NXP Semiconductors
User's Guide

Document Number: MCUXSDKUSBCOMDUG
Rev. 10, June 2019

MCUXpresso SDK USB Stack Composite Device

User's Guide

Contents

Chapter 1
Overview

This document describes steps to implement a composite device based on the USB stack.

The USB Stack provides five composite device demos, *HID+audio*, *MSC+CDC, MSC_SDCARD+CDC, CDC_VCOM +CDC_VCOMAND*, and *mouse + keyboard*. However, users can create composite devices to fit their needs. This document is a step-by-step guide to create a customizable composite device**.**

Chapter 2
Introduction

A composite device combines multiple independent functionalities by unifying independent functionality code into one implementation. For example, the single functionality code for CDC is provided in the CDC example and the single functionality code for MSC is provided in the MSC example. Creating the CDC+MSC composite device example requires combining the CDC example code and MSC example code into a single example.

Composite device descriptors are combined from the single-function device descriptors. There are two single-function devices. Each device has an interfaces descriptor in a configuration descriptor. If the composite device is combined using two single function devices, the interface descriptor of each device should be merged into the composite device configuration descriptor.

Implementing a composite device involves combining the descriptors and the functionality of the single function devices.

Chapter 3
Setup

Before developing the composite device, the user needs to:

1. Decide how many classes to include in this composite device.
2. Decide which types of classes to include in this composite device, for example, HID + AUDIO, HID + HID, and so on.
3. Prepare the device descriptor depending on the use case. In particular, the IAD should be used for AUDIO/VIDEO class. For more information, see www.usb.org/developers/docs/whitepapers/iadclasscode_r10.pdf.
4. Ensure that the functionality of the single function device code is valid.

3.1 Design steps

1. A new composite device application should use the existing examples as a template.
2. Prepare the descriptor-related data structure to ensure that the correct information about the customized composite device is relayed to the USB device stack. See Section 4, "*USB composite device structures*".
3. Prepare the descriptors array and ensure that the descriptors are consistent with the descriptor-related data structure. See Section 5, "*USB descriptor functions*".
4. Implement the specific descriptor-related callback function, which the USB device stack calls to get the device descriptor. See Section 5, "*USB descriptor functions*"..

Chapter 4
USB composite device structures

The USB composite device structures are defined in the USB stack code. The structures describe the class and are consistent with the descriptor. They are also used in single function examples.

4.1 usb_device_class_config_list_struct_t

This structure is required for the composite device and relays device callback, class callback, interface numbers, and endpoint numbers of each interface to the class driver. The structure should be placed in the “composite.c” file.

This is an example for a composite device MSD + CDC:

```
usb_device_class_config_list_struct_t g_compositeDeviceConfigList =
       {
       .config = g_compositeDevice,
        .deviceCallback = USB_DeviceCallback,
        .count = 2,
        };
```

The variable “count” holds the number of classes included in the composite device. Because the composite device MSD+CDC includes two classes, the value of variable "count" is 2.

The type of “config” is usb_device_class_config_struct_t. See the next Section for more information.

4.2 usb_device_class_config_struct_t

This structure is required for the composite device and provides information about each class. The structure should be placed in the “composite.c” file.

This is an example for the composite device MSD + CDC:

```
usb_device_class_config_struct_t g_compositeDevice[2] =
        {
    {
            .classCallback = USB_DeviceCdcVcomCallback,
        .classHandle = (class_handle_t)NULL,
        .classInfomation = &g_UsbDeviceCdcVcomConfig,
    },
    {
            .classCallback = USB_DeviceMscCallback,
        .classHandle = (class_handle_t)NULL,
        .classInfomation = &g_mscDiskClass,
}
};
```

classCallback is the callback function pointer of each class.

classHandle is the class handle. This value is NULL and updated by the USB_DeviceClassInit function.

The type of *classInfomation* is usb_device_class_struct_t, including the configuration count, class type, and the interface list for this class.

4.3 usb_device_class_struct_t

This structure is required for each class including the class type, supported configuration count, and interface list for each configuration. The structure should be placed in the “usb_device_descriptor.c” file.

This is an example for MSD in the composite MSD + CDC device example.

```
usb_device_class_struct_t g_mscDiskClass =
{
    .interfaceList = g_mscDiskInterfaceList,
    .type = kUSB_DeviceClassTypeMsc,
    .configurations = USB_DEVICE_CONFIGURATION_COUNT,
};
```

Type represents the type of each class included in the composite device. For example, the type of MSD class is kUSB_DeviceClassTypeMsc.

interfaceList is the interface list pointer, which points to the type usb_device_interface_list_t. It includes detailed interface information about the class including interface count, alternate setting count for each interface, and ep count, ep type, and ep direction for each alternate setting. See the next section for more information.

Configurations member indicates the count of the class supported.

4.4 usb_device_interface_list_t

This structure is required for the composite device and provides information about each class. The structure should be placed in the “usb_device_descriptor.c” file.

This is an example for MSC in the composite MSC + CDC device example.

```
usb_device_interface_list_t g_mscDiskInterfaceList[USB_DEVICE_CONFIGURATION_COUNT] =
{
    {
        .count = USB_MSC_DISK_INTERFACE_COUNT,
        .interfaces = g_mscDiskInterfaces,
    },
};
```

Count means the interface count this class supports in each configuration.

Interfaces member indicates the interface list for each configuration.

4.5 usb_device_interfaces_struct_t

This structure provides alternate setting interface information about each interface. All structures should be placed in the “usb_device_descriptor.c” file.

Prototype:

```
typedef struct _usb_device_interfaces_struct
{
    uint8_t                          classCode;
    uint8_t                          subclassCode;
```

```
    uint8_t                         protocolCode;
    uint8_t                         interfaceNumber;
    usb_device_interface_struct_t*  interface;
    uint8_t                         count;
} usb_device_interfaces_struct_t;
```

Description:

- classCode: The class code for this interface.
- subclassCode: The subclass code for this interface.
- protocolCode: The protocol code for this interface.
- interfaceNumber: Interface index in the interface descriptor.
- interfaceNumber: Interface index in the interface descriptor.
- count: Number of interfaces in the current interface.

This is an example for the composite device MSD + CDC:

MSD:

```
usb_device_interfaces_struct_t g_mscDiskInterfaces[USB_MSC_DISK_INTERFACE_COUNT] =
{
    {
        USB_MSC_DISK_CLASS,
        USB_MSC_DISK_SUBCLASS,
        USB_MSC_DISK_PROTOCOL,
        USB_MSC_DISK_INTERFACE_INDEX,
        g_mscDiskInterface,
        sizeof(g_mscDiskInterface) / sizeof(usb_device_interfaces_struct_t),
    }
};
```

`USB_MSC_DISK_INTERFACE_INDEX` is the interface index of this interface in a current configuration. In other words, in the interface descriptor, the interface number is `USB_MSC_DISK_INTERFACE_INDEX`.

`"g_mscDiskInterface"` is the interface detailed information structure. See Section 4.6 for more information.

CDC:

```
usb_device_interfaces_struct_t g_cdcVcomInterfaces[USB_CDC_VCOM_INTERFACE_COUNT] =
{
    {
        USB_CDC_VCOM_CIC_CLASS,
        USB_CDC_VCOM_CIC_SUBCLASS,
        USB_CDC_VCOM_CIC_PROTOCOL,
        USB_CDC_VCOM_CIC_INTERFACE_INDEX,
        g_cdcVcomCicInterface, sizeof(g_cdcVcomCicInterface) / sizeof(usb_device_interfaces_struct_t)
    },
    {
        USB_CDC_VCOM_DIC_CLASS,
        USB_CDC_VCOM_DIC_SUBCLASS,
        USB_CDC_VCOM_DIC_PROTOCOL,
        USB_CDC_VCOM_DIC_INTERFACE_INDEX,
        g_cdcVcomDicInterface, sizeof(g_cdcVcomDicInterface) / sizeof(usb_device_interfaces_struct_t)
    },
};
```

`USB_CDC_VCOM_CIC_INTERFACE_INDEX` is the interface index of the control interface in a current configuration. In other words, in the interface descriptor, the interface number is `USB_CDC_VCOM_CIC_INTERFACE_INDEX`.

`USB_CDC_VCOM_DIC_INTERFACE_INDEX` is the interface index of the data interface in a current configuration. In other words, in the interface descriptor, the interface number is

```
USB_CDC_VCOM_DIC_INTERFACE_INDEX
```

.

`"g_cdcVcomCicInterface"` is the control interface detail information structure. See Section 4.6 for more information.

`"g_cdcVcomDicInterface"` is the data interface detail information structure. See Section 4.6 for more information.

4.6 usb_device_interface_struct_t

This structure provides information about each alternate setting interface for the current interface. All structures should be placed in the "usb_device_descriptor.c" file.

Prototype:

```
typedef struct _usb_device_interface_struct
{
    uint8_t                          alternateSetting;
    usb_device_endpoint_list_t       endpointList;
    void*                            classSpecific;
 } usb_device_interface_struct_t;
```

Description:

- alternateSetting: The alternate value of this interface.
- endpointList: endpoint list structure. See the usb_device_endpoint_list_t structure.
- classSpecific: The class-specific structure pointer.

Prototype:

```
typedef struct _usb_device_endpoint_list
{
    uint8_t                          count;
    usb_device_endpoint_struct_t*    endpoint;
} usb_device_endpoint_list_t;
```

Description:

- count: Number of endpoints in the current interface.
- endpoint: Endpoint information structure.

This is an example for the composite device MSD + CDC:

MSD:

```
usb_device_interface_struct_t g_mscDiskInterface[] =
{
    {
        0,
        {
            USB_MSC_DISK_ENDPOINT_COUNT,
            g_mscDiskEndpoints,
        },
```

```
    },
};
```

Number “0” holds the alternate setting value of the MSD interface.

USB_MSC_DISK_ENDPOINT_COUNT is the endpoint number for MSD interface when the alternate setting is 0.

“g_mscDiskEndpoints” is the endpoint detailed information structure. See Section 4.7 for more information.

CDC:

For control interface:

```
/* Define interface for communication class */
usb_device_interface_struct_t g_cdcVcomCicInterface[] =
{
    {
        0,
        {
            USB_CDC_VCOM_CIC_ENDPOINT_COUNT,
            g_cdcVcomCicEndpoints,
        },
    }
};
```

Number “0” holds the alternate setting value of the CDC control interface.

USB_CDC_VCOM_CIC_ENDPOINT_COUNT is the endpoint number for control interface when the alternate setting is 0.

“g_cdcVcomCicEndpoints" is the endpoint detailed information structure. See Section 4.7 for more information.

For data interface:

```
/* Define interface for data class */
usb_device_interface_struct_t g_cdcVcomDicInterface[] =
{
    {
       0,
       {
           USB_CDC_VCOM_DIC_ENDPOINT_COUNT,
           g_cdcVcomDicEndpoints,
       },
    }
};
```

Number “0” holds the alternate setting value of the CDC data interface.

USB_CDC_VCOM_DIC_ENDPOINT_COUNT is the endpoint number for control interface when the alternate setting is 0.

“g_cdcVcomDicEndpoints” is the endpoint detailed information structure. See Section 4.7 for more information.

4.7 usb_device_endpoint_struct_t

This structure is required for the composite device and provides ep information. All structures should be placed in the “usb_device_descriptor.c” file.

Prototype:

```
typedef struct _usb_device_endpoint_struct
{
    uint8_t               endpointAddress;    /*! endpoint address*/
```

```
    uint8_t               transferType;       /*! endpoint transfer type*/
    uint16_t              maxPacketSize;     /*! endpoint max packet size */
} usb_device_endpoint_struct_t;
```

Description:

- endpointAddress: Endpoint address (b7, 0 – USB_OUT, 1 – USB_IN).
- transferType: The transfer type of this endpoint.
- maxPacketSize: The maximum packet size of this endpoint.

This is an example for the composite device MSD + CDC:

MSD:

```
usb_device_endpoint_struct_t g_mscDiskEndpoints[USB_MSC_DISK_ENDPOINT_COUNT] =
{
    {
        USB_MSC_DISK_BULK_IN_ENDPOINT | (USB_IN << 7U),
        USB_ENDPOINT_BULK,
        FS_MSC_DISK_BULK_IN_PACKET_SIZE,
    },
    {
        USB_MSC_DISK_BULK_OUT_ENDPOINT | (USB_OUT << 7U),
        USB_ENDPOINT_BULK,
        FS_MSC_DISK_BULK_OUT_PACKET_SIZE,
    }
};
```

CDC:

This is CDC class control interface endpoint information.

```
/* Define endpoint for communication class */
usb_device_endpoint_struct_t g_cdcVcomCicEndpoints[USB_CDC_VCOM_CIC_ENDPOINT_COUNT] =
{
    {
        USB_CDC_VCOM_CIC_INTERRUPT_IN_ENDPOINT | (USB_IN << 7U),
        USB_ENDPOINT_INTERRUPT,
        HS_CDC_VCOM_BULK_IN_PACKET_SIZE,
    },
};
```

This is the CDC class data interface endpoint information.

```
/* Define endpoint for data class */
usb_device_endpoint_struct_t g_cdcVcomDicEndpoints[USB_CDC_VCOM_DIC_ENDPOINT_COUNT] =
{
    {
        USB_CDC_VCOM_DIC_BULK_IN_ENDPOINT | (USB_IN << 7U),
        USB_ENDPOINT_BULK,
        FS_CDC_VCOM_BULK_IN_PACKET_SIZE,
    },
    {
        USB_CDC_VCOM_DIC_BULK_OUT_ENDPOINT | (USB_OUT << 7U),
        USB_ENDPOINT_BULK,
        FS_CDC_VCOM_BULK_OUT_PACKET_SIZE,
    },
};
```

Chapter 5
USB descriptor functions

All USB device descriptor and functions are placed in the “usb_device_descriptor.c” file.

5.1 USB descriptor

The descriptors for each class can be obtained from the class-related examples and class specification. For the composite device, combine multiple class descriptors.

NOTE

The interface number in the configuration descriptor must be the correct interface number value. The endpoint number value in each endpoint descriptor must be consistent with the structures in Section 1.

5.2 USB_DeviceGetDeviceDescriptor

This function is used to get the device descriptor. All devices must implement this function.

```
usb_status_t USB_DeviceGetDeviceDescriptor(usb_device_handle handle,
                                           usb_device_get_device_descriptor_struct_t
*deviceDescriptor)
{
    deviceDescriptor->buffer = g_UsbDeviceDescriptor;
    deviceDescriptor->length = USB_DESCRIPTOR_LENGTH_DEVICE;
    return kStatus_USB_Success;
}
```

5.3 USB_DeviceGetConfigurationDescriptor

This function is used to get the configuration descriptor. All devices must implement this function.

```
/* Get device configuration descriptor request */
usb_status_t USB_DeviceGetConfigurationDescriptor(
    usb_device_handle handle, usb_device_get_configuration_descriptor_struct_t
*configurationDescriptor)
{
    if (USB_COMPOSITE_CONFIGURE_INDEX > configurationDescriptor->configuration)
    {
        configurationDescriptor->buffer = g_UsbDeviceConfigurationDescriptor;
        configurationDescriptor->length = USB_DESCRIPTOR_LENGTH_CONFIGURATION_ALL;
        return kStatus_USB_Success;
    }
    return kStatus_USB_InvalidRequest;
}
```

5.4 USB_DeviceGetStringDescriptor

This function is used to get the string descriptor. All devices must implement this function.

```
/* Get device string descriptor request */
usb_status_t USB_DeviceGetStringDescriptor(usb_device_handle handle,
                                           usb_device_get_string_descriptor_struct_t
*stringDescriptor)
{
    if (stringDescriptor->stringIndex == 0U)
    {
        stringDescriptor->buffer = (uint8_t *)g_UsbDeviceLanguageList.languageString;
        stringDescriptor->length = g_UsbDeviceLanguageList.stringLength;
    }
    else
    {
        uint8_t languageId = 0U;
        uint8_t languageIndex = USB_DEVICE_STRING_COUNT;

        for (; languageId < USB_DEVICE_STRING_COUNT; languageId++)
        {
            if (stringDescriptor->languageId ==
g_UsbDeviceLanguageList.languageList[languageId].languageId)
            {
                if (stringDescriptor->stringIndex < USB_DEVICE_STRING_COUNT)
                {
                    languageIndex = stringDescriptor->stringIndex;
                }
                break;
            }
        }

        if (USB_DEVICE_STRING_COUNT == languageIndex)
        {
            return kStatus_USB_InvalidRequest;
        }
        stringDescriptor->buffer = (uint8_t
*)g_UsbDeviceLanguageList.languageList[languageId].string[languageIndex];
        stringDescriptor->length =
g_UsbDeviceLanguageList.languageList[languageId].length[languageIndex];
    }
    return kStatus_USB_Success;
}
```

5.5 USB_DeviceGetHidDescriptor

```
/* Get HID descriptor request */
usb_status_t USB_DeviceGetHidDescriptor(usb_device_handle handle,
                                        usb_device_get_hid_descriptor_struct_t *hidDescriptor)
{
    /* If this request is not supported, return the error code “kStatus_USB_InvalidRequest”.
Otherwise, fill the hidDescriptor with the descriptor buffer address and length based on the
```

```
interface number. */
    return kStatus_USB_InvalidRequest;
}
```

5.6 USB_DeviceGetHidReportDescriptor

```
/* Get the HID report descriptor request */
usb_status_t USB_DeviceGetHidReportDescriptor(usb_device_handle handle,
                                              usb_device_get_hid_report_descriptor_struct_t
*hidReportDescriptor)
{
    if (USB_HID_GENERIC_INTERFACE_INDEX == hidReportDescriptor->interfaceNumber)
    {
        hidReportDescriptor->buffer = g_UsbDeviceHidGenericReportDescriptor;
        hidReportDescriptor->length = USB_DESCRIPTOR_LENGTH_HID_GENERIC_REPORT;
    }
    else if (USB_HID_KEYBOARD_INTERFACE_INDEX == hidReportDescriptor->interfaceNumber)
    {
        hidReportDescriptor->buffer = g_UsbDeviceHidKeyboardReportDescriptor;
        hidReportDescriptor->length = USB_DESCRIPTOR_LENGTH_HID_KEYBOARD_REPORT;
    }
    else
    {
        return kStatus_USB_InvalidRequest;
    }
    return kStatus_USB_Success;
}
```

5.7 USB_DeviceGetHidPhysicalDescriptor

```
/* Get the HID physical descriptor request */
usb_status_t USB_DeviceGetHidPhysicalDescriptor(
    usb_device_handle handle, usb_device_get_hid_physical_descriptor_struct_t *hidPhysicalDescriptor)
{
    /* If this request is not supported, return the error code “kStatus_USB_InvalidRequest”.
Otherwise, fill the hidPhysicalDescriptor with the descriptor buffer address and length based on the
interface number and the physical index. */
    return kStatus_USB_InvalidRequest;
}
```

5.8 USB_DeviceSetSpeed

```
/* Because HS and FS descriptors are different, update the device descriptors and configurations to
match the current speed.
 * By default, the device descriptors and configurations are configured by using FS parameters for
both EHCI and KHCI.
 * When the EHCI is enabled, the application needs to call this function to update the device by
using current speed.
 * The updated information includes the endpoint max packet size, endpoint interval, and so on. */
```

```
usb_status_t USB_DeviceSetSpeed(usb_device_handle handle, uint8_t speed)
{
    usb_descriptor_union_t *descriptorHead;
    usb_descriptor_union_t *descriptorTail;

    descriptorHead = (usb_descriptor_union_t *)&g_UsbDeviceConfigurationDescriptor[0];
    descriptorTail = (usb_descriptor_union_t *)
(&g_UsbDeviceConfigurationDescriptor[USB_DESCRIPTOR_LENGTH_CONFIGURATION_ALL - 1U]);

    while (descriptorHead < descriptorTail)
    {
        if (descriptorHead->common.bDescriptorType == USB_DESCRIPTOR_TYPE_ENDPOINT)
        {
            if (USB_SPEED_HIGH == speed)
            {
                if (USB_HID_KEYBOARD_ENDPOINT_IN == (descriptorHead->endpoint.bEndpointAddress &
USB_ENDPOINT_NUMBER_MASK))
                {
                    descriptorHead->endpoint.bInterval = HS_HID_KEYBOARD_INTERRUPT_IN_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(HS_HID_KEYBOARD_INTERRUPT_IN_PACKET_SIZE,
                                                       descriptorHead->endpoint.wMaxPacketSize);
                }
                else if (((descriptorHead->endpoint.bEndpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN) ==
                          USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN) &&
                         (USB_HID_GENERIC_ENDPOINT_IN == (descriptorHead->endpoint.bEndpointAddress &
USB_ENDPOINT_NUMBER_MASK)))
                {
                    descriptorHead->endpoint.bInterval = HS_HID_GENERIC_INTERRUPT_IN_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(HS_HID_GENERIC_INTERRUPT_IN_PACKET_SIZE,
                                                       descriptorHead->endpoint.wMaxPacketSize);
                }
                else if (((descriptorHead->endpoint.bEndpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_OUT) ==
                          USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_OUT) &&
                         (USB_HID_GENERIC_ENDPOINT_OUT == (descriptorHead->endpoint.bEndpointAddress
& USB_ENDPOINT_NUMBER_MASK)))
                {
                    descriptorHead->endpoint.bInterval = HS_HID_GENERIC_INTERRUPT_OUT_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(HS_HID_GENERIC_INTERRUPT_OUT_PACKET_SIZE,
                                                       descriptorHead->endpoint.wMaxPacketSize);
                }
            }
            else
            {
                if (USB_HID_KEYBOARD_ENDPOINT_IN == (descriptorHead->endpoint.bEndpointAddress &
USB_ENDPOINT_NUMBER_MASK))
                {
                    descriptorHead->endpoint.bInterval = FS_HID_KEYBOARD_INTERRUPT_IN_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(FS_HID_KEYBOARD_INTERRUPT_IN_PACKET_SIZE,
                                                       descriptorHead->endpoint.wMaxPacketSize);
                }
                else if (((descriptorHead->endpoint.bEndpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN) ==
                          USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN) &&
                         (USB_HID_GENERIC_ENDPOINT_IN == (descriptorHead->endpoint.bEndpointAddress &
USB_ENDPOINT_NUMBER_MASK)))
                {
                    descriptorHead->endpoint.bInterval = FS_HID_GENERIC_INTERRUPT_IN_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(FS_HID_GENERIC_INTERRUPT_IN_PACKET_SIZE,
```

```
                                                    descriptorHead->endpoint.wMaxPacketSize);
                }
                else if (((descriptorHead->endpoint.bEndpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_OUT) ==
                          USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_OUT) &&
                         (USB_HID_GENERIC_ENDPOINT_OUT == (descriptorHead->endpoint.bEndpointAddress
& USB_ENDPOINT_NUMBER_MASK)))
                {
                    descriptorHead->endpoint.bInterval = FS_HID_GENERIC_INTERRUPT_OUT_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(FS_HID_GENERIC_INTERRUPT_OUT_PACKET_SIZE,
                                                    descriptorHead->endpoint.wMaxPacketSize);
                }
            }
        }
        descriptorHead = (usb_descriptor_union_t *)((uint8_t *)descriptorHead + descriptorHead-
>common.bLength);
    }

    for (int i = 0U; i < USB_HID_GENERIC_ENDPOINT_COUNT; i++)
    {
        if (USB_SPEED_HIGH == speed)
        {
            if (g_UsbDeviceHidGenericEndpoints[i].endpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN)
            {
                g_UsbDeviceHidGenericEndpoints[i].maxPacketSize =
HS_HID_GENERIC_INTERRUPT_IN_PACKET_SIZE;
            }
            else
            {
                g_UsbDeviceHidGenericEndpoints[i].maxPacketSize =
HS_HID_GENERIC_INTERRUPT_OUT_PACKET_SIZE;
            }
        }
        else
        {
            if (g_UsbDeviceHidGenericEndpoints[i].endpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN)
            {
                g_UsbDeviceHidGenericEndpoints[i].maxPacketSize =
HS_HID_GENERIC_INTERRUPT_OUT_PACKET_SIZE;
            }
            else
            {
                g_UsbDeviceHidGenericEndpoints[i].maxPacketSize =
FS_HID_GENERIC_INTERRUPT_OUT_PACKET_SIZE;
            }
        }
    }

    if (USB_SPEED_HIGH == speed)
    {
        g_UsbDeviceHidKeyboardEndpoints[0].maxPacketSize = HS_HID_KEYBOARD_INTERRUPT_IN_PACKET_SIZE;
    }
    else
    {
        g_UsbDeviceHidKeyboardEndpoints[0].maxPacketSize = FS_HID_KEYBOARD_INTERRUPT_IN_PACKET_SIZE;
    }
```

```
    return kStatus_USB_Success;
}
```

Chapter 6
USB stack configurations

Class configuration:

This section describes a use case where two or more of the same classes are used in the composite device.

To reduce the footprint, the released USB stack does not support multiple instances of the same class in the default configuration. If two or more same classes are used in the composite device, the user needs to configure the class.

- For HID class, USB_DEVICE_CONFIG_HID must be configured in the usb_device_config.h.
- For CDC class, USB_DEVICE_CONFIG_CDC_ACM must be configured in the usb_device_config.h.
- For MSD class, USB_DEVICE_CONFIG_MSC must be configured in the usb_device_config.h.
- For AUDIO class, USB_DEVICE_CONFIG_AUDIO must be configured in the usb_device_config.h.
- For PHDC class, USB_DEVICE_CONFIG_PHDC must be configured in the usb_device_config.h.
- For VIDEO class, USB_DEVICE_CONFIG_VIDEO must be configured in the usb_device_config.h.
- For CCID class, USB_DEVICE_CONFIG_CCID must be configured in the usb_device_config.h.

The value of the configuration depends on use cases and user requirements. For example, for the composite device HID+HID, the USB_DEVICE_CONFIG_HID must be set to 2.

NOTE

USBCFG_DEV_MAX_ENDPOINTS must not be less than “max used endpoint number + 1”. “max used endpoint number” indicates the maximum endpoint number that the example uses.

Chapter 7
Application template

The redesigned USB stack makes the composite device application easy to implement and aligned with the general device.

7.1 Application structure template

For a general device, a demo contains only one class. However, for the composite device, a demo contains more than one class. Likewise, a structure is required to manage the application involving more than one class.

```
typedef struct composite_device_struct
{
    usb_device_handle                  deviceHandle;
    class_handle_t                     classHandle1;
    class_handle_t                     classHandle2;
    …
    class_handle_t                     classHandlen;
    uint8_t                            speed;
    uint8_t                            attach;
    uint8_t                            currentConfiguration;
    uint8_t
currentInterfaceAlternateSetting[USB_COMPOSITE_INTERFACE_COUNT];
}composite_device_struct_t;
```

deviceHandle: The handle pointer to a device, which is returned by the USB_DeviceClassInit.

speed: Speed of the USB device. USB_SPEED_FULL/USB_SPEED_LOW/USB_SPEED_HIGH.

attach: Indicates whether the device is attached or not.

currentConfiguration: The current device configuration value.

currentInterfaceAlternateSetting: The current alternate setting for each interface.

classHandlen: The pointer to a class.

This is an example for a composite device HID mouse + HID keyboard:

This structure is in the “composite.h” file.

Prototype:

```
typedef struct _usb_device_composite_struct
{
    usb_device_handle                  deviceHandle;
    class_handle_t                     hidMouseHandle;
    class_handle_t                     hidKeyboardHandle;
    uint8_t                            speed;
    uint8_t                            attach;
    uint8_t                            currentConfiguration;
    uint8_t
currentInterfaceAlternateSetting[USB_COMPOSITE_INTERFACE_COUNT];
} usb_device_composite_struct_t;
```

7.2 Application initialization process

1. Before initializing the USB stack by calling the USB_DeviceClassInit function, the usb_device_class_config_list_struct_tand usb_device_class_config_struct_tare assigned values respectively. For example, for MSC + CDC, the steps are as follows:

   - Declare the g_compositeDeviceConfigList as global variables of the type usb_device_class_config_list_struct_t.

   ```
   usb_device_class_config_list_struct_t g_compositeDeviceConfigList =
   {
       g_compositeDevice,
       USB_DeviceCallback,
       2,
   };
   ```

   - Declare the g_compositeDevice as global variables of the type usb_device_class_config_struct_t.

   ```
   usb_device_class_config_struct_t g_compositeDevice[2] =
   {
       {
           USB_DeviceCdcVcomCallback,
           (class_handle_t)NULL,
           &g_UsbDeviceCdcVcomConfig,
       },
       {
           USB_DeviceMscCallback,
           (class_handle_t)NULL,
           &g_mscDiskClass,
       }
   };
   ```

   - Add a function for the USB device ISR.

     For EHCI,

     ```
     #if defined(USB_DEVICE_CONFIG_EHCI) && (USB_DEVICE_CONFIG_EHCI > 0U)
     void USBHS_IRQHandler(void)
     {
         USB_DeviceEhciIsrFunction(g_composite.deviceHandle);
     }
     #endif
     ```

     For KHC1,

     ```
     #if defined(USB_DEVICE_CONFIG_KHCI) && (USB_DEVICE_CONFIG_KHCI > 0U)
     void USB0_IRQHandler(void)
     {
         USB_DeviceKhciIsrFunction(g_composite.deviceHandle);
     }
     #endif
     ```

     For LPC IP3511,

     ```
     #if defined(USB_DEVICE_CONFIG_LPC3511IP) && (USB_DEVICE_CONFIG_LPC3511IP > 0U)
     void USB0_IRQHandler(void)
     {
         USB_DeviceLpc3511IpIsrFunction (g_composite.deviceHandle);
     ```

```
}
#endif
```

2. Enable the USB device clock.

   For EHC1,

```
CLOCK_EnableUsbhs0Clock(kCLOCK_UsbSrcPll0, CLOCK_GetFreq(kCLOCK_PllFllSelClk));
USB_EhciPhyInit(CONTROLLER_ID, BOARD_XTAL0_CLK_HZ);
```

   For KHC1,

```
#if ((defined FSL_FEATURE_USB_KHCI_IRC48M_MODULE_CLOCK_ENABLED) &&
(FSL_FEATURE_USB_KHCI_IRC48M_MODULE_CLOCK_ENABLED))
CLOCK_EnableUsbfs0Clock(kCLOCK_UsbSrcIrc48M, 48000000U);
#else
CLOCK_EnableUsbfs0Clock(kCLOCK_UsbSrcPll0, CLOCK_GetFreq(kCLOCK_PllFllSelClk));
#endif /* FSL_FEATURE_USB_KHCI_IRC48M_MODULE_CLOCK_ENABLED */
```

   for LPC IP3511,

```
CLOCK_EnableUsbfs0Clock(kCLOCK_UsbSrcFro, CLOCK_GetFreq(kCLOCK_FroHf));
```

3. Call the USB_DeviceClassInit function.

```
if (kStatus_USB_Success != USB_DeviceClassInit(CONTROLLER_ID, &g_compositeDeviceConfigList,
&g_composite.deviceHandle))
{
    usb_echo("USB device composite demo init failed\r\n");
    return;
}
else
{
    usb_echo("USB device composite demo\r\n");
    ……
}
```

4. Get a handle for each class. For example,

   CDC virtual com:

```
g_composite.cdcVcom.cdcAcmHandle = g_compositeDeviceConfigList.config[0].classHandle;
```

   MSC ramdisk:

```
g_composite.mscDisk.mscHandle = g_compositeDeviceConfigList.config[1].classHandle;
```

5. Initialize each class application.

   Such as,

   CDC virtual com:

```
USB_DeviceCdcVcomInit(&g_composite);
```

   MSC ramdisk:

```
USB_DeviceMscDiskInit(&g_composite);
```

6. Set the interrupt priority and enable the USB device interrupt

```
NVIC_SetPriority((IRQn_Type)irqNo, USB_DEVICE_INTERRUPT_PRIORITY);
NVIC_EnableIRQ((IRQn_Type)irqNo);
```

7. Enable the USB device funtionally:

```
USB_DeviceRun(g_composite.deviceHandle);
```

Chapter 8
HID keyboard + HID generic composite device example

In this section, HID keyboard + HID generic composite device are used as an example.

8.1 USB composite device structure examples

```
/* Two HID classes */
usb_device_class_config_list_struct_t g_UsbDeviceCompositeConfigList =
{
    g_CompositeClassConfig,
    USB_DeviceCallback,
    2U,
};

/* Two HID classes definition */
usb_device_class_config_struct_t g_CompositeClassConfig[2] =
 {
    {
        USB_DeviceHidKeyboardCallback,
        (class_handle_t)NULL,
        &g_UsbDeviceHidKeyboardConfig,
    },
    {
        USB_DeviceHidGenericCallback,
        (class_handle_t)NULL,
        &g_UsbDeviceHidGenericConfig,
    }
};

/* HID generic device config */
usb_device_class_struct_t g_UsbDeviceHidGenericConfig =
{
    g_UsbDeviceHidGenericInterfaceList, /* The interface list of the HID generic */
    kUSB_DeviceClassTypeHid,             /* The HID class type */
    USB_DEVICE_CONFIGURATION_COUNT,          /* The configuration count */
};

/* HID generic device interface list */
usb_device_interface_list_t g_UsbDeviceHidGenericInterfaceList[USB_DEVICE_CONFIGURATION_COUNT] =
{
    {
        USB_HID_GENERIC_INTERFACE_COUNT, /* The interface count of the HID generic */
        g_UsbDeviceHidGenericInterfaces,        /* The interfaces handle */
    },
};

/* HID generic device interfaces */
usb_device_interfaces_struct_t g_UsbDeviceHidGenericInterfaces[USB_HID_GENERIC_INTERFACE_COUNT] =
{
    USB_HID_GENERIC_CLASS,           /* HID generic class code */
    USB_HID_GENERIC_SUBCLASS,        /* HID generic subclass code */
    USB_HID_GENERIC_PROTOCOL,        /* HID generic protocol code */
```

```
    USB_HID_GENERIC_INTERFACE_INDEX, /* The interface number of the HID generic */
    g_UsbDeviceHidGenericInterface,         /* Interfaces handle */
    sizeof(g_UsbDeviceHidGenericInterface) / sizeof(usb_device_interfaces_struct_t),
};

/* HID generic device interface and alternate setting device information */
usb_device_interface_struct_t g_UsbDeviceHidGenericInterface[] =
{
    {
        0U, /* The alternate setting of the interface */
        {
            USB_HID_GENERIC_ENDPOINT_COUNT, /* Endpoint count */
            g_UsbDeviceHidGenericEndpoints,        /* Endpoints handle */
        },
    }
};

/* HID generic device endpoint information for interface USB_HID_GENERIC_INTERFACE_INDEX and
alternate setting is 0. */
usb_device_endpoint_struct_t g_UsbDeviceHidGenericEndpoints[USB_HID_GENERIC_ENDPOINT_COUNT] =
{
    /* HID generic interrupt IN pipe */
    {
        USB_HID_GENERIC_ENDPOINT_IN | (USB_IN << USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_SHIFT),
        USB_ENDPOINT_INTERRUPT,
        FS_HID_GENERIC_INTERRUPT_IN_PACKET_SIZE,
    },
    /* HID generic interrupt OUT pipe */
    {
        USB_HID_GENERIC_ENDPOINT_OUT | (USB_OUT << USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_SHIFT),
        USB_ENDPOINT_INTERRUPT,
        FS_HID_GENERIC_INTERRUPT_OUT_PACKET_SIZE,
    },
};

/* HID keyboard device config */
usb_device_class_struct_t g_UsbDeviceHidKeyboardConfig =
{
    g_UsbDeviceHidKeyboardInterfaceList, /* The interface list of the HID keyboard */
    kUSB_DeviceClassTypeHid,             /* The HID class type */
    USB_DEVICE_CONFIGURATION_COUNT,          /* The configuration count */
};

/* HID keyboard device interface list */
usb_device_interface_list_t g_UsbDeviceHidKeyboardInterfaceList[USB_DEVICE_CONFIGURATION_COUNT] =
{
    {
        USB_HID_KEYBOARD_INTERFACE_COUNT, /* The interface count of the HID keyboard */
        g_UsbDeviceHidKeyboardInterfaces,        /* The interfaces handle */
    },
};

/* HID generic device interfaces */
usb_device_interfaces_struct_t g_UsbDeviceHidKeyboardInterfaces[USB_HID_KEYBOARD_INTERFACE_COUNT] =
{
    USB_HID_KEYBOARD_CLASS,           /* HID keyboard class code */
    USB_HID_KEYBOARD_SUBCLASS,        /* HID keyboard subclass code */
    USB_HID_KEYBOARD_PROTOCOL,        /* HID keyboard protocol code */
    USB_HID_KEYBOARD_INTERFACE_INDEX, /* The interface number of the HID keyboard */
    g_UsbDeviceHidKeyboardInterface,         /* Interfaces handle */
```

```
    sizeof(g_UsbDeviceHidKeyboardInterface) / sizeof(usb_device_interfaces_struct_t),
};

/* HID generic device interface and alternate setting device information */
usb_device_interface_struct_t g_UsbDeviceHidKeyboardInterface[] =
{
    {
        0U, /* The alternate setting of the interface */
        {
            USB_HID_KEYBOARD_ENDPOINT_COUNT, /* Endpoint count */
            g_UsbDeviceHidKeyboardEndpoints,        /* Endpoints handle */
        },
    }
};

/* HID generic device endpoint information for interface USB_HID_GENERIC_INTERFACE_INDEX and
alternate setting is 0. */
usb_device_endpoint_struct_t g_UsbDeviceHidKeyboardEndpoints[USB_HID_KEYBOARD_ENDPOINT_COUNT] =
{
    /* HID keyboard interrupt IN pipe */
    {
        USB_HID_KEYBOARD_ENDPOINT_IN | (USB_IN << USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_SHIFT),
        USB_ENDPOINT_INTERRUPT,
        FS_HID_KEYBOARD_INTERRUPT_IN_PACKET_SIZE,
    },
};
```

8.2 USB composite device descriptor examples

Modify the vendor ID and product ID for the device descriptor in the “usb_device_descriptor.c” file.

Change the interface number as shown in the configuration descriptor in the “usb_device_descriptor.c” file.

Merge the HID keyboard and HID generic configuration descriptor (in the “usb_device_descriptor.c” file) from the HID mouse + HID keyboard example and hid_generic example and change the endpoint number to be consistent with Section 8.1., "*USB composite device structure examples*".

8.2.1 USB_DeviceGetDeviceDescriptor

This function is used to get the device descriptor. All devices must implement this function.

```
usb_status_t USB_DeviceGetDeviceDescriptor(usb_device_handle handle,
                                           usb_device_get_device_descriptor_struct_t
*deviceDescriptor)
{
    deviceDescriptor->buffer = g_UsbDeviceDescriptor;
    deviceDescriptor->length = USB_DESCRIPTOR_LENGTH_DEVICE;
    return kStatus_USB_Success;
}
```

8.2.2 USB_DeviceGetConfigurationDescriptor

This function is used to get the configuration descriptor. All devices must implement this function.

```
/* Get device configuration descriptor request */
usb_status_t USB_DeviceGetConfigurationDescriptor(
```

```
    usb_device_handle handle, usb_device_get_configuration_descriptor_struct_t
*configurationDescriptor)
{
    if (USB_COMPOSITE_CONFIGURE_INDEX > configurationDescriptor->configuration)
    {
        configurationDescriptor->buffer = g_UsbDeviceConfigurationDescriptor;
        configurationDescriptor->length = USB_DESCRIPTOR_LENGTH_CONFIGURATION_ALL;
        return kStatus_USB_Success;
    }
    return kStatus_USB_InvalidRequest;
}
```

8.2.3 USB_DeviceGetStringDescriptor

This function is used to get the string descriptor. All devices must implement this function.

```
/* Get device string descriptor request */
usb_status_t USB_DeviceGetStringDescriptor(usb_device_handle handle,
                                           usb_device_get_string_descriptor_struct_t
*stringDescriptor)
{
    if (stringDescriptor->stringIndex == 0U)
    {
        stringDescriptor->buffer = (uint8_t *)g_UsbDeviceLanguageList.languageString;
        stringDescriptor->length = g_UsbDeviceLanguageList.stringLength;
    }
    else
    {
        uint8_t languageId = 0U;
        uint8_t languageIndex = USB_DEVICE_STRING_COUNT;

        for (; languageId < USB_DEVICE_STRING_COUNT; languageId++)
        {
            if (stringDescriptor->languageId ==
g_UsbDeviceLanguageList.languageList[languageId].languageId)
            {
                if (stringDescriptor->stringIndex < USB_DEVICE_STRING_COUNT)
                {
                    languageIndex = stringDescriptor->stringIndex;
                }
                break;
            }
        }

        if (USB_DEVICE_STRING_COUNT == languageIndex)
        {
            return kStatus_USB_InvalidRequest;
        }
        stringDescriptor->buffer = (uint8_t
*)g_UsbDeviceLanguageList.languageList[languageId].string[languageIndex];
        stringDescriptor->length =
g_UsbDeviceLanguageList.languageList[languageId].length[languageIndex];
    }
    return kStatus_USB_Success;
}
```

8.2.4 USB_DeviceGetHidDescriptor

```
/* Get HID descriptor request */
usb_status_t USB_DeviceGetHidDescriptor(usb_device_handle handle,
                                        usb_device_get_hid_descriptor_struct_t *hidDescriptor)
{
    /* If this request is not supported, return the error code “kStatus_USB_InvalidRequest”.
Otherwise, fill the hidDescriptor with the descriptor buffer address and length based on the
interface number. */
    return kStatus_USB_InvalidRequest;
}
```

8.2.5 USB_DeviceGetHidReportDescriptor

```
/* Get the HID report descriptor request */
usb_status_t USB_DeviceGetHidReportDescriptor(usb_device_handle handle,
                                              usb_device_get_hid_report_descriptor_struct_t
*hidReportDescriptor)
{
    if (USB_HID_GENERIC_INTERFACE_INDEX == hidReportDescriptor->interfaceNumber)
    {
        hidReportDescriptor->buffer = g_UsbDeviceHidGenericReportDescriptor;
        hidReportDescriptor->length = USB_DESCRIPTOR_LENGTH_HID_GENERIC_REPORT;
    }
    else if (USB_HID_KEYBOARD_INTERFACE_INDEX == hidReportDescriptor->interfaceNumber)
    {
        hidReportDescriptor->buffer = g_UsbDeviceHidKeyboardReportDescriptor;
        hidReportDescriptor->length = USB_DESCRIPTOR_LENGTH_HID_KEYBOARD_REPORT;
    }
    else
    {
        return kStatus_USB_InvalidRequest;
    }
    return kStatus_USB_Success;
}
```

8.2.6 USB_DeviceGetHidPhysicalDescriptor

```
/* Get the HID physical descriptor request */
usb_status_t USB_DeviceGetHidPhysicalDescriptor(
    usb_device_handle handle, usb_device_get_hid_physical_descriptor_struct_t *hidPhysicalDescriptor)
{
    /* If this request is not supported, return the error code “kStatus_USB_InvalidRequest”.
Otherwise, fill the hidPhysicalDescriptor with the descriptor buffer address and length based on the
interface number and the physical index. */
    return kStatus_USB_InvalidRequest;
}
```

8.2.7 USB_DeviceSetSpeed

```
/* Because HS and FS descriptors are different, update the device descriptors and configurations to
match the current speed.
 * By default, the device descriptors and configurations are configured by using FS parameters for
both EHCI and KHCI.
 * When the EHCI is enabled, the application needs to call this function to update the device by
```

```
using current speed.
 * The updated information includes the endpoint max packet size, endpoint interval, and so on. */
usb_status_t USB_DeviceSetSpeed(usb_device_handle handle, uint8_t speed)
{
    usb_descriptor_union_t *descriptorHead;
    usb_descriptor_union_t *descriptorTail;

    descriptorHead = (usb_descriptor_union_t *)&g_UsbDeviceConfigurationDescriptor[0];
    descriptorTail = (usb_descriptor_union_t *)
(&g_UsbDeviceConfigurationDescriptor[USB_DESCRIPTOR_LENGTH_CONFIGURATION_ALL - 1U]);

    while (descriptorHead < descriptorTail)
    {
        if (descriptorHead->common.bDescriptorType == USB_DESCRIPTOR_TYPE_ENDPOINT)
        {
            if (USB_SPEED_HIGH == speed)
            {
                if (USB_HID_KEYBOARD_ENDPOINT_IN == (descriptorHead->endpoint.bEndpointAddress &
USB_ENDPOINT_NUMBER_MASK))
                {
                    descriptorHead->endpoint.bInterval = HS_HID_KEYBOARD_INTERRUPT_IN_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(HS_HID_KEYBOARD_INTERRUPT_IN_PACKET_SIZE,
                                                       descriptorHead->endpoint.wMaxPacketSize);
                }
                else if (((descriptorHead->endpoint.bEndpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN) ==
                          USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN) &&
                         (USB_HID_GENERIC_ENDPOINT_IN == (descriptorHead->endpoint.bEndpointAddress &
USB_ENDPOINT_NUMBER_MASK)))
                {
                    descriptorHead->endpoint.bInterval = HS_HID_GENERIC_INTERRUPT_IN_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(HS_HID_GENERIC_INTERRUPT_IN_PACKET_SIZE,
                                                       descriptorHead->endpoint.wMaxPacketSize);
                }
                else if (((descriptorHead->endpoint.bEndpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_OUT) ==
                          USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_OUT) &&
                         (USB_HID_GENERIC_ENDPOINT_OUT == (descriptorHead->endpoint.bEndpointAddress
& USB_ENDPOINT_NUMBER_MASK)))
                {
                    descriptorHead->endpoint.bInterval = HS_HID_GENERIC_INTERRUPT_OUT_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(HS_HID_GENERIC_INTERRUPT_OUT_PACKET_SIZE,
                                                       descriptorHead->endpoint.wMaxPacketSize);
                }
            }
            else
            {
                if (USB_HID_KEYBOARD_ENDPOINT_IN == (descriptorHead->endpoint.bEndpointAddress &
USB_ENDPOINT_NUMBER_MASK))
                {
                    descriptorHead->endpoint.bInterval = FS_HID_KEYBOARD_INTERRUPT_IN_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(FS_HID_KEYBOARD_INTERRUPT_IN_PACKET_SIZE,
                                                       descriptorHead->endpoint.wMaxPacketSize);
                }
                else if (((descriptorHead->endpoint.bEndpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN) ==
                          USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN) &&
                         (USB_HID_GENERIC_ENDPOINT_IN == (descriptorHead->endpoint.bEndpointAddress &
USB_ENDPOINT_NUMBER_MASK)))
                {
```

```
                    descriptorHead->endpoint.bInterval = FS_HID_GENERIC_INTERRUPT_IN_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(FS_HID_GENERIC_INTERRUPT_IN_PACKET_SIZE,
                                                       descriptorHead->endpoint.wMaxPacketSize);
                }
                else if (((descriptorHead->endpoint.bEndpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_OUT) ==
                          USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_OUT) &&
                         (USB_HID_GENERIC_ENDPOINT_OUT == (descriptorHead->endpoint.bEndpointAddress
& USB_ENDPOINT_NUMBER_MASK)))
                {
                    descriptorHead->endpoint.bInterval = FS_HID_GENERIC_INTERRUPT_OUT_INTERVAL;
                    USB_SHORT_TO_LITTLE_ENDIAN_ADDRESS(FS_HID_GENERIC_INTERRUPT_OUT_PACKET_SIZE,
                                                       descriptorHead->endpoint.wMaxPacketSize);
                }
            }
        }
        descriptorHead = (usb_descriptor_union_t *)((uint8_t *)descriptorHead + descriptorHead-
>common.bLength);
    }

    for (int i = 0U; i < USB_HID_GENERIC_ENDPOINT_COUNT; i++)
    {
        if (USB_SPEED_HIGH == speed)
        {
            if (g_UsbDeviceHidGenericEndpoints[i].endpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN)
            {
                g_UsbDeviceHidGenericEndpoints[i].maxPacketSize =
HS_HID_GENERIC_INTERRUPT_IN_PACKET_SIZE;
            }
            else
            {
                g_UsbDeviceHidGenericEndpoints[i].maxPacketSize =
HS_HID_GENERIC_INTERRUPT_OUT_PACKET_SIZE;
            }
        }
        else
        {
            if (g_UsbDeviceHidGenericEndpoints[i].endpointAddress &
USB_DESCRIPTOR_ENDPOINT_ADDRESS_DIRECTION_IN)
            {
                g_UsbDeviceHidGenericEndpoints[i].maxPacketSize =
HS_HID_GENERIC_INTERRUPT_OUT_PACKET_SIZE;
            }
            else
            {
                g_UsbDeviceHidGenericEndpoints[i].maxPacketSize =
FS_HID_GENERIC_INTERRUPT_OUT_PACKET_SIZE;
            }
        }
    }

    if (USB_SPEED_HIGH == speed)
    {
        g_UsbDeviceHidKeyboardEndpoints[0].maxPacketSize = HS_HID_KEYBOARD_INTERRUPT_IN_PACKET_SIZE;
    }
    else
    {
        g_UsbDeviceHidKeyboardEndpoints[0].maxPacketSize = FS_HID_KEYBOARD_INTERRUPT_IN_PACKET_SIZE;
    }
```

```
    return kStatus_USB_Success;
}
```

8.3 USB composite device application example

8.3.1 Class configuration

USB_DEVICE_CONFIG_HID is set to 2 in usb_device_config.h

USB_DEVICE_CONFIG_ENDPOINTS is set to 4 in usb_device_config.h

8.3.2 HID + HID Application structure

```
typedef struct _usb_device_composite_struct
{
    usb_device_handle                  deviceHandle;
    class_handle_t                     hidKeyboardHandle;
    class_handle_t                     hidGenericHandle;
    uint8_t                            speed;
    uint8_t                            attach;
    uint8_t                            currentConfiguration;
    uint8_t
currentInterfaceAlternateSetting[USB_COMPOSITE_INTERFACE_COUNT];
} usb_device_composite_struct_t;

/* HID keyboard structure */
typedef struct _usb_device_hid_keyboard_struct
{
    uint8_t                   buffer[USB_HID_KEYBOARD_IN_BUFFER_LENGTH];
    uint8_t                   idleRate;
} usb_device_hid_keyboard_struct_t;

/* HID generic structure */
typedef struct _usb_device_hid_generic_struct
{
    uint32_t                  buffer[2][USB_HID_GENERIC_IN_BUFFER_LENGTH>>2];
    uint8_t                   bufferIndex;
    uint8_t                   idleRate;
} usb_device_hid_generic_struct_t;
```

8.3.3 HID + HID application

1. Define and initialize the configuration structure.

```
static usb_device_composite_struct_t g_UsbDeviceComposite;
usb_device_class_struct_t g_UsbDeviceHidGenericConfig;
usb_device_class_struct_t g_UsbDeviceHidKeyboardConfig;

usb_device_class_config_struct_t g_CompositeClassConfig[2] =
{
    {
        USB_DeviceHidKeyboardCallback,
```

```
        (class_handle_t)NULL,
        &g_UsbDeviceHidKeyboardConfig,
    },
    {
        USB_DeviceHidGenericCallback,
        (class_handle_t)NULL,
        &g_UsbDeviceHidGenericConfig,
    }
};

usb_device_class_config_list_struct_t g_UsbDeviceCompositeConfigList =
{
    g_CompositeClassConfig,
    USB_DeviceCallback,
    2U,
};
```

2. Add USB ISR.

```
#if defined(USB_DEVICE_CONFIG_EHCI) && (USB_DEVICE_CONFIG_EHCI > 0U)
void USBHS_IRQHandler(void) { USB_DeviceEhciIsrFunction(g_UsbDeviceComposite.deviceHandle); }
#endif
#if defined(USB_DEVICE_CONFIG_KHCI) && (USB_DEVICE_CONFIG_KHCI > 0U)
void USB0_IRQHandler(void) { USB_DeviceKhciIsrFunction(g_UsbDeviceComposite.deviceHandle); }
#endif
#if defined(USB_DEVICE_CONFIG_LPC3511IP) && (USB_DEVICE_CONFIG_LPC3511IP > 0U)
void USB0_IRQHandler(void)
{
    USB_DeviceLpc3511IpIsrFunction(g_UsbDeviceHidMouse.deviceHandle);
}
#endif
```

3. Enable the USB device clock.

```
#if defined(USB_DEVICE_CONFIG_EHCI) && (USB_DEVICE_CONFIG_EHCI > 0U)
    CLOCK_EnableUsbhs0Clock(kCLOCK_UsbSrcPll0, CLOCK_GetFreq(kCLOCK_PllFllSelClk));
    USB_EhciPhyInit(CONTROLLER_ID, BOARD_XTAL0_CLK_HZ);
#endif
#if defined(USB_DEVICE_CONFIG_KHCI) && (USB_DEVICE_CONFIG_KHCI > 0U)
#if ((defined FSL_FEATURE_USB_KHCI_IRC48M_MODULE_CLOCK_ENABLED) &&
(FSL_FEATURE_USB_KHCI_IRC48M_MODULE_CLOCK_ENABLED))
    CLOCK_EnableUsbfs0Clock(kCLOCK_UsbSrcIrc48M, 48000000U);
#else
    CLOCK_EnableUsbfs0Clock(kCLOCK_UsbSrcPll0, CLOCK_GetFreq(kCLOCK_PllFllSelClk));
#endif /* FSL_FEATURE_USB_KHCI_IRC48M_MODULE_CLOCK_ENABLED */
#endif
#if defined(USB_DEVICE_CONFIG_LPC3511IP) && (USB_DEVICE_CONFIG_LPC3511IP > 0U)
    CLOCK_EnableUsbfs0Clock(kCLOCK_UsbSrcFro, CLOCK_GetFreq(kCLOCK_FroHf));
#endif
```

4. Set the default state.

```
    g_UsbDeviceComposite.speed = USB_SPEED_FULL;
    g_UsbDeviceComposite.attach = 0U;
    g_UsbDeviceComposite.hidGenericHandle = (class_handle_t)NULL;
    g_UsbDeviceComposite.hidKeyboardHandle = (class_handle_t)NULL;
    g_UsbDeviceComposite.deviceHandle = NULL;
```

5. Initialize the USB device.

```
    if (kStatus_USB_Success !=
        USB_DeviceClassInit(CONTROLLER_ID, &g_UsbDeviceCompositeConfigList,
&g_UsbDeviceComposite.deviceHandle))
    {
        usb_echo("USB device composite demo init failed\r\n");
        return;
    }
    else
    {
        usb_echo("USB device composite demo\r\n");
        …
    }
```

6. Save each class handle when the device is initialized successfully.

```
        /* Get the HID keyboard class handle */
        g_UsbDeviceComposite.hidKeyboardHandle =
g_UsbDeviceCompositeConfigList.config[0].classHandle;
        /* Get the HID generic class handle */
        g_UsbDeviceComposite.hidGenericHandle =
g_UsbDeviceCompositeConfigList.config[1].classHandle;
```

7. Initialize the HID keyboard and HID generic application.

```
        USB_DeviceHidKeyboardInit(&g_UsbDeviceComposite);
        USB_DeviceHidGenericInit(&g_UsbDeviceComposite);
```

8. Set the device ISR priority and enable the device interrupt.

```
    NVIC_SetPriority((IRQn_Type)irqNumber, USB_DEVICE_INTERRUPT_PRIORITY);
    NVIC_EnableIRQ((IRQn_Type)irqNumber);
```

9. Start the device functionality.

```
USB_DeviceRun(g_UsbDeviceComposite.deviceHandle);
```

10. Poll the device task when the "USB_DEVICE_CONFIG_USE_TASK" is non-zero. Poll the HID keyboard and HID generic task when these tasks are implemented.

```
#if USB_DEVICE_CONFIG_USE_TASK
#if defined(USB_DEVICE_CONFIG_EHCI) && (USB_DEVICE_CONFIG_EHCI > 0U)
        USB_DeviceEhciTaskFunction(g_UsbDeviceComposite.deviceHandle);
#endif
#if defined(USB_DEVICE_CONFIG_KHCI) && (USB_DEVICE_CONFIG_KHCI > 0U)
        USB_DeviceKhciTaskFunction(g_UsbDeviceComposite.deviceHandle);
#endif
#if defined(USB_DEVICE_CONFIG_LPC3511IP) && (USB_DEVICE_CONFIG_LPC3511IP > 0U)
        USB_DeviceLpc3511IpTaskFunction(g_UsbDeviceHidMouse.deviceHandle);
#endif
#endif
```

Chapter 9
Revision history

This table summarizes revisions to this document.

Table 1. Revision history

Revision number	Date	Substantive changes
0	12/2014	Initial release
1	04/2015	Substantive changes
2	09/2015	Section 5.3, Section 6, Section 8.2.2, Section 8.3.1
3	11/2015	Updated for KV5x release
4	01/2016	Updated Section 1
5	09/2016	Added LPC content for release
6	03/2017	Updates for MCUXpresso SDK release
7	11/2017	Updates for MCUXpresso SDK 2.3.0 release
8	05/2018	Updated Section 4.5, "usb_device_interfaces_struct_t", for MCUXpresso SDK 2.4.0 release
9	12/2018	Updated Section 8.3, "USB composite device application example" for MCUXpresso SDK 2.5.0
10	06/2019	Updated 'Overview' section for MCUXpresso SDK 2.6.0

How To Reach Us

Home Page:

www.nxp.com

Web Support:

www.nxp.com/support

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: June 2019
Document identifier: MCUXSDKUSBCOMDUG

arm