**Freescale Semiconductor** Document Number: USBSDNAUG
User's Guide Rev. 1, 04/2015

# Freescale KSDK USB Stack Developing New Application User's Guide

## 1 Read Me First

This document provides the detailed steps to develop a new application based on the existing classes in the USB Unified Stack. There are two parts in this document:

- Developing a new USB device application
- Developing a new USB host application

**Contents**

# 2 Developing a New USB Device Application

## 2.1 Application interfaces

The interface definition between the application and the classes includes the calls shown in the following table:

| API Call | Description |
|---|---|
| Class Initialize | This API is used to initialize not only the class but also the lower driver layers. |
| Receive Data | This API is used by the application to receive the data from the host system. |
| Send Data | This API is used by the application to send the data to the host system. |
| USB descriptor related callback | Handles the callback to get the descriptor. |
| USB Device call back function | Handles the callback by the class driver to inform the application about various USB bus events. |
| USB Class specific call back function | Handles the specific callback of the class. |
| USB Vendor call back function | This is an optional callback and is not mandatory for the application to support it. This callback is used to propagate any vendor specific request that the host system sends. |
| Periodic Task | This is an API call by the application to the class, so that it can complete some tasks that it may want to execute in non-interrupt context. |

## 2.2 How to develop a new device application

Perform these steps to develop a new device application:

1. Create a new application directory under `…/usb/example/device/xxx` to `<install_dir>/usb/example/device/xxx` locate the application source files and header files. The xxx is the class name, such as HID and CDC. The name of this directory is the same as the class name that the application is based on, such as HID and CDC. For example,

   `<install_dir>/usb/example/device/hid/hid_test`

2. Copy the following files from the similar existing applications to the application directory that is created in Step 1.

   `usb_descriptor.c`

`usb_descriptor.h`

The `usb_descriptor.c` and `usb_descriptor.h` files contain the USB descriptors that are dependent on the application and the class driver.

3. Copy the bm directory from the similar existing application directory to the new application directory. Remove the unused project directory from the bm directory. Modify the project directory name to the new application project name. For example, if we want to create `toolchain-IAR, board-frdmk64 class-hid related application`, then we can create the new application `hid_test` based on a similar existing application `hid_mouse`.

   Change `<install_dir>/examples/frdmk64f/demo_apps/usb/device/hid/hid_mouse/bm/iar` to `<install_dir>/examples/frdmk64f/demo_apps/usb/device/hid/hid_test/bm/iar`

4. Modify the project file name to the new application project file name, for example, from `dev_hid_mouse_frdmk64f_bm.ewp` to `dev_hid_test_frdmk64f.ewp`. You can globally replace the existing name to the new project name by editing the project files. The `dev_hid_test_frdmk64f_bm.ewp` file includes the new application project setting.
5. Create a new source file to implement the main application functions and callback functions. The name of this file is similar with the new application name, such as `mouse.c` and `keyboard.c`.

The following sections describe the detailed steps to change application files created in the steps above to correspond with the new application.

### 2.2.1 Changing the usb_descriptor.c file

This file contains the class driver interface. It also contains USB standard descriptors such as device descriptor, configuration descriptor, string descriptor, and the other class specific descriptors that are provided to class driver when required.

The lists below show user modifiable variables for an already implemented class driver. The user should also modify the corresponding MACROs defined in the `usb_descriptor.h` file

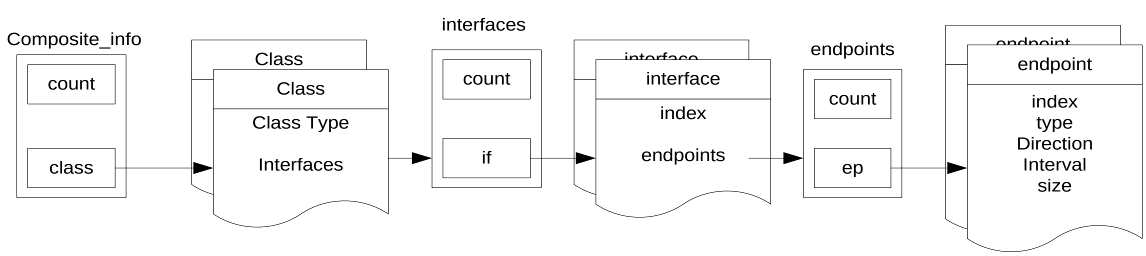
- Endpoint structures: Endpoint structure describes the property of endpoint such as the endpoint number, size, direction, and type. This array should contain all the mandatory endpoints defined by USB class specifications.

  Data structure of endpoint descriptor:

```
typedef struct _usb_ep_struct
{
    uint8_t              ep_num;      /* endpoint number         */
    uint8_t              type;        /* type of endpoint        */
    uint8_t              direction;   /* direction of endpoint   */
    uint32_t             size;        /* buffer size of endpoint */
} usb_ep_struct_t;

/* Strucutre Representing Endpoints and number of endpoints user want*/
typedef struct _usb_endpoints
{
```

```
    uint8_t                 count;
    usb_ep_struct_t*        ep;
} usb_endpoints_t;

/* Strucutre Representing interface*/
typedef struct _usb_if_struct
{
    uint8_t                 index;
    usb_endpoints_t         endpoints;
} usb_if_struct_t;

/* Strucutre Representing how many interfaces in one class type*/
typedef struct _usb_interfaces_struct
{
    uint8_t                 count;
    usb_if_struct_t*        interface;
} usb_interfaces_struct_t;

/* Strucutre Representing class info*/
typedef struct _usb_class_struct
{
    class_type                type;
    usb_interfaces_struct_t interfaces;
} usb_class_struct_t;

/* Strucutre Representing composite info*/
typedef struct _usb_composite_info_struct
{
    uint8_t                 count;
    usb_class_struct_t*     class;
} usb_composite_info_struct_t;
```

A brief diagram about the relationship between these items is as follows:




Sample code implementation of endpoint descriptor for HID class is given below:

```
usb_ep_struct_t g_ep[HID_DESC_ENDPOINT_COUNT] =
{
```

```
    HID_ENDPOINT,
    USB_INTERRUPT_PIPE,
    USB_SEND,
    HID_ENDPOINT_PACKET_SIZE
};

/* structure containing details of all the endpoints used by this device */
usb_endpoints_t g_usb_desc_ep =
{
    HID_DESC_ENDPOINT_COUNT,
    g_ep
};

static usb_if_struct_t g_usb_if[1];

usb_class_struct_t g_usb_dec_class =
{
    USB_CLASS_HID,
        {
            1,
           g_usb_if
        }
 };
```

- g_device_descriptor

  This variable contains the USB Device Descriptor.

  Sample code implementation of device descriptor for HID class is given below:

```
uint8_t  g_device_descriptor[DEVICE_DESCRIPTOR_SIZE] =
{
    DEVICE_DESCRIPTOR_SIZE,               /*  Device Descriptor Size        */
    USB_DEVICE_DESCRIPTOR,                /*  Device  Type of descriptor    */
    0x00, 0x02,                           /*  BCD USB version               */
    0x00,                                 /*  Device Class is indicated in
                                              the interface descriptors     */
    0x00,                                 /*  Device Subclass is indicated
                                              in the interface descriptors  */
    0x00,                                 /*  Device Protocol               */
    CONTROL_MAX_PACKET_SIZE,              /*  Max Packet size               */
    0xA2,0x15,                            /*  Vendor ID                     */
    0x01,0x01,                            /*  Product ID (0x0101 for KBD)   */
    0x02,0x00,                            /*  BCD Device version            */
    0x01,                                 /*  Manufacturer string index     */
    0x02,                                 /*  Product string index          */
    0x00,                                 /*  Serial number string index    */
```

```
    0x01                                  /*  Number of configurations      */
};
```

- g_config_descriptor

  This variable contains the USB Configuration Descriptor.

  Sample code implementation of configuration descriptor for HID class is given below:

```
uint8_t  g_config_descriptor[CONFIG_DESC_SIZE] =
{
    CONFIG_ONLY_DESC_SIZE,  /*  Configuration Descriptor Size - always 9 bytes*/
    USB_CONFIG_DESCRIPTOR,  /* "Configuration" type of descriptor */
    CONFIG_DESC_SIZE, 0x00, /*  Total length of the Configuration descriptor */
    1,                      /*  NumInterfaces */
    1,                      /*  Configuration Value */
    0,                      /*  Configuration Description String Index*/
    (USBCFG_DEV_SELF_POWER << USB_DESC_CFG_ATTRIBUTES_SELF_POWERED_SHIFT) |
(USBCFG_DEV_REMOTE_WAKEUP << USB_DESC_CFG_ATTRIBUTES_REMOTE_WAKEUP_SHIFT),
    /* S08/CFv1 are both self powered (its compulsory to set bus powered)*/
    /* Attributes.supportRemoteWakeup and self power */
    0x32,                   /*  Current draw from bus */

    /* Interface Descriptor */
    IFACE_ONLY_DESC_SIZE,
    USB_IFACE_DESCRIPTOR,
    0x00,
    0x00,
    HID_DESC_ENDPOINT_COUNT,
    0x03,
    0x01,
    0x01, /* 0x01 for keyboard */
    0x00,

    /* HID descriptor */
    HID_ONLY_DESC_SIZE,
    USB_HID_DESCRIPTOR,
    0x00,0x01,
    0x00,
    0x01,
    0x22,
    0x3F,0x00, /* report descriptor size to follow */

    /*Endpoint descriptor */
    ENDP_ONLY_DESC_SIZE,
    USB_ENDPOINT_DESCRIPTOR,
```

```
    HID_ENDPOINT|(USB_SEND << 7),
    USB_INTERRUPT_PIPE,
    HID_ENDPOINT_PACKET_SIZE, 0x00,
    0x0A
};
```

- String Descriptors

  Users can modify string descriptors to customize their product. String descriptors are written in the UNICODE format. An appropriate language identification number is specified in USB_STR_0. Multiple languages support can also be added.

  Sample code implementation of string descriptors for the HID class application is given below:

```
/* number of strings in the table not including 0 or n. */
uint8_t g_usb_str_0[USB_STR_0_SIZE+USB_STR_DESC_SIZE] =
{
    sizeof(g_usb_str_0),
    USB_STRING_DESCRIPTOR,
    0x09,
    0x04/*equivalent to 0x0409*/
};

uint8_t g_usb_str_1[USB_STR_1_SIZE+USB_STR_DESC_SIZE] =
{
    sizeof(g_usb_str_1),
    USB_STRING_DESCRIPTOR,
    'F',0,
    'R',0,
    'E',0,
    'E',0,
    'S',0,
    'C',0,
    'A',0,
    'L',0,
    'E',0,
    ' ',0,
    'S',0,
    'E',0,
    'M',0,
    'I',0,
    'C',0,
    'O',0,
    'N',0,
    'D',0,
    'U',0,
    'C',0,
```

```
    'T',0,
    'O',0,
    'R',0,
    ' ',0,
    'I',0,
    'N',0,
    'C',0,
    '.',0
};

uint8_t g_usb_str_2[USB_STR_2_SIZE+USB_STR_DESC_SIZE] =
{
    sizeof(g_usb_str_2),
    USB_STRING_DESCRIPTOR,
    'M',0,
    'C',0,
    'U',0,
    ' ',0,
    'K',0,
    'E',0,
    'Y',0,
    'B',0,
    'O',0,
    'A',0,
    'R',0,
    'D',0,
    ' ',0,
    'D',0,
    'E',0,
    'M',0,
    'O',0,
    ' ',0
};

uint8_t g_usb_str_n[USB_STR_n_SIZE+USB_STR_DESC_SIZE] =
{
    sizeof(g_usb_str_n),
    USB_STRING_DESCRIPTOR,
    'B',0,
    'A',0,
    'D',0,
    ' ',0,
    'S',0,
    'T',0,
```

```
    'R',0,
    'I',0,
    'N',0,
    'G',0,
    ' ',0,
    'I',0,
    'N',0,
    'D',0,
    'E',0,
    'X',0
};
uint8_t g_string_desc_size[USB_MAX_STRING_DESCRIPTORS+1] =
{
    sizeof(g_usb_str_0),
    sizeof(g_usb_str_1),
    sizeof(g_usb_str_2),
    sizeof(g_usb_str_n)
};

uint8_t *g_string_descriptors[USB_MAX_STRING_DESCRIPTORS+1] =
{
    g_usb_str_0,
    g_usb_str_1,
    g_usb_str_2,
g_usb_str_n
};

usb_language_t g_usb_language[USB_MAX_SUPPORTED_INTERFACES] =
{
    (uint16_t)0x0409,
    g_string_descriptors,
    g_string_desc_size
};

usb_all_languages_t g_languages =
{
    g_usb_str_0,
    sizeof(g_usb_str_0),
    USB_MAX_LANGUAGES_SUPPORTED,
    g_usb_language
};
```

- Standard Descriptor Table

Users can modify the standard descriptor table to support additional class specific descriptors and vendor specific descriptors.

Sample implementation for HID Class application is given below:

```
uint32_t g_std_desc_size[USB_MAX_STD_DESCRIPTORS+1] =
{
    0,
    DEVICE_DESCRIPTOR_SIZE,
    CONFIG_DESC_SIZE,
    0, /* string */
    0, /* Interfdace */
    0, /* Endpoint */
    #if HIGH_SPEED_DEVICE
        DEVICE_QUALIFIER_DESCRIPTOR_SIZE,
        OTHER_SPEED_CONFIG_DESCRIPTOR_SIZE,
    #else
        0, /* Device Qualifier */
        0, /* other sppedconfig */
    #endif
    REPORT_DESC_SIZE
};

uint8_t *g_std_descriptors[USB_MAX_STD_DESCRIPTORS+1] =
{
    NULL,
    g_device_descriptor,
    g_config_descriptor,
    NULL, /* string */
    NULL, /* Interfdace */
    NULL, /* Endpoint */
    #if HIGH_SPEED_DEVICE
        g_device_qualifier_descriptor,
        g_other_speed_config_descriptor,
    #else
        NULL, /* Device Qualifier */
        NULL, /* other sppedconfig*/
    #endif
    g_report_descriptor
};
```

- g_valid_config_values

  This variable contains valid configurations for a device. This value remains fixed for a device.

  ```
  uint_8 constg_valid_config_values[USB_MAX_CONFIG_SUPPORTED+1]={0,1};
  ```

- g_alternate_interface

This variable contains valid alternate interfaces for a given configuration. Sample implementation uses a single configuration. If the user implements additional alternate interfaces, the USB_MAX_SUPPORTED_INTERFACES macro (usb_descriptor.h) should be changed accordingly.

```
static uint_8 g_alternate_interface[USB_MAX_SUPPORTED_INTERFACES];
```

The following interfaces are required to be implemented by the application in `usb_descriptor.c`. These interfaces are called by class drivers.

- USB_Desc_Get_Descriptor

  This interface function is invoked by the Class driver. This call is made when the Class driver receives the GET_DESCRIPTOR call from the Host. Mandatory descriptors that an application is required to implement are as follows:

  - Device Descriptor
  - Configuration Descriptor
  - Class Specific Descriptors (For example, for HID class implementation, Report Descriptor, and HID Descriptor)

  Apart from the mandatory descriptors, an application should also implement various string descriptors as specified by the Device Descriptor and other configuration descriptors.

  Sample code for HID class application is given below:

```
/**************************************************************************//*!
 *
 * @name  USB_Desc_Get_Descriptor
 *
 * @brief The function returns the correponding descriptor
 *
 * @param handle:        handle
 * @param type          : type of descriptor requested
 * @param sub_type      : string index for string descriptor
 * @param index         : string descriptor language Id
 * @param descriptor    : output descriptor pointer
 * @param size          : size of descriptor returned
 *
 * @return USB_OK                      When Successfull
 *         USBERR_INVALID_REQ_TYPE     when Error
 *****************************************************************************/
uint8_t USB_Desc_Get_Descriptor
(
    hid_handle handle,
    uint8_t type,
    uint8_t str_num,
    uint16_t index,
    uint8_t * *descriptor,
    uint32_t *size
```

```
)
{
    UNUSED_ARGUMENT (handle)

    switch(type)
    {
      case USB_REPORT_DESCRIPTOR:
        {
          type = USB_MAX_STD_DESCRIPTORS;
          *descriptor = (uint8_t *)g_std_descriptors [type];
          *size = g_std_desc_size[type];
        }
        break;
      case USB_HID_DESCRIPTOR:
        {
          type = USB_CONFIG_DESCRIPTOR ;
          *descriptor = (uint8_t *)(g_std_descriptors [type]+
                               CONFIG_ONLY_DESC_SIZE+IFACE_ONLY_DESC_SIZE);
          *size = HID_ONLY_DESC_SIZE;
        }
        break;
      case USB_STRING_DESCRIPTOR:
        {
            if(index == 0)
            {
                /* return the string and size of all languages */
                    *descriptor =
                        (uint8_t *)g_languages.languages_supported_string;
                *size = g_languages.languages_supported_size;
                }
                else
            {
                uint8_t lang_id=0;
                uint8_t lang_index=USB_MAX_LANGUAGES_SUPPORTED;

                for(;lang_id< USB_MAX_LANGUAGES_SUPPORTED;lang_id++)
                {
                    /* check whether we have a string for this language */
                        if(index ==
                            g_languages.usb_language[lang_id].language_id)
                    {   /* check for max descriptors */
                        if(str_num < USB_MAX_STRING_DESCRIPTORS)
                        {   /* setup index for the string to be returned */
                            lang_index=str_num;
                        }
                        break;
                    }
                }
```

```
                    /* set return val for descriptor and size */
                    *descriptor = (uint8_t *)
                        g_languages.usb_language[lang_id].lang_desc[lang_index];
                    *size =
                        g_languages.usb_language[lang_id].
                                            lang_desc_size[lang_index];
                }
            }
            break;
          default :
            if (type < USB_MAX_STD_DESCRIPTORS)
            {
                /* set return val for descriptor and size*/
                *descriptor = (uint8_t *)g_std_descriptors [type];

                /* if there is no descriptor then return error */
                    *size = g_std_desc_size[type];

                if(*descriptor == NULL)
                {
                    return USBERR_INVALID_REQ_TYPE;
                }
            }
            else /* invalid descriptor */
            {
                return USBERR_INVALID_REQ_TYPE;
            }
            break;
    }/* End Switch */
    return USB_OK;
}
```

- USB_Desc_Get_Interface

  This interface function is invoked by the Class driver. This function returns a pointer to the alternate interface for the specified interface. This routine is called when the Class driver receives the GET_INTERFACE request from the Host.

  Sample code for the HID class application is given below:

```
/**************************************************************************//*!
 *
 * @name  USB_Desc_Get_Interface
 *
 * @brief The function returns the alternate interface
 *
 * @param handle:         handle
 * @param interface:      interface number
 * @param alt_interface:  output alternate interface
```

```
 *
 * @return USB_OK                     When Successfull
 *         USBERR_INVALID_REQ_TYPE    when Error
 *****************************************************************************/
uint8_t USB_Desc_Get_Interface
(
    hid_handle handle,
    uint8_t interface,
    uint8_t * alt_interface
)
{
    UNUSED_ARGUMENT (handle)
    /* if interface valid */
    if(interface < USB_MAX_SUPPORTED_INTERFACES)
    {
        /* get alternate interface*/
        *alt_interface = g_alternate_interface[interface];
        return USB_OK;
    }
    return USBERR_INVALID_REQ_TYPE;
}
```

- USB_Desc_Set_Interface

  This interface function is called from the Class driver. This function sets an alternate interface for a specified interface. This routine is called when the Class driver receives the SET_INTERFACE request from the host.

  Sample code for the HID class application is given below:

```
uint8_t USB_Desc_Set_Interface
(
    hid_handle handle,
    uint8_t interface,
    uint8_t alt_interface
)
{
    UNUSED_ARGUMENT (handle)
    /* if interface valid */
    if(interface < USB_MAX_SUPPORTED_INTERFACES)
    {
        /* set alternate interface*/
        g_alternate_interface[interface]=alt_interface;
        return USB_OK;
    }
```

```
    return USBERR_INVALID_REQ_TYPE;
}
```

- USB_Set_Configation

  This function is used to set the device configuration.

  Sample code for the HID class application is given below:

```
uint8_t USB_Set_Configation
(
    hid_handle handle, uint8_t config
)
{
    UNUSED_ARGUMENT(handle)

    return USB_OK;
}
```

- USB_Desc_Get_Entity

  This function is used to get some descriptor related entities which will be used in the USB stack. For example, for the eventual endpoint configuration, one device may have several different configuration descriptors, so the endpoint configuration may be different. The finalized configuration can be obtained only after the Host calls the SET_CONFIGURATION, and then the USB stack can get all the correct information about the interfaces and endpoints through the USB_CLASS_INFO selector.

  Sample code for the HID class application is given below:

```
uint8_t USB_Desc_Get_Entity(hid_handle handle,entity_type type, uint32_t * object)
{

    switch (type)
    {
        case USB_CLASS_INFO:
            g_usb_if[0].index = 1;
            g_usb_if[0].endpoints = g_usb_desc_ep;
            *object = (unsigned long)&g_usb_dec_class;
            break;
        default :
            break;
    }/* End Switch */
    return USB_OK;
}
```

### 2.2.2 Changing the usb_descriptor.h file

This file is mandatory for the application to implement. The usb_descriptor.c file includes this file for function prototype definitions. When the user modifies usb_descriptor.c, MACROs in this file should also be modified.

### 2.2.3 Changing the application file

1. Main application function

   The main application function is provided by two functions: APP_init and APP_task.

   Sample code for the HID class application is given below:

```
void APP_init(void)
{
    hid_config_struct_t config_struct;
    OS_Mem_zero(&g_keyboard,sizeof(keyboard_global_variable_struct_t));
    OS_Mem_zero(&config_struct,sizeof(hid_config_struct_t));

#if (OS_ADAPTER_ACTIVE_OS == OS_ADAPTER_MQX)
    g_keyboard.rpt_buf = (uint8_t*)OS_Mem_alloc_uncached_align(KEYBOARD_BUFF_SIZE,
32);
    if(NULL == g_keyboard.rpt_buf)
    {
        printf("\nMalloc error in APP_init\n");
        return;
    }
    OS_Mem_zero(g_keyboard.rpt_buf,KEYBOARD_BUFF_SIZE);
#endif
    printf("\nbegin to test keyboard\n");
    config_struct.hid_application_callback.callback = USB_App_Callback;
    config_struct.hid_application_callback.arg = &g_keyboard.app_handle;
    config_struct.class_specific_callback.callback = USB_App_Param_Callback;
    config_struct.class_specific_callback.arg = &g_keyboard.app_handle;
    config_struct.desc_callback_ptr = &g_desc_callback;

    USB_Class_HID_Init(CONTROLLER_ID, &config_struct, &g_keyboard.app_handle);
}

void APP_task()
{
    USB_HID_Periodic_Task();
}
```

2. USB device call back function

   Sample code for the HID class application is given below:

```
void USB_App_Callback(uint8_t event_type, void* val,void* arg)
{
    UNUSED_ARGUMENT (arg)
    UNUSED_ARGUMENT (val)

    switch(event_type)
{
        case USB_DEV_EVENT_BUS_RESET:
            g_keyboard.keyboard_init = FALSE;
            break;
        case USB_DEV_EVENT_ENUM_COMPLETE:
            g_keyboard.keyboard_init = TRUE;
            g_process_times = 1;
            KeyBoard_Events_Process();/* run the coursor movement code */
            break;
        case USB_DEV_EVENT_ERROR:
            /* user may add code here for error handling
               NOTE : val has the value of error from h/w*/
            break;
        default:
            break;
    }
    return;
    }
```

3. USB Class specific call back function

   Sample code for the HID class application is given below:

```
uint8_t USB_App_Param_Callback
(
    uint8_t request,
    uint16_t value,
    uint8_t ** data,
    uint32_t* size,
    void* arg
)
{
    uint8_t error = USB_OK;

    uint8_t index = (uint8_t)((request - 2) & USB_HID_REQUEST_TYPE_MASK);
    if ((request == USB_DEV_EVENT_SEND_COMPLETE) && (value == USB_REQ_VAL_INVALID))
    {
        if((g_keyboard.keyboard_init)&& (arg != NULL))
        {
                #if COMPLIANCE_TESTING
```

```
                    uint32_t g_compliance_delay = 0x009FFFFF;
                    while(g_compliance_delay--);
                #endif

                KeyBoard_Events_Process();/* run the coursor movement code */
        }
        return error;
    }

    /* index == 0 for get/set idle, index == 1 for get/set protocol */
    *size =0;
    /* handle the class request */
    switch (request)
    {
        case USB_HID_GET_REPORT_REQUEST :
            *data = &g_keyboard.rpt_buf[0]; /* point to the report to send */
            *size = KEYBOARD_BUFF_SIZE; /* report size */
            break;

        case USB_HID_SET_REPORT_REQUEST :
            for (index = 0; index < KEYBOARD_BUFF_SIZE ; index++)
            {   /* copy the report sent by the host */
    //            g_keyboard.rpt_buf[index] = *(*data + index);
            }
            break;

        case USB_HID_GET_IDLE_REQUEST :
            /* point to the current idle rate */
            *data = &g_keyboard.app_request_params[index];
            *size = REQ_DATA_SIZE;
            break;
case USB_HID_SET_IDLE_REQUEST :
            /* set the idle rate sent by the host */
            g_keyboard.app_request_params[index] =(uint8_t)((value & MSB_MASK) >>
                                                    HIGH_BYTE_SHIFT);
            break;

        case USB_HID_GET_PROTOCOL_REQUEST :
            /* point to the current protocol code
               0 = Boot Protocol
               1 = Report Protocol*/
            *data = &g_keyboard.app_request_params[index];
            *size = REQ_DATA_SIZE;
            break;
```

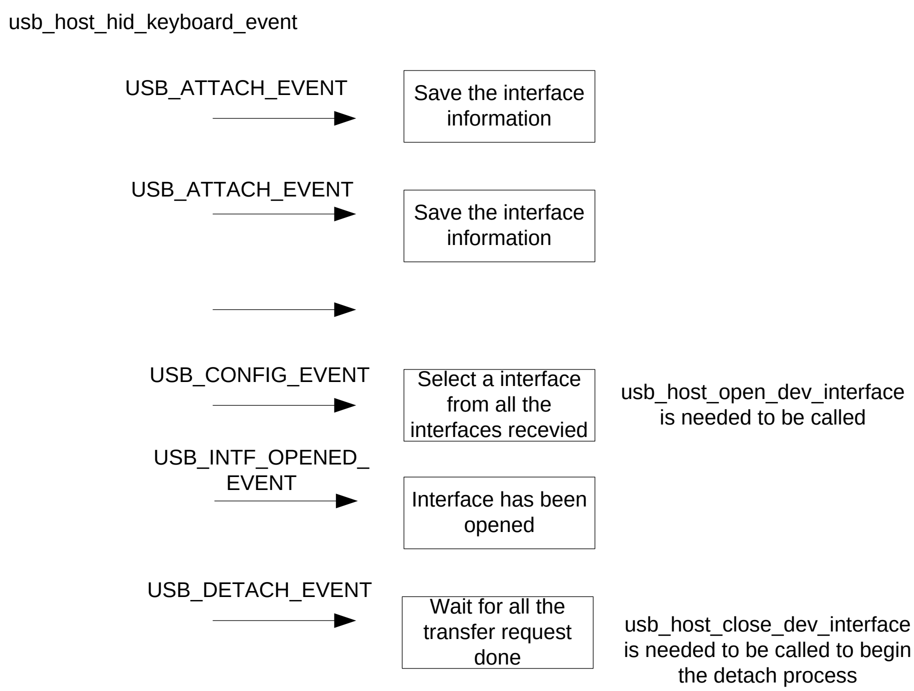
```
        case USB_HID_SET_PROTOCOL_REQUEST :
            /* set the protocol sent by the host
                 0 = Boot Protocol
                 1 = Report Protocol*/
               g_keyboard.app_request_params[index] = (uint8_t)(value);
            break;
    }
    return error;
}
```

# 3 Developing a New USB Host Application

## 3.1 Background

In the USB system, the host software controls the bus and talks to the target devices under the rules defined by the specification. A device is represented by a configuration that is a collection of one or more interfaces. Each interface comprises one or more endpoints. Each endpoint is represented as a logical pipe from the application software perspective.

The host application software registers for services with the USB host stack and describes the callback routines inside the driver info table. The following figure shows the enumeration and detach flow.




The USB host stack is a few lines of code executed before starting communication with the USB device. The examples on the USB stack are written with class drivers APIs. Class drivers work with the host API as a supplement to the functionality. They make it easy to achieve the target functionality (see example sources for details) without dealing with the implementation of standard routines. The following code steps are taken inside a host application driver for any specific device.

## 3.2 How to develop a new host application

### 3.2.1 Creating a project

Perform the following steps to create a project.

1. Create a new application directory under <install_dir>/usb/example/host/xxx to locate the application source files and header files. The xxx is the class name, such as HID and CDC. The name

of this directory is the same as the class name that the application is based on, such as HID and CDC. For example,

`<install_dir>/usb/example/host/hid/test`

2. Copy the bm directory from the similar existing applications directory to the new application directory. Remove the unused project directory from the bm directory. Modify the project directory name to the new application project name. For example, if we want to create toolchain-IAR, board-frdmk64 class-hid related application, then we can create the new application `test` base on a similar existing application `mouse`.

   Change `usb <install_dir>/examples/frdmk64f/demo_apps/usb/host/hid/mouse/bm/iar` to `install_dir>/examples/frdmk64f/demo_apps/usb/example/host/hid/test/bm/iar`

3. Modify the project file name to the new application project file name, for example, from `host_hid_mouse_frdmk64f_bm.ewp` to `host_hid_test_frdmk64f_bm.ewp`. Then you can globally replace the existing name to the new project name by editing the project files. The `host_hid_test_frdmk64f_bm.ewp` file includes the new application project setting.

4. Create a new source file to implement the main application function and the callback function. The `new_app.h` file contains the application types and definitions. The `new_app.c` file contains the driver information, callback functions, event functions, and main function.

### 3.2.2 Defining a driver information table

A driver information table defines the devices that are supported and handled by this target application. This table defines the PID, VID, class, and subclass of the USB device. The host/device stack generates an attached callback when a device matches this table entry. The application now can communicate with the device. The following structure defines one member of the table. If the Vendor-Product pair does not match a device, then Class, Subclass, and Protocol are checked to match. Use 0xFF in Subclass and Protocol structure member to match any Subclass/Protocol.

The following is a sample driver information table. See the example source code for samples. The following table defines all HID KEYBOARD devices that are boot subclasses. A terminating NULL entry in the table is always created for search end.

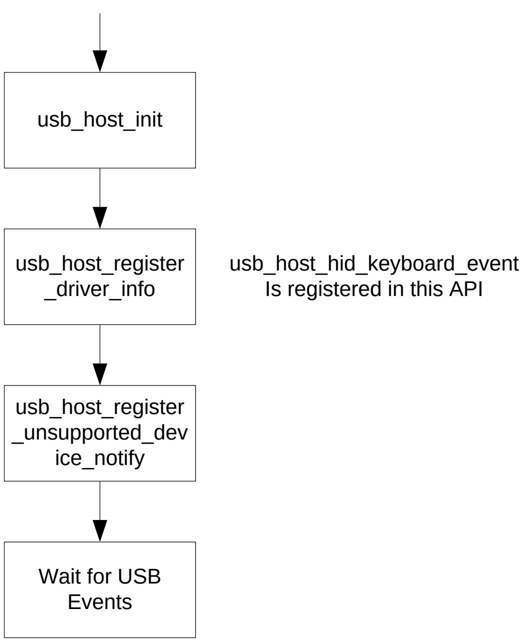
Because two classes (HID and HUB) are used in the HID KEYBOARD application, the DriverInfoTable variable has three elements. There are two event callback functions for two classes: `usb_host_hid_keyboard_event` for the HID class and `usb_host_hub_device_event` for the HUB class.

```
/* Table of driver capabilities this application wants to use */
static usb_host_driver_info_tDriverInfoTable[] = {
    {
        {0x00, 0x00},       /* Vendor ID per USB-IF             */
        {0x00, 0x00},       /* Product ID per manufacturer      */
        USB_CLASS_HID,      /* Class code                       */
        USB_SUBCLASS_HID_BOOT, /* Sub-Class code                   */
        USB_PROTOCOL_HID_KEYBOARD, /* Protocol                          */
        0,                  /* Reserved                         */
        usb_host_hid_keyboard_event /* Application call back function   */
```

```
    },
    /* USB 1.1 hub */
    {
        {0x00, 0x00},       /* Vendor ID per USB-IF             */
        {0x00, 0x00},       /* Product ID per manufacturer      */
        USB_CLASS_HUB,      /* Class code                       */
        USB_SUBCLASS_HUB_NONE, /* Sub-Class code                   */
        USB_PROTOCOL_HUB_ALL, /* Protocol                         */
        0,                  /* Reserved                         */
        usb_host_hub_device_event /* Application call back function   */
    },
    {
        {0x00, 0x00},       /* All-zero entry terminates        */
        {0x00, 0x00},       /* driver info list.                */
        0,
        0,
        0,
        0,
        NULL
    }
};
```

### 3.2.3 Main application function flow

In the main application function, it is necessary to follow these steps




1. Initialize the host controller.

The first step required to act as a host is to initialize the stack in a host mode. This allows the stack to install a host interrupt handler and initialize the necessary memory required to run the stack. The following example illustrates this:

```
status = usb_host_init(CONTROLLER_ID, &host_handle);
```

2. Register services.

   Once the host is initialized, the USB host stack is ready to provide services. An application can register for services as documented in *Freescale USB Stack Host API Reference Manual* (USBHOSTAPIRM). The host API document describes how the application is registered for this device, because the driver information table already registers a callback routine. The following example shows how to register for a service on the host stack:

```
status = usb_host_register_driver_info(host_handle, (void *)DriverInfoTable);
```

3. Register the unsupported device notify. Register a callback function to get all the information about the unsupported device.

```
status = usb_host_register_unsupported_device_notify(host_handle,
usb_host_hid_unsupported_device_event);
```

### 3.2.4 Event callback function

After the software has registered the driver info table and register for other services, it is ready to handle devices. In the USB Host stack, customers do not have to write any enumeration code. When the device is connected to the host controller, the USB Host stack enumerates the device and finds how many interfaces are supported. In addition, for each interface, it scans the registered driver information tables and finds which application has registered for the device. It provides a callback if the device criteria matches the table. The application software has to choose the interface. You can implement the event callback function as follows:

```
void usb_host_hid_keyboard_event(
/* [IN] pointer to device instance */
_usb_device_instance_handledev_handle,
/* [IN] pointer to interface descriptor */
_usb_interface_descriptor_handleintf_handle,
/* [IN] code number for event causing callback */
uint_32 event_code)
{
INTERFACE_DESCRIPTOR_PTR intf_ptr = (INTERFACE_DESCRIPTOR_PTR) intf_handle;
switch (event_code) {
case USB_ATTACH_EVENT:
case USB_CONFIG_EVENT:
<Add your code here>
break;
case USB_INTF_EVENT:
<Add your code here>
break;
case USB_DETACH_EVENT:
```

```
<Add your code here>
break;
}
}
```

Here is the sample code for the HID KEYBOARD application. In this code, the kbd_hid_device variable contains all the states and pointers used by the application to control or operate the device:

```
static void usb_host_hid_keyboard_event
(
    /* [IN] pointer to device instance */
    usb_device_instance_handledev_handle,
    /* [IN] pointer to interface descriptor */
    usb_interface_descriptor_handleintf_handle,
    /* [IN] code number for event causing callback */
    uint32_t event_code
)
{
usb_device_interface_struct_t* pHostIntf =
(usb_device_interface_struct_t*)intf_handle;
interface_descriptor_t* intf_ptr = pHostIntf->lpinterfaceDesc;

    switch (event_code)
    {
        case USB_ATTACH_EVENT:
            kbd_interface_info[kbd_interface_number] = pHostIntf;
            kbd_interface_number++;
            printf("----- Attach Event -----\r\n");
            printf("State = %d", kbd_hid_device.DEV_STATE);
            printf("  Interface Number = %d", intf_ptr->bInterfaceNumber);
            printf("  Alternate Setting = %d", intf_ptr->bAlternateSetting);
            printf("  Class = %d", intf_ptr->bInterfaceClass);
            printf("  SubClass = %d", intf_ptr->bInterfaceSubClass);
            printf("  Protocol = %d\r\n", intf_ptr->bInterfaceProtocol);
            break;
        case USB_CONFIG_EVENT:
            if(kbd_hid_device.DEV_STATE == USB_DEVICE_IDLE)
            {
                kbd_hid_device.DEV_HANDLE = dev_handle;
                kbd_hid_device.INTF_HANDLE = kbd_hid_get_interface();
                kbd_hid_device.DEV_STATE = USB_DEVICE_ATTACHED;
            }
            else
            {
                printf("HID device already attached - DEV_STATE = %d\r\n",
kbd_hid_device.DEV_STATE);
```

```
            }
            break;

        case USB_INTF_OPENED_EVENT:
            printf("----- Interfaced Event -----\r\n");
            kbd_hid_device.DEV_STATE = USB_DEVICE_INTERFACE_OPENED;
            break;

        case USB_DETACH_EVENT:
            <Add your code here>
        default:
            printf("HID Device state = %d??\r\n", kbd_hid_device.DEV_STATE);
            kbd_hid_device.DEV_STATE = USB_DEVICE_IDLE;
            break;
    }

    /* notify application that status has changed */
    OS_Event_set(kbd_usb_event, USB_EVENT_CTRL);
}
```

### 3.2.5 Selecting an interface on the device

If the interface handle is obtained, the application software can select the interface that a retrieve pipe handles. The following code demonstrates this procedure:

```
case USB_DEVICE_ATTACHED:
printf("\nKeyboard device attached\n");
kbd_hid_device.DEV_STATE = USB_DEVICE_SET_INTERFACE_STARTED;
            status = usb_host_open_dev_interface(host_handle,
kbd_hid_device.DEV_HANDLE,              kbd_hid_device.INTF_HANDLE,
(class_handle*)&kbd_hid_device.CLASS_HANDLE);
            if (status != USB_OK)
            {
                printf("\nError in _usb_hostdev_open_interface: %x\n", status);
                return;
            }
            break;
```

### 3.2.6 Sending/Receiving data to/from the device

The transfer flow is quite simple: Call the usb_class_xxx_xxxx API to begin the transfer. The transfer result will be notified by the callback function registered in the usb_class_xxx_xxxx API parameter.

The HID Keyboard host uses the following code to receive data from the device:

```
kbd_hid_com->class_ptr = kbd_hid_device.CLASS_HANDLE;
kbd_hid_com->callback_fn = usb_host_hid_keyboard_recv_callback;
kbd_hid_com->callback_param = 0;
```

```
status = usb_class_hid_recv_data(kbd_hid_com, kbd_buffer, kbd_size);
```

# 4 Revision History

This table summarizes revisions to this document.

**Table 1 Revision History**

| Revision number | Date | Substantial changes |
|---|---|---|
| 1 | 04/2015 | Kinetis SDK 1.2.0 release |
| 0 | 12/2014 | Kinetis SDK 1.1.0 release |

**How to Reach Us:**

**Home Page:**
www.freescale.com

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



Document Number: USBSDNAUG
Rev. 1
04/2015