USBD_EP6_TX_SIZE         20   /* Endpoint 6 send FIFO, 20*4 Byte */
#define USBD_EP7_TX_SIZE         20   /* Endpoint 7 send FIFO, 20*4 Byte */
```

2.3 USB device initialization

Before using USB, developers need to initialize USB registers by calling USB initialization functions. For USBFS and OTGFS, functions to be called during initialization are different.

2.3.1 USBFS initialization

The USBFS initialization function usbd_core_init includes five parameters:

```
void usbd_core_init(usbd_core_type *udev,
                    usb_reg_type *usb_reg,
                    usbd_class_handler *class_handler,
                    usbd_desc_handler *desc_handler,
                    uint8_t core_id)
```

The initialization of vcp_loopback routine is as follows:
usbd_core_init (&usb_core_dev, USB, &class_handler, &desc_handler, 0);

2.3.2 OTGFS initialization

The OTGFS initialization function usbd_init includes five parameters:

```
usb_sts_type usbd_init(otg_core_type *otgdev,
                       uint8_t core_id, uint8_t usb_id,
                       usbd_class_handler *class_handler,
```

```
                    usbd_desc_handler *desc_handler)
```

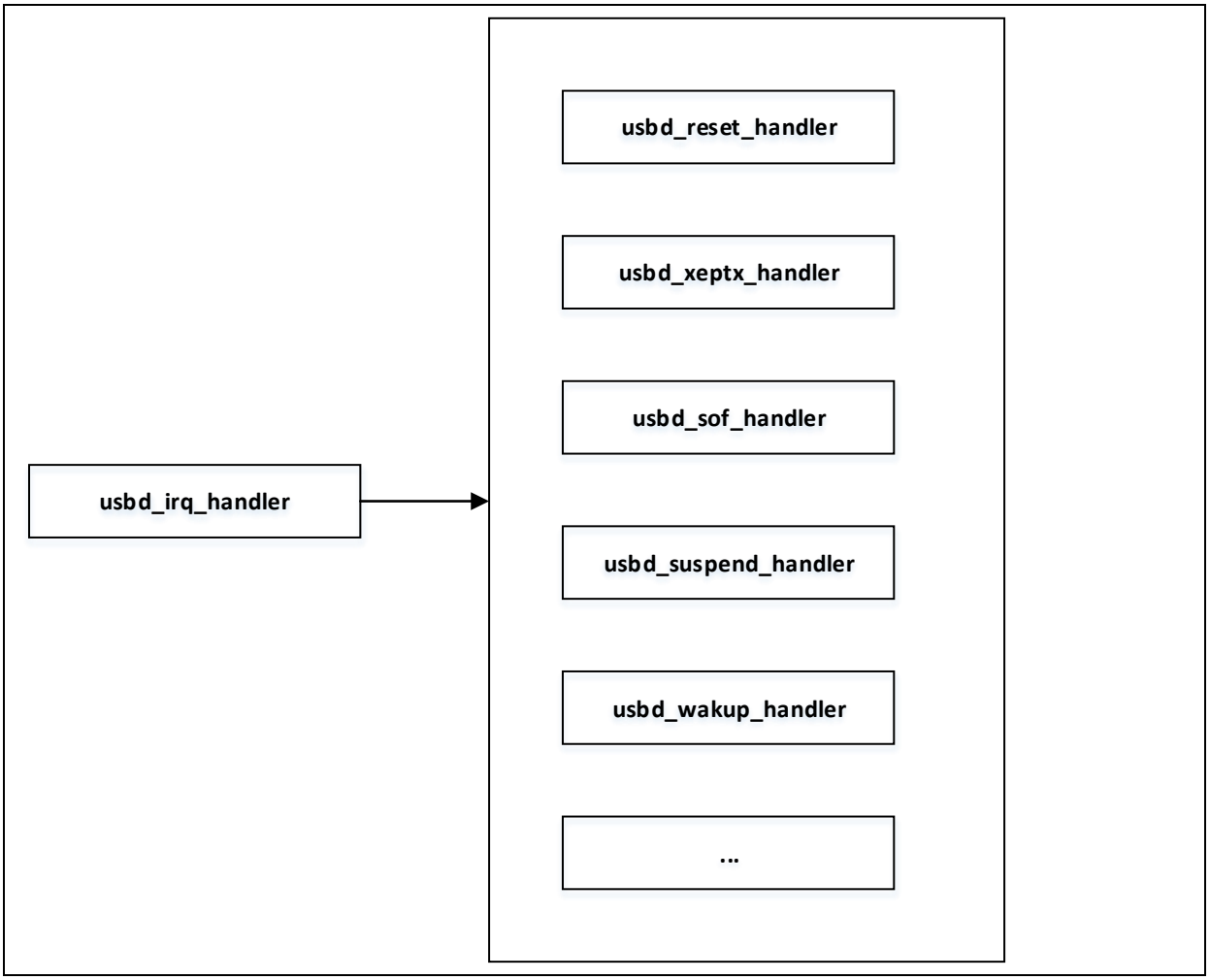
The initialization of vcp_loopback routine is as follows:

```
usbd_init(&otg_core_struct,
          USB_FULL_SPEED_CORE_ID,
          USB_ID,
          &class_handler,
          &desc_handler);
```

2.4 USB device interrupt processing

The USB interrupt entry function usbd_irq_handler is used to handle all USB interrupts, including Reset, endpoint receive/send data, SOF, suspend and wakeup. The typical interrupt handlers are shown as follows.

Figure 7 USB interrupt processing functions



2.4.1 Reset interrupt processing

When USB device detects a Reset signal on the bus, a Reset interrupt is generated. When the software receives the reset interrupt, it is necessary to implement basic initialization for subsequent enumeration processing.

Reset interrupt processing function usbd_reset_handler:

- Endpoint FIFO initialization
- Set device address to 0
- Endpoint 0 initialization
- Call device class event function
  udev->class_handler->event_handler(udev, USBD_RESET_EVENT);

2.4.2 Endpoint interrupt processing

When the USB endpoint data receive/send is completed, a corresponding endpoint complete interrupt is generated. The endpoint complete interrupt handles the data sent and received.
The interrupt processing function is **usbd_xeptx_handler.**

2.4.3 SOF interrupt processing

After the SOF interrupt is enabled, a SOF interrupt will be generated every time the USB device receives a SOF sent by the host.
The interrupt processing function is usbd_sof_handler.

- The interrupt processing function will call the device class SOF processing function
  udev->class_handler->sof_handler(udev);

2.4.4 Suspend interrupt processing

When the bus meets the suspend condition, the USB device will generate a suspend interrupt, which can be used by developers to determine whether to enter low-power mode.
Interrupt processing function: usbd_suspend_handler

- Set connect state suspend
- Set device to suspend state
- Call the device class event processing function
  udev->class_handler->event_handler(udev, USBD_SUSPEND_EVENT);

2.4.5 Wakeup interrupt processing

When the device is in suspend state, if there is a wakeup signal on the bus, the USB device will generate a wakeup interrupt.
Interrupt processing function: usbd_wakeup_handler

- Device exists suspend state
- Connect state is set to the state before entering suspend state
- Call the device class event processing function
  udev->class_handler->event_handler(udev, USBD_WAKEUP_EVENT);

2.5 USB device endpoint data processing flow

After the USB device receives the data packet sent from the host, the endpoint 0 data (IN/OUT/SETUP) will be processed separately, and the data of other endpoints will be processed by calling the device class IN/OUT handler.
The data processing flow is shown below.

Figure 8 Endpoint data processing flow

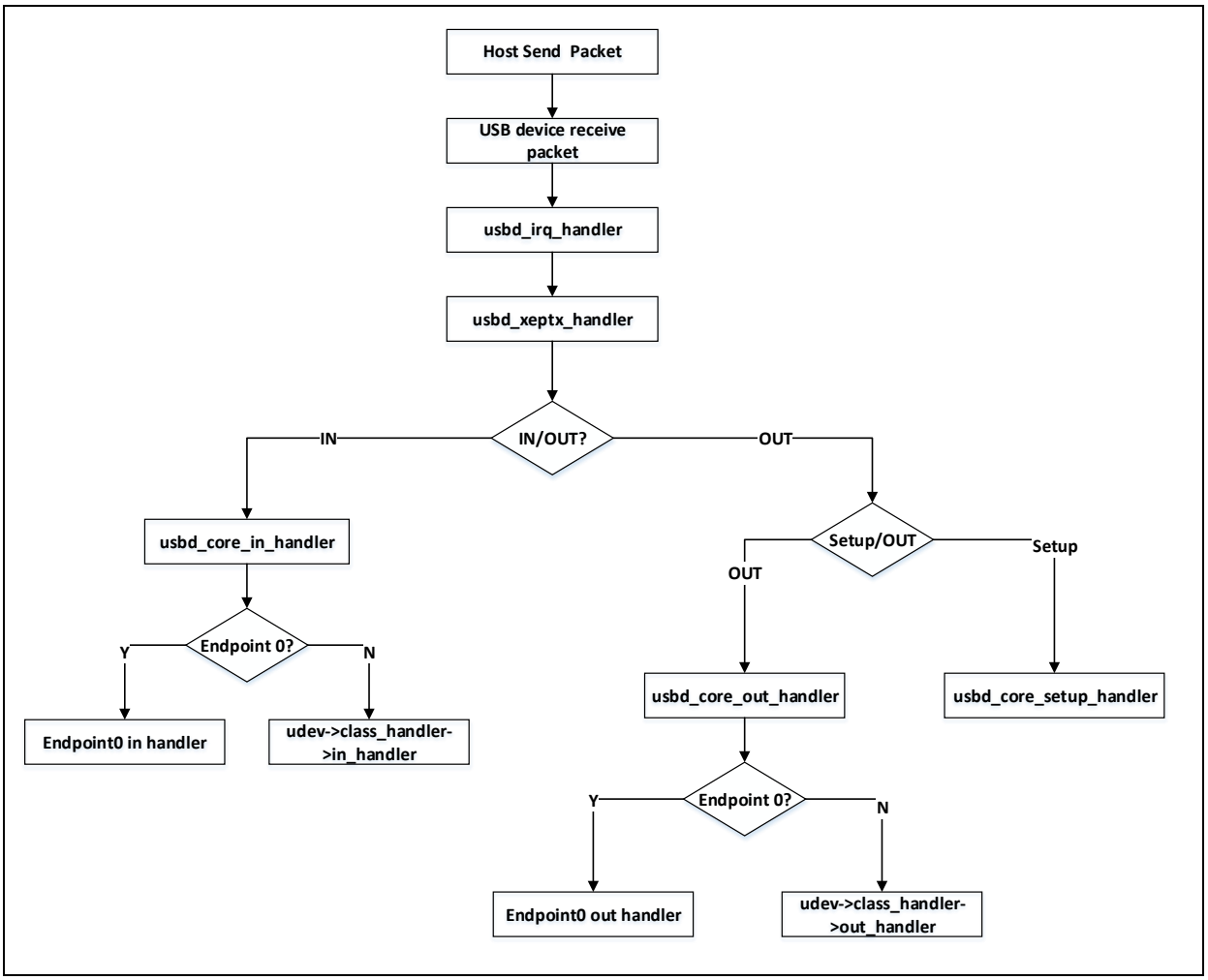
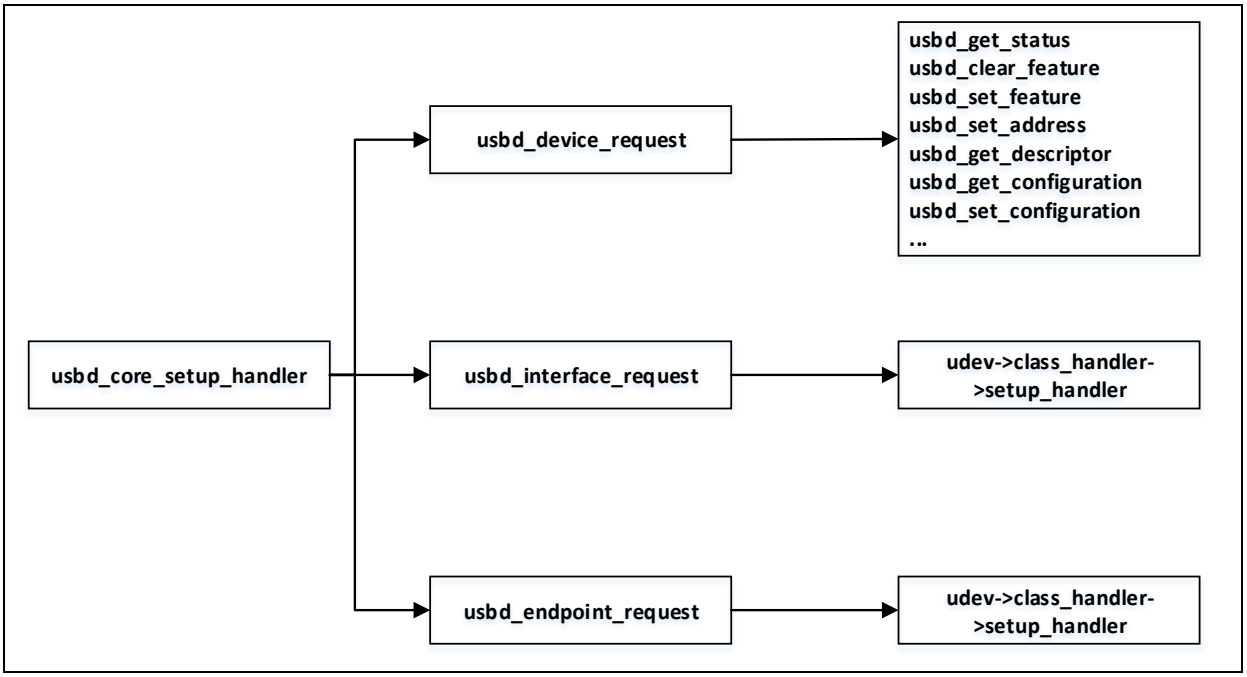


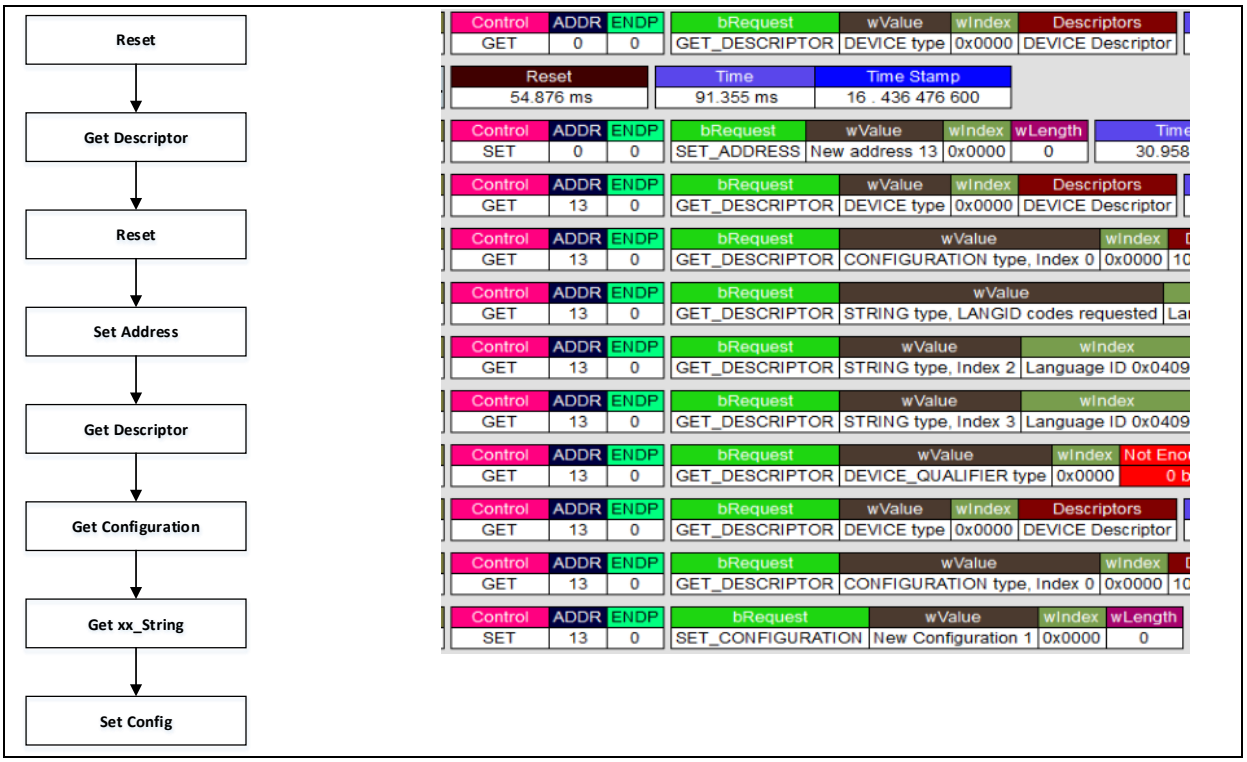
Figure 9 Setup processing flow



2.5.1 USB control endpoint enumeration process

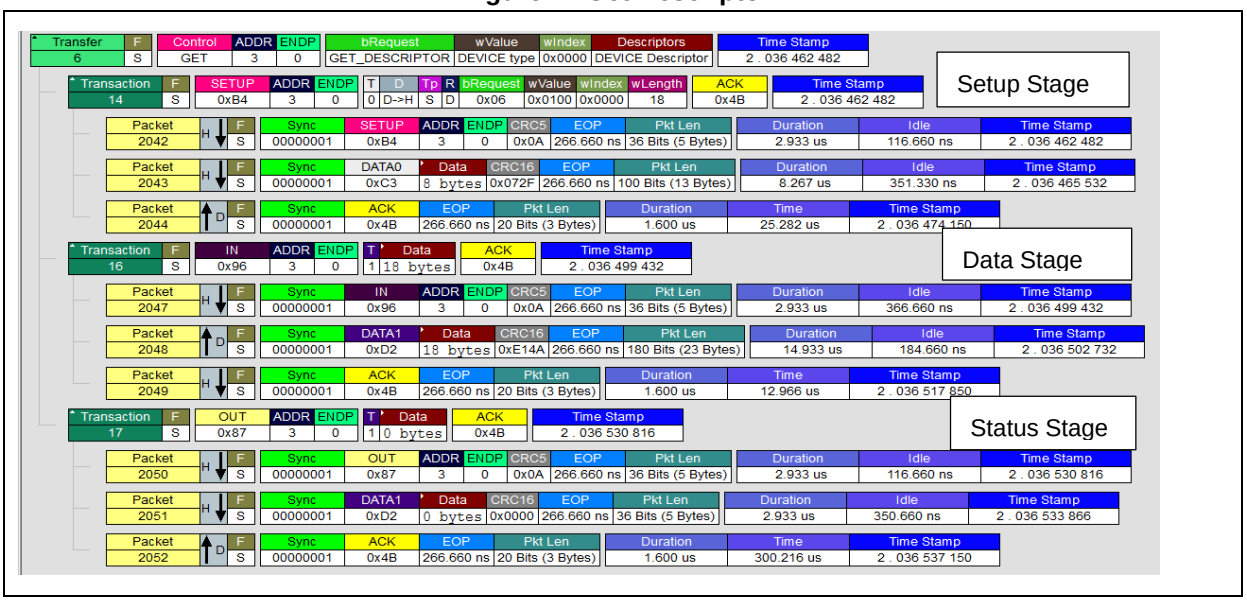
After the device is connected to the host, perform enumeration through the control endpoint (endpoint 0). The typical enumeration process is as follows.

Figure 10 USB enumeration process



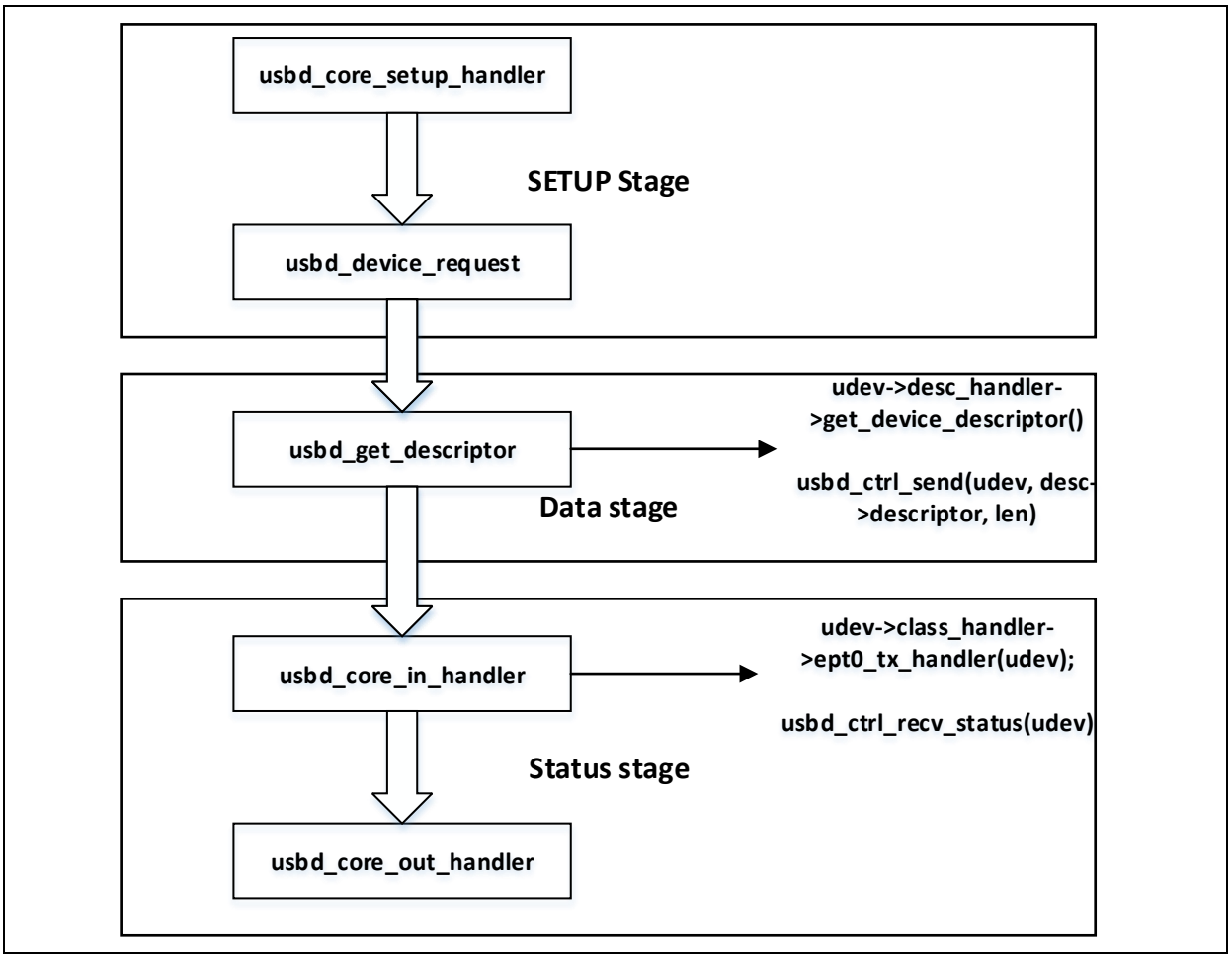
USB control transfer process includes the SETUP-DATA-STATUS stages. The process GET_DESCRIPTOR of the host to get the device information is shown below.

Figure 11 Get Descriptor



The USB library handles the Get Descriptor process, as shown below.

Figure 12 USB library handles Get Descriptor process



USB device request format (Setup request)

Figure 13 Setup request format

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

2.5.2 USB application endpoint processing flow

The application endpoint refers to the non-0 endpoint used by customers in actual application, such as Bulk, interrupt, ISO and other types of endpoints. The data of these application endpoints are processed by using the in_handler and out_handler callback functions. Developers only need to implement data processing for specific endpoints in class_in_handler and class_out_handler in xxx_class.c.

IN endpoint data processing:

Figure 14 IN endpoint data processing

```
usb_sts_type class_in_handler(void *udev, uint8_t ept_num)
{
  usbd_core_type *pudev = (usbd_core_type *)udev;
  usb_sts_type status = USB_OK;
  if((ept_num & 0x7F) == 0x1)
  {
     /* IN endpoint 1 transfer complete*/
    /*trans next packet data
      usbd_ept_send(pudev, 0x1, send_data, len);
    */
  }
  if((ept_num & 0x7F) == 0x2)
  {
     /* IN endpoint 2 transfer complete*/
    /*trans next packet data
      usbd_ept_send(pudev, 0x2, send_data, len);
    */
  }
  ….
  return status;
}
```

OUT endpoint data processing:

Figure 15 OUT endpoint data processing

```
usb_sts_type class_out_handler(void *udev, uint8_t ept_num)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  if((ept_num & 0x7F) == 0x1)
  {
    /* get endpoint receive data length   */
    g_rxlen = usbd_get_recv_len(pudev, ept_num);
    /* start receive next data
    usbd_ept_recv(pudev, 0x1, buffer, max_packet_size);
    */
  }
```

```
    if((ept_num & 0x7F) == 0x2)
    {
      /* get endpoint receive data length    */
      g_rxlen = usbd_get_recv_len(pudev, ept_num);
      /* start receive next data
      usbd_ept_recv(pudev, 0x2, buffer, max_packet_size);
      */
    }
    …
    return status;
}
```

3 USB device class routines

This chapter introduces different device class routines that are implemented by using the AT32 USB. The following routines are implemented:

- audio
- custom_hid
- keyboard
- mouse
- msc (mass storage)
- printer
- vcp_loopback
- virtual_msc_iap
- composite_vcp_keyboard
- hid_iap
- composite_audio_hid
- virtual_comport

3.1 Audio routine

Audio routine implements the Speaker and Microphone according to Audio V1.0. Audio data uses isochronous transfer; Speaker uses isochronous OUT transfer, and Microphone uses isochronous IN transfer.

This routine runs on AT-START evaluation board, and Audio Speaker and Microphone are implemented based on Audio Arduino Demo Board. The AT-START and Audio Arduino Board need to be connected during experiment. For details about the development board, refer to *UM_Audio Arduino Daughter Board_V1.0/V2.0*. For related Audio protocol, refer to *Universal Serial Bus Device Class Define for Audio Device V1.0*.

3.1.1 Implement functions

Implement a composite audio device of Speaker and Microphone to perform audio playback and recording synchronously.

Speaker features:

- Support 16 K and 48 K sampling rates
- Support sampling rate switching

- Support 16-bit sampling
- Support mute mode
- Support volume adjustment
- Support feedback function
- Support dual-channel

Microphone features:

- Support 16 K and 48 K sampling rates
- Support sampling rate switching
- Support 16-bit sampling
- Support mute mode
- Support volume adjustment
- Support dual-channel

3.1.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration and partial control of audio
- Endpoint 1 IN: for Microphone recording data
- Endpoint 1 OUT: for Speaker playback data
- Endpoint 2 IN: for Feedback data

I2C:

- Use I2C to send control information to the audio Board.

I2S:

- Use I2S1 to send data to the audio board (speaker);
- Use I2S2 to receive data from the audio board (microphone).

DMA:

- Use DMA1 channel 3 to transfer I2S1 data;
- Use DMA1 channel 4 to transfer I2S2 data.

TIMER:

- Use TIMER to generate a clock required by Codec.

3.1.3 Audio implementation

USB Audio device class implementation source files are in audio_class.c and audio_desc.c. The control of external codec and audio data processing are implemented in audio_codec.c, and the specific setting function in audio_codec.c will be called according to the host request setting. Note that the USB device endpoint FIFO size is configured in usb_conf.h and is allocated according to the maximum packet length transmitted on the specific endpoint.

3.1.3.1 Device description: (audio_desc.c/h)

- Audio device description (g_usbd_descriptor)
- Audio device configuration description (g_usbd_configuration)
  AC interface
  Microphone Streaming interface
  Microphone Terminal INPUT/OUTPUT
  Microphone Feature Unit

Microphone Endpoint
Speaker Streaming interface
Speaker Terminal INPUT/OUTPUT
Speaker Feature Unit
Speaker Endpoint
Feedback Endpoint

- Lang id (g_string_lang_id)
- Serial number (g_string_serial)
- Vendor/product ID (audio_desc.h)

```
#define USBD_AUDIO_VENDOR_ID                    0x2E3C
#define USBD_AUDIO_PRODUCT_ID                   0x5730
```

- Manufacturer, product name, configuration description and interface description (audio_desc.h)

```
#define USBD_AUDIO_DESC_MANUFACTURER_STRING      "Artery"
#define USBD_AUDIO_DESC_PRODUCT_STRING           "AT32 Audio"
#define USBD_AUDIO_DESC_CONFIGURATION_STRING     "Audio Config"
#define USBD_AUDIO_DESC_INTERFACE_STRING         "Audio Interface"
```

3.1.3.2 Data processing (audio_class.c/h audio_codec.c/h)

- Endpoint initialization (class_init_handler)

```
/* open microphone in endpoint */
usbd_ept_open(pudev, USBD_AUDIO_MIC_IN_EPT, EPT_ISO_TYPE, AUDIO_MIC_IN_MAXPACKET_SIZE);
/* open speaker out endpoint */
usbd_ept_open(pudev, USBD_AUDIO_SPK_OUT_EPT, EPT_ISO_TYPE, AUDIO_SPK_OUT_MAXPACKET_SIZE);
/* open speaker feedback endpoint */
usbd_ept_open(pudev, USBD_AUDIO_FEEDBACK_EPT, EPT_ISO_TYPE, AUDIO_FEEDBACK_MAXPACKET_SIZE);
```

- Endpoint initialization (class_clear_handler)

```
/* close in endpoint */
usbd_ept_close(pudev, USBD_AUDIO_MIC_IN_EPT);
/* close in endpoint */
usbd_ept_close(pudev, USBD_AUDIO_FEEDBACK_EPT);
/* close out endpoint */
usbd_ept_close(pudev, USBD_AUDIO_SPK_OUT_EPT);
```

- Audio control request (class_setup_handler)

Implement the following audio control requests:

Request	Description
GET_CUR	Get mute mode state and volume level
SET_CUR	Set mute mode and volume
GET_MIN	Get the minimum volume property
GET_MAX	Get the maximum volume property
GET_RES	Get volume resolution property

```
switch(setup->bRequest)
{
      case AUDIO_REQ_GET_CUR:
            audio_req_get_cur(pudev, setup);
            break;
      case AUDIO_REQ_SET_CUR:
            audio_req_set_cur(pudev, setup);
            break;
       case AUDIO_REQ_GET_MIN:
             audio_req_get_min(pudev, setup);
             break;
       case AUDIO_REQ_GET_MAX:
             audio_req_get_max(pudev, setup);
             break;
       case AUDIO_REQ_GET_RES:
             audio_req_get_res(pudev, setup);
             break;
        default:
             usbd_ctrl_unsupport(pudev);
             break;
}
```

- Audio volume, mute mode and sampling rate setting (class_ept0_rx_handler)
  This function is used for subsequent processing after receiving the setting data sent from the host, including the volume, mute mode and sampling rate settings.

```
switch(audio_struct.request_no)
{
  case AUDIO_VOLUME_CONTROL:
  if(audio_struct.interface == AUDIO_SPK_FEATURE_UNIT_ID)
  {
    …
  }
  else
  {
    …
  }
  case AUDIO_MUTE_CONTROL:
  if(audio_struct.interface == AUDIO_SPK_FEATURE_UNIT_ID)
  {
    …
  }
  else
  {
    …
  }
```

```
    break;
    case AUDIO_FREQ_SET_CONTROL:
    if(audio_struct.enpd == USBD_AUDIO_MIC_IN_EPT)
    {
      …
    }
    else
    {
      …
    }
    break;
    default:
    break;
}
```

- Microphone and Feedback data transfer (class_in_handler)

```
usb_audio_type *paudio = (usb_audio_type *)pudev->class_handler->pdata;
if((ept_num & 0x7F) == (USBD_AUDIO_MIC_IN_EPT & 0x7F))
{
    len = audio_codec_mic_get_data(paudio->audio_mic_data);
    usbd_flush_tx_fifo(udev, USBD_AUDIO_MIC_IN_EPT);
    usbd_ept_send(udev, USBD_AUDIO_MIC_IN_EPT, paudio->audio_mic_data, len);
}

else if((ept_num & 0x7F) == (USBD_AUDIO_FEEDBACK_EPT & 0x7F))
{
    paudio->audio_feedback_state = 0;
}
```

- Speaker data receive (class_out_handler)

```
/* get endpoint receive data length    */
g_rxlen = usbd_get_recv_len(pudev, ept_num);

if((ept_num & 0x7F) == (USBD_AUDIO_SPK_OUT_EPT & 0x7F))
{
    /* speaker data*/
    audio_codec_spk_fifo_write(paudio->audio_spk_data, g_rxlen);
    paudio->audio_spk_out_stage = 1;
    /* get next data */
    usbd_ept_recv(pudev, USBD_AUDIO_SPK_OUT_EPT, paudio->audio_spk_data,
AUDIO_SPK_OUT_MAXPACKET_SIZE);
}
```

- To implement specific control of codec and data processing in audio_codec.c, developers need to implement the following functions:

```
void audio_codec_spk_fifo_write(uint8_t *data, uint32_t len);
uint32_t audio_codec_mic_get_data(uint8_t *buffer);
uint8_t audio_codec_spk_feedback(uint8_t *feedback);
void audio_codec_spk_alt_setting(uint32_t alt_seting);
void audio_codec_mic_alt_setting(uint32_t alt_seting);
void audio_codec_set_mic_mute(uint8_t mute);
void audio_codec_set_spk_mute(uint8_t mute);
void audio_codec_set_mic_volume(uint16_t volume);
void audio_codec_set_spk_volume(uint16_t volume);
void audio_codec_set_mic_freq(uint32_t freq);
void audio_codec_set_spk_freq(uint32_t freq);
```

The above functions can be implemented according to related methods in the routine or modified according to the codec actually used by developers. The codec initialization is not detailed in this application note.

- audio routine function configuration

  The current audio routine can be configured in audio_conf.h, such as the speaker function and sampling rate. Optional configurations are as follows:

```
#define AUDIO_SUPPORT_SPK                1
#define AUDIO_SUPPORT_MIC                1
#define AUDIO_SUPPORT_FEEDBACK           0

#define AUDIO_SUPPORT_FREQ_16K           1
#define AUDIO_SUPPORT_FREQ_48K           1


#define AUDIO_SUPPORT_FREQ          (AUDIO_SUPPORT_FREQ_16K + \
                                     AUDIO_SUPPORT_FREQ_48K \
                                    )

#define AUDIO_FREQ_16K                16000
#define AUDIO_FREQ_48K                48000
#define AUDIO_BITW_16                 16

#define AUDIO_MIC_CHANEL_NUM           2
#define AUDIO_MIC_DEFAULT_BITW        AUDIO_BITW_16

#define AUDIO_SPK_CHANEL_NUM           2
#define AUDIO_SPK_DEFAULT_BITW        AUDIO_BITW_16


#define AUDIO_SUPPORT_MAX_FREQ        48
#define AUDIO_DEFAULT_FREQ            AUDIO_FREQ_48K
#define AUDIO_DEFAULT_BITW             AUDIO_BITW_16
```

3.1.4 How to develop based on audio routine

This section briefly introduces how to modify the code of audio routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify audio configuration (audio_conf.h)
- According to specific functional requirements to modify device descriptions (audio_desc.c, audio_desc.h)
  Device descriptions (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Other descriptions
- According to specific functional requirements to modify endpoints (audio_class.c, audio_class.h)
  Endpoint definition (audio_class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)
- Modify audio control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- Audio data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)
- According to specific requirements to implement codec function interface (audio_codec.c)

3.2 custom_hid routine

The custom_hid implements a HID (human interface device) function, and communicates with the upper computer (Artery_UsbHid_Demo) to complete some simple interactive operations. HID communicates with the upper computer through interrupt transfer, and this routine runs on the AT-START development board. Please download the upper computer from the official website, and refer to *Human Interface Devices (HID) V1.11* for HID protocol.

3.2.1 Implement functions

- Upper computer display key states
- Control the development board LED on/off state through the upper computer
- HID data loopback function

3.2.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration
- Endpoint 1 IN: for data sending
- Endpoint 1 OUT: for data receiving

3.2.3 custom_hid implementation

The custom_hid device class implementation source code is mainly in custom_hid_class.c and custom_hid_desc.c, and these two source files implement device descriptions and device

processing.

3.2.3.1 Device description (custom_hid_desc.c/h)

- custom hid device description (g_usbd_descriptor)
- custom hid device configuration description (g_usbd_configuration)
  HID interface
  HID Endpoint
- custom hid report descriptor (g_usbd_hid_report)
  HID_REPORT_ID_2 (LED2)
  HID_REPORT_ID_3 (LED3)
  HID_REPORT_ID_4 (LED4)
  HID_REPORT_ID_5 (BUTTON)
  HID_REPORT_ID_6 (LOOPBACK DATA)
- Lang id (g_string_lang_id)
- Serial number (g_string_serial)
- Vendor/product ID (custom_hid_desc.h)

```
#define USBD_CUSHID_VENDOR_ID                0x2E3C
#define USBD_CUSHID_PRODUCT_ID               0x5745
```

- Manufacturer, product name, configuration description and interface description (custom_hid_desc.h)

```
#define USBD_CUSHID_DESC_MANUFACTURER_STRING     "Artery"
#define USBD_CUSHID_DESC_PRODUCT_STRING          "Custom HID"
#define USBD_CUSHID_DESC_CONFIGURATION_STRING    "Custom HID Config"
#define USBD_CUSHID_DESC_INTERFACE_STRING        "Custom HID Interface"
```

3.3.3.2 Data processing (custom_hid_class.c/h)

- Endpoint initialization (class_init_handler)

```
/* open custom hid in endpoint */
usbd_ept_open(pudev, USBD_CUSTOM_HID_IN_EPT, EPT_INT_TYPE, USBD_CUSTOM_IN_MAXPACKET_SIZE);
/* open custom hid out endpoint */
usbd_ept_open(pudev, USBD_CUSTOM_HID_OUT_EPT, EPT_INT_TYPE, USBD_CUSTOM_OUT_MAXPACKET_SIZE);
```

- Endpoint clear (class_clear_handler)

```
/* close custom hid in endpoint */
usbd_ept_close(pudev, USBD_CUSTOM_HID_IN_EPT);
/* close custom hid out endpoint */
usbd_ept_close(pudev, USBD_CUSTOM_HID_OUT_EPT);
```

- HID device class request (class_setup_handler)
  Implement the following requests:
  SET_PROTOCOL
  GET_PROTOCOL
  SET_IDLE
  GET_IDLE

SET_REPORT

The code is as follows:

```
switch(setup->bRequest)
{
  case HID_REQ_SET_PROTOCOL:
    hid_protocol = (uint8_t)setup->wValue;
    break;
  case HID_REQ_GET_PROTOCOL:
    usbd_ctrl_send(pudev, (uint8_t *)&pcshid->hid_protocol, 1);
    break;
  case HID_REQ_SET_IDLE:
    hid_set_idle = (uint8_t)(setup->wValue >> 8);
    break;
  case HID_REQ_GET_IDLE:
    usbd_ctrl_send(pudev, (uint8_t *)&hid_set_idle, 1);
    break;
  case HID_REQ_SET_REPORT:
    hid_state = HID_REQ_SET_REPORT;
    usbd_ctrl_recv(pudev, pcshid->hid_set_report, setup->wLength);
    break;
  default:
    usbd_ctrl_unsupport(pudev);
    break;
}
```

- Custom_HID send data

```
usb_sts_type custom_hid_class_send_report(void *udev, uint8_t *report, uint16_t len)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  if(usbd_connect_state_get(pudev) == USB_CONN_STATE_CONFIGURED)
    usbd_ept_send(pudev, USBD_CUSTOM_HID_IN_EPT, report, len);
  return status;
}
```

- Custom_HID receive data

```
/* get endpoint receive data length  */
uint32_t recv_len = usbd_get_recv_len(pudev, ept_num);
/* hid buffer process */
usb_hid_buf_process(udev, pcshid->g_rxhid_buff, recv_len);
/* start receive next packet */
usbd_ept_recv(pudev, USBD_CUSTOM_HID_OUT_EPT, pcshid->g_rxhid_buff, recv_len);
```

- Data processing

```
void usb_hid_buf_process(void *udev, uint8_t *report, uint16_t len)
{
```

```
  uint32_t i_index;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  switch(report[0])
  {
    case HID_REPORT_ID_2:
      if(g_rxhid_buff[1] == 0)
      {
        at32_led_off(LED2);
      }
      else
      {
        at32_led_on(LED2);
      }
      break;
    case HID_REPORT_ID_3:
      if(g_rxhid_buff[1] == 0)
      {
        at32_led_off(LED3);
      }
      else
      {
        at32_led_on(LED3);
      }
      break;
    case HID_REPORT_ID_4:
      if(g_rxhid_buff[1] == 0)
      {
        at32_led_off(LED4);
      }
      else
      {
        at32_led_on(LED4);
      }
      break;
    case HID_REPORT_ID_6:
      for(i_index = 0; i_index < len; i_index ++)
      {
        g_txhid_buff[i_index] = report[i_index];
      }
      usbd_ept_send(pudev, USBD_CUSTOM_HID_IN_EPT, pcshid->g_txhid_buff, len);
    default:
      break;
  }
}
```

3.2.4 How to develop based on custom hid routine

This section briefly introduces how to modify the code of custom_hid routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify device descriptions (custom_hid_desc.c, custom_hid_desc.h)
  Device description (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Device report descriptor (g_usbd_hid_report)
  Other descriptions
- According to specific functional requirements to modify the endpoints (custom_hid_class.c, custom_hid_class.h)
  Endpoint definition (custom_hid _class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)
- Modify custom_hid control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- custom_hid send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)
- Modify the data processing

3.3 Keyboard routine

The keyboard implements a keyboard function, and communicates with the upper computer through interrupt transfer. This routine runs on the AT-START development board, and sends a string to the host by using a key.

3.3.1 Implement functions

- Send a string (Keyboard Demo) to the host by using a key.

3.3.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration
- Endpoint 1 IN: for data sending

3.3.3 Keyboard implementation

The keyboard device class implementation source code is mainly in keyboard_class.c and keyboard_desc.c, and these two source files implement device descriptions and device processing.

3.3.3.1 Device description (keyboard_desc.c/h)

- keyboard device description (g_usbd_descriptor)
- keyboard device configuration description (g_usbd_configuration)

keyboard interface
keyboard endpoint

- keyboard report descriptor (g_usbd_hid_report)
- Lang id (g_string_lang_id)
- Serial number (g_string_serial)
- Vendor/product ID (keyboard_desc.h)

```
#define USBD_KEYBOARD_VENDOR_ID                0x2E3C
#define USBD_KEYBOARD_PRODUCT_ID               0x6040
```

- Manufacturer, product name, configuration description and interface description (keyboard_desc.h)

```
#define USBD_KEYBOARD_DESC_MANUFACTURER_STRING      "Artery"
#define USBD_KEYBOARD_DESC_PRODUCT_STRING           "Keyboard"
#define USBD_KEYBOARD_DESC_CONFIGURATION_STRING     "Keyboard Config"
#define USBD_KEYBOARD_DESC_INTERFACE_STRING         "Keyboard Interface"
```

3.3.3.2 Data processing (keyboard_class.c/h)

- Endpoint initialization (class_init_handler)

```
/* open hid in endpoint */
usbd_ept_open(pudev, USBD_KEYBOARD_IN_EPT, EPT_INT_TYPE,
USBD_KEYBOARD_IN_MAXPACKET_SIZE);
```

- Endpoint clear (class_clear_handler)

```
/* close hid in endpoint */
usbd_ept_close(pudev, USBD_KEYBOARD_IN_EPT);
```

- HID device class request (class_setup_handler)

  Implement the following requests:
  SET_PROTOCOL
  GET_PROTOCOL
  SET_IDLE
  GET_IDLE
  SET_REPORT

- keyboard send data

```
usb_sts_type usb_keyboard_class_send_report(void *udev, uint8_t *report, uint16_t len)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  if(usbd_connect_state_get(pudev) == USB_CONN_STATE_CONFIGURED)
    usbd_ept_send(pudev, USBD_KEYBOARD_IN_EPT, report, len);
  return status;
}
```

- keyboard character data processing

```
void usb_hid_keyboard_send_char(void *udev, uint8_t ascii_code)
{
        uint8_t key_data = 0;
```

```
        keyboard_type *pkeyboard = (keyboard_type *)pudev->class_handler->pdata;
        if(ascii_code > 128)
        {
          ascii_code = 0;
        }
        else
        {
          ascii_code = _asciimap[ascii_code];
          if(!ascii_code)
          {
            ascii_code = 0;
          }

          if(ascii_code & SHIFT)
          {
            key_data = 0x2;
            ascii_code &= 0x7F;
          }
        }
       if((pkeyboard->temp == ascii_code) && (ascii_code != 0))
       {
          pkeyboard->keyboard_buf[0] = 0;
          pkeyboard->keyboard_buf[2] = 0;
          usb_keyboard_class_send_report(udev, pkeyboard->keyboard_buf, 8);
        }
        else
        {
          pkeyboard->keyboard_buf[0] = key_data;
          pkeyboard->keyboard_buf[2] = ascii_code;
          usb_keyboard_class_send_report(udev, pkeyboard->keyboard_buf, 8);
    }
}
```

3.3.4 How to develop based on keyboard routine

This section briefly introduces how to modify the code of keyboard routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify device descriptions (keyboard_desc.c, keyboard_desc.h)
  Device description (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Device report descriptor (g_usbd_hid_report)
  Other descriptions
- According to specific functional requirements to modify endpoints (keyboard_class.c, keyboard_class.h)
  Endpoint definition (keyboard _class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)

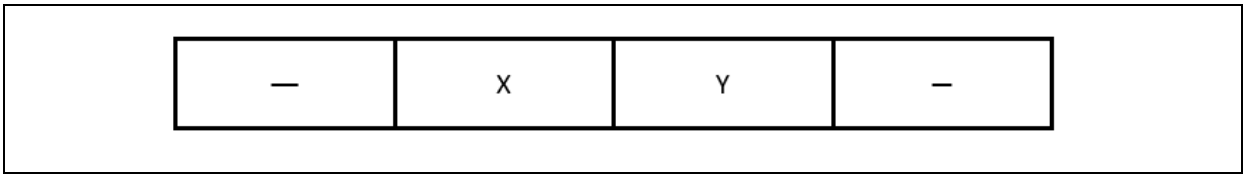
- Modify keyboard control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- keyboard send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)
- Modify the data processing

3.4 Mouse routine

3.4.1 Implement functions

The mouse implements a simple mouse function, and communicates with the upper computer through interrupt transfer. This routine runs on the AT-START development board, and send the right mouse button function through the key on development board.

Figure 16 Mouse transfer format



PC mouse movement is usually controlled by setting X and Y values.

3.4.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration
- Endpoint 1 IN: for data sending

3.4.3 Mouse implementation

The mouse device class implementation source code is mainly in mouse_class.c and mouse_desc.c, and these two source files implement device descriptions and device processing.

3.4.3.1 Device description (mouse_desc.c/h)

- Mouse device description (g_usbd_descriptor)
- Mouse device configuration description (g_usbd_configuration)
  Mouse interface
  Mouse endpoint
- Mouse report descriptor (g_usbd_hid_report)
- Lang id (g_string_lang_id)
- Serial number (g_string_serial)
- Vendor/product ID (mouse_desc.h)

#define USBD_MOUSE_VENDOR_ID	0x2E3C
#define USBD_MOUSE_PRODUCT_ID	0x5710

- Manufacturer, product name, configuration description and interface description (mouse_desc.h)

```
#define USBD_MOUSE_DESC_MANUFACTURER_STRING      "Artery"
#define USBD_MOUSE_DESC_PRODUCT_STRING           "Mouse"
#define USBD_MOUSE_DESC_CONFIGURATION_STRING     "Mouse Config"
#define USBD_MOUSE_DESC_INTERFACE_STRING         "Mouse Interface"
```

3.4.3.2 Data processing (mouse_class.c/h)

- Endpoint initialization (class_init_handler)

```
/* open hid in endpoint */
usbd_ept_open(pudev, USBD_MOUSE_IN_EPT, EPT_INT_TYPE,
USBD_MOUSE_IN_MAXPACKET_SIZE);
```

- Endpoint clear (class_clear_handler)

```
/* close hid in endpoint */
usbd_ept_close(pudev, USBD_MOUSE_IN_EPT);
```

- HID device class request (class_setup_handler)
  Implement the following requests:
  SET_PROTOCOL
  GET_PROTOCOL
  SET_IDLE
  GET_IDLE
  SET_REPORT
- Mouse send data

```
usb_sts_type usb_mouse_class_send_report(void *udev, uint8_t *report, uint16_t len)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  if(usbd_connect_state_get(pudev) == USB_CONN_STATE_CONFIGURED)
  {
    usbd_flush_tx_fifo(pudev, USBD_MOUSE_IN_EPT);
    usbd_ept_send(pudev, USBD_MOUSE_IN_EPT, report, len);
  }
  return status;
}
```

- Mouse data processing

```
void usb_hid_mouse_send(void *udev, uint8_t op)
{
  usbd_core_type *pudev = (usbd_core_type *)udev;
  mouse_type *pmouse = (mouse_type *)pudev->class_handler->pdata;
  int8_t posx = 0, posy = 0, button = 0;
  switch(op)
  {
    case LEFT_BUTTON:
      button = 0x01;
```

```
        break;
      case RIGHT_BUTTON:
        button = 0x2;
        break;
      case UP_MOVE:
        posy -= MOVE_STEP;
        break;
      case DOWN_MOVE:
        posy += MOVE_STEP;
        break;
      case LEFT_MOVE:
        posx -= MOVE_STEP;
        break;
      case RIGHT_MOVE:
        posx += MOVE_STEP;
        break;
      default:
        break;
    }
    pmouse->mouse_buffer[0] = button;
    pmouse->mouse_buffer[1] = posx;
    pmouse->mouse_buffer[2] = posy;

    usb_mouse_class_send_report(udev, pmouse->mouse_buffer, 4);
}
```

3.4.4 How to develop based on mouse routine

This section briefly introduces how to modify the code of mouse routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify device description (mouse_desc.c, mouse_desc.h)
  Device description (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Device report descriptor (g_usbd_hid_report)
  Other descriptions
- According to specific functional requirements to modify endpoints (mouse_class.c, mouse_class.h)
  Endpoint definition (mouse_class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)
- Modify mouse control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- Mouse send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)

- Modify the data processing

3.5 MSC routine

The msc (mass storage) routine shows how to communicate with PC host and AT-START through USB BULK transfer. This routine supports BOT (Bulk only transfer) protocol and SCSI (small computer system interface) instructions.

Figure 17 BOT command/data/status process

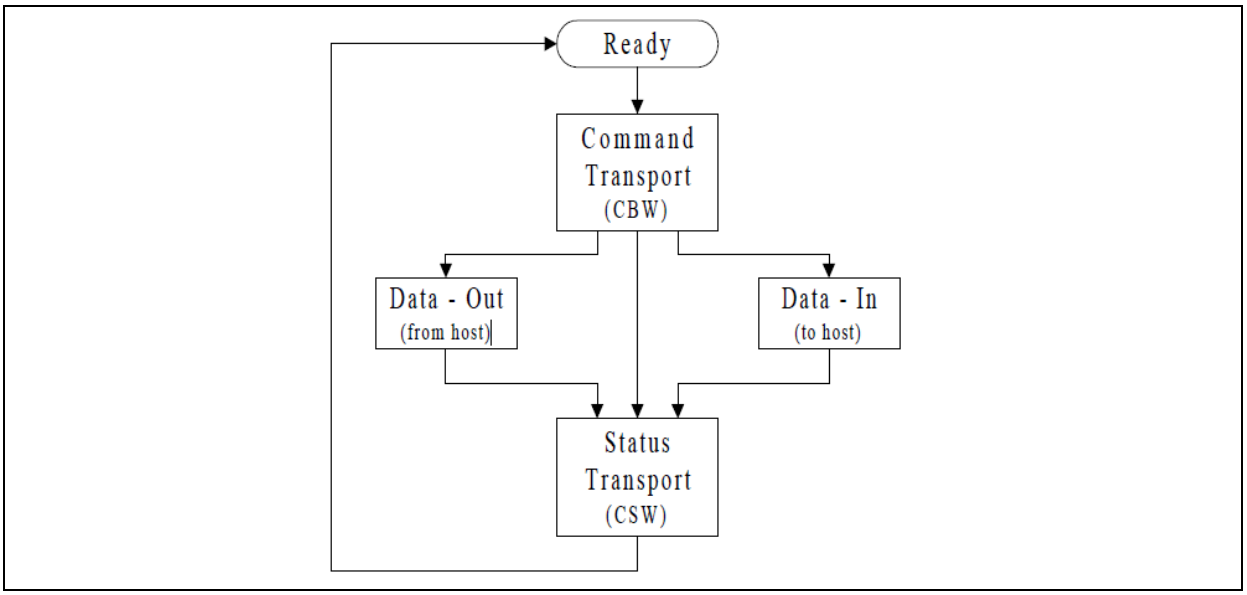
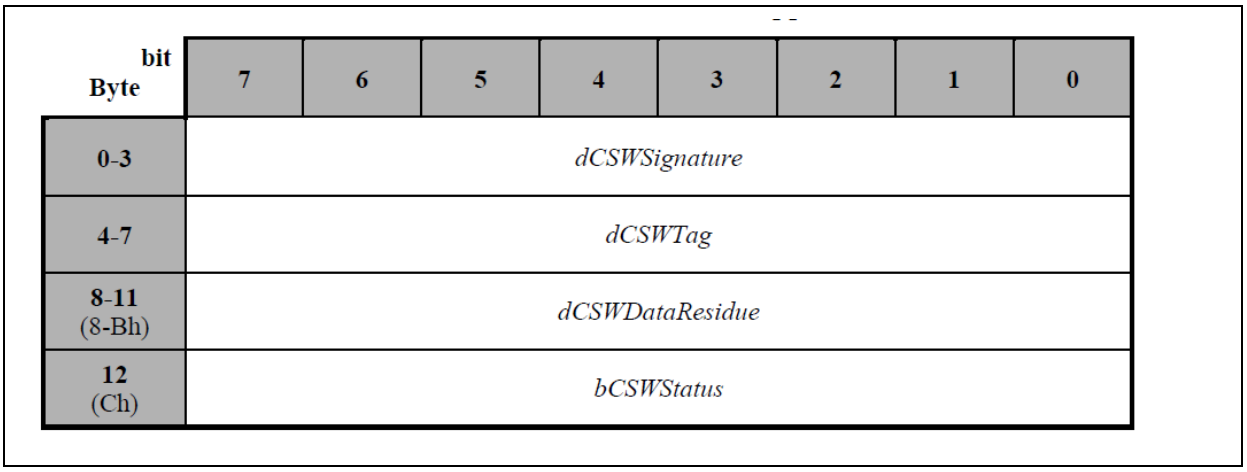


Figure 18 BOT command format

bit / Byte	7	6	5	4	3	2	1	0
0-3	*dCBWSignature*							
4-7	*dCBWTag*							
8-11 (08h-0Bh)	*dCBWDataTransferLength*							
12 (0Ch)	*bmCBWFlags*							
13 (0Dh)	Reserved (0)					*bCBWLUN*		
14 (0Eh)	Reserved (0)					*bCBWCBLength*		
15-30 (0Fh-1Eh)	*CBWCB*							

Figure 19 BOT status format



bit / Byte	7	6	5	4	3	2	1	0
0-3	*dCSWSignature*							
4-7	*dCSWTag*							
8-11 (8-Bh)	*dCSWDataResidue*							
12 (Ch)	*bCSWStatus*							

3.5.1 Implement functions

- Virtualize the internal FLASH into a disk
- Implement bulk-only transfer protocol
- Implement subclass SCSI transfer commands

  MSC_CMD_INQUIRY
  MSC_CMD_START_STOP
  MSC_CMD_MODE_SENSE6
  MSC_CMD_MODE_SENSE10
  MSC_CMD_ALLOW_MEDIUM_REMOVAL
  MSC_CMD_READ_10
  MSC_CMD_READ_CAPACITY
  MSC_CMD_REQUEST_SENSE
  MSC_CMD_TEST_UNIT
  MSC_CMD_VERIFY
  MSC_CMD_WRITE_10
  MSC_CMD_READ_FORMAT_CAPACITY

3.5.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration
- Endpoint 1 IN: for data sending
- Endpoint 1 OUT: for data receiving

3.5.3 MSC implementation

3.5.3.1 Device description (msc_desc.c/h)

- msc device description (g_usbd_descriptor)
- msc device configuration description (g_usbd_configuration)

  msc interface

msc endpoint

- Lang id (g_string_lang_id)
- Serial number (g_string_serial)
- Vendor/product ID (msc_desc.h)

```
#define USBD_MSC_VENDOR_ID                0x2E3C
#define USBD_MSC_PRODUCT_ID               0x5745
```

- Manufacturer, product name, configuration description and interface description (msc_desc.h)

```
#define USBD_MSC_DESC_MANUFACTURER_STRING      "Artery"
#define USBD_MSC_DESC_PRODUCT_STRING           "AT32 Mass Storage"
#define USBD_MSC_DESC_CONFIGURATION_STRING     "Mass Storage Config"
#define USBD_MSC_DESC_INTERFACE_STRING         "Mass Storage Interface"
```

3.5.3.2 Data processing (msc_class.c/h)

- Endpoint initialization (class_init_handler)

```
/* open in endpoint */
usbd_ept_open(pudev, USBD_MSC_BULK_IN_EPT, EPT_BULK_TYPE, USBD_OUT_MAXPACKET_SIZE);
/* open out endpoint */
usbd_ept_open(pudev, USBD_MSC_BULK_OUT_EPT, EPT_BULK_TYPE,
USBD_OUT_MAXPACKET_SIZE);
```

- Endpoint clear (class_clear_handler)

```
/* close in endpoint */
usbd_ept_close(pudev, USBD_MSC_BULK_IN_EPT);
/* close out endpoint */
usbd_ept_close(pudev, USBD_MSC_BULK_OUT_EPT);
```

- MSC device request (class_setup_handler)

  GET_MAX_LUN

  BO_RESET

  The code is as follows:

```
switch(setup->bRequest)
 {
    case MSC_REQ_GET_MAX_LUN:
      usbd_ctrl_send(pudev, (uint8_t *)&pmsc->max_lun, 1);
      break;
    case MSC_REQ_BO_RESET:
      usbd_ctrl_send_status(pudev);
      break;
    default:
      usbd_ctrl_unsupport(pudev);
      break;
 }
```

- IN transfer processing

```
usb_sts_type class_in_handler(void *udev, uint8_t ept_num)
{
```

```
    usb_sts_type status = USB_OK;
    usbd_core_type *pudev = (usbd_core_type *)udev;
    usbd_flush_tx_fifo(pudev, ept_num&0x7F);
    bot_scsi_datain_handler(udev, ept_num);
    return status;
}
```

- OUT transfer processing (receive data)

```
usb_sts_type class_out_handler(void *udev, uint8_t ept_num)
{
    usb_sts_type status = USB_OK;
    bot_scsi_dataout_handler(udev, ept_num);
    return status;
}
```

3.5.3.3 BOT and SCSI command processing

Bulk-only transfer and SCSI command processing is in the library file “msc_bot_scsi.c/h”.

Table 11 msc_bot_scsi functions

Function	Description
bot_scsi_init	Initialize bulk-only transfer
bot_scsi_datain_handler	Bulk-only data in
bot_scsi_dataout_handler	Bulk-only data out
bot_scsi_cmd_process	SCSI command processing
bot_scsi_cmd_state	SCSI command state
bot_scsi_test_unit	Test unit command
bot_scsi_inquiry	Inquiry command
bot_scsi_start_stop	Start stop command
bot_scsi_allow_medium_removal	Allow medium removal command
bot_scsi_mode_sense6	Mode sense6 command
bot_scsi_mode_sense10	Mode sense10 command
bot_scsi_read10	Read10 command
bot_scsi_capacity	Capacity command
bot_scsi_format_capacity	Format capacity command
bot_scsi_request_sense	Request sense command
bot_scsi_verify	Verify command
bot_scsi_write10	Write10 command
bot_scsi_unsupport	Unsupported command

3.5.3.4 diskio processing

This part mainly handles interfaces with the memory device. In this routine, the memory control of internal Flash is used for demonstration. Select the msc_diskio.c/h file according to the memory used by developers, and implement read/write functions of the corresponding memory.

Table 12 inquiry description

```
uint8_t scsi_inquiry[MSC_SUPPORT_MAX_LUN][SCSI_INQUIRY_DATA_LENGTH] =
{
  /* lun = 0 */
  {
    0x00,              /* peripheral device type (direct-access device) */
    0x80,              /* removable media bit */
    0x00,              /* ansi version, ecma version, iso version */
    0x01,              /* respond data format */
    SCSI_INQUIRY_DATA_LENGTH - 5, /* additional length */
    0x00, 0x00, 0x00, /* reserved */
    'A', 'T', '3', '2', ' ', ' ', ' ', ' ', /* vendor information "AT32" */
    'D', 'i', 's', 'k', '0', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', /* Product identification "Disk" */
    '2', '.', '0', '0'   /* product revision level */
  }
};
```

Table 13 diskio operating functions

Function	Description
get_inquiry	Get inquiry description
msc_disk_read	Read data from memory space
msc_disk_write	Write functions to memory space
msc_disk_capacity	Get memory capacity

3.5.4 How to develop based on MSC routine

This section briefly introduces how to modify the code of msc routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify device description (msc_desc.c, msc_desc.h)
  Device description (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Other descriptions
- According to specific functional requirements to modify endpoints (msc_class.c, msc_class.h)
  Endpoint definition (msc_class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)
- Modify msc control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- MSC send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)
- Modify diskio, and implement functions (msc_diskio.c/h) listed in Table 13

3.6 Printer routine

The printer routine shows that the USB Device is used as a printer. This demo can identify a printer on the PC side and respond to the printer class status request command (such as paper/no paper fed) sent from the PC side.

3.6.1 Implement functions

- Implement a printer.

3.6.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration
- Endpoint 1 IN: for data sending
- Endpoint 1 OUT: for data receiving

3.6.3 Printer implementation

3.6.3.1 Device description (printer_desc.c/h)

- Printer device description (g_usbd_descriptor)
- Printer device configuration description (g_usbd_configuration)
  printer interface
  printer endpoint
- Lang id (g_string_lang_id)
- Serial number (g_string_serial)
- Vendor/product ID (printer_desc.h)

```
#define USBD_PRINTER_VENDOR_ID                0x2E3C
#define USBD_PRINTER_PRODUCT_ID               0x57FF
```

- Manufacturer, product name, configuration description and interface description (printer_desc.h)

```
#define USBD_PRINTER_DESC_MANUFACTURER_STRING     "Artery"
#define USBD_PRINTER_DESC_PRODUCT_STRING          "AT32 Printer"
#define USBD_PRINTER_DESC_CONFIGURATION_STRING    "Printer Config"
#define USBD_PRINTER_DESC_INTERFACE_STRING        "Printer Interface"
```

3.6.3.2 Data processing (printer_class.c/h)

- Endpoint initialization (class_init_handler)

```
/* open in endpoint */
usbd_ept_open(pudev, USBD_PRINTER_BULK_IN_EPT, EPT_BULK_TYPE,
USBD_PRINTER_IN_MAXPACKET_SIZE);
/* open out endpoint */
usbd_ept_open(pudev, USBD_PRINTER_BULK_OUT_EPT, EPT_BULK_TYPE,
USBD_PRINTER_OUT_MAXPACKET_SIZE);
```

- Endpoint clear (class_clear_handler)

```
/* close in endpoint */
usbd_ept_close(pudev, USBD_PRINTER_BULK_IN_EPT);
/* close out endpoint */
usbd_ept_close(pudev, USBD_PRINTER_BULK_OUT_EPT);
```

- Printer device request (class_setup_handler)
  GET_DEVICE_ID
  PORT_STATUS
  SOFT_RESET
  The code is as follows:

```
switch(setup->bRequest)
{
  case PRINTER_REQ_GET_DEVICE_ID:
    usbd_ctrl_send(pudev, printer_device_id, PRINTER_DEVICE_ID_LEN);
    break;
  case PRINTER_REQ_GET_PORT_STATUS:
    usbd_ctrl_send(pudev, (uint8_t *)&pprter->g_printer_port_status, 1);
    break;
  case PRINTER_REQ_GET_SOFT_RESET:
    usbd_ctrl_recv(pudev, pprter->g_printer_data, USBD_PRINTER_OUT_MAXPACKET_SIZE);
    break;
  default:
    usbd_ctrl_unsupport(pudev);
    break;
}
```

- IN transfer processing

```
usb_sts_type class_in_handler(void *udev, uint8_t ept_num)
{
  usb_sts_type status = USB_OK;

  return status;
}
```

- OUT transfer processing (receive data)

```
usb_sts_type class_out_handler(void *udev, uint8_t ept_num)
{
  usb_sts_type status = USB_OK;
  return status;
}
```

3.6.4 How to develop based on printer routine

This section briefly introduces how to modify the code of printer routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify device description (printer_desc.c, printer_desc.h)
  Device description (g_usbd_descriptor)

Device configuration description (g_usbd_configuration)
Other descriptions

- According to specific functional requirements to modify endpoints (printer_class.c, printer_class.h)
  Endpoint definition (printer_class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)
- Modify printer control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- printer send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)

3.7 vcp_loopback routine

In modern PCs, USB is the standard interface for communicating with most peripherals. Nonetheless, most industrial software still uses the COM interface (UART) to communicate. The vcp_loopback routine provides a solution of simulating COM interface using USB devices, and shows how to perform USB data sending/receiving through the CDC protocol. This routine requires a virtual serial port driver, which can be downloaded from the official website.

3.7.1 Implement functions

- Implement a virtual serial port, and return the received data directly to the upper computer.

3.7.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration
- Endpoint 1 IN: for data sending
- Endpoint 1 OUT: for data receiving
- Endpoint 2 IN: monitor interrupt transfer

3.7.3 vcp_loopback implementation

3.7.3.1 Device description (cdc_desc.c/h)

- cdc device description (g_usbd_descriptor)
- cdc device configuration description (g_usbd_configuration)
  cdc interface
  cdc endpoint
- Lang id (g_string_lang_id)
- Serial number (g_string_serial)
- Vendor/product ID (cdc_desc.h)

```
#define USBD_CDC_VENDOR_ID          0x2E3C
#define USBD_CDC_PRODUCT_ID         0x5740
```

- Manufacturer, product name, configuration description and interface description (cdc_desc.h)

```
#define USBD_CDC_DESC_MANUFACTURER_STRING      "Artery"
#define USBD_CDC_DESC_PRODUCT_STRING           "AT32 Virtual Com Port   "
#define USBD_CDC_DESC_CONFIGURATION_STRING     "Virtual ComPort Config"
#define USBD_CDC_DESC_INTERFACE_STRING         "Virtual ComPort Interface"
```

3.7.3.2 Data processing (cdc_class.c/h)

- Endpoint initialization (class_init_handler)

```
/* open in endpoint */
usbd_ept_open(pudev, USBD_CDC_INT_EPT, EPT_INT_TYPE, USBD_CDC_CMD_MAXPACKET_SIZE);
/* open in endpoint */
usbd_ept_open(pudev, USBD_CDC_BULK_IN_EPT, EPT_BULK_TYPE,
USBD_CDC_IN_MAXPACKET_SIZE);
/* open out endpoint */
usbd_ept_open(pudev, USBD_CDC_BULK_OUT_EPT, EPT_BULK_TYPE,
USBD_CDC_OUT_MAXPACKET_SIZE);
```

- Endpoint clear (class_clear_handler)

```
/* close in endpoint */
usbd_ept_close(pudev, USBD_CDC_INT_EPT);
/* close in endpoint */
usbd_ept_close(pudev, USBD_CDC_BULK_IN_EPT);
/* close out endpoint */
usbd_ept_close(pudev, USBD_CDC_BULK_OUT_EPT);
```

- cdc device request (class_setup_handler)

  SET_LINE_CODING

  GET_LINE_CODING

  The code is as follows:

```
 case USB_REQ_TYPE_CLASS:
      if(setup->wLength)
      {
        if(setup->bmRequestType & USB_REQ_DIR_DTH)
        {
          usb_vcp_cmd_process(udev, setup->bRequest, pcdc->g_cmd, setup->wLength);
          usbd_ctrl_send(pudev, pcdc->g_cmd, setup->wLength);
        }
        else
        {
          pcdc->g_req = setup->bRequest;
          pcdc->g_len = setup->wLength;
          usbd_ctrl_recv(pudev, pcdc->g_cmd, pcdc->g_len);
        }
      }
```

- IN transfer processing

```
static usb_sts_type class_in_handler(void *udev, uint8_t ept_num)
```

```
{
  usbd_core_type *pudev = (usbd_core_type *)udev;
  cdc_struct_type *pcdc = (cdc_struct_type *)pudev->class_handler->pdata;
  usb_sts_type status = USB_OK;

  /* ...user code...
    trans next packet data
  */
  usbd_flush_tx_fifo(pudev, ept_num);
  pcdc->g_tx_completed = 1;

  return status;
}
```

Send data:

```
error_status usb_vcp_send_data(void *udev, uint8_t *send_data, uint16_t len)
{
  error_status status = SUCCESS;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  cdc_struct_type *pcdc = (cdc_struct_type *)pudev->class_handler->pdata;
  if(pcdc->g_tx_completed)
  {
    pcdc->g_tx_completed = 0;
    usbd_ept_send(pudev, USBD_CDC_BULK_IN_EPT, send_data, len);
  }
  else
  {
    status = ERROR;
  }
  return status;
}
```

- OUT transfer processing (receive data)

```
static usb_sts_type class_out_handler(void *udev, uint8_t ept_num)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  cdc_struct_type *pcdc = (cdc_struct_type *)pudev->class_handler->pdata;
  /* get endpoint receive data length  */
  pcdc->g_rxlen = usbd_get_recv_len(pudev, ept_num);

  /*set recv flag*/
  pcdc->g_rx_completed = 1;

  return status;
}

uint16_t usb_vcp_get_rxdata(void *udev, uint8_t *recv_data)
```

```
{
  uint16_t i_index = 0;
  uint16_t tmp_len = 0;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  cdc_struct_type *pcdc = (cdc_struct_type *)pudev->class_handler->pdata;

  if(pcdc->g_rx_completed == 0)
  {
    return 0;
  }
  pcdc->g_rx_completed = 0;
  tmp_len = pcdc->g_rxlen;
  for(i_index = 0; i_index < pcdc->g_rxlen; i_index ++)
  {
    recv_data[i_index] = pcdc->g_rx_buff[i_index];
  }

  usbd_ept_recv(pudev, USBD_CDC_BULK_OUT_EPT, pcdc->g_rx_buff,
USBD_CDC_OUT_MAXPACKET_SIZE);
  return tmp_len;
}
```

3.7.4 How to develop based on vcp_loopback routine

This section briefly introduces how to modify the code of cdc routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify device description (cdc_desc.c, cdc_desc.h)
  Device description (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Other descriptions
- According to specific functional requirements to modify endpoints (cdc_class.c, cdc_class.h)
  Endpoint definition (cdc_class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)
- Modify cdc control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- cdc send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)

3.8 virtual_msc_iap routine

The virtual msc iap implements a device for function upgrading, which do not depend on the upper computer. After connecting to the PC, copy the firmware to disk for upgrading.

3.8.1 Implement functions

- Virtualize the Flash into a disk for upgrading
- Iap reserves 20 Kbytes of space
- After the upgrade is complete, reset USB device to the upgrade state
- Support download address setting
- Support jumping to APP after the upgrade is complete
- Support bin file upgrading

3.8.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration
- Endpoint 1 IN: for data sending
- Endpoint 1 OUT: for data receiving

3.8.3 virtual_msc_iap implementation

3.8.3.1 Device description (msc_desc.c/h)

Refer to 3.5.3.1.

3.8.3.2 Data processing (msc_class.c/h)

Refer to 3.5.3.2.

3.8.3.3 BOT and SCSI command processing

Refer to 3.5.3.3.

3.8.3.4 diskio processing

Refer to 3.5.3.4.

3.8.3.5 Flash upgrade (flash_fat16.c/h)

- Upgrade states

```
typedef enum
{
  UPGRADE_READY                    = 0,
  UPGRAGE_ONGOING,
  UPGRADE_SUCCESS,
  UPGRADE_FAILED,
  UPGRADE_LARGE,
  UPGRADE_UNKNOWN,
  UPGRADE_DONE,
```

```
  UPGRADE_JUMP
}upgrade_status_type;
```

After connecting to the Host, the current state is displayed in TXT text on the response disk.

Ready to upgrade (Ready.TXT)

Upgrade successful (Success.TXT)

Upgrade failed (Failed.TXT)

Error: unknown file (Unkonwn.TXT)

File being upgraded larger than FLASH (Large.TXT)

- FAT16 partition table description

```
static uint8_t fat16_sector[FAT16_SECTOR_SIZE] =
{
  0xEB,   /*0*/
  0x3C,   /*1*/
  0x90,   /*2*/
  0x4D,   /*3*/
  0x53,   /*4*/
  0x44,   /*5*/
  0x4F,   /*6*/
  0x53,   /*7*/
  0x35,   /*8*/
  0x2E,   /*9*/
  0x30,   /*10*/
  0x00,   /*11*/
  0x08,   /*12*/ //2K
  0x04,   /*13*/
  0x06,   /*14*/
  0x00,   /*15*/

  0x02,   /*16*/
  0x00,   /*17*/
  0x02,   /*18*/
  0xFF,   /*19*/
  0x0F,   /*20*/
  0xF8,   /*21*/
  0x01,   /*22*/
  0x00,   /*23*/
  0x01,   /*24*/
  0x00,   /*25*/
  0x01,   /*26*/
  0x00,   /*27*/
  0x00,   /*28*/
  0x00,   /*29*/
  0x00,   /*30*/
  0x00,   /*31*/

  0x00,   /*32*/
```

```
    0x00,   /*33*/
    0x00,   /*34*/
    0x00,   /*35*/
    0x00,   /*36*/
    0x00,   /*37*/
    0x29,   /*38*/
    0x96,   /*39*/
    0x16,   /*40*/
    0x66,   /*41*/
    0xD3,    /*42*/
    0x20,   /*43*/
    0x20,   /*44*/
    0x20,   /*45*/
    0x20,   /*46*/
    0x20,   /*47*/

    0x20,   /*48*/
    0x20,   /*49*/
    0x20,   /*50*/
    0x20,   /*51*/
    0x20,   /*52*/
    0x20,   /*53*/
    0x46,   /*54*/
    0x41,   /*55*/
    0x54,   /*56*/
    0x31,   /*57*/
    0x36,   /*58*/
    0x20,   /*59*/
    0x20,   /*60*/
    0x20     /*61*/
};

const uint8_t fat16_root_dir_sector[FAT16_DIR_SIZE]=
{
        0x20,                                   /*11 - Archive Attribute set */
        0x00,                                   /*12 - Reserved */
        0x4B,                                   /*13 - Create Time Tenth */
        0x9C,                                   /*14 - Create Time */
        0x42,                                   /*15 - Create Time */
        0x92,                                   /*16 - Create Date */
        0x38,                                   /*17 - Create Date */
        0x92,                                   /*18 - Last Access Date */
        0x38,                                   /*19 - Last Access Date */
        0x00,                                   /*20 - Not used in FAT16 */
        0x00,                                   /*21 - Not used in FAT16 */
        0x9D,                                   /*22 - Write Time */
```

```
        0x42,                                   /*23 - Write Time */
        0x92,                                   /*24 - Write Date */
        0x38,                                   /*25 - Write Date */
        0x00,                                   /*26 - First Cluster (none, because file is empty) */
        0x00,                                   /*27 - First Cluster (none, because file is empty) */
        0x00,                                   /*28 - File Size */
        0x00,                                   /*29 - File Size */
        0x00,                                   /*30 - File Size */
        0x00,                                   /*31 - File Size */
        'A','T','3','2',' ','I','A','P',' ',' ',' ',   /*32-42 - Volume label */
        0x08,                                   /*43 - File attribute = Volume label */
        0x00,                                   /*44 - Reserved */
        0x00,                                   /*45 - Create Time Tenth */
        0x00,                                   /*46 - Create Time */
        0x00,                                   /*47 - Create Time */
        0x00,                                   /*48 - Create Date */
        0x00,                                   /*49 - Create Date */
        0x00,                                   /*50 - Last Access Date */
        0x00,                                   /*51 - Last Access Date */
        0x00,                                   /*52 - Not used in FAT16 */
        0x00,                                   /*53 - Not used in FAT16 */
        0x9D,                                    /*54 - Write Time */
        0x42,                                   /*55 - Write Time */
        0x92,                                   /*56 - Write Date */
        0x38,                                   /*57 - Write Date */
};
```

- Upgrade interface function

Function	Description
flash_fat16_init	Initialization
flash_fat16_loop_status	Upgrade state monitoring
flash_fat16_set_name	Modify TXT file name of the corresponding state
flash_fat16_get_upgrade_flag	Get upgrade state
flash_fat16_write	Write data to flash
flash_fat16_read	Read data from flash

3.8.4 How to develop based on virtual_msc_iap routine

This section briefly introduces how to modify the code of virtual_msc_iap routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify device description (msc_desc.c, msc_desc.h)
  Device description (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Other descriptions
- According to specific functional requirements to modify endpoints (msc_class.c, msc_class.h)

Endpoint definition (msc_class.h)
Endpoint initialization (class_init_handler, class_clear_handler)

- Modify msc control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- msc send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)
- Modify diskio, and implement functions (msc_diskio.c/h) listed in Table 13
- Modify upgrade parameters in flash_fat16.c/h, such as APP start address, IAP occupied space, to that the IAP and APP addresses do not overlap

3.9 composite_vcp_keyboard routine

Definition: A composite device refers to a device with multiple independent interfaces.
A composite device has multiple combined features. For example, the composite vcp keyboard demo provides HID and CDC functions (keyboard and serial port communication).

3.9.1 Implement functions

- Implement an USB virtual serial port (refer to 3.7)
- Implement an USB keyboard device (refer to 3.3)

3.9.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration
- Endpoint 1 IN: for data sending
- Endpoint 1 OUT: for data receiving
- Endpoint 2 IN: CDC command interrupt transfer
- Endpoint 3 IN: Keyboard send data

3.9.3 composite_vcp_keyboard implementation

3.9.3.1 Device description (cdc_keyboard_desc.c/h)

- cdc_keyboard device description (g_usbd_descriptor)
- cdc_keyboard device configuration description (g_usbd_configuration)
  cdc interface
  cdc endpoint
  keyboard interface
  keyboard endpoint
- Lang id (g_string_lang_id)
- Serial number (g_string_serial)
- Vendor/product ID (cdc_keyboard_desc.h)

```
#define USBD_VCPKYBRD_VENDOR_ID                 0x2E3C
```

```
#define USBD_VCPKYBRD_PRODUCT_ID                    0x5750
```

- Manufacturer, product name, configuration description and interface description

```
#define USBD_VCPKYBRD_DESC_MANUFACTURER_STRING      "Artery"
#define USBD_VCPKYBRD_DESC_PRODUCT_STRING           "AT32 Composite VCP and Keyboard"
#define USBD_VCPKYBRD_DESC_CONFIGURATION_STRING     "Composite VCP and Keyboard Config"
#define USBD_VCPKYBRD_DESC_INTERFACE_STRING         "Composite VCP and Keyboard Interface"
```

3.9.3.2 Data processing (cdc_keyboard_class.c/h)

- Endpoint initialization (class_init_handler)

```
/* open in endpoint */
usbd_ept_open(pudev, USBD_VCPKYBRD_CDC_INT_EPT, EPT_INT_TYPE,
USBD_VCPKYBRD_CMD_MAXPACKET_SIZE);
/* open in endpoint */
usbd_ept_open(pudev, USBD_VCPKYBRD_CDC_BULK_IN_EPT, EPT_BULK_TYPE,
USBD_VCPKYBRD_IN_MAXPACKET_SIZE);
/* open out endpoint */
usbd_ept_open(pudev, USBD_VCPKYBRD_CDC_BULK_OUT_EPT, EPT_BULK_TYPE,
USBD_VCPKYBRD_OUT_MAXPACKET_SIZE);
/* open hid in endpoint */
usbd_ept_open(pudev, USBD_VCPKYBRD_HID_IN_EPT, EPT_INT_TYPE,
USBD_VCPKYBRD_IN_MAXPACKET_SIZE);
```

- Endpoint clear (class_clear_handler)

```
/* close in endpoint */
usbd_ept_close(pudev, USBD_VCPKYBRD_CDC_INT_EPT);
/* close in endpoint */
usbd_ept_close(pudev, USBD_VCPKYBRD_CDC_BULK_IN_EPT);
/* close out endpoint */
usbd_ept_close(pudev, USBD_VCPKYBRD_CDC_BULK_OUT_EPT);
/* close in endpoint */
usbd_ept_close(pudev, USBD_VCPKYBRD_HID_IN_EPT);
```

- Device request (class_setup_handler)

  cdc device class requests:
  SET_LINE_CODING
  GET_LINE_CODING

  Keyboard hid device class requests:
  SET_PROTOCOL
  GET_PROTOCOL
  SET_IDLE
  GET_IDLE
  SET_REPORT

  The code is as follows:

```
switch(setup->bmRequestType & USB_REQ_RECIPIENT_MASK)
```

```
{
    case USB_REQ_RECIPIENT_INTERFACE:
      if(setup->wIndex == VCPKYBRD_KEYBOARD_INTERFACE)
      {
        keyboard_class_setup_handler(udev, setup);
      }
      else
      {
        cdc_class_setup_handler(pudev, setup);
      }
      break;
    case USB_REQ_RECIPIENT_ENDPOINT:
      break;

}
```

- IN transfer processing

```
static usb_sts_type class_in_handler(void *udev, uint8_t ept_num)
{
  usb_sts_type status = USB_OK;
  vcp_keyboard_type *vcpkybrd = (vcp_keyboard_type *)pudev->class_handler->pdata;

  /* ...user code...
    trans next packet data
  */
  if((ept_num & 0x7F) == (USBD_VCPKYBRD_CDC_BULK_IN_EPT & 0x7F))
  {
    vcpkybrd->g_tx_completed = 1;
  }
  if((ept_num & 0x7F) == (USBD_VCPKYBRD_HID_IN_EPT & 0x7F))
  {
    vcpkybrd->g_keyboard_tx_completed = 1;
  }  return status;
}
```

CDC send data:

```
error_status usb_vcpkybrd_vcp_send_data(void *udev, uint8_t *send_data, uint16_t len)
{
  error_status status = SUCCESS;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  vcp_keyboard_type *vcpkybrd = (vcp_keyboard_type *)pudev->class_handler->pdata;
  if(vcpkybrd->g_tx_completed)
  {
    vcpkybrd->g_tx_completed = 0;
    usbd_ept_send(pudev, USBD_VCPKYBRD_CDC_BULK_IN_EPT, send_data, len);
```

```
    }
    else
    {
      status = ERROR;
    }
    return status;
}

Keyboard send data:
usb_sts_type usb_vcpkybrd_class_send_report(void *udev, uint8_t *report, uint16_t len)
{
    usb_sts_type status = USB_OK;
    usbd_core_type *pudev = (usbd_core_type *)udev;

    if(usbd_connect_state_get(pudev) == USB_CONN_STATE_CONFIGURED)
      usbd_ept_send(pudev, USBD_VCPKYBRD_HID_IN_EPT, report, len);

    return status;
```

- OUT transfer processing (receive data)

```
static usb_sts_type class_out_handler(void *udev, uint8_t ept_num)
{
    usb_sts_type status = USB_OK;
    usbd_core_type *pudev = (usbd_core_type *)udev;
    vcp_keyboard_type *vcpkybrd = (vcp_keyboard_type *)pudev->class_handler->pdata;

    /* get endpoint receive data length    */
    vcpkybrd->g_rxlen = usbd_get_recv_len(pudev, ept_num);

    /*set recv flag*/
    vcpkybrd->g_rx_completed = 1;

    return status;
}
uint16_t usb_vcpkybrd_vcp_get_rxdata(void *udev, uint8_t *recv_data)
{
    uint16_t i_index = 0;
    usbd_core_type *pudev = (usbd_core_type *)udev;
    vcp_keyboard_type *vcpkybrd = (vcp_keyboard_type *)pudev->class_handler->pdata;
    uint16_t tmp_len = 0;
    if(vcpkybrd->g_rx_completed == 0)
    {
      return 0;
    }
    vcpkybrd->g_rx_completed = 0;

    tmp_len = vcpkybrd->g_rxlen;
```

```
    for(i_index = 0; i_index < vcpkybrd->g_rxlen; i_index ++)
    {
      recv_data[i_index] = vcpkybrd->g_rx_buff[i_index];
    }

    usbd_ept_recv(pudev, USBD_VCPKYBRD_CDC_BULK_OUT_EPT, vcpkybrd->g_rx_buff,
USBD_VCPKYBRD_OUT_MAXPACKET_SIZE);

    return tmp_len;
}
```

3.9.4 How to develop based on composite_vcp_keyboard routine

This section briefly introduces how to modify the code of composite_vcp_keyboard routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify device description (cdc_keyboard_desc.c, cdc_keyboard _desc.h)
  Device description (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Other descriptions
- According to specific functional requirements to modify endpoints (cdc_keyboard _class.c, cdc_keyboard _class.h)
  Endpoint definition (cdc_keyboard class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)
- Modify control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- cdc_keyboard send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)

3.10 hid_iap routine

The hid_iap routine implements IAP upgrade function by using usb hid together with the upper computer. Developers can download the IAP_Programmer from the official website. The hid iap routine code is in BSP firmware library (utilities\at32f435_437_usb_iap_demo). Refer to *AN0007_AT32_IAP_using_the_USB_HID_ZH_V2.x.x.pdf* for the usage.

3.10.1 Implement functions

- Implement using HID for device upgrade

3.10.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration

- Endpoint 1 IN: for data sending
- Endpoint 1 OUT: for data receiving

3.10.3 hid_iap implementation

3.10.3.1 Device description (hid_iap_desc.c/h)

- hid iap device description (g_usbd_descriptor)
- hid iap device configuration description (g_usbd_configuration)
  HID interface
  HID Endpoint
- hid iap report descriptor (g_usbd_hid_report)
- Lang id (g_string_lang_id)
- Serial number (g_string_serial)
- Vendor/product ID (hid_iap_desc.h)

```
#define USBD_HIDIAP_VENDOR_ID                0x2E3C
#define USBD_HIDIAP_PRODUCT_ID               0xAF01
```

- Manufacturer, product name, configuration description and interface description (hid_iap_desc.h)

```
#define USBD_HIDIAP_DESC_MANUFACTURER_STRING     "Artery"
#define USBD_HIDIAP_DESC_PRODUCT_STRING          "HID IAP"
#define USBD_HIDIAP_DESC_CONFIGURATION_STRING    "HID IAP Config"
#define USBD_HIDIAP_DESC_INTERFACE_STRING        "HID IAP Interface"
```

3.10.3.2 Data processing (hid_iap_class.c/h)

- Endpoint initialization (class_init_handler)

```
/* open hid iap in endpoint */
usbd_ept_open(pudev, USBD_HIDIAP_IN_EPT, EPT_INT_TYPE, USBD_HIDIAP_IN_MAXPACKET_SIZE);
/* open hid iap out endpoint */
usbd_ept_open(pudev, USBD_HIDIAP_OUT_EPT, EPT_INT_TYPE,
USBD_HIDIAP_OUT_MAXPACKET_SIZE);
```

- Endpoint clear (class_clear_handler)

```
/* close hid iap   in endpoint */
usbd_ept_close(pudev, USBD_HIDIAP_IN_EPT);
/* close hid iap   out endpoint */
usbd_ept_close(pudev, USBD_HIDIAP_OUT_EPT);
```

- HID device class request (class_setup_handler)
  Implement the following requests:
  SET_PROTOCOL
  GET_PROTOCOL
  SET_IDLE
  GET_IDLE

SET_REPORT

The code is as follows:

```
switch(setup->bRequest)
{
  case HID_REQ_SET_PROTOCOL:
    hid_protocol = (uint8_t)setup->wValue;
    break;
  case HID_REQ_GET_PROTOCOL:
    usbd_ctrl_send(pudev, (uint8_t *)&piap->hid_protocol, 1);
    break;
  case HID_REQ_SET_IDLE:
    hid_set_idle = (uint8_t)(setup->wValue >> 8);
    break;
  case HID_REQ_GET_IDLE:
    piap->hid_set_idle = (uint8_t)(setup->wValue >> 8);
    break;
  case HID_REQ_SET_REPORT:
    piap->hid_state = HID_REQ_SET_REPORT;
    usbd_ctrl_recv(pudev, piap->hid_set_report, setup->wLength);
    break;
  default:
    usbd_ctrl_unsupport(pudev);
    break;
}
```

- hid iap send data

```
Send complete processing:
static usb_sts_type class_in_handler(void *udev, uint8_t ept_num)
{
  usb_sts_type status = USB_OK;

  /* ...user code...
    trans next packet data
  */
  usbd_hid_iap_in_complete(udev);

  return status;
}

Send data:
usb_sts_type usb_iap_class_send_report(void *udev, uint8_t *report, uint16_t len)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;

  if(usbd_connect_state_get(pudev) == USB_CONN_STATE_CONFIGURED)
```

```
    usbd_ept_send(pudev, USBD_HIDIAP_IN_EPT, report, len);

  return status;
}
```

- hid iap receive data

```
static usb_sts_type class_out_handler(void *udev, uint8_t ept_num)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  iap_info_type *piap = (iap_info_type *)pudev->class_handler->pdata;

  /* get endpoint receive data length  */
  uint32_t recv_len = usbd_get_recv_len(pudev, ept_num);

  /* hid iap process */
  usbd_hid_iap_process(udev, piap->g_rxhid_buff, recv_len);

  /* start receive next packet */
  usbd_ept_recv(pudev, USBD_HIDIAP_OUT_EPT, piap->g_rxhid_buff,
USBD_HIDIAP_OUT_MAXPACKET_SIZE);

  return status;
}
```

- Upgrade command data processing

```
iap_result_type usbd_hid_iap_process(void *udev, uint8_t *pdata, uint16_t len)
{
  iap_result_type status = IAP_SUCCESS;
  uint16_t iap_cmd;

  if(len < 2)
  {
    return IAP_FAILED;
  }

  iap_info.respond_flag = 0;

  iap_cmd = (pdata[0] << 8) | pdata[1];

  switch(iap_cmd)
  {
    case IAP_CMD_IDLE:
      iap_idle();
      break;
    case IAP_CMD_START:
```

```
            iap_start();
            break;
        case IAP_CMD_ADDR:
            iap_address(pdata, len);
            break;
        case IAP_CMD_DATA:
            iap_data_write(pdata, len);
            break;
        case IAP_CMD_FINISH:
            iap_finish();
            break;
        case IAP_CMD_CRC:
            iap_crc(pdata, len);
            break;
        case IAP_CMD_JMP:
            iap_jump();
            break;
        case IAP_CMD_GET:
            iap_get();
            break;
        default:
            status = IAP_FAILED;
            break;
    }

    if(iap_info.respond_flag)
    {
        usb_iap_class_send_report (udev, iap_info.iap_tx, 64);
    }
    return status;
}
```

3.10.3.3 hid iap upgrade protocol

Table 14 hid iap upgrade commands

Command value	Description
0x5AA0	Enter IAP mode
0x5AA1	Start download
0x5AA2	Set address (aligned with 1K)
0x5AA3	Download data command
0x5AA4	Download complete
0x5AA5	CRC check
0x5AA6	Jump command (jump to user code)
0x5AA7	Get User Code address set by IAP

1. 0x5AA0: enter IAP mode

   This is a specific command. After receiving this command, the user APP enters IAP mode by erasing the flag and then performing reset.

   Upper computer: [0x5A, 0xA0]

   IAP device response: [0x5A, 0XA0, ACK/NACK]

2. 0x5AA1: start download

   Upper computer: [0x5A,0xA1]

   IAP device response: [0x5A,0xA1,ACK/NACK]

3. 0x5AA2: set download address

   Set the download address to be aligned with 1 KB. Every time 1 Kbyte of data is downloaded, the download address needs to be reset.

   Upper computer (command+address): [0x5A, 0xA2, 0x08, 0x00, 0x40, 0x00]

   IAP device response: [0x5A,0xA2, ACK/NACK]

4. 0x5AA3: download data command (aligned by 1 KB, sent by multiple packets)

   The download data command is sent in the format of “command + length + data”, and the maximum data size per packet is 60 bytes (64-command-length). When the sent data reaches 1 KB, the upper computer needs to wait for ACK response from the device. At this point, the device needs to write the 1 KB of data to FLASH.

   Upper computer (command (2 Bytes) + length (2 Bytes) + data (n byte)): [0x5A,0xA3,LEN1, LEN0,DATA0….DATAn]

   IAP device response after receiving 1 KB of data: [0x5A, 0XA3, ACK/NACK]

5. 0x5AA4: download complete

   Upper computer: [0x5A, 0xA4]

   IAP device response: [0x5A, 0xA4, ACK/NACK]

6. 0x5AA5: firmware CRC check

   The firmware start address and size/1 KB (firmware size aligned with 1 KB; or 0xFF stuffing if it is insufficient) are transferred by the upper computer, and then sent back to the upper computer after CRC check by IAP.

   Upper computer: [0x5A,0xA5, 0x08, 0x00, 0x40, 0x00, LEN1, LEN0]

   IAP device response: [0x5A, 0xA5, ACK/NACK, CRC3, CRC2, CRC1, CRC0]

7. 0x5AA6: jump command

   The jump command is used to jump to run the user code.

   Upper computer: [0x5A,0xA6, 0x08, 0x00, 0x40, 0x00]

   IAP device response: [0x5A,0xA6,ACK/NACK]

8. 0x5AA7: get app address set by IAP

   Return to the app address set by IAP.

   Upper computer: [0x5A, 0xA7]

   IAP device response: [0x5A, 0xA7, ACK/NACK, 0x08, 0x00, 0x40, 0x00]

3.10.4 How to develop based on hid_iap routine

This section briefly introduces how to modify the code of hid_iap routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify device description (hid_iap_desc.c, hid_iap_desc.h)
  Device description (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Other descriptions
- According to specific functional requirements to modify endpoints (hid_iap_class.c, hid_iap_class.h)
  Endpoint definition (hid_iap_class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)
- Modify hid control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- hid_iap send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)
- Modify upgrade parameters in hid_iap_user.h, such as APP start address, IAP occupied space, to ensure that the IAP and APP addresses do not overlap

3.11 composite_audio_hid routine

Definition: A composite device refers to a device with multiple independent interfaces.
A composite device has multiple combined features. For example, the composite audio hid demo provides AUDIO and HID functions.

3.11.1 Implement functions

- Implement an AUDIO device (refer to 3.1)
- Implement a HID device (refer to 3.2)

3.11.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration
- Endpoint 1 IN: for audio data sending
- Endpoint 2 OUT: for audio data receiving
- Endpoint 2 IN: for HID data sending
- Endpoint 1 OUT: for HID data receiving
- Endpoint 3 IN: for audio feedback data sending

3.11.3 composite_audio_hid implementation

3.9.3.1 Device description (audio_hid_desc.c/h)

- audio_hid device description (g_usbd_descriptor)
- audio_hid device configuration description (g_usbd_configuration)
  audio interface
  audio endpoint
  hid interface
  hid endpoint
- Lang id (g_string_lang_id)
- Serial number (g_string_serial)
- Vendor/product ID (audio_hid_desc.h)

```
#define USBD_AUHID_VENDOR_ID                    0x2E3C
#define USBD_AUHID_PRODUCT_ID                   0x5555
```

- Manufacturer, product name, configuration description and interface description

```
#define USBD_AUHID_DESC_MANUFACTURER_STRING      "Artery"
#define USBD_AUHID_DESC_PRODUCT_STRING           "AT32 Audio"
#define USBD_AUHID_DESC_CONFIGURATION_STRING     "Audio Config"
#define USBD_AUHID_DESC_INTERFACE_STRING         "Audio Interface"
```

3.9.3.2 Data processing (audio_hid_class.c/h)

- Endpoint initialization (class_init_handler)

```
/* open in endpoint */
/* open microphone in endpoint */
usbd_ept_open(pudev, USBD_AUHID_AUDIO_MIC_IN_EPT, EPT_ISO_TYPE,
AUDIO_MIC_IN_MAXPACKET_SIZE);

/* open speaker out endpoint */
usbd_ept_open(pudev, USBD_AUHID_AUDIO_SPK_OUT_EPT, EPT_ISO_TYPE,
AUDIO_SPK_OUT_MAXPACKET_SIZE);
#if AUDIO_SUPPORT_FEEDBACK
/* open speaker feedback endpoint */
usbd_ept_open(pudev, USBD_AUHID_AUDIO_FEEDBACK_EPT, EPT_ISO_TYPE,
AUDIO_FEEDBACK_MAXPACKET_SIZE);
#endif
/* start receive speaker out data */
usbd_ept_recv(pudev, USBD_AUHID_AUDIO_SPK_OUT_EPT, paudio_hid->audio_spk_data,
AUDIO_SPK_OUT_MAXPACKET_SIZE);

   /*open hid endpoint */
/* open custom hid in endpoint */
usbd_ept_open(pudev, USBD_AUHID_HID_IN_EPT, EPT_INT_TYPE,
```

```
USBD_AUHID_IN_MAXPACKET_SIZE);

/* open custom hid out endpoint */
usbd_ept_open(pudev, USBD_AUHID_HID_OUT_EPT, EPT_INT_TYPE,
USBD_AUHID_OUT_MAXPACKET_SIZE);
```

- Endpoint clear (class_clear_handler)

```
/* close in endpoint */
usbd_ept_close(pudev, USBD_AUHID_AUDIO_MIC_IN_EPT);
#if AUDIO_SUPPORT_FEEDBACK
/* close in endpoint */
usbd_ept_close(pudev, USBD_AUHID_AUDIO_FEEDBACK_EPT);
#endif
/* close out endpoint */
usbd_ept_close(pudev, USBD_AUHID_AUDIO_SPK_OUT_EPT);
/* close custom hid in endpoint */
usbd_ept_close(pudev, USBD_AUHID_HID_IN_EPT);
/* close custom hid out endpoint */
usbd_ept_close(pudev, USBD_AUHID_HID_OUT_EPT);
```

- Device request (class_setup_handler)

audio device class requests:
GET_CUR
SET_CUR
GET_MIN
GET_MAX
GET_RES

hid device class requests:
SET_PROTOCOL
GET_PROTOCOL
SET_IDLE
GET_IDLE
SET_REPORT

The code is as follows:

```
switch(setup->bmRequestType & USB_REQ_RECIPIENT_MASK)
{
  case USB_REQ_RECIPIENT_INTERFACE:
    if(setup->wIndex == HID_INTERFACE_NUMBER)
    {
      class_hid_setup_handler(udev, setup);
    }
    else
    {
      class_audio_setup_handler(pudev, setup);
    }
    break;
```

```
    case USB_REQ_RECIPIENT_ENDPOINT:
      class_audio_setup_handler(pudev, setup);
      break;
    }
```

- IN transfer processing

```
static usb_sts_type class_in_handler(void *udev, uint8_t ept_num)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  usb_audio_hid_type *paudio_hid = (usb_audio_hid_type *)pudev->class_handler->pdata;
  uint32_t len = 0;

  /* ...user code...
     trans next packet data
  */
  if((ept_num & 0x7F) == (USBD_AUHID_AUDIO_MIC_IN_EPT & 0x7F))
  {
    len = audio_codec_mic_get_data(paudio_hid->audio_mic_data);
    usbd_flush_tx_fifo(udev, USBD_AUHID_AUDIO_MIC_IN_EPT);
    usbd_ept_send(udev, USBD_AUHID_AUDIO_MIC_IN_EPT, paudio_hid->audio_mic_data, len);
  }

  else if((ept_num & 0x7F) == (USBD_AUHID_AUDIO_FEEDBACK_EPT & 0x7F))
  {
    paudio_hid->audio_feedback_state = 0;
  }
  else if((ept_num & 0x7F) == (USBD_AUHID_HID_IN_EPT & 0x7F))
  {
    usbd_flush_tx_fifo(udev, USBD_AUHID_HID_IN_EPT);
  }
  return status;
}
```

- OUT transfer processing (receive data)

```
static usb_sts_type class_out_handler(void *udev, uint8_t ept_num)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  usb_audio_hid_type *paudio_hid = (usb_audio_hid_type *)pudev->class_handler->pdata;
  uint16_t g_rxlen;

  /* get endpoint receive data length  */
  g_rxlen = usbd_get_recv_len(pudev, ept_num);

  if((ept_num & 0x7F) == (USBD_AUHID_AUDIO_SPK_OUT_EPT & 0x7F))
  {
```

```
      /* speaker data*/
      audio_codec_spk_fifo_write(paudio_hid->audio_spk_data, g_rxlen);
      paudio_hid->audio_spk_out_stage = 1;
      /* get next data */
      usbd_ept_recv(pudev, USBD_AUHID_AUDIO_SPK_OUT_EPT, paudio_hid->audio_spk_data,
AUDIO_SPK_OUT_MAXPACKET_SIZE);
   }
   else if((ept_num & 0x7F) == (USBD_AUHID_HID_OUT_EPT & 0x7F))
   {
      /* hid buffer process */
      usb_hid_buf_process(udev, paudio_hid->g_rxhid_buff, g_rxlen);

      /* start receive next packet */
      usbd_ept_recv(pudev, USBD_AUHID_HID_OUT_EPT, paudio_hid->g_rxhid_buff,
USBD_AUHID_OUT_MAXPACKET_SIZE);
   }
   return status;
}
```

3.11.4 How to develop based on composite_audio_hid routine

This section briefly introduces how to modify the code of composite_audio_hid routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify device description (audio_hid_desc.c, audio_hid_desc.h)
  Device description (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Other descriptions
- According to specific functional requirements to modify endpoints (audio_hid _class.c, audio_hid _class.h)
  Endpoint definition (audio_hid _class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)
- Modify control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- Send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)

3.12 virtual_comport routine

In modern PCs, USB is the standard interface for communicating with most peripherals. Nonetheless, most industrial software still uses the COM interface (UART) to communicate. The virtual_comport routine provides a solution of simulating COM interface using USB devices, and shows how to perform USB data sending/receiving and USART forwarding through the CDC protocol. This routine requires a virtual serial port driver, which can be downloaded from the official

website.

3.12.1 Implement functions

- Implement a virtual serial port, forward the data received through the upper computer to the USART, and forward the data received from the USART to the upper computer.

3.12.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration
- Endpoint 1 IN: for data sending
- Endpoint 1 OUT: for data receiving
- Endpoint 2 IN: interrupt transfer monitoring

USART:

- USART2: data sending/receiving through serial port

3.12.3 virtual_comport implementation

3.12.3.1 Device description (cdc_desc.c/h)

Refer to 3.7.3.1.

3.12.3.2 Data processing (cdc_class.c/h)

Refer to 3.7.3.2.

3.12.4 How to develop based on virtual_comport routine

This section briefly introduces how to modify the code of cdc routine for development according to the specific requirements of the application.

- According to specific functional requirements to modify device description (cdc_desc.c, cdc_desc.h)
  Device description (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Other descriptions
- According to specific functional requirements to modify endpoints (cdc_class.c, cdc_class.h)
  Endpoint definition (cdc_class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)
- Modify cdc control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- cdc send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)

3.13 composite_vcp_msc routine

Definition: A composite device refers to a device with multiple independent interfaces.
A composite device has multiple combined features. For example, the composite vcp msc demo provides MSC and CDC functions.

3.13.1 Implement functions

- Implement an USB virtual serial port (refer to 3.7)
- Implement an USB MSC device (refer to 3.5)

3.13.2 Peripherals

USB peripherals:

- Endpoint 0 IN/OUT: for USB enumeration
- Endpoint 1 IN: for CDC data sending
- Endpoint 1 OUT: for CDC data receiving
- Endpoint 2 IN: CDC command interrupt transfer
- Endpoint 2 OUT: for MSC data receiving
- Endpoint 3 IN: for MSC data sending

3.13.3 composite_vcp_msc implementation

3.9.3.1 Device description (cdc_msc_desc.c/h)

- cdc_msc device description (g_usbd_descriptor)
- cdc_msc device configuration description (g_usbd_configuration)
  cdc interface
  cdc endpoint
  msc interface
  msc endpoint
- Lang id (g_string_lang_id)
- Serial number (g_string_serial)
- Vendor/product ID (cdc_msc_desc.h)

```
#define USBD_CDC_MSC_VENDOR_ID                0x2E3C
#define USBD_CDC_MSC_PRODUCT_ID               0x5760
```

- Manufacturer, product name, configuration description and interface description

```
#define USBD_CDC_DESC_MANUFACTURER_STRING     "Artery"
#define USBD_CDC_DESC_PRODUCT_STRING          "AT32 Composite VCP and MSC   "
#define USBD_CDC_DESC_CONFIGURATION_STRING    "Composite VCP and MSC Config"
#define USBD_CDC_DESC_INTERFACE_STRING        "Composite VCP and MSC Interface"
```

3.9.3.2 Data processing (cdc_msc_class.c/h)

- Endpoint initialization (class_init_handler)

```
/* open in endpoint */
```

```
/* open in endpoint */
  usbd_ept_open(pudev, USBD_CDC_INT_EPT, EPT_INT_TYPE,
USBD_CDC_CMD_MAXPACKET_SIZE);
  /* open in endpoint */
  usbd_ept_open(pudev, USBD_CDC_BULK_IN_EPT, EPT_BULK_TYPE,
USBD_CDC_MSC_IN_MAXPACKET_SIZE);
  /* open out endpoint */
  usbd_ept_open(pudev, USBD_CDC_BULK_OUT_EPT, EPT_BULK_TYPE,
USBD_CDC_MSC_OUT_MAXPACKET_SIZE);
  /* set out endpoint to receive status */
  usbd_ept_recv(pudev, USBD_CDC_BULK_OUT_EPT, pcdcmsc->g_rx_buff,
USBD_CDC_MSC_OUT_MAXPACKET_SIZE);
  /* open in endpoint */
  usbd_ept_open(pudev, USBD_MSC_BULK_IN_EPT, EPT_BULK_TYPE,
USBD_CDC_MSC_IN_MAXPACKET_SIZE);
  /* open out endpoint */
  usbd_ept_open(pudev, USBD_MSC_BULK_OUT_EPT, EPT_BULK_TYPE,
USBD_CDC_MSC_OUT_MAXPACKET_SIZE);
```

- Endpoint clear (class_clear_handler)

```
  /* close in endpoint */
  usbd_ept_close(pudev, USBD_CDC_INT_EPT);
  /* close in endpoint */
  usbd_ept_close(pudev, USBD_CDC_BULK_IN_EPT);
  /* close out endpoint */
  usbd_ept_close(pudev, USBD_CDC_BULK_OUT_EPT);
  /* close in endpoint */
  usbd_ept_close(pudev, USBD_MSC_BULK_IN_EPT);
  /* close out endpoint */
  usbd_ept_close(pudev, USBD_MSC_BULK_OUT_EPT);
```

- Device request (class_setup_handler)

  cdc device class requests:
  SET_LINE_CODING
  GET_LINE_CODING

  MSC device class requests:
  GET_MAX_LUN
  BO_RESET

  The code is as follows:

```
switch(setup->bmRequestType & USB_REQ_RECIPIENT_MASK)
  {
    case USB_REQ_RECIPIENT_INTERFACE:
      if(setup->wIndex == VCPMSC_MSC_INTERFACE)
      {
        msc_class_setup_handler(udev, setup);
      }
```

```
        else
        {
          cdc_class_setup_handler(pudev, setup);
        }
        break;
      case USB_REQ_RECIPIENT_ENDPOINT:
        if(setup->wIndex == (USBD_MSC_BULK_IN_EPT | USBD_MSC_BULK_OUT_EPT))
        {
          msc_class_setup_handler(udev, setup);
        }
        else
        {
          cdc_class_setup_handler(pudev, setup);
        }
        break;

    }
```

- IN transfer processing

```
static usb_sts_type class_in_handler(void *udev, uint8_t ept_num)
{
  usbd_core_type *pudev = (usbd_core_type *)udev;
  cdc_msc_struct_type *pcdcmsc = (cdc_msc_struct_type *)pudev->class_handler->pdata;
  usb_sts_type status = USB_OK;

  /* ...user code...
    trans next packet data
  */
  usbd_flush_tx_fifo(pudev, ept_num);

  if((ept_num & 0x7F) == (USBD_CDC_BULK_IN_EPT & 0x7F))
  {
    pcdcmsc->g_tx_completed = 1;
  }
  if((ept_num & 0x7F) == (USBD_MSC_BULK_IN_EPT & 0x7F))
  {
    bot_scsi_datain_handler(udev, ept_num);
  }

  return status;
}
```

CDC send data:

```
error_status usb_vcp_send_data(void *udev, uint8_t *send_data, uint16_t len)
{
  error_status status = SUCCESS;
  usbd_core_type *pudev = (usbd_core_type *)udev;
```

```
  cdc_msc_struct_type *pcdc = (cdc_msc_struct_type *)pudev->class_handler->pdata;
  if(pcdc->g_tx_completed)
  {
    pcdc->g_tx_completed = 0;
    usbd_ept_send(pudev, USBD_CDC_BULK_IN_EPT, send_data, len);
  }
  else
  {
    status = ERROR;
  }
  return status;
}
```

- OUT transfer processing (receive data)

```
static usb_sts_type class_out_handler(void *udev, uint8_t ept_num)
{
  usb_sts_type status = USB_OK;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  cdc_msc_struct_type *pcdcmsc = (cdc_msc_struct_type *)pudev->class_handler->pdata;
  if((ept_num & 0x7F) == (USBD_CDC_BULK_OUT_EPT & 0x7F))
  {
    /* get endpoint receive data length    */
    pcdcmsc->g_rxlen = usbd_get_recv_len(pudev, ept_num);

    /*set recv flag*/
    pcdcmsc->g_rx_completed = 1;
  }
  if((ept_num & 0x7F) == (USBD_MSC_BULK_OUT_EPT & 0x7F))
  {
    bot_scsi_dataout_handler(udev, ept_num);
  }

  return status;
}

uint16_t usb_vcp_get_rxdata(void *udev, uint8_t *recv_data)
{
  uint16_t i_index = 0;
  uint16_t tmp_len = 0;
  usbd_core_type *pudev = (usbd_core_type *)udev;
  cdc_msc_struct_type *pcdc = (cdc_msc_struct_type *)pudev->class_handler->pdata;

 if(pcdc->g_rx_completed == 0)
  {
    return 0;
  }
```

```
    pcdc->g_rx_completed = 0;
    tmp_len = pcdc->g_rxlen;
    for(i_index = 0; i_index < pcdc->g_rxlen; i_index ++)
    {
      recv_data[i_index] = pcdc->g_rx_buff[i_index];
    }

    usbd_ept_recv(pudev, USBD_CDC_BULK_OUT_EPT, pcdc->g_rx_buff,
USBD_CDC_MSC_OUT_MAXPACKET_SIZE);

    return tmp_len;
}
```

3.13.4 How to develop based on composite_vcp_msc routine

This section briefly introduces how to modify the code of composite_vcp_msc routine for composite device development according to the specific requirements of the application.

- According to specific functional requirements to modify device description (cdc_msc_desc.c, cdc_msc _desc.h)
  Device description (g_usbd_descriptor)
  Device configuration description (g_usbd_configuration)
  Other descriptions
- According to specific functional requirements to modify endpoints (cdc_msc_class.c, cdc_msc_class.h)
  Endpoint definition (cdc_msc_class.h)
  Endpoint initialization (class_init_handler, class_clear_handler)
- Modify control requests
  Control request modification (class_setup_handler)
  Control request setting (class_ept0_rx_handler)
- cdc_msc send/receive data processing/modification
  IN data processing (class_in_handler)
  OUT data processing (class_out_handler)
- According to specific requirements to modify endpoint FIFO size allocation (usb_conf.h)

4 Revision history

Table 15. Document revision history

Date	Version	Revision note
2021.11.05	2.0.0	Initial release
2022.04.14	2.0.1	Modified device class structures and added “pdata” parameter; Modified macro definitions and names of each device class; Modified data processing of certain device class; Added composite_audio_hid routine; Added virtual_comport routine.
2022.05.16	2.0.2	Added composite_vcp_msc routine.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers are solely responsible for the selection and use of ARTERY's products and services; ARTERY assumes no liability for purchasers' selection or use of the products and the relevant services.

No license, express or implied, to any intellectual property right is granted by ARTERY herein regardless of the existence of any previous representation in any forms. If any part of this document involves third party's products or services, it does NOT imply that ARTERY authorizes the use of the third party's products or services, or permits any of the intellectual property, or guarantees any uses of the third party's products or services or intellectual property in any way.

Except as provided in ARTERY's terms and conditions of sale for such products, ARTERY disclaims any express or implied warranty, relating to use and/or sale of the products, including but not restricted to liability or warranties relating to merchantability, fitness for a particular purpose (based on the corresponding legal situation in any unjudicial districts), or infringement of any patent, copyright, or other intellectual property right.

ARTERY's products are not designed for the following purposes, and thus not intended for the following uses: (A) Applications that have specific requirements on safety, for example: life-support applications, active implant devices, or systems that have specific requirements on product function safety; (B) Aviation applications; (C) Auto-motive application or environment; (D) Aerospace applications or environment, and/or (E) weapons. Since ARTERY products are not intended for the above-mentioned purposes, if purchasers apply ARTERY products to these purposes, purchasers are solely responsible for any consequences or risks caused, even if any written notice is sent to ARTERY by purchasers; in addition, purchasers are solely responsible for the compliance with all statutory and regulatory requirements regarding these uses.

Any inconsistency of the sold ARTERY products with the statement and/or technical features specification described in this document will immediately cause the invalidity of any warranty granted by ARTERY products or services stated in this document by ARTERY, and ARTERY disclaims any responsibility in any form.