Freescale Semiconductor
Application Note

Document Number: AN4748
Rev. 0, 10/2013

USB and Using Freescale USB Stack with Kinetis L devices

by: William Jiang

1 Introduction

Universal Serial Bus (USB) is a low-cost, fast, bi-directional, isochronous, and dynamically attachable serial interface that is consistent with the requirements of the PC platform of today and tomorrow. It is widely used in the PC connection world as its name implied.

Freescale's Kinetis L series KL2x and KL4x family devices are cost effective and energy efficient, which integrate USB On-The-Go (OTG) controller to facilitate the customers to interconnect their devices to PCs or any other devices with USB. Freescale also provides USB Stack software for all devices with USB function, which can be downloaded free from Freescale public website. This total USB hardware and software solution makes Freescale devices easy to use, and enables rapid time to market for products.

This application note explains the overview of USB protocol and the key features of Kinetis L USB OTG Controller. In addition, it introduces Freescale USB Stack with focus on USB device stack and HID class. In the later sections, it describes how to use Freescale USB stack to create a HID joystick demo.

Contents

2 USB Overview

USB is a cable bus that supports data exchange between a host computer and a wide range of simultaneously accessible peripherals. USB allows dynamic attachment and detachment of peripherals. The attached peripherals share USB bandwidth through a host scheduled and token-based protocol. Within a USB system, a host is a master and all other peripherals are slaves.

A USB device is logically composed of USB bus interface, USB logical device, and function layer. A USB logical device appears to the USB host as a collection of endpoints. Endpoints are grouped into endpoint sets that implement an interface. Interfaces are views to the function, that is, the function is a collection of interfaces. The USB host system software manages the device using the Default Control Pipe (endpoint 0). The host client software manages an interface using pipe bundles (associated with an endpoint set). The host client software requests that data be moved across the USB between a buffer on the host and an endpoint on the USB device. The Host Controller (or USB device, depending on transfer direction) packetizes the data prior to transfer. The Host Controller also coordinates when to move the packet of data over the USB.

USB connection topology supports seven tiers. There can be maximum five non-root hubs in a communication path between host and any device.

USB supports three types of bit rates in USB specification version 2.0: High Speed (480 MHz), Full Speed (12 MHz), and Low Speed (1.5 MHz). The communication between a host and a peripheral is initiated by the host via USB transfers. Kinetis L devices support Full Speed bit rate.

USB defines the following four different types of transfer:

- Control transfer (bi-directional) - Control transfers are burst, non-periodic, host software-initiated request/response communication, and typically used for command/status operations.
- Interrupt transfer (uni-directional) - Interrupt transfers are low-frequency, bounded-latency communication.
- Isochronous transfer (uni-directional) - Isochronous Transfers are periodic, continuous communication between host and device, typically used for time-relevant information.
- Bulk transfer (uni-directional) - Bulk transfers are non-periodic, large-packet burst communication, typically used for data that can use any available bandwidth and can also be delayed until bandwidth is available.

All peripherals must support control transfer. In addition, USB low speed peripherals support interrupt transfer where as USB full speed and high speed peripherals support all four types of transfers.

USB transfer is comprised of a few USB transactions. USB transaction is the smallest part of USB transfer and is made up of three phases: token phase, data phase, and handshake phase. Under certain circumstances, there is no data phase, for example, the device is unable to complete a USB command.

The host commands a peripheral what to do next by sending USB device requests through control transfers. The host can also poll the state of a peripheral through interrupt transfers. And it can also send or receive bulk data through bulk transfers. In addition, it can exchange the time-related data (for example, audio and video) with a peripheral through isochronous transfers. The USB device can only respond to the host requests and has no capability to initiate any bus transaction.

USB has defined following six states for a USB device:

- Attached state - If the device is attached to the USB, but is not powered by the USB, the device is in Attached state.
- Powered state - Once the device is attached to the USB and powered by the USB, but has not been reset, it is in Powered state.
- Default state - If the device is attached to the USB and powered and has been reset, but has not been assigned a unique address, the device responds at the default address and is in Default state.
- Address state - If the device is in Default state and a unique device address has been assigned, and the device is not configured, then the device is transitioned to Address state.
- Configured state - If the device is already in Address state and configured, and is not suspended, then the device enters Configured state in which the host can use the function provided by the device.
- Suspended state - If the device is in one of the following states: Powered, Default, Address, Configured, and has not seen bus activity for 3 ms, it will go to suspend state. Because the device is suspended, the host may not use the device's function.

When a USB device is attached to the USB, the host uses a process to identify and manage the necessary device state changes known as bus enumeration. The following actions can be taken when a USB device is attached to a port:

1. The device is plugged into a host (in Attached state). The host provides power to the device with a current limit of 100 mA.
2. The host determines low-speed/full-speed capability by pullup resistors connected to either D+ or D– lines. At this point, the device is in the powered state.
3. The host sends a reset to the device by setting D+ and D– low for at least 10 ms. When the host removes the reset, the device goes into the default state.
4. In the default state, the device is ready to respond to control transfers at Endpoint 0. The host communicates with the device using the default address 00. The device can draw up to 100 mA from the host.
5. The host sends a GET_DESCRIPTOR request to Endpoint 0 and address 0 to get the device descriptor.
   The eighteen-byte device descriptor contains maximum packet size supported by Endpoint 0 and other important information for proper communication.
6. The host assigns a unique address to the device by sending a SET_ADDRESS request. The device is in the address state.
7. The host sends a GET_DESCRIPTOR request to the new address to read the full device descriptor.

8. The host then requests any additional descriptors specified in the device descriptor, each descriptor begins its length and type.
9. The host assigns a device driver based on the data in the descriptors. Windows will use the devices vendor ID and product ID to find an appropriate INF file to determine what drivers to load. If there is no match, Windows will use a default driver according to class.
10. If the device supports multiple configurations, the host will send a SET_CONFIGURATION request to the device to select the desired configuration.

There are various classes defined by the USB: HID, CDC, mass-storage, audio, and video. Each of these classes has its class specific data that are exchanged between the host and device, and the related class requests for the host to request the device to execute.

HID class primarily consists of devices that are used by humans to control the operation of computer systems, for example, mouse, keyboard, joystick, slider, knob, throttle, bar-code reader, and many more.

HID class specification defines how the HID class driver should extract data from USB devices by introducing the HID class descriptors (report descriptors, physical descriptors, and physical descriptor sets) and HID class-specific requests.

HID report descriptors are self-describing data structures containing different items with associated tags, types, sizes, and data, which enable the host to recognize and handle all data report that are coming through the USB. HID physical descriptor is a data structure that provides information about the specific part or parts of the human body that are activating control or controls. A report descriptor may be associated with a physical descriptor using report descriptor's designator index items (indicate which part of the body affects the item). A HID descriptor specifies the number of class descriptors (always at least one, that is, Report descriptor.) A report descriptor may specify three different types of reports: Input, Output, and Feature reports.

A report is same as transfer. It returns the structure or structure(s) in which each data field is sequentially represented as described by the report descriptor or physical descriptor. Only **Input** reports are sent via the **Interrupt In** pipe. **Feature** and **Output** reports must be initiated by the host via **Control** pipe or an optional **Interrupt Out** pipe.

Note

For more detailed information on the USB protocol, please refer to USB specification 2.0 at http://www.usb.org. All USB classes related documents can also be found on this website.

3 Kinetis L USB OTG controller

Kinetis L family have similar USB OTG controller to other Kinetis K families with an addition of host frame adjust capability to compensate for frame inaccuracies in the 48 MHz USB clock through adjusting the frame cycles by -128 to +127 in full speed 12 MHz clock. It is compliant with USB1.1 and USB2.0 full speed device specification and the USB 2.0 OTG supplement with 16-bidirectional endpoints and DMA or FIFO data stream interfaces. It provides a single-chip solution for full-speed

(12Mbps) USB device applications, and integrates the required transceiver with serial interface engine (SIE), 3.3 V regulator, and other control logics.

Kinetis Peripheral Module Quick Reference (KQRUG), available on freescale.com.

4 Freescale USB stack

Freescale USB stack is an open source software available to all the customers, supporting all Freescale MCUs with USB which includes ColdFire, HCS08, and Kinetis ARM® Cortex™- M4 microcontroller families. It also supports many popular development tools such as CodeWarrior Development Studio for Microcontrollers, IAR Embedded Workbench for ARM, and Keil uVision4 Integrated Development Environment.

It includes two driver stacks: Freescale USB device stack and host stack, and one supplement part: OTG.

There are API Reference Manual which details all driver APIs and the User Guide which provides general guidelines for the use of the stack to develop new class drivers and applications.

This document focuses on Freescale USB device stack. For others, please refer to the related users guide. For more details see References.

This chapter first describes the architecture of the USB device stack with the examples provided in stack. And then it describes the introduction of USB Driver, HID Class Driver, and the resource usage of the Freescale USB Device Stack for HID class.

4.1 Freescale USB device stack architecture

The following figure shows the USB stack directory structure:

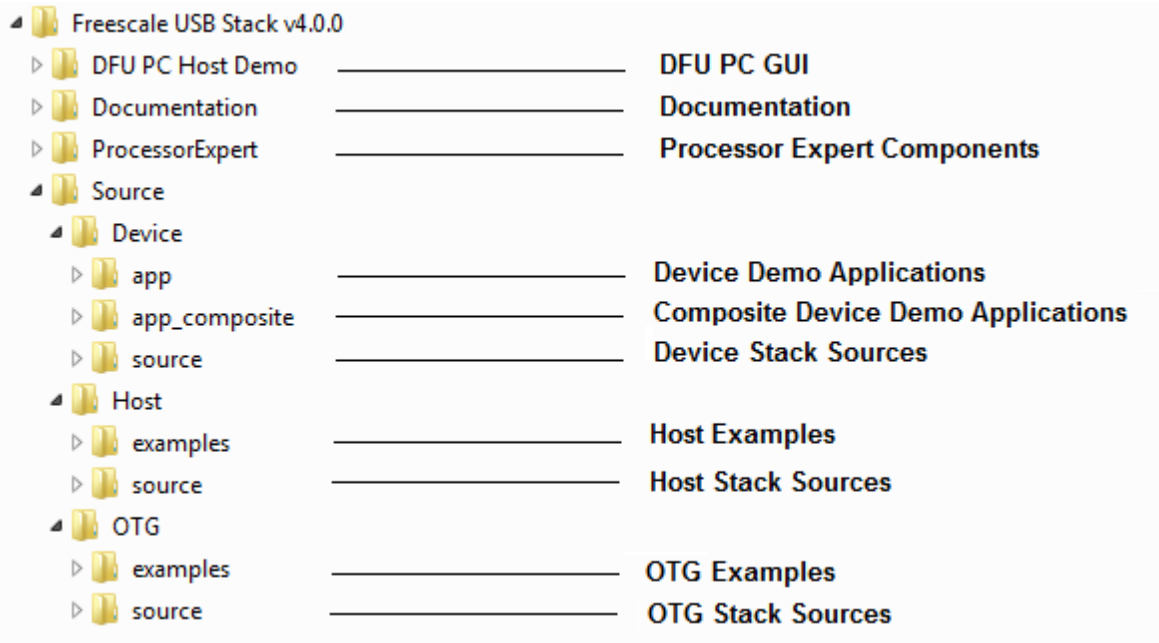

Figure 1. Freescale USB stack directory structure

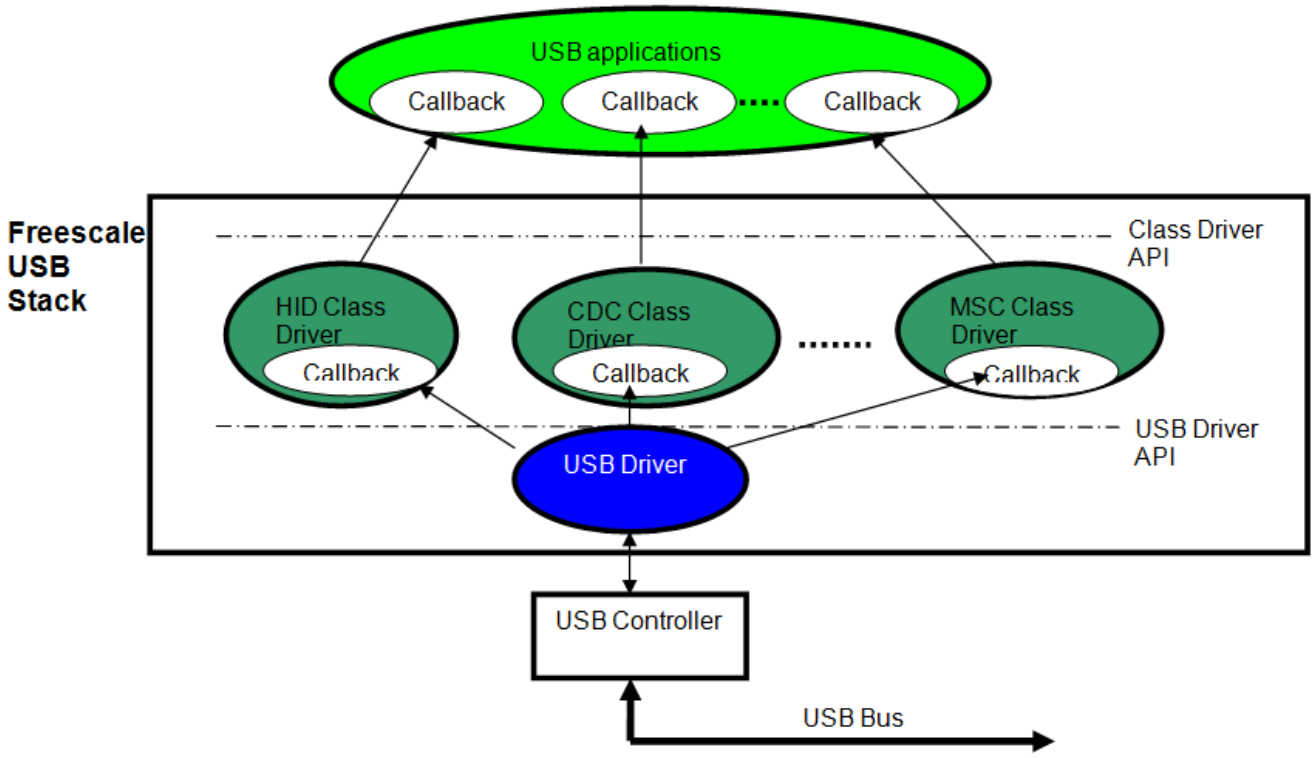

Figure 2. Freescale USB device stack architecture

The USB stack is a layered architecture that helps the application developers concentrate on developing the application without being concerned about other layers. The applications can also be seamlessly ported over to other cores after the low-level driver for that core is available.

Figure 2 shows the USB Device Stack Architecture. It includes three layers: USB Driver, Class Drivers, and USB applications.

USB Driver or Low-Level Driver is hardware dependent, sits on the USB Controller, initialize or de-initialize a USB device controller, initializes endpoints, reads SETUP data for an endpoint, sends or receives data on an endpoint, sets the address of a USB device controller, stalls or unstalls an endpoint, cancels a pending transfer, and shuts down the USB device controller. It also allows the high-layer software to register or unregister callback service routines for an event or an endpoint. When an event occurs at the bus, or data is sent or received at an endpoint, the USB driver calls the corresponding service callback function registered for this event or endpoint.

USB driver also contains USB Device Controller Interface (DCI) APIs which are used to interface with numbers of USB controllers that a device may contain.

The class driver layer includes framework module, common class module, and class specific module. The framework module handles all requests to the control endpoint and implements all requests defined in USB Chapter 9 “USB Device Framework” of the USB specification 1.1 available on www.usb.org.

The common class module contains implementation independent to application specific classes. It handles functions like suspend/resume, reset, stall, and SOF that needs to be present for all classes. When an event occurs, the common class module will notify the class specific module or higher layer

software via service callback routines. Such events include USB bus reset, enumeration complete, configuration change, data send complete, and data received.

The class specific module implements class specific functionality. It implements all interactions with non control endpoints. The data sent and received on these endpoints are class specific. This module also implements the class specific requests on the control end point.

HID Class driver handles the HID protocols, while the CDC class driver handles the communication class protocols by implementing the Abstract Control Model Serial Emulation. The MSC Class driver handles the mass storage specification. There are other class drivers supported by the USB stack such as Personal Healthcare Device Class (PHDC), Device Firmware Update (DFU) Class, USB Video Class and so on.

USB applications use the services provided by the class drivers to implement application specific functions based on the USB communication link. This stack provides a lot of example applications to facilitate the use of all drivers. Typical examples are HID mouse, keyboard, SD card, USB audio, battery charging, personal health care, video device, MSD, and CDC composite.
The following section describes typical HID examples. For other examples, please consult Freescale USB Device Stack Users Guide.

4.2 Freescale USB device stack HID examples

HID class demos include: HID keyboard and HID mouse.

Table 1. HID demo

Demos	Description
HID keyboard device	Emulates a keyboard
HID mouse device	Emulates a mouse

4.3 USB stack USB driver and HID class driver files

The USB Stack contains a lot of files in different folders. The following table lists all source files with their description to understand the usage and structure of files:

Table 2. USB stack files for HID

Files		Description
USB Driver files	Usb_driver.c	USB driver API and the related code
	Usb_dci_kinetis.c	USB device controller interface (DCI) code for specific device (Kinetis in this example)
	Usb_dci_kinetis.h	Contains Macro's and functions needed by the DCI layer , device specific (Kinetis in this example)

	Usb_bdt_kineits.h	Buffer Descriptor Table definition for specific device (Kinetis in this example)
	Usb_dciapi.h	DCI APIs function definitions
	Usb_devapi.h	Header file for device layer APIs
HID class driver files	Usb_framework.c	Code to handle USB device framework as described in USB specification Chapter 9
	Usb_framework.h	USB device framework header file
	Usb_hid.c	HID class specific driver
	Usb_hid.h	Header file for HID class specific driver
	Usb_class.c	Class layer class independent driver code
	Usb_class.h	Macros and function prototypes for class independent driver code
Application	Usb_descriptor.c	Template code contains USB Framework Module interface (all descriptor APIs required to be implemented in an application). It also contains various descriptors defined by USB Standards like device descriptor, configuration descriptor, string descriptor, and other class specific descriptors that are provided to Framework Module when requested. For customization, user can modify these variables and function implementations to suit the requirement
	Usb_descriptor.h	Mandatory header files for the application to implement. Framework and class drivers include this file for function prototype definitions and data structures described in usb_descriptor.c. User modifying usb_descriptor.c should also modify MACROs in this file as well
	user_config.h	Required header file to define various compile time macros. These parameters are essential for successful compilation of source code. Mandatory macros that need to be defined are: – LONG_SEND_TRANSACTION — This macro is defined for classes that have to send data more than the endpoint size over USB bus. It allows Low-Level Device Framework to use split transaction for sending

		data over USB bus. – LONG_RECIEVE_TRANSACTION — This macro is defined for classes that have to receive data more than the endpoint size over USB bus. It allows Low-Level Device Framework to use split transaction for receiving data over USB bus. – USB_PACKET_SIZE — This macro defines maximum transfer packet size of USB Data Transaction. – DOUBLE_BUFFERING_USED — For S08 devices, if application is making use of endpoint 5 and 6 (Double Buffered) then this macro has to be defined. For Kinetis, CFV1 and CFV2 devices, this macro is not required, as all the endpoints are double buffered. – MULTIPLE_DEVICES — if defined, the same endpoint can dynamically have multiple function assignments depending on user input via registering/unregistering service callback routines for an event or endpoint; if not defined, the endpoint or service can only have single function assignment and all the service callback routines must be defined in usb_config.h
	Usb_config.h	Required header file if MULTIPLE_DEVICES macro is not defined in user_config.h. It defines all related service callbacks that are essential for successful compilation of source code.

These files shall be added in the application project with some required changes.

4.4 HID driver

The HID class driver provides the following four APIs to the higher layer software:

Table 3. HID class APIs

API Function	Description
USB_Class_HID_Init()	Initialize the HID class with controller ID and some callbacks
USB_Class_HID_DeInit()	De-initialize the HID class
USB_Class_HID_Periodic_Task()	Periodic call to the class driver to complete pending tasks

USB_Class_HID_Send_Data()	Sends the HID report to the host

The class driver also requires the following API functions (USB descriptor related for framework module) to be implemented by the higher layer software:

API Function	Description
USB_Desc_Get_Descriptor()	Gets various descriptors as specified from the application
USB_Desc_Get_Endpoints()	Gets the endpoints used and their properties
USB_Desc_Get_Interface()	Gets the currently configured interface
USB_Desc_Set_Interface()	Sets new interface
USB_Desc_Remote_Wakeup()	Checks whether the application supports remote wakeup or not
USB_Desc_Valid_Configation()	Checks whether the configuration being set is valid or not
USB_Desc_Get_Interface()	Checks whether the interface being set is valid or not

The reference code implementing all the preceding required APIs is defined in usb_descriptor.c of a demo code.

To use HID class layer API functions, the higher layer software must follow steps below:

- Call "*USB_Class_HID_Init()*" to initialize the class driver, all the layers below it, and the device controller. Event callback functions are also passed as a parameter to this function.
- When the callback function is called with the USB_APP_ENUM_COMPLETE event, the higher layer software should move into the ready state.
- Call "*USB_Class_HID_Send_Data()*" to send data/report to the host through the underlying layer drivers when required.

For detailed information on all APIs, please consult API Reference Manual.

4.4.1 HID driver callbacks

HID driver callback routines are required to handle USB generic class events, USB vendor specific requests, and USB class specific request.
The prototype of generic class callback routine is as follows:

```
typedef void(_CODE_PTR_ USB_CLASS_CALLBACK)(
uint_8 controller_ID,   // USB controller ID
```

```
uint_8 type,        // type of event
void* data          // event data based on the event type
```

The caller implementing this callback shall typecast the data parameter to the data type or structure, based on the type of event before reading it.

The prototype of vendor specific request callback is as follows:

```
typedef uint_8 (_CODE_PTR_ USB_REQ_FUNC)(
uint_8 controller_ID,                   // USB controller ID
USB_SETUP_STRUCT *setup_packet,  // SETUP packet received from host
uint_8_ptr *buff,                       // pointer to the buffer to be returned with data
USB_PACKET_SIZE *size);          // size of data to be received or sent
```

The prototype of class specific request callback is as follows:

```
typedef uint_8 (_CODE_PTR_ USB_CLASS_SPECIFIC_HANDLER_FUNC)(
uint_8 request,             // request code from SETUP packet
uint_16 value,              // value code from SETUP packet
uint_8_ptr *buff,           // pointer to the buffer to be returned with data
USB_PACKET_SIZE *size);     // size of data to be received or sent
```

4.4.2 Service events

There are following service events defined by USB driver:

Table 4. USB driver events

Service event	Description
USB_SERVICE_BUS_RESET	Signals a bus reset is just occurred
USB_SERVICE_EP<n>	Signals an endpoint transaction is just occurred, where <n> is the endpoint number the device supports.
USB_SERVICE_ERROR	Signals an error is just occurred
USB_SERVICE_RESUME	Signals a device resumes from suspend state
USB_SERVICE_SLEEP	Device sleep event
USB_SERVICE_SOF	Start of Frame event
USB_SERVICE_SPEED_DETECTION	Speed detection event
USB_SERVICE_STALL	Endpoint stall event
USB_SERVICE_SUSPEND	Device suspend event

The preceding events are intended for class drivers, not for applications. For HID class driver, the service callbacks for the related events are defined in usb_config.h.

There are following service events exposed to the higher layer software by the class driver:

Table 5. HID events

Service event	Description
USB_APP_BUS_RESET	Bus reset event
USB_APP_CONFIG_CHANGED	Configuration is changed
USB_APP_ENUM_COMPLETE	Device enumeration is completed and ready for use
USB_APP_SEND_COMPLETE	Data transfer is completed
USB_APP_DATA_RECEIVED	Data is received. Not used for HID class
USB_APP_ERROR	Error occurs
USB_APP_GET_DATA_BUFF	Get data buffer. Not used for HID class
USB_APP_EP_STALLED	Endpoint is stalled
USB_APP_EP_UNSTALLED	Endpoint is unstalled
USB_APP_GET_TRANSFER_SIZE	Get the transfer size. Not used for HID class

Typical events that should be checked in a class callback routine are *USB_APP_BUS_RESET*, *USB_APP_CONFIG_CHANGED*, and *USB_APP_ENUM_COMPLETE*

4.5 Resource usage

The USB Stack occupies very small amount of memory in bytes as listed in Table 6. These data are calculated directly from the linker map file generated by CodeWarrior 10.3. It includes RAM used by USB driver, HID driver, and application specific code. Flash memory includes code, initialized data, and constant variables like USB descriptors. RAM includes initialized data, uninitialized data, and heap.

Table 6. Resource usage for HID driver, USB driver and HID mouse

Memory	USB Driver	HID Driver	HID mouse
Flash	3436	3236	12196
RAM	2580	116	3076

Note

The size of HID demo counts in the size of all of its components except the stack size which is fixed to 0x800.

5 HID joystick demo

HID joystick demonstrates joystick control with two axis (X and Y), one throttle, one hat switch, and four buttons.

The demo board is the low-cost FREEDOM-KL25Z, which contain a 3-axis accelerometer - MMA8451Q, a smart low-power and 3-axis capacitive micro-machined accelerometer with 14 bits of resolution. The accelerometer is used to emulate the mouse movement in X and Y direction and the throttle control. The demo board also contains a slider touch pad E1, which is used to simulate the four buttons.

To simulate the hat switch, additional hardware named Potentiometer is needed and the connection to the board is as follows:

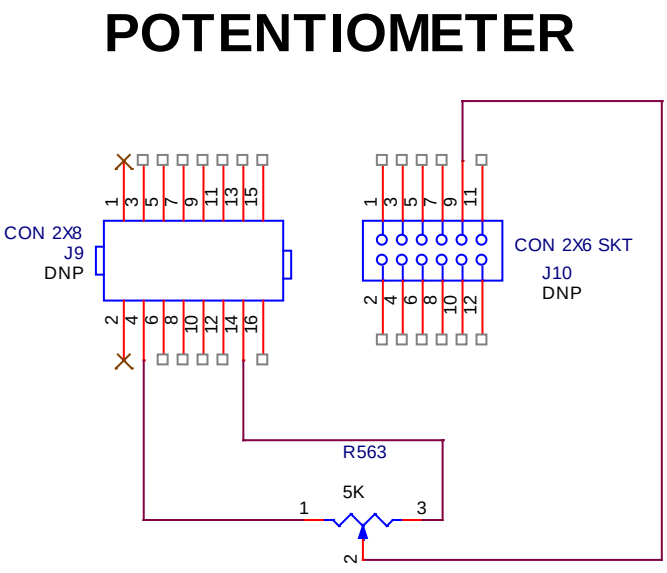

Figure 3. External potentiometer connection

The following list describes the usage of HID Joystick Demo:

- Tilt the board left or right to move X left or right.
- Tilt the board forward or backwards to move Y up or down.
- Pull up or down the board to move throttle up or down.
- Touch the slider pad E1 and slide on it to select different buttons.
- Rotate the external potentiometer to wheel the hat switch.

The demo code is created by referring to the USB stack HID mouse demo with some changes that will be described in the following subsections.

5.1 Add and change HID class required files

As described in the previous sections, *usb_descriptor.c*, *usb_descriptor.h*, *user_config.h,* and *usb_config.h* shall be added to the demo project in addition to all the HID class driver and USB driver code. In addition, some changes will be made to fulfill the demo application, this section describes such changes.

The first change is made to the USB descriptors in *usb_descriptor.c*. The USB descriptors include standard USB device descriptor, configuration descriptor, interface descriptor, HID class descriptor, endpoint descriptor, string descriptor, and HID report descriptor. The demo simply modifies the device descriptor *g_device_descriptor*, configuration descriptor *g_config_descriptor*, and the string descriptor *USB_STR_2*, replaces the existing HID report descriptor *g_report_descriptor* with a joystick HID report descriptor *g_joy_report_descriptor*.

In order to make the device a joystick or game pad device on Windows system, it requires a self-explained HID report descriptor to be sent to the host and it must declare its top-level collection as belonging to the Generic Desktop Page (0x01), and deploy usage of Joystick (0x04) or Game Pad (0x05) respectively.

The joystick HID report descriptor is defined as the following format:

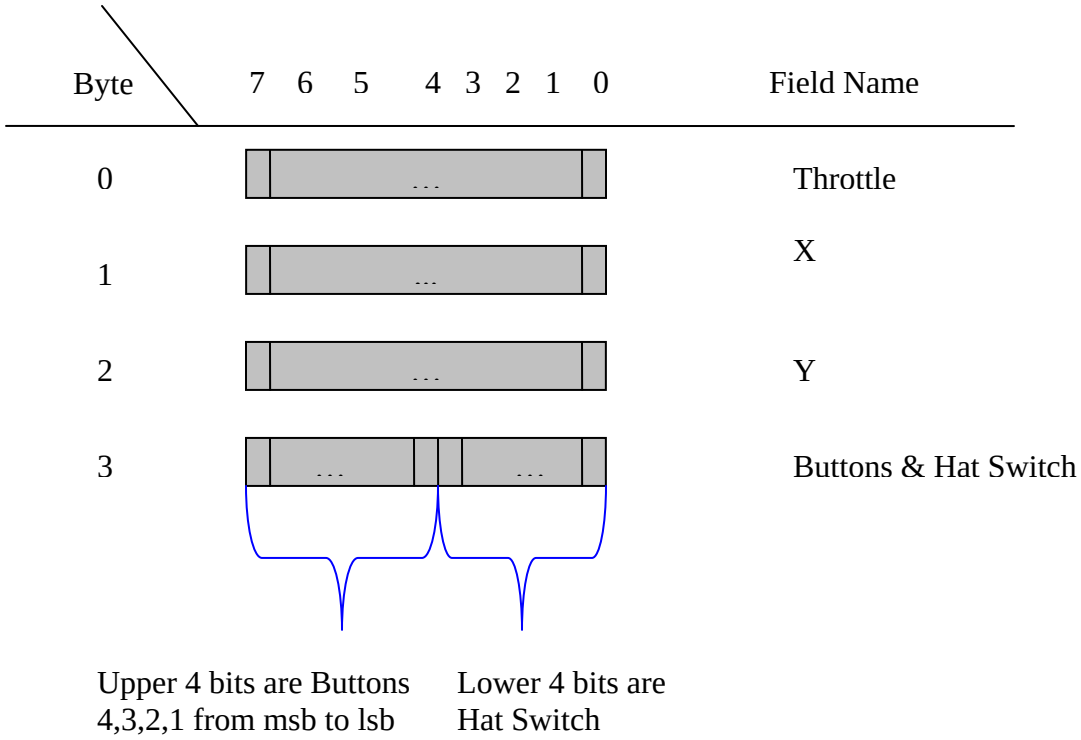

Figure 4. Joystick report structure

This report structure has 4 bytes with each field representing values of throttle, X, Y, buttons or hat switch.

The global variable *hid_joy_report_in* is defined in this structure format to store the real-time HID report data which will be transferred via interrupt pipe to the host:

```
static signed char hid_joy_report_in[HID_JOY_REPORT_IN_SIZE] = {
0};
```

where, HID_JOY_REPORT_IN_SIZE = 4:

Some utility macros are also defined to facilitate the access to this structure. For example, *DIR_REP_X(h)* macro is used to access the X field, *CHANGE_RPT_IN_X(h,x)* macro is used to change the X field of the report h.

Following is the joystick HID report descriptor that describes the aforementioned report structure, which can be created using USB HID Descriptor Tool DT.exe (http://www.usb.org/developers/hidpage/dt2_4.zip):

```
const uint_8 g_joy_report_descriptor[76] = {
  0x05, 0x01,          // USAGE_PAGE (Generic Desktop)
  0x09, 0x04,          // USAGE (Joystick)
```

```
0xa1, 0x01,          // COLLECTION (Application)
0x05, 0x02,          //   USAGE_PAGE (Simulation Controls)
0x09, 0xbb,          //   USAGE (Throttle)
0x15, 0x81,          //   LOGICAL_MINIMUM (-127)
0x25, 0x7f,          //   LOGICAL_MAXIMUM (127)
0x35, 0x00,          //   PHYSICAL_MINIMUM (0)
0x46, 0xff, 0x00,    //   PHYSICAL_MAXIMUM (255)
0x75, 0x08,          //   REPORT_SIZE (8)
0x95, 0x01,          //   REPORT_COUNT (1)
0x81, 0x02,          //   INPUT (Data,Var,Abs)
0x05, 0x01,          //   USAGE_PAGE (Generic Desktop)
0x09, 0x01,          //   USAGE (Pointer)
0xa1, 0x00,          //   COLLECTION (Physical)
0x09, 0x30,          //     USAGE (X)
0x09, 0x31,          //     USAGE (Y)
0x95, 0x02,          //     REPORT_COUNT (2)
0x81, 0x02,          //     INPUT (Data,Var,Abs)
0xc0,                //     END_COLLECTION
0x09, 0x39,          //   USAGE (Hat switch)
0x15, 0x00,          //   LOGICAL_MINIMUM (0)
0x25, 0x03,          //   LOGICAL_MAXIMUM (3)
0x35, 0x00,          //   PHYSICAL_MINIMUM (0)
0x46, 0x0e, 0x01,    //   PHYSICAL_MAXIMUM (270)
0x65, 0x14,          //   UNIT (Eng Rot:Angular Pos)
0x75, 0x04,          //   REPORT_SIZE (4)
0x95, 0x01,          //   REPORT_COUNT (1)
0x81, 0x02,          //   INPUT (Data,Var,Abs)
0x05, 0x09,          //   USAGE_PAGE (Button)
0x19, 0x01,          //   USAGE_MINIMUM (Button 1)
0x29, 0x04,          //   USAGE_MAXIMUM (Button 4)
```

```
    0x25, 0x01,             //  LOGICAL_MAXIMUM (1)
    0x15, 0x00,             //  LOGICAL_MINIMUM (0)
    0x75, 0x01,             //  REPORT_SIZE (1)
    0x95, 0x04,             //  REPORT_COUNT (4)
    0x81, 0x02,             //  INPUT (Data,Var,Abs)
    0xc0                    // END_COLLECTION
};
```

5.2 Add callbacks

The generic class callback and the class specific request callback must be added to the demo project. This demo reuses the same callback names as in the HID demo: *USB_App_Callback* as a generic class callback**,** *USB_App_Param_Callback* as a class specific request callback. The pointers to these callbacks will be passed to *USB_Class_HID_Init()* as parameters. Following is the code snippet for *USB_App_Param_Callback* that highlight the required changes:

```
uint_8 USB_App_Param_Callback(
    uint_8 request,      /* [IN] request type */
    uint_16 value,       /* [IN] report type and ID */
    uint_8_ptr* data,    /* [OUT] pointer to the data */
    USB_PACKET_SIZE* size  /* [OUT] size of the transfer */
)
{
…
   /* handle the class request */
   switch(request)
   {
      case USB_HID_GET_REPORT_REQUEST :
         *data = &hid_joy_report_in[0]; /* point to the report to send */
         *size = HID_JOY_REPORT_IN_SIZE; /* report size */
         break;
```

```
        case USB_HID_SET_REPORT_REQUEST :
            for(index = 0; index < HID_JOY_REPORT_IN_SIZE ; index++)
            {   /* copy the report sent by the host */
                    hid_joy_report_in[index] = *(*data + index);
            }
            break;
….
```

5.3 Add application task

The application task is the main task which will execute repetitively to fulfill the required application specific tasks. It is required to call *USB_Class_HID_Periodic_Task()* as follows:

```
void hid_joy_task(void)
{
        /* call the periodic task function */
        USB_Class_HID_Periodic_Task();
        if(joy_init) /*check whether enumeration is complete or not */
        {
                /* run the button emulation code */
                Emulate_Joy_WithButton();
        }
}
```

The *Emulate_Joy_WithButton()* is used to emulate the joystick control by sending the HID report to the host if there is any change in the control. The following code snippet shows how to send the HID report with changed X and Y fields:

```
if(xy_state_changed) {
    xy_state_changed = FALSE;
    /* Write report data.
     */
(void)USB_Class_HID_Send_Data(CONTROLLER_ID,HID_ENDPOINT,
```

```
    hid_joy_report_in,  HID_JOY_REPORT_IN_SIZE);
}
```

The demo will start a timer (TPM1) to periodically check the joystick controls and change the joystick HID report once a change event is detected. For example, the following code reads 14-bit x-axis data from the accelerator using *hal_dev_mma8451_read_reg()* and then check changes on x-axis. If a change is made on x-axis, then write to the joystick HID report with the new value, and at last change the state variable *xy_state_changed* to flag it is changed:

```
AdcResult = hal_dev_mma8451_read_reg(0x01) << 8;
AdcResult |= hal_dev_mma8451_read_reg(0x02);
AdcResult >>= 6;
delta =  ((AdcResult)-x);
if( ((delta) >= XY_THRESHOLD) ||
   ((delta) < -XY_THRESHOLD)
 )
 {
 …
 CHANGE_RPT_IN_X(hid_joy_report_in,delta);
  xy_state_changed = TRUE;
 }
```

5.4 Add driver initialization code

Before requesting service from the USB stack drivers, it is required to initialize class driver as follows:

```
error = USB_Class_HID_Init(CONTROLLER_ID, USB_App_Callback, NULL,
USB_App_Param_Callback);
```

This will initialize the class driver, all the layers below it, and the device controller. The class event callback functions are also passed as parameters.

In addition, before the class driver initialization, the USB device controller must be enabled. This is completed by selecting a right clock source, enabling the clock to the controller, and the USB regulator as well as the USB interrupt. This work is performed in *usb_init()* as follows:

```
/* Enable USB-OTG IP clocking */
```

```
SIM_SCGC4 |= (SIM_SCGC4_USBOTG_MASK);
/* Configure USB to be clocked from PLL */
SIM_SOPT2  |= (SIM_SOPT2_USBSRC_MASK |
SIM_SOPT2_PLLFLLSEL_MASK);
/* Configure enable USB regulator for device */
SIM_SOPT1 |= SIM_SOPT1_USBREGEN_MASK;
 NVIC_ICER = (1<<24);       /* Clear any pending interrupts on USB */
 NVIC_ISER = (1<<24);       /* Enable interrupts from USB module */
```

5.5 Code summary

To summarize the aforementioned steps, the following table lists the key code modules:

Table 7. Key modules

Software Modules	Description
Hid_joy.c	Application main task module. It includes the aforementioned driver initialization code, application task, timer task and so on.
usb_descriptor.c	USB Framework Module interface (all descriptor APIs required to be implemented in an application). It also contains various descriptors defined by USB Standards such as device descriptor, configuration descriptor, string descriptor, the joystick HID report descriptor, and other required descriptors.
main.c	Is the entry point of the application which performs device initialization and then go to application main task module.
TSIdrv.c	Touch sensing input driver
Hal_dev_mma8451.c, hal_i2c.c	Accelerometer driver code and IIC driver code
Adc.c	Is the adc driver which contains adc operation routines: ADC_Init, ADC_Cvt, ADC_Poll.

The demo code must be compiled with the following two predefined preprocessors:

LITTLE_ENDIAN, __MK_xxx_H__

5.6 Run the demo

Click the Windows Start menu, select “Settings →Control Panel” followed by clicking “Game Controller”. The Game Controllers window appears with empty in the Installed game controllers list.

Plug two USB cables from two USB ports on the freedom board to PC. The “MK HID Joystick DEMO” in the Installed game controller list will appear as follows:

Figure 5. Game controller

Select this controller, and click the Properties button, the MK HID Joystick Demo properties window appears as shown in the following figure:

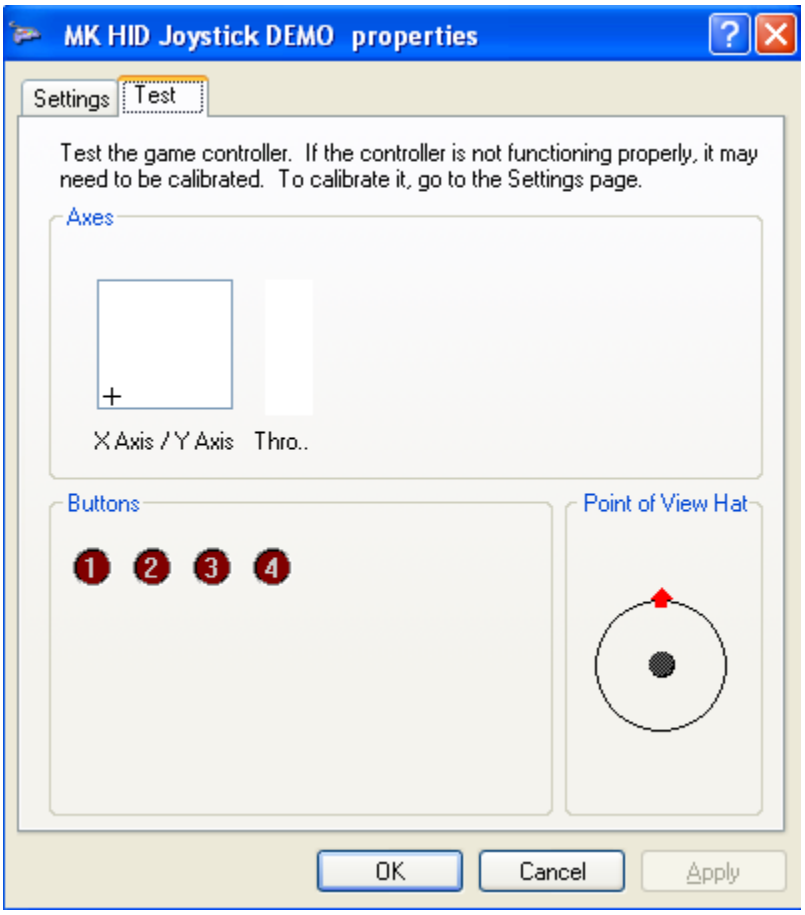

Figure 6. Joystick demo properties

Follow steps below while watching on the window:

1. Tilt the board left or right to move X left or right.
2. Tilt the board forward or backwards to move Y up or down.
3. Pull up or down the board to move throttle up or down.
4. Touch the slider pad E1 and slide on it to select different buttons.

6 Conclusion

Freescale USB Stack is an easy to use software with a lot of demos and documents.

This document mainly describes the Freescale USB device stack architecture and the HID class driver usage. At last, it describes how to use the USB stack to build a HID joystick demo.

7 References

(1) Kinetis Peripheral Module Quick Reference, KQRUG, Rev.0
(2) Freescale USB Device Stack Users Guide, USBUG, Rev.12
(3) Freescale USB Host Stack Users Guide, USBHOSTUG, Rev.6
(4) Freescale USB Stack OTG Users Guide, USBOTGUG, Rev.2
(5) Freescale USB Stack Device API Reference Manual, USBAPIRM, Rev.10
(6) Freescale USB Stack OTG API Reference Manual, USBOTGAPIRM, Rev.2
(7) Freescale USB Host Stack API Reference Manual, USBHOSTAPIRM, Rev.5
(8) Freescale Processor Expert USB Components Quick Start Guide
(9) Freescale USB Stack Hardware Configuration

8 Glossary

USB	*Universal Serial Bus*
OTG	*On-The-Go*
DCI	*Device Controller Interface*

9 Revision history

Revision Number	Date	Substantial Changes
0	10/2013	Initial release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4748
Rev. 0
10/2013