Application Note: AN00146

xCORE-XA - USB HID Class

This application note shows how to create a USB device compliant to the standard USB Human Interface Device (HID) class on an XMOS xCORE-XA multicore microcontroller.

The code associated with this application note provides an example of using the Silicon Labs EFM32 USB library and associated USB class descriptors to provide a framework for the creation of a USB HID.

The USB HID example provide a demonstration of a simple keyboard example running over full speed USB. The code used in the application note creates a device which supports the standard requests associated with this class of USB devices. The code which forms the basis of the application note is taken from an existing Silicon Labs application note for the EFM32 device family.

The application operates as a simple keyboard which when running uses a button on the development board to generate keyboard events on the host machine. This demonstrates the simple way in which PC peripheral devices can easily be deployed using an xCORE device.

Note: This application note provides a standard USB HID class device and as a result does not require drivers to run on Windows, Mac or Linux.

The example code in this application note does not communicate between the xCORE tile and the ARM core. The example demonstrates how the xCORE tile can still be used whilst the ARM core is running a full speed USB device.

Required tools and libraries

- xTIMEcomposer Tools - Version 13.2.0
- EFM32 USB library - Version 3.20.2 (supplied with application note)

Required hardware

This application note is designed to run on an XMOS xCORE-XA series device.

The example code provided with this application note has been implemented and tested on the xCORE-XA core module board but there is no dependancy on this board and it can be modified to run on any development board which uses an xCORE-XA series device.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, the Universal Serial Bus 2.0 Specification (and related specifications), the XMOS tool chain and the xC language. Documentation related to these aspects which are not specific to this application note are linked to in the *References* appendix.
- For descriptions of XMOS related terms found in this document please see the XMOS Glossary[1].
- The XMOS tools manual contains information regarding the use of xCORE-XA devices[2].

[1] http://www.xmos.com/published/glossary
[2] http://www.xmos.com/published/xtimecomposer-user-guide

1 Overview

1.1 Introduction

The HID class consists primarily of devices that are used by humans to control the operation of computer systems. Typical examples of HID class include:

- Keyboards and pointing devices, for example, standard mouse devices, trackballs, and joysticks.
- Front-panel controls, for example: knobs, switches, buttons, and sliders.
- Controls that might be found on devices such as telephones, VCR remote controls, games or simulation devices, for example: data gloves, throttles, steering wheels, and rudder pedals.
- Devices that may not require human interaction but provide data in a similar format to HID class devices, for example, bar-code readers, thermometers, or voltmeters.

Many typical HID class devices include indicators, specialized displays, audio feedback, and force or tactile feedback. Therefore, the HID class definition includes support for various types of output directed to the end user.

The USB specification provides a standard device class for the implementation of HID class devices.

(http://www.usb.org/developers/devclass_docs/HID1_11.pdf)

1.2 Block diagram

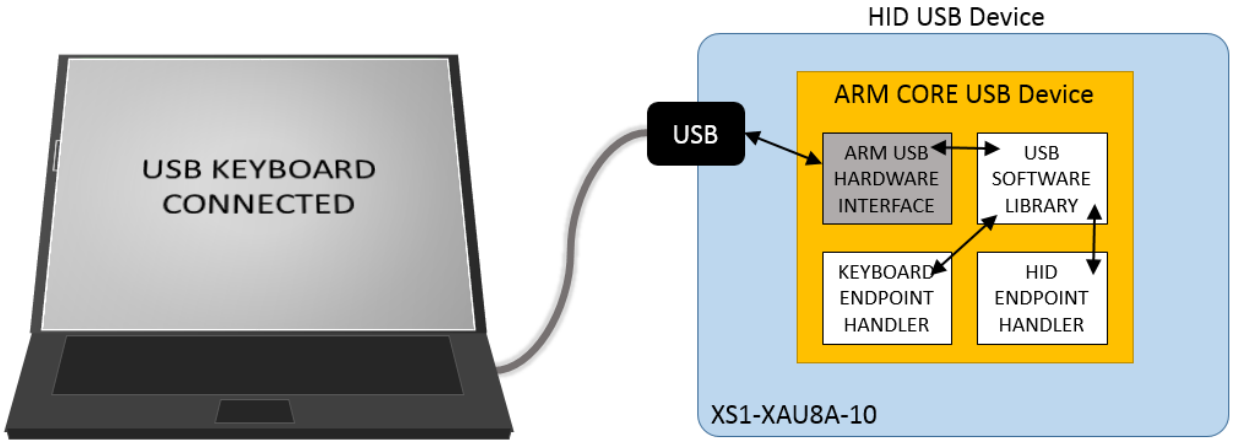

Figure 1: Block diagram of xCORE-XA USB HID application example

2 xCORE-XA USB HID Class application note

The example in this application note uses the EFM32 usb library with an XMOS xCORE-XA silicon device in order to demonstrate a basic USB keyboard which can output text characters to a host PC.

For the USB HID device class example, the system comprises of 2 tasks, one running on the ARM core of an xCORE-XA and one running on a single logical core of the xCORE tile. In this example there is no communication between the ARM core and the xCORE tile in order to provide a simple demonstration of using the USB functionality of the ARM core. The xCORE task in this example simply flashes an LED demonstrating that the xCORE tile can be used whilst the ARM core is running a USB application.

This example is implemented using 2 tasks which are partitioned so that one executes on the ARM core and one on the xCORE tile.

The tasks perform the following operations.

- A task running a USB HID keyboard example on the ARM core
- A task executing a loop and toggling a GPIO on the xCORE tile

This can be seen in the following task diagram.

Figure 2: Task diagram of xCORE-XA USB HID example

2.1 Declaring resource and setting up the ARM core and xCORE tile

The example code in this application note is split between the ARM core and the xCORE tile. The ARM code directory contains the source code for the simple USB keyboard example along with the source code for the USB library required to interface with the USB hardware of the ARM core in the xCORE-XA device.

The xCORE code contains a simple main function which drives an LED connected to the xCORE tile.

The Makefile and source code structure for building application for xCORE-XA based devices within the xTIMEcomposer development tools is detailed in the XMOS tools user guide and also the xCORE-XA development application note AN00141.

2.2 The xCORE tile application main() function

The main() function for the xCORE tile is contained within the file `main_xcore.xc` and is as follows,

```
int main() {
  while (1) {
    set_led_on();
    delay_seconds(1);
    set_led_off();
    delay_seconds(1);
  }
  return 0;
}
```

Looking at this function in more detail you can see the following:

- A single task executes an infinite loop
- An LED is toggled on and off
- A delay is added using the delay_seconds() library call

2.3 The ARM core application main() function

The main() function for the ARM core is contained within the file `main_arm.c` and is as follows,

```
int main(void)
{
  CMU_ClockEnable(cmuClock_GPIO, true);
  GPIO_PinModeSet(BUTTON, gpioModeInputPull, 1);
  GPIO_PinModeSet(ACTIVITY_LED, gpioModePushPull,  0);
  CMU_ClockSelectSet( cmuClock_HF, cmuSelect_HFXO );

  USBD_Init(&initstruct);

  /*
   * When using a debugger it is practical to uncomment the following three
   * lines to force host to re-enumerate the device.
   */
   //USBD_Disconnect();
   //USBTIMER_DelayMs(1000);
   //USBD_Connect();

  for (;;)
  {
  }
}
```

Looking at this function in more detail you can see the following:

- The GPIO clock for the ARM core is explicitly enabled
- The GPIO used for a button is set to pull up with the value 1
- The GPIO used for the USB activity LED is set to 0
- A library function is used to enable the high frequency crystal oscillator
- There is a call to the initialization function of the USB library
- There is code that can be enabled if running in a interactive debugging session to enable the USB device to enumerate correctly
- The function does not return it waits forever

2.4 Setting up ARM GPIO for the application

The GPIO on the ARM core is accessed from software using a port and pin identifier.

In this example the following GPIO ports and pins are used.

USB activity LED

```
#define ACTIVITY_LED            gpioPortB, 13
```

USB keyboard button

```
#define BUTTON                  gpioPortC, 8
```

Functions for accessing the GPIO of the ARM core are provided with the XMOS development tools and can be accessed using the following header file:

```
#include <em_gpio.h>
```

2.5 USB configuration for the ARM core

In order to use the USB library on the ARM core there are a set of defines that need to be used to configure the USB library. These are declared in the file usbconfig.h.

```
#define BUSPOWERED      /* Uncomment to build buspowered device */

#if defined(BUSPOWERED)
  #define USB_PWRSAVE_MODE (USB_PWRSAVE_MODE_ONSUSPEND | USB_PWRSAVE_MODE_ENTEREM2)
#else
  #define USB_PWRSAVE_MODE  (USB_PWRSAVE_MODE_ONVBUSOFF | USB_PWRSAVE_MODE_ONSUSPEND)
#endif

#define USB_USBC_32kHz_CLK   USB_USBC_32kHz_CLK_LFRCO

#define USB_DEVICE        /* Compile stack for device mode. */

/***************************************************************************
**                                                                        **
** Specify number of endpoints used (in addition to EP0).                 **
**                                                                        **
***************************************************************************/
#define NUM_EP_USED 1

/***************************************************************************
**                                                                        **
** Specify number of application timers you need.                         **
**                                                                        **
***************************************************************************/
#define NUM_APP_TIMERS 2
```

Looking at the defines you can see the following:

- The USB device is defined to be bus powered which is then used to set defines for the USB suspend and power on behaviour
- There is a define for the USB controller clock
- There is a define selecting that the software should run in USB device mode
- The number of endpoints in the USB HID device is declared as 1
- The number of timers used within the USB HID device is declared as 2

2.6 The ARM core IdleTimeout() function

The IdleTimeout() function is used to implement the functionality to report back to the host HID events at a rate specified by the host machine. These are setup using a standard USB device request for the HID class. This functionality is implemented using a hardware timer on the ARM core which will interrupt and call the IdleTimeout() function at a fixed rate.

```
static void IdleTimeout(void)
```

Once this function has been called on a timer interrupt the following code is executed which checks if there has been a keyboard event generated by the user on the device and if there is one queued, reports this back. Otherwise an empty key report is sent to the host to denote that no key has been pressed.

The code uses the USB library function USBD_Write to transfer a buffer from the ARM core to the USB host machine via USB.

```
int reportIndex;

/* If there is a keyboard event in the queue, we send it to the host. */
/* If not we send the previous key event which can be the empty key event. */
if ( !QueueEmpty() )
{
  /* A new keyboard event. */
  QueueGet( &reportIndex );
  lastReportIndex = reportIndex;
}
else
{
  /* The previous keyboard event. */
  reportIndex = lastReportIndex;
}

if ( reportIndex == -1 )
{
  /* Send an empty (key released) report */
  USBD_Write( INTR_IN_EP_ADDR, (void*) &noKeyReport,
              sizeof(KeyReport_TypeDef), NULL);
}
else
{
  /* Send a key pushed report */
  USBD_Write( INTR_IN_EP_ADDR, (void*) &reportTable[ reportIndex ],
              sizeof(KeyReport_TypeDef), NULL);
}
```

Once the idle timer has been serviced the IdleTimeout function reschedules a new timer event to fire an interrupt after the next time period has elapsed.

```
/* Schedule next idle event at current idle rate, idleRate unit is 4 ms. */
USBTIMER_Start( IDLE_TIMER, idleRate * 4,  IdleTimeout );
```

2.7 The ARM core ScanTimeout() function

The ScanTimeout() function is used to implement the hardware keyboard interface on the ARM core. This function is called at a fixed rate using a hardware timer which will interrupt the device and call the ScanTimeout() function when the timer condition is met.

```
static void ScanTimeout(void)
```

The main code in this function is used to handle and record a user key press event on the USB keyboard. In this example a key press is generated when a GPIO is toggled on the development board. The GPIO is connected to a simple push button in order to generate these events.

The code uses the standard GPIO access function to read from the GPIO and records if there has been a key press. The keyboard event will be stored in a queue to be processed by the IdleTimeout() function. If the USB host has set the USB HID idle rate to be 0 then the event is processed in this function and sent back to the host using the USBD_Write function.

```
bool pushed;

/* Check pushbutton */
pushed = GPIO_PinInGet( BUTTON ) == 0;

if (!keyPushed)
  GPIO_PinOutToggle(ACTIVITY_LED);

if (pushed != keyPushed)  /* Any change in keyboard status ? */
{
  if ( idleRate != 0 )    /* Put keyboard events into a queue. */
  {
    /* Put the kbd event in the queue, it will be retrieved and reported */
    /* to host in the idleRate timeout function.                         */
    if (pushed)
    {
      QueuePut( keySeqNo );
    }
    else
    {
      QueuePut( -1 ); /* Use -1 as a key release marker. */
    }
  }
  else /* idleRate == 0, send report immediately. */
  {
    if (pushed)
    {
      /* Send a key pushed report */
      USBD_Write(INTR_IN_EP_ADDR, (void*) &reportTable[ keySeqNo ],
                 sizeof(KeyReport_TypeDef), NULL);
    }
    else
    {
      /* Send an empty (key released) report */
      USBD_Write(INTR_IN_EP_ADDR,
                 (void*) &noKeyReport, sizeof(KeyReport_TypeDef), NULL);
    }
  }
}
```

As there is only a single button on this USB keyboard there is an array of keys which is used to generate key report data which is sent to the USB host. There is code in the ScanTimeout() function to keep track of the last key sent.

```
/* Keep track of the new keypush event (if any) */
if (pushed && !keyPushed)
{
  /* Advance to next position in report table */
  keySeqNo++;
  if (keySeqNo == (sizeof(reportTable) / sizeof(KeyReport_TypeDef)))
  {
    keySeqNo = 0;
  }

  GPIO_PinOutSet(ACTIVITY_LED);
}
keyPushed = pushed;
```

Once the keyboard scan timer has been serviced the ScanTimeout function reschedules a new timer event to fire an interrupt after the next time period has elapsed.

```
/* Restart keyboard scan timer */
USBTIMER_Start( SCAN_TIMER, SCAN_RATE, ScanTimeout);
```

2.8 The ARM core keyboard StateChange() function

The StateChange() function is used to keep track of USB state events on the device in order to update events that are occuring. This function is registered as a callback in the USB library and will be called whenever the device state changes. This function is used to start and stop the timers used for the HID idle timeout and the keyboard scan timeout. It will react to USB events such as the device being configured, de-configured and suspended.

```
static void StateChange(USBD_State_TypeDef oldState,
                        USBD_State_TypeDef newState)
{
  if (newState == USBD_STATE_CONFIGURED)
  {
    /* We have been configured, start HID functionality ! */
    if (oldState != USBD_STATE_SUSPENDED)   /* Resume ?   */
    {
      keySeqNo        = 0;
      keyPushed       = false;
      lastReportIndex = -1;
      idleRate        = DEFAULT_IDLE_RATE / 4;     /* Unit is 4 millisecond. */

      GPIO_PinOutSet(ACTIVITY_LED);

      QueueInit();
    }
    USBTIMER_Start( SCAN_TIMER, SCAN_RATE, ScanTimeout);
    if ( idleRate )
    {
      USBTIMER_Start( IDLE_TIMER, idleRate * 4,  IdleTimeout);
    }
  }

  else if ((oldState == USBD_STATE_CONFIGURED) &&
           (newState != USBD_STATE_SUSPENDED))
  {
    /* We have been de-configured, stop HID functionality */
    USBTIMER_Stop(SCAN_TIMER);
    USBTIMER_Stop(IDLE_TIMER);
    GPIO_PinOutClear(ACTIVITY_LED);
  }

  else if (newState == USBD_STATE_SUSPENDED)
  {
    /* We have been suspended, stop HID functionality */
    /* Reduce current consumption to below 2.5 mA.    */
    GPIO_PinOutClear(ACTIVITY_LED);
    USBTIMER_Stop(SCAN_TIMER);
    USBTIMER_Stop(IDLE_TIMER);
  }
}
```

2.9 The ARM core HID SetupCmd() function

The SetupCmd() function is used to process the USB HID specific requests sent to the USB device from the USB host. This function is a callback which is registered with the USB interface library and will be called when there is a USB setup packet which is not one of the standard USB requests handled in the USB library. This function receives a USB setup packet pointer which is passed from the USB library when called.

This callback allows the functionality required for a specific USB device class to be implemented outside of the base USB library. The function reports that it has successfully handled the request by returning USB_STATUS_OK.

```
static int SetupCmd(const USB_Setup_TypeDef *setup)
```

The GET_DESCRIPTOR request reports the HID descriptor back to the USB host, the code for this is implemented as follows.

```
case GET_DESCRIPTOR:
  /*********************/
  if ((setup->wValue >> 8) == USB_HID_REPORT_DESCRIPTOR)
  {
    USBD_Write(0, (void*) ReportDescriptor,
               EFM32_MIN(sizeof(ReportDescriptor), setup->wLength),
               NULL);
    retVal = USB_STATUS_OK;
  }
  else if ((setup->wValue >> 8) == USB_HID_DESCRIPTOR)
  {
    /* The HID descriptor might be misaligned ! */
    memcpy( hidDesc,
            &configDesc[ USB_CONFIG_DESCSIZE + USB_INTERFACE_DESCSIZE ],
            USB_HID_DESCSIZE );
    USBD_Write(0, hidDesc, EFM32_MIN(USB_HID_DESCSIZE, setup->wLength),
               NULL);
    retVal = USB_STATUS_OK;
  }
  break;
```

The SET_REPORT request posts a request from the USB host back to the USB device to report on information which can be used by the USB device.

```
case USB_HID_SET_REPORT:
  /*********************/
  if (((setup->wValue >> 8) == 2) &&            /* Output report */
      ((setup->wValue & 0xFF) == 0) &&          /* Report ID     */
      (setup->wIndex == 0) &&                   /* Interface no. */
      (setup->wLength == 1) &&                  /* Report length */
      (setup->Direction != USB_SETUP_DIR_IN))
  {
    USBD_Read(0, (void*) &tmpBuffer, 1, OutputReportReceived);
    retVal = USB_STATUS_OK;
  }
  break;
```

The GET_REPORT request is used to send a HID keyboard report back to the USB host. This containes the keyboard report information based on the information generated in the function ScanTimeout() which reads from the hardware keyboard and stores the current state internally.

```
case USB_HID_GET_REPORT:
  /*********************/
  if (((setup->wValue >> 8) == 1) &&            /* Input report  */
      ((setup->wValue & 0xFF) == 0) &&          /* Report ID     */
      (setup->wIndex == 0) &&                   /* Interface no. */
      (setup->wLength == 8) &&                  /* Report length */
      (setup->Direction == USB_SETUP_DIR_IN))
  {
    if (keyPushed)
    {
      /* Send a key pushed report */
      USBD_Write(0, (void*) &reportTable[ keySeqNo ],
                 sizeof(KeyReport_TypeDef), NULL);
    }
    else
    {
      /* Send an empty (key released) report */
      USBD_Write(0, (void*) &noKeyReport,
                 sizeof(KeyReport_TypeDef), NULL);
    }
    retVal = USB_STATUS_OK;
  }
  break;
```

The SET_IDLE request (as mentioned earlier in the IdleTimeout() function) is used by the host to set the polling rate of the USB device. The polling rate specified is stored in a global variable and used to configure the hardware timer used to generate events at this rate.

The hardware timer is used to generate interrupts and call the IdleTimeout() function when the timer condition is met. This allows a set of events to be generated at the rate the USB host has requested.

```
case USB_HID_SET_IDLE:
  /*********************/
  if (((setup->wValue & 0xFF) == 0) &&          /* Report ID     */
      (setup->wIndex == 0) &&                   /* Interface no. */
      (setup->wLength == 0) &&
      (setup->Direction != USB_SETUP_DIR_IN))
  {
    idleRate = setup->wValue >> 8;
    if ( ( idleRate != 0 ) && ( idleRate < ( POLL_RATE / 4 ) ) )
    {
      idleRate = POLL_RATE / 4;
    }
    USBTIMER_Stop( IDLE_TIMER );
    if ( idleRate != 0 )
    {
      IdleTimeout();
    }
    retVal = USB_STATUS_OK;
  }
  break;
```

The GET_IDLE request is used to report back the current idle polling rate that is set in the USB device. This information is not tracked by the USB host and at any point it can request this information from the device. The global idleRate variable is simply copied into a USB packet and sent to the USB host using the USB library.

```
case USB_HID_GET_IDLE:
  /*********************/
  if ((setup->wValue == 0) &&                    /* Report ID     */
      (setup->wIndex == 0) &&                    /* Interface no. */
      (setup->wLength == 1) &&
      (setup->Direction == USB_SETUP_DIR_IN))
  {
    *(uint8_t*)&tmpBuffer = idleRate;
    USBD_Write(0, (void*) &tmpBuffer, 1, NULL);
    retVal = USB_STATUS_OK;
  }
  break;
```

2.10 The ARM core HID OutputReportReceived() function

The OutputReportRecieved() function is called when a SET_REPORT request has been sent from the USB host to the USB device. This function is a callback which allows the USB HID to be notified from the USB host of state information useful to the USB device. In this example the information is not acted upon but for example with a full keyboard this would be where the USB host has requested an action such as CapsLock being turned on.

```
static int OutputReportReceived(USB_Status_TypeDef status,
                                uint32_t xferred,
                                uint32_t remaining)
{
  uint8_t *p = (uint8_t*) &tmpBuffer;
  (void) remaining;

  /* We have received new data for NumLock, CapsLock and ScrollLock LED's */
  if ((status == USB_STATUS_OK) && (xferred == 1))
  {
    // Update status leds here
  }
  return USB_STATUS_OK;
}
```

2.11 USB Descriptor setup for the ARM core

Any USB device contains a set of descriptors to allow the USB host to configure the driver in use on the host machine and also to report back device capability to any host applications which wish to query it. These descriptors follow a standard layout which are described in the USB specification and USB HID specification. The USB descriptors are defined in the file usbdescriptors.h.

2.12 HID Report Descriptor

```
static const char ReportDescriptor[69] __attribute__ ((aligned(4)))=
{
    0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)
    0x09, 0x06,                    // USAGE (Keyboard)
    0xa1, 0x01,                    // COLLECTION (Application)
    0x05, 0x07,                    //   USAGE_PAGE (Keyboard)
    0x19, 0xe0,                    //   USAGE_MINIMUM (Keyboard LeftControl)
    0x29, 0xe7,                    //   USAGE_MAXIMUM (Keyboard Right GUI)
    0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
    0x25, 0x01,                    //   LOGICAL_MAXIMUM (1)
    0x75, 0x01,                    //   REPORT_SIZE (1)
    0x95, 0x08,                    //   REPORT_COUNT (8)
    0x81, 0x02,                    //   INPUT (Data,Var,Abs)
    0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
    0x25, 0x01,                    //   LOGICAL_MAXIMUM (1)
    0x75, 0x01,                    //   REPORT_SIZE (1)
    0x95, 0x08,                    //   REPORT_COUNT (8)
    0x81, 0x01,                    //   INPUT (Cnst,Ary,Abs)
    0x19, 0x00,                    //   USAGE_MINIMUM (Reserved (no event indicated))
    0x29, 0x65,                    //   USAGE_MAXIMUM (Keyboard Application)
    0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
    0x25, 0x65,                    //   LOGICAL_MAXIMUM (101)
    0x75, 0x08,                    //   REPORT_SIZE (8)
    0x95, 0x06,                    //   REPORT_COUNT (6)
    0x81, 0x00,                    //   INPUT (Data,Ary,Abs)
    0x05, 0x08,                    //   USAGE_PAGE (LEDs)
    0x19, 0x01,                    //   USAGE_MINIMUM (Num Lock)
    0x29, 0x03,                    //   USAGE_MAXIMUM (Scroll Lock)
    0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
    0x25, 0x01,                    //   LOGICAL_MAXIMUM (1)
    0x75, 0x01,                    //   REPORT_SIZE (1)
    0x95, 0x03,                    //   REPORT_COUNT (3)
    0x91, 0x02,                    //   OUTPUT (Data,Var,Abs)
    0x75, 0x01,                    //   REPORT_SIZE (1)
    0x95, 0x05,                    //   REPORT_COUNT (5)
    0x91, 0x01,                    //   OUTPUT (Cnst,Ary,Abs)
    0xc0                           // END_COLLECTION
};
```

2.13 USB Device Descriptor

```
static const USB_DeviceDescriptor_TypeDef deviceDesc __attribute__ ((aligned(4)))=
{
  .bLength            = USB_DEVICE_DESCSIZE,
  .bDescriptorType    = USB_DEVICE_DESCRIPTOR,
  .bcdUSB             = 0x0200,
  .bDeviceClass       = 0,
  .bDeviceSubClass    = 0,
  .bDeviceProtocol    = 0,
  .bMaxPacketSize0    = USB_EP0_SIZE,
  .idVendor           = 0x20b1,
  .idProduct          = 0xff10,
  .bcdDevice          = 0x0000,
  .iManufacturer      = 1,
  .iProduct           = 2,
  .iSerialNumber      = 3,
  .bNumConfigurations = 1
};
```

2.14 USB Config Descriptor

```
static const uint8_t configDesc[] __attribute__ ((aligned(4)))=
{
  /*** Configuration descriptor ***/
  USB_CONFIG_DESCSIZE,    /* bLength                                */
  USB_CONFIG_DESCRIPTOR,  /* bDescriptorType                        */

  USB_CONFIG_DESCSIZE +   /* wTotalLength (LSB)                     */
  USB_INTERFACE_DESCSIZE +
  USB_HID_DESCSIZE +
  (USB_ENDPOINT_DESCSIZE * NUM_EP_USED),

  (USB_CONFIG_DESCSIZE +  /* wTotalLength (MSB)                     */
  USB_INTERFACE_DESCSIZE +
  USB_HID_DESCSIZE +
  (USB_ENDPOINT_DESCSIZE * NUM_EP_USED))>>8,

  1,                      /* bNumInterfaces                         */
  1,                      /* bConfigurationValue                    */
  0,                      /* iConfiguration                         */

#if defined(BUSPOWERED)
  CONFIG_DESC_BM_RESERVED_D7,    /* bmAttrib: Bus powered           */
#else
  CONFIG_DESC_BM_RESERVED_D7 |   /* bmAttrib: Self powered          */
  CONFIG_DESC_BM_SELFPOWERED,
#endif

  CONFIG_DESC_MAXPOWER_mA( 100 ), /* bMaxPower: 100 mA             */

  /*** Interface descriptor ***/
  USB_INTERFACE_DESCSIZE, /* bLength               */
  USB_INTERFACE_DESCRIPTOR,/* bDescriptorType      */
  0,                      /* bInterfaceNumber      */
  0,                      /* bAlternateSetting     */
  NUM_EP_USED,            /* bNumEndpoints         */
  0x03,                   /* bInterfaceClass (HID) */
  0,                      /* bInterfaceSubClass    */
  1,                      /* bInterfaceProtocol    */
  0,                      /* iInterface            */

  /*** HID descriptor ***/
  USB_HID_DESCSIZE,       /* bLength               */
  USB_HID_DESCRIPTOR,     /* bDescriptorType       */
  0x11,                   /* bcdHID (LSB)          */
  0x01,                   /* bcdHID (MSB)          */
  0,                      /* bCountryCode          */
  1,                      /* bNumDescriptors       */
  USB_HID_REPORT_DESCRIPTOR,     /* bDecriptorType        */
  sizeof( ReportDescriptor ),    /* wDescriptorLength(LSB)*/
  sizeof( ReportDescriptor )>>8, /* wDescriptorLength(MSB)*/

  /*** Endpoint descriptor ***/
  USB_ENDPOINT_DESCSIZE,  /* bLength               */
  USB_ENDPOINT_DESCRIPTOR,/* bDescriptorType       */
```

```
  INTR_IN_EP_ADDR,         /* bEndpointAddress (IN) */
  USB_EPTYPE_INTR,         /* bmAttributes          */
  USB_MAX_EP_SIZE,         /* wMaxPacketSize (LSB)  */
  0,                       /* wMaxPacketSize (MSB)  */
  POLL_RATE,               /* bInterval             */
};
```

2.15 USB String Descriptor

```
STATIC_CONST_STRING_DESC_LANGID( langID, 0x04, 0x09 );
STATIC_CONST_STRING_DESC( iManufacturer, 'X','M','O','S',' ',         \
                                         'x','C','O','R','E',' ','X','A' );
STATIC_CONST_STRING_DESC( iProduct     , 'x','C','O','R','E',' ','X','A',' ','U','S','B', \
                                         ' ','H','I','D',' ','K','e','y','b', \
                                         'o','a','r','d' );
STATIC_CONST_STRING_DESC( iSerialNumber, '0','0','0','0','0','0',            \
                                         '0','0','1','2','3','4' );

static const void * const strings[] =
{
  &langID,
  &iManufacturer,
  &iProduct,
  &iSerialNumber
};
```

2.16 USB library initialization and callbacks

The file usbdescriptors.h also contains the USB library callback registration for USB events this USB device would like to be notified about and also the initialization structure for the USB library which sets up the various descriptors used within the USB device.

```
static const USBD_Callbacks_TypeDef callbacks =
{
  .usbReset        = NULL,
  .usbStateChange  = StateChange,
  .setupCmd        = SetupCmd,
  .isSelfPowered   = NULL,
  .sofInt          = NULL
};

static const USBD_Init_TypeDef initstruct =
{
  .deviceDescriptor    = &deviceDesc,
  .configDescriptor    = configDesc,
  .stringDescriptors   = strings,
  .numberOfStrings     = sizeof(strings)/sizeof(void*),
  .callbacks           = &callbacks,
  .bufferingMultiplier = bufferingMultiplier,
  .reserved            = 0
};
```

2.17 USB HID keyboard report

The USB hid keyboard report contains the key mapping which is sent to the USB host when the button is not being pressed and also the buffer of key reports which will be cycled through when the key is pressed on the USB device.

The key mapping format is described in the USB HID specification.

```
static const KeyReport_TypeDef noKeyReport =
{ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

/* A sequence of keystroke input reports. */
EFM32_ALIGN(4)
static const KeyReport_TypeDef reportTable[] __attribute__ ((aligned(4)))=
{
  { 0x02, 0x00, 0x1B, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* 'X'   */
  { 0x02, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* 'M'   */
  { 0x02, 0x00, 0x12, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* 'O'   */
  { 0x02, 0x00, 0x16, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* 'S'   */
  { 0x00, 0x00, 0x2C, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* space */
  { 0x00, 0x00, 0x1B, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* 'x'   */
  { 0x02, 0x00, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* 'C'   */
  { 0x02, 0x00, 0x12, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* 'O' */
  { 0x02, 0x00, 0x15, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* 'R'   */
  { 0x02, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* 'E'   */
  { 0x00, 0x00, 0x2D, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* '-'   */
  { 0x02, 0x00, 0x1B, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* 'X'   */
  { 0x02, 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* 'A'   */
  { 0x00, 0x00, 0x2C, 0x00, 0x00, 0x00, 0x00, 0x00 },  /* space */
};
```

APPENDIX A - Example Hardware Setup

This application example is designed to run on the xCORE-XA core module board. The xCORE-XA core module board should be connected to both power and have the development adapters connected to a host machine to allow program download.

In this example a simple push button has been soldered onto the xCORE-XA core module board between port C8 and GND. This will report a button press when the by driving the GPIO port low when the button is pressed. There is no requirement to use this specific GPIO port but using another one will require the software to be modified to select the different GPIO port for the button.

This can be seen in the following image.

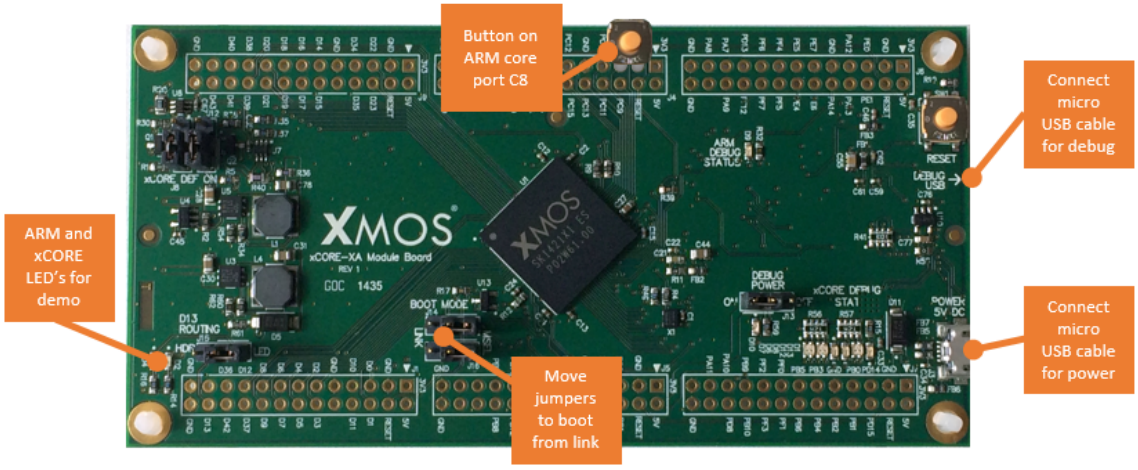

Figure 3: XMOS xCORE-XA core module board setup

The hardware should be configured as displayed above for this example:

- Both the power and debug adapter USB cables should be connected
- The boot mode of the board should have the jumpers selecting boot from msel
- The ARM led will flash when the example is executing
- The xCORE led will flash when the example is executing
- A push button should be connected onto ARM core GPIO port C8

APPENDIX B - Launching the example application

Once the example has been built either from the command line using xmake or via the build mechanism of xTIMEcomposer Studio the application can be executed on the xCORE-XA core module board.

Once built there will be a `bin` directory within the project which contains the binary for both the ARM core and xCORE tile. The xCORE binary has a XMOS standard .xe extension.

In order to allow access to the debug interface of the ARM core the SEGGER gdb server application needs to be running. The documentation for setting up and executing this can be found in the XMOS development tools user guide and the xCORE-XA development chapter.

B.1 Launching from the command line

From the command line the `xrun` tool is used to download code to both the ARM core and xCORE tile. Changing into the bin directory of the project will allow the application to be executed on the xCORE-XA microcontroller as follows:

```
> xrun app_xcore_xa_usb_keyboard            <-- Download and execute the ARM code
> xrun app_xcore_xa_usb_keyboard.xe         <-- Download and execute the xCORE code
```

Once these two commands have been executed the LED's connected to ARM core and xCORE tile will be flashing.

B.2 Launching from xTIMEcomposer Studio

From xTIMEcomposer Studio there is the run mechanism to download code to both the ARM core and xCORE tile. By selecting the ARM binary contained within the project bin directory, right click and then run as ARM application, the application will be downloaded and executed on the ARM core. Selecting the xCORE binary, right click and then run as xCORE application will perform the same operation for the xCORE tile.

Once these two commands have been executed the LED's connected to ARM core and xCORE tile will be flashing.

B.3 Using the USB HID keyboard

Once the application has been launched either from the command line or from within xTIMEcomposer studio there will be a new USB HID device enumerated on your host machine.

Selecting an open window that can accept text input will allow you to write characters into the window by pressing the button on the xCORE-XA module board.

As the button is pressed the text "XMOS xCORE-XA" will be repeatedly printed to the active window on your host machine.

APPENDIX C - References

XMOS Tools User Guide

http://www.xmos.com/published/xtimecomposer-user-guide

XMOS xCORE Programming Guide

http://www.xmos.com/published/xmos-programming-guide

USB HID Class Specification, USB.org:

http://www.usb.org/developers/devclass_docs/HID1_11.pdf

USB 2.0 Specification

http://www.usb.org/developers/docs/usb20_docs/usb_20_081114.zip

APPENDIX D - Full source code listing

D.1 Source code for main_xcore.xc

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved
#include <platform.h>
#include <xs1.h>

out port led = XS1_PORT_1F;

void set_led_on() {
  led <: 0;
}

void set_led_off() {
  led <: 1;
}

int main() {
  while (1) {
    set_led_on();
    delay_seconds(1);
    set_led_off();
    delay_seconds(1);
  }
  return 0;
}
```

D.2 Source code for main_arm.c

```
/*
 * XA_hid_keyboard.c
 *
 *  Created on: 9 Apr 2014
 *      Author: matt
 */

/**************************************************************************//**
 * @file main.c
 * @brief USB HID keyboard device example.
 * @author Energy Micro AS
 * @version 3.20.3
 ******************************************************************************
 * @section License
 * <b>(C) Copyright 2014 Silicon Labs, http://www.silabs.com</b>
 ******************************************************************************
 *
 * This file is licensed under the Silicon Labs Software License Agreement. See
 * "http://developer.silabs.com/legal/version/v11/Silicon_Labs_Software_License_Agreement.txt"
 * for details. Before using this software for any purpose, you must agree to the
 * terms of that agreement.
 *
 *****************************************************************************/

#include "em_device.h"
#include "em_usb.h"
#include "em_cmu.h"
#include "em_gpio.h"

#include <stdio.h>

/**************************************************************************//**
 *
 * This example shows how a HID keyboard can be implemented.
```

```
 *
 ******************************************************************************/

/*** Typedef's and defines. ***/

#define SCAN_TIMER              1       /* Timer used to scan keyboard. */
#define SCAN_RATE               50

#define IDLE_TIMER              0       /* Timer used to implement the idle-  */
#define DEFAULT_IDLE_RATE       500     /* rate defined in the HID class doc. */

#define POLL_RATE               24      /* The bInterval reported with the    */
                                        /* interrupt IN endpoint descriptor.  */

#define INTR_IN_EP_ADDR         0x81

#define ACTIVITY_LED            gpioPortB, 13
#define BUTTON                  gpioPortC, 8

/*** Function prototypes. ***/

static int  OutputReportReceived(USB_Status_TypeDef status,
                                 uint32_t xferred,
                                 uint32_t remaining);
static int  SetupCmd(const USB_Setup_TypeDef *setup);
static void StateChange(USBD_State_TypeDef oldState,
                        USBD_State_TypeDef newState);

static bool QueueEmpty( void );
static bool QueueFull( void );
static bool QueueGet( int *element );
static void QueueInit( void );
static bool QueuePut( int element );

/*** Include device descriptor definitions. ***/
#include "usbdescriptors.h"

/*** Variables ***/

static int      keySeqNo;           /* Current position in report table. */
static bool     keyPushed;          /* Current pushbutton status. */
static int      lastReportIndex;    /* Last report sent to host. */
static uint8_t  idleRate;
static uint32_t tmpBuffer;

/******************************************************************************//**
 * @brief main - the entrypoint after reset.
 *****************************************************************************/
int main(void)
{
  CMU_ClockEnable(cmuClock_GPIO, true);
  GPIO_PinModeSet(BUTTON, gpioModeInputPull, 1);
  GPIO_PinModeSet(ACTIVITY_LED, gpioModePushPull,  0);
  CMU_ClockSelectSet( cmuClock_HF, cmuSelect_HFXO );

  USBD_Init(&initstruct);

  /*
   * When using a debugger it is practical to uncomment the following three
   * lines to force host to re-enumerate the device.
   */
   //USBD_Disconnect();
   //USBTIMER_DelayMs(1000);
   //USBD_Connect();

  for (;;)
  {
  }
}

/******************************************************************************//**
 * @brief
 *   Timeout function for the idleRate timer. The idleRate is set by the host
 *   device driver with the SET_IDLE setup command.
 *   If idleRate is set to 0 the idleRate timer is completely stopped.
```

```
 *   This function will always send a keyboard report to host.
 ******************************************************************************/
/* IdleTimeout() */
static void IdleTimeout(void)
{
  int reportIndex;

  /* If there is a keyboard event in the queue, we send it to the host. */
  /* If not we send the previous key event which can be the empty key event. */
  if ( !QueueEmpty() )
  {
    /* A new keyboard event. */
    QueueGet( &reportIndex );
    lastReportIndex = reportIndex;
  }
  else
  {
    /* The previous keyboard event. */
    reportIndex = lastReportIndex;
  }

  if ( reportIndex == -1 )
  {
    /* Send an empty (key released) report */
    USBD_Write( INTR_IN_EP_ADDR, (void*) &noKeyReport,
                sizeof(KeyReport_TypeDef), NULL);
  }
  else
  {
    /* Send a key pushed report */
    USBD_Write( INTR_IN_EP_ADDR, (void*) &reportTable[ reportIndex ],
                sizeof(KeyReport_TypeDef), NULL);
  }

  /* Schedule next idle event at current idle rate, idleRate unit is 4 ms. */
  USBTIMER_Start( IDLE_TIMER, idleRate * 4,  IdleTimeout );
}

/******************************************************************************//**
 * @brief
 *   Timeout function for keyboard scan timer.
 *   Scan keyboard to check for key press/release events.
 *   This function is called at a fixed rate.
 ******************************************************************************/
/* ScanTimeout() */
static void ScanTimeout(void)
{
  bool pushed;

  /* Check pushbutton */
  pushed = GPIO_PinInGet( BUTTON ) == 0;

  if (!keyPushed)
    GPIO_PinOutToggle(ACTIVITY_LED);

  if (pushed != keyPushed)  /* Any change in keyboard status ? */
  {
    if ( idleRate != 0 )    /* Put keyboard events into a queue. */
    {
      /* Put the kbd event in the queue, it will be retrieved and reported */
      /* to host in the idleRate timeout function.                         */
      if (pushed)
      {
        QueuePut( keySeqNo );
      }
      else
      {
        QueuePut( -1 ); /* Use -1 as a key release marker. */
      }
    }
    else /* idleRate == 0, send report immediately. */
    {
      if (pushed)
      {
        /* Send a key pushed report */
```

```
        USBD_Write(INTR_IN_EP_ADDR, (void*) &reportTable[ keySeqNo ],
                   sizeof(KeyReport_TypeDef), NULL);
      }
      else
      {
        /* Send an empty (key released) report */
        USBD_Write(INTR_IN_EP_ADDR,
                   (void*) &noKeyReport, sizeof(KeyReport_TypeDef), NULL);
      }
    }
  }

  /* Keep track of the new keypush event (if any) */
  if (pushed && !keyPushed)
  {
    /* Advance to next position in report table */
    keySeqNo++;
    if (keySeqNo == (sizeof(reportTable) / sizeof(KeyReport_TypeDef)))
    {
      keySeqNo = 0;
    }

    GPIO_PinOutSet(ACTIVITY_LED);
  }
  keyPushed = pushed;

  /* Restart keyboard scan timer */
  USBTIMER_Start( SCAN_TIMER, SCAN_RATE, ScanTimeout);
}

/**************************************************************************//**
 * @brief
 *   Callback function called each time the USB device state is changed.
 *   Starts HID operation when device has been configured by USB host.
 *
 * @param[in] oldState The device state the device has just left.
 * @param[in] newState The new device state.
 *****************************************************************************/
/* StateChange() */
static void StateChange(USBD_State_TypeDef oldState,
                        USBD_State_TypeDef newState)
{
  if (newState == USBD_STATE_CONFIGURED)
  {
    /* We have been configured, start HID functionality ! */
    if (oldState != USBD_STATE_SUSPENDED)   /* Resume ?   */
    {
      keySeqNo        = 0;
      keyPushed       = false;
      lastReportIndex = -1;
      idleRate        = DEFAULT_IDLE_RATE / 4;     /* Unit is 4 millisecond. */

      GPIO_PinOutSet(ACTIVITY_LED);

      QueueInit();
    }
    USBTIMER_Start( SCAN_TIMER, SCAN_RATE, ScanTimeout);
    if ( idleRate )
    {
      USBTIMER_Start( IDLE_TIMER, idleRate * 4,  IdleTimeout);
    }
  }

  else if ((oldState == USBD_STATE_CONFIGURED) &&
           (newState != USBD_STATE_SUSPENDED))
  {
    /* We have been de-configured, stop HID functionality */
    USBTIMER_Stop(SCAN_TIMER);
    USBTIMER_Stop(IDLE_TIMER);
    GPIO_PinOutClear(ACTIVITY_LED);
  }

  else if (newState == USBD_STATE_SUSPENDED)
  {
    /* We have been suspended, stop HID functionality */
```

```
    /* Reduce current consumption to below 2.5 mA.    */
    GPIO_PinOutClear(ACTIVITY_LED);
    USBTIMER_Stop(SCAN_TIMER);
    USBTIMER_Stop(IDLE_TIMER);
  }
}

/**************************************************************************//**
 * @brief
 *   Handle USB setup commands. Implements HID class specific commands.
 *
 * @param[in] setup Pointer to the setup packet received.
 *
 * @return USB_STATUS_OK if command accepted.
 *         USB_STATUS_REQ_UNHANDLED when command is unknown, the USB device
 *         stack will handle the request.
 *****************************************************************************/
/* SetupCmd() */
static int SetupCmd(const USB_Setup_TypeDef *setup)
{
  STATIC_UBUF( hidDesc, USB_HID_DESCSIZE );

  int retVal = USB_STATUS_REQ_UNHANDLED;

  if ((setup->Type == USB_SETUP_TYPE_STANDARD) &&
      (setup->Direction == USB_SETUP_DIR_IN) &&
      (setup->Recipient == USB_SETUP_RECIPIENT_INTERFACE))
  {
    /* A HID device must extend the standard GET_DESCRIPTOR command   */
    /* with support for HID descriptors.                              */
    switch (setup->bRequest)
    {
    case GET_DESCRIPTOR:
      /*********************/
      if ((setup->wValue >> 8) == USB_HID_REPORT_DESCRIPTOR)
      {
        USBD_Write(0, (void*) ReportDescriptor,
                   EFM32_MIN(sizeof(ReportDescriptor), setup->wLength),
                   NULL);
        retVal = USB_STATUS_OK;
      }
      else if ((setup->wValue >> 8) == USB_HID_DESCRIPTOR)
      {
        /* The HID descriptor might be misaligned ! */
        memcpy( hidDesc,
                &configDesc[ USB_CONFIG_DESCSIZE + USB_INTERFACE_DESCSIZE ],
                USB_HID_DESCSIZE );
        USBD_Write(0, hidDesc, EFM32_MIN(USB_HID_DESCSIZE, setup->wLength),
                   NULL);
        retVal = USB_STATUS_OK;
      }
      break;
    }
  }

  else if ((setup->Type == USB_SETUP_TYPE_CLASS) &&
           (setup->Recipient == USB_SETUP_RECIPIENT_INTERFACE))
  {
    /* Implement the necessary HID class specific commands.           */
    switch (setup->bRequest)
    {
    case USB_HID_SET_REPORT:
      /*********************/
      if (((setup->wValue >> 8) == 2) &&            /* Output report */
          ((setup->wValue & 0xFF) == 0) &&          /* Report ID     */
          (setup->wIndex == 0) &&                   /* Interface no. */
          (setup->wLength == 1) &&                  /* Report length */
          (setup->Direction != USB_SETUP_DIR_IN))
      {
        USBD_Read(0, (void*) &tmpBuffer, 1, OutputReportReceived);
        retVal = USB_STATUS_OK;
      }
      break;

    case USB_HID_GET_REPORT:
```

```
      /*******************/
      if (((setup->wValue >> 8) == 1) &&            /* Input report  */
          ((setup->wValue & 0xFF) == 0) &&          /* Report ID     */
          (setup->wIndex == 0) &&                   /* Interface no. */
          (setup->wLength == 8) &&                  /* Report length */
          (setup->Direction == USB_SETUP_DIR_IN))
      {
        if (keyPushed)
        {
          /* Send a key pushed report */
          USBD_Write(0, (void*) &reportTable[ keySeqNo ],
                     sizeof(KeyReport_TypeDef), NULL);
        }
        else
        {
          /* Send an empty (key released) report */
          USBD_Write(0, (void*) &noKeyReport,
                     sizeof(KeyReport_TypeDef), NULL);
        }
        retVal = USB_STATUS_OK;
      }
      break;

    case USB_HID_SET_IDLE:
      /*******************/
      if (((setup->wValue & 0xFF) == 0) &&          /* Report ID     */
          (setup->wIndex == 0) &&                   /* Interface no. */
          (setup->wLength == 0) &&
          (setup->Direction != USB_SETUP_DIR_IN))
      {
        idleRate = setup->wValue >> 8;
        if ( ( idleRate != 0 ) && ( idleRate < ( POLL_RATE / 4 ) ) )
        {
          idleRate = POLL_RATE / 4;
        }
        USBTIMER_Stop( IDLE_TIMER );
        if ( idleRate != 0 )
        {
          IdleTimeout();
        }
        retVal = USB_STATUS_OK;
      }
      break;

    case USB_HID_GET_IDLE:
      /*******************/
      if ((setup->wValue == 0) &&                   /* Report ID     */
          (setup->wIndex == 0) &&                   /* Interface no. */
          (setup->wLength == 1) &&
          (setup->Direction == USB_SETUP_DIR_IN))
      {
        *(uint8_t*)&tmpBuffer = idleRate;
        USBD_Write(0, (void*) &tmpBuffer, 1, NULL);
        retVal = USB_STATUS_OK;
      }
      break;
    }
  }

  return retVal;
}

/*****************************************************************************//**
 * @brief
 *   Callback function called when the data stage of a USB_HID_SET_REPORT
 *   setup command has completed.
 *
 * @param[in] status    Transfer status code.
 * @param[in] xferred   Number of bytes transferred.
 * @param[in] remaining Number of bytes not transferred.
 *
 * @return USB_STATUS_OK.
 ******************************************************************************/
/* OutputReportReceived() */
static int OutputReportReceived(USB_Status_TypeDef status,
```

```
                                uint32_t xferred,
                                uint32_t remaining)
{
  uint8_t *p = (uint8_t*) &tmpBuffer;
  (void) remaining;

  /* We have received new data for NumLock, CapsLock and ScrollLock LED's */
  if ((status == USB_STATUS_OK) && (xferred == 1))
  {
    // Update status leds here
  }
  return USB_STATUS_OK;
}

/* Minimal circular buffer implementation. */

#define QUEUE_SIZE 16             /* Must be 2^n !! */

typedef struct ringBuffer_t
{
  unsigned int putIdx;
  unsigned int getIdx;
  int          buf[ QUEUE_SIZE ];
} ringBuffer_t;

ringBuffer_t queue;

static bool QueueEmpty( void )
{
  return ( queue.putIdx - queue.getIdx ) == 0;
}

static bool QueueFull( void )
{
  return ( queue.putIdx - queue.getIdx ) >= QUEUE_SIZE;
}

static bool QueueGet( int *element )
{
  if ( !QueueEmpty() )
  {
    *element = queue.buf[ queue.getIdx++ & (QUEUE_SIZE - 1) ];
    return true;
  }
  return false;
}

static void QueueInit( void )
{
  queue.getIdx = 0;
  queue.putIdx = 0;
}

static bool QueuePut( int element )
{
  if ( !QueueFull() )
  {
    queue.buf[ queue.putIdx++ & (QUEUE_SIZE - 1) ] = element;
    return true;
  }
  return false;
}
```