# AN00136: Example USB Vendor Specific Device

Publication Date: 2024/9/26
Document Number: XM-006260-AN v3.0.0

IN THIS DOCUMENT

1 Introduction . . . 1
2 Application detail . . . 4
3 Hardware setup . . . 11
4 Host detail . . . 12
5 Running the example . . . 13
6 Advanced example . . . 15
7 Further reading . . . 17

## 1 Introduction

### 1.1 USB Basics

USB class specifications define standardised protocols for different types of USB devices, ensuring compatibility across various platforms and devices. Each USB class specification describes how a particular type of device should behave, what descriptors it should use, and how data transfers should occur. Examples of USB Class Specifications include Audio, Mass Storage and Human Interface Device.

However, some devices do not naturally fit into this class framework and the USB specification allows for the creation of completely custom USB devices which do not conform to any of the USB device class standards.

There are many reasons why a vendor specific implementation might be used over a standard class, including, but not limited to, the following:

- Custom Functionality: Unique device features that don't fit within existing USB classes.
- Performance: Optimised data transfer and lower latency for specific use cases.
- Proprietary Technology: Protection of intellectual property and confidential protocols.
- Flexibility: Full control over device behavior and protocol, allowing customization.
- Legacy Support: Maintain compatibility with existing systems or protocols.
- Advanced Features: More complex or real-time data handling beyond class limitations.
- Avoid Class Overhead: Simpler implementation by bypassing unnecessary class complexities.
- Differentiation: Create unique product features for competitive advantage.
- Driver Control: Custom drivers allow consistent behavior across platforms.

Examples of such devices might include:

- Adapters which bridge custom debug interfaces to a host PC, for example *XMOS xSCOPE*
- Devices which control a variety of custom interfaces from a host PC

- Systems which stream large amounts of captured data to a host PC

These devices typically implement a custom command allowing the host to instruct it to perform specific operations.

Since the operation of a vendor specific devices is entirely specified by the developer, no native support can be included with any operating system. The developer is responsible for building their own driver and host application.

A host's operating system will typically provide a method of programmatically accessing USB devices, with libraries such as *libusb*, *pyUSB* and *openUSB* providing cross platform API's wrapping these access methods.

Windows operating systems provide a unique obstacle in the requirement of a driver to be loaded against a device which the host application uses to access the device.

The flexible nature of the *xcore* architecture lends itself to these custom applications. Furthermore the *xcore* allows for high-bandwidth, low-latency communications with a fast development time.

## 1.2 This note

This application note describes how to create a vendor specific USB device for a *xcore* device. An accompanying example application (`app_an00136`) is provided that uses the *XMOS USB Device Library* (lib_xud).

The example application and associated host application(s) demonstrate USB bulk transfers running over high speed USB. The example application also deals with the standard requests associated with this type of USB device.

The application includes simple data transfers, transmitting and receiving buffers to/from the USB host.

**Note:** This example uses the open source *libusb* host library and Windows driver to allow the device to be accessed from the host machine. On other host platforms supported by this application example a host driver is not required to interact with libusb.

## 1.3 Required hardware

The example code accompanying this application note has been written for the *XK-EVK-XU316* board (*xcore.ai* device), however, since there are no hardware interfaces used (other than USB), it can be trivially ported to other hardware platforms.

## 1.4 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* and *C* languages.

Documentation related to these aspects which are not specific to this application note are linked to in *Further Reading*.

- For the full API listing of the XMOS USB Device (XUD) Library and for information of designing devices using this library please see the document XMOS USB Device (XUD) Library[1].

[1] http://www.xmos.com/file/lib_xud?version=latest

## 1.5 Block diagram

Fig. 1 depicts a system block diagram showing the fundamental functional units of the design.

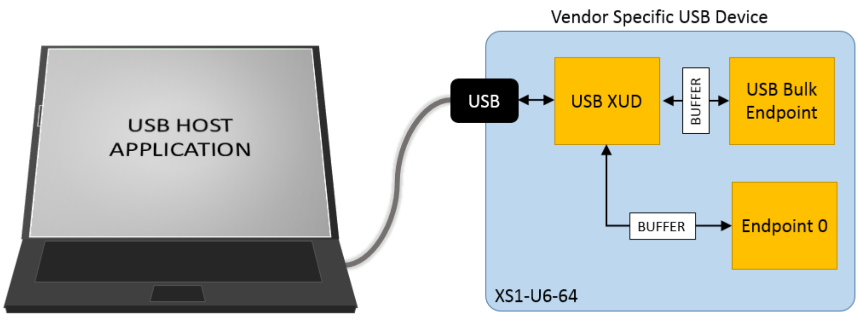

Fig. 1: System block diagram

## 1.6 Endpoint Types

A vendor specific device can contain a number of endpoints and endpoint types. When building a vendor specific device some consideration should be given to endpoint type selection and the transfer mechanisms they employ.

These can be chosen based on the description given in Table 1.

Table 1: Endpoint type descriptions

| Endpoint type | Description | Use case |
|---|---|---|
| Control | ▶ Low transfer rate<br>▶ No need of new endpoint which can reduce application footprint | Configuration of the device |
| Bulk | ▶ High (but variable) transfer rate<br>▶ Guaranteed data integrity | Large amount of data transfer |
| Interrupt | ▶ Very low transfer rate<br>▶ Frequency guaranteed | Real time constraints and low bandwidth transfer |
| Isochronous | ▶ Guarantee of timing<br>▶ Data integrity not guaranteed | Streaming application e.g. video |

**Note:** The provided application provides an example usage of a control and two bulk endpoints (one in each direction).

# 2 Application detail

The application accompanying this note uses the *XMOS* USB Device (XUD) library to provide a simple program that creates a basic vendor specific device which responds to data transfer requests from the host PC.

The application comprises three tasks running on separate threads of the *xcore* device.

The tasks perform the following operations:

- A task containing the USB library functionality to communicate over USB
- A task implementing Endpoint 0 responding to standard USB control requests
- A task implementing the application code for our custom bulk interface

Fig. 2 shows the task and communication structure for the application example. Tasks communicate via *xCONNECT* channels, denoted by the arrows in the diagram.

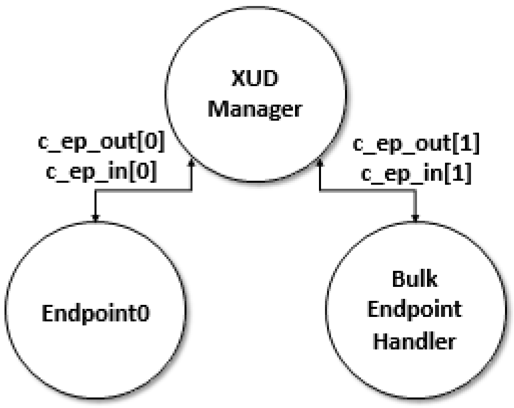

Fig. 2: Task diagram for vendor specific device

## 2.1 CMakeLists.txt additions

*XMOS* applications use the xcommon-cmake build and dependency management system. This is bundled with the *XMOS* XTC tools.

In order for an application to use the USB library, `lib_xud` must be added to the to the application's dependency list in *CMakeLists.txt*:

```
set(APP_DEPENDENT_MODULES   "lib_xud")
```

The application can then access USB functions via the `xud_device.h` header file:

```
#include "xud_device.h"
```

## 2.2 Setting up the USB library

`app.xc` contains the core application implementation for the USB vendor specific device.

For convenience two defines are created for for the endpoint count of the device. These are used in various points in the code:

```
#define EP_COUNT_OUT   (2)
#define EP_COUNT_IN    (2)
```

Initially the code declares tables that describe the endpoint types, as required by `lib_xud`.

This example has bi-directional communication with the host via the mandatory control endpoint 0. It also has a bulk endpoint IN and OUT endpoint. Note, USB nomenclature dictates traffic direction is always described from the point view of the host.

```
XUD_EpType epTypeTableOut[EP_COUNT_OUT] = {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE, XUD_EPTYPE_BUL | XUD_STATUS_
↪ENABLE};
XUD_EpType epTypeTableIn[EP_COUNT_IN] =   {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE, XUD_EPTYPE_BUL | XUD_STATUS_
↪ENABLE};
```

These defines and tables are later are passed to the `XUD_Main()` function from `main()`.

Following *XMOS* convention an optional header file, `xud_conf.h` is used to configure options for `lib_xud`. This file is automatically detected by the build system.

The contents of this file simply inform `lib_xud` what tile it is to be run on. This is not strictly necessary since it replicates the default value, but is included for completeness.

```
#define USB_TILE tile[0]
```

## 2.3 Application main() function

Like all *C* and *XC* programs, application entry is at `main()`. A source code listing for the application `main()` function is shown below:

```
int main()
{
    /* Declare channels for each endpoint */
    chan c_ep_out[EP_COUNT_OUT], c_ep_in[EP_COUNT_IN];

    par
    {
        on USB_TILE: XUD_Main(c_ep_out, EP_COUNT_OUT, c_ep_in, EP_COUNT_IN,
                      null, epTypeTableOut, epTypeTableIn,
                      XUD_SPEED_HS, XUD_PWR_BUS);

        on USB_TILE: Endpoint0(c_ep_out[0], c_ep_in[0]);

#ifdef BENCHMARK
        on USB_TILE: bulk_endpoint(c_ep_in[1]);
#else
        on USB_TILE: bulk_endpoint(c_ep_out[1], c_ep_in[1]);
#endif
    }

    return 0;
}
```

Inspecting this function should yield the following observations:

- The *par* functionality describes running three separate tasks/threads in parallel, these are:
  - A function to configure and execute the USB library: `XUD_Main()`
  - A function to run the Endpoint 0 code: `Endpoint0()`
  - A function to deal with the custom bulk endpoints `bulk_endpoint()`
- The define `USB_TILE` describes the tile on which the individual tasks will run
- All tasks run on the same tile (this is requirement of `lib_xud`)
- The *xCONNECT* communication channels used by the application are declared at the beginning of `main()`
- The endpoint type tables, discussed earlier, are passed into the function `XUD_Main()`
- Some additional parameters are also passed to `XUD_Main()`. Refer to `lib_xud` for full documentation.

**Note:** There are two implementations of `bulk_endpoint()` in the application, guarded by `BENCHMARK`. This is discussed later in this document.

## 2.4 USB device descriptor

A **USB device descriptor** is a structured data block that provides essential information about a USB device to the host when the device is connected. This information allows the host to recognize the device and determine how to communicate with it.

Key details provided by a USB device descriptor include:

1. **Vendor ID (VID)**: A unique identifier assigned to the manufacturer of the USB device.
2. **Product ID (PID)**: A unique identifier assigned to the specific product made by the manufacturer.
3. **Device Class, Subclass, and Protocol**: These fields categorize the type of device (e.g., mass storage, human interface device like a keyboard or mouse, etc.) and help the host load the appropriate driver.
4. **Device Release Number**: The version of the device's firmware or hardware.
5. **Maximum Packet Size**: The size of the data packets the device can handle in one transaction.
6. **Number of Configurations**: How many different configurations (sets of interfaces and endpoints) the device supports.
7. **Manufacturer, Product, and Serial Number Strings**: These fields are optional but provide human-readable information about the device (e.g., manufacturer name, product name).

When a USB device is connected, the host reads the device descriptor as part of the enumeration process, which allows it to load the proper drivers and establish communication.

The application's device descriptor is declared in `endpoint0.xc` and is listed below:

```
static unsigned char devDesc[] =
{
    0x12,                     /* 0  bLength */
    USB_DESCTYPE_DEVICE,      /* 1  bdescriptorType */
    0x00,                     /* 2  bcdUSB */
    0x02,                     /* 3  bcdUSB */
    USB_CLASS_VENDOR_SPECIFIC,/* 4  bDeviceClass (from xud_std_descriptors.h) */
    0x00,                     /* 5  bDeviceSubClass */
    0x00,                     /* 6  bDeviceProtocol */
    0x40,                     /* 7  bMaxPacketSize */
    (VENDOR_ID & 0xFF),       /* 8  idVendor */
    (VENDOR_ID >> 8),         /* 9  idVendor */
    (PRODUCT_ID & 0xFF),      /* 10 idProduct */
    (PRODUCT_ID >> 8),        /* 11 idProduct */
    (0x00),                   /* 12 bcdDevice: v1.0.0*/
    (0x01),                   /* 13 bcdDevice */
    STR_INDEX_MANUFACTURER,   /* 14 iManufacturer */
    STR_INDEX_PRODUCT,        /* 15 iProduct */
    0x00,                     /* 16 iSerialNumber */
    0x01                      /* 17 bNumConfigurations */
};
```

Note the use of `VENDOR_ID` and `PRODUCT_ID` defines, see below.

## 2.5 Configuring the USB Device ID's

The values used for Vendor ID (VID) and Product ID (PID) are defined in the file `endpoint0.xc`. These are used by the host machine to identify the product on the bus.

```
#define VENDOR_ID                  0x20B1
#define PRODUCT_ID                 0x00B1
```

**Warning:** When developing USB devices, obtaining a Vendor ID (VID) is crucial. A VID uniquely identifies a company and ensures proper device recognition. VIDs are issued by the USB Implementers Forum (USB-IF) and must be legally purchased. Using an unauthorised or duplicate VID can cause device conflicts, malfunction, or legal repercussions. Always ensure you have a legitimate VID for commercial devices to avoid potential issues with compatibility, certification, and intellectual property.

## 2.6 USB class codes

The USB Implementers Forum (USB-IF) define values for each class (https://usb.org/defined-class-codes). These are used in the *bDeviceClass* value in the device descriptor. In the case of a vendor specific device this should the value *0xff*. `lib_xud` maintains a list of class codes in *xud_standard_descriptors.h*.

Typically Each class has various sub-classes (*bDeviceSubClass*) and protocols (*bDeviceProtocol*). For a vendor specific device the developer has the freedom to set these values as they choose, potentially allowing for multiple custom protocols.

This example code simply uses *0* for both *bDeviceSubClass* and *bDeviceProtocol*, a developer may wish to modify to match their requirements.

## 2.7 USB configuration descriptor

A **USB configuration descriptor** is a data structure that provides detailed information about one specific configuration of a USB device. USB devices can support multiple configurations, each of which may define different sets of functionality (e.g., different modes of operation). The configuration descriptor is part of the hierarchy of descriptors sent to the host during the device enumeration process.

Key elements in a USB configuration descriptor include:

1. **Total Length**: The total size of the configuration descriptor, including all its subordinate descriptors (such as interface and endpoint descriptors).
2. **Number of Interfaces**: Specifies how many interfaces are part of the configuration. Each interface typically represents a functional unit of the device (e.g., a keyboard interface, a mouse interface, etc.).
3. **Configuration Value**: A unique number that identifies this configuration. This value is used by the host to select a particular configuration.
4. **Attributes**: This field specifies whether the device is bus-powered, self-powered, or supports remote wakeup (a feature that allows the device to wake up the host system from sleep).
5. **Maximum Power**: Indicates the maximum amount of power the device will draw from the bus when this configuration is active.

The configuration descriptor also points to subordinate descriptors, such as:

- **Interface Descriptors**: Define specific interfaces within the configuration, each of which may support a specific function (e.g., a speaker or microphone in an audio device).
- **Endpoint Descriptors**: Define communication channels (endpoints) used for data transfer between the device and the host.

When a host reads the configuration descriptor, it learns about the device's power requirements, the number of interfaces, and the communication endpoints available for use in that configuration.

The configuration descriptor used is listed below:

```
static unsigned char cfgDesc[] =
{
    0x09,                           /* 0  bLength */
    USB_DESCTYPE_CONFIGURATION,     /* 1  bDescriptortype */
    0x20, 0x00,                     /* 2  wTotalLength */
    0x01,                           /* 4  bNumInterfaces */
    0x01,                           /* 5  bConfigurationValue */
    0x00,                           /* 6  iConfiguration */
    0x80,                           /* 7  bmAttributes (bus-powered) */
    0xFA,                           /* 8  bMaxPower */

    0x09,                           /* 0  bLength */
    USB_DESCTYPE_INTERFACE,         /* 1  bDescriptorType */
    0x00,                           /* 2  bInterfacecNumber */
    0x00,                           /* 3  bAlternateSetting */
    0x02,                           /* 4: bNumEndpoints */
    0xFF,                           /* 5: bInterfaceClass */
    0xFF,                           /* 6: bInterfaceSubClass */
    0xFF,                           /* 7: bInterfaceProtocol*/
    0x00,                           /* 8  iInterface */

    0x07,                           /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,          /* 1  bDescriptorType */
    0x01,                           /* 2  bEndpointAddress */
    0x02,                           /* 3  bmAttributes (bulk) */
    0x00,                           /* 4  wMaxPacketSize */
    0x02,                           /* 5  wMaxPacketSize */
    0x01,                           /* 6  bInterval */

    0x07,                           /* 0  bLength */
    USB_DESCTYPE_ENDPOINT,          /* 1  bDescriptorType */
    0x81,                           /* 2  bEndpointAddress */
    0x02,                           /* 3  bmAttributes (bulk) */
    0x00,                           /* 4  wMaxPacketSize */
    0x02,                           /* 5  wMaxPacketSize */
    0x01                            /* 6  bInterval */
};
```

It is very basic, describing a single interface with 2 bulk (see *bmAttributes*) endpoints - one in each direction (denoted by bit 8 of the endpoint address).

## 2.8 USB string descriptors

The final table is used to hold strings for the device. It should be noted that the device descriptor contains indexes into this table for the manufacturer (*iManufacturer*) and product (*iProduct*) strings.

```
static char * unsafe stringTable[] =
{
    "\x09\x04",                         // Language ID string (US English)
    "XMOS",                             // iManufacturer
    "XMOS Simple Bulk Transfer Example",    // iProduct
```

**Note:** The string table is passed to the function `USB_StandardRequests()` which handles the conversion of the raw strings to valid USB string descriptors.

**Note:** The string at index 0 must always contain the *Language ID Descriptor*. This descriptor indicates the languages that the device supports for string descriptors.

## 2.9 Endpoint 0

The function `Endpoint0()` contains the code for handling control requests from the host to the mandatory control endpoint 0.

The function `USB_StandardRequests()` from `lib_xud` handles all the required standard requests. These include requests for mandatory descriptors, setting the device address etc.

There are no additional requests which need to be handled for a vendor specific device. However, a developer may choose to add additional requests to control custom device parameters etc.

```
void Endpoint0(chanend chan_ep0_out, chanend chan_ep0_in)
{
    USB_SetupPacket_t sp;
    XUD_BusSpeed_t usbBusSpeed;
    XUD_ep ep0_out = XUD_InitEp(chan_ep0_out);
    XUD_ep ep0_in  = XUD_InitEp(chan_ep0_in);

    while(1)
    {
        /* Returns XUD_RES_OKAY on success */
        XUD_Result_t result = USB_GetSetupPacket(ep0_out, ep0_in, sp);

        if(result == XUD_RES_OKAY)
        {
            /* Returns  XUD_RES_OKAY if handled okay,
             *          XUD_RES_ERR if request was not handled (i.e. STALLed),
             *          XUD_RES_RST if USB Reset */
            result = USB_StandardRequests(ep0_out, ep0_in, devDesc,
                        sizeof(devDesc), cfgDesc, sizeof(cfgDesc),
                        null, 0,
                        null, 0,
                        stringTable, sizeof(stringTable)/sizeof(stringTable[0]),
                        sp, usbBusSpeed);
        }

        /* USB bus reset detected, reset EP and get new bus speed */
        if(result == XUD_RES_RST)
        {
            usbBusSpeed = XUD_ResetEndpoint(ep0_out, ep0_in);
        }
    }
}
```

## 2.10 Bulk endpoints

The application endpoints for receiving and transmitting to the host machine are implemented in the file `app.xc`. This is contained within the function `bulk_endpoint()`, shown below:

```
void bulk_endpoint(chanend chan_ep_from_host, chanend chan_ep_to_host)
{
    int host_transfer_buf[BUFFER_SIZE_WORDS];
    unsigned host_transfer_length = 0;
    XUD_Result_t result;

    XUD_ep ep_from_host = XUD_InitEp(chan_ep_from_host);
    XUD_ep ep_to_host = XUD_InitEp(chan_ep_to_host);

    while(1)
    {
        /* Receive a buffer (512-bytes) of data from the host */
        if((result = XUD_GetBuffer(ep_from_host, (host_transfer_buf, char[BUFFER_SIZE_CHARS]), host_transfer_
↪length)) == XUD_RES_RST)
        {
            XUD_ResetEndpoint(ep_from_host, ep_to_host);
            continue;
        }

        /* Perform basic processing (increment data values) */
        for (int i = 0; i < host_transfer_length/4; i++)
            host_transfer_buf[i]++;

        /* Send the modified buffer back to the host */
        if((result = XUD_SetBuffer(ep_to_host, (host_transfer_buf, char[BUFFER_SIZE_CHARS]), host_transfer_
↪length)) == XUD_RES_RST)
        {
            XUD_ResetEndpoint(ep_from_host, ep_to_host);
        }
    }
}
```

Inspection of this function should yield the following observations:

- A buffer is declared to communicate and transfer data with the host `host_transfer_buf` of size BUFFER_SIZE.

- This task operates inside a `while (1)` loop which repeatedly deals with a sequence of requests from the host to send data to the device and then from the host to then read data from the device.
- A blocking call is made to `lib_xud` to receive (using `XUD_GetBuffer()`) and send data (using `XUD_SetBuffer()`) to the host machine at every loop iteration.
- The function performs some basic processing on the received host buffer and simply increments the values in the buffer received from the host and then sends it back.
- This simple processing could easily be replaced or extended to include access some external hardware device connected to the *xcore* GPIO or communication with another parallel task.

## Keeping in sync

Some consideration should be given to handling errors or connection issues, such as unexpected device disconnection or a host program exception/termination.

The example code runs two endpoints in a single thread to implement a simple request/response protocol. Consider a scenario where the device receives a request from the host but is disconnected from the USB bus while preparing it's response. In the case of a self-powered device, a naive implementation might have the device attempting to send a response while the host, recognising the disconnection, sends a new request. This could result in a deadlock.

While developing a fully robust and error-resilient communication protocol is beyond the scope of this document, the example code mitigates this issue by registering for bus updates. If a bus reset is detected, the code waits for a new request. The critical part to note is the `continue` statement following the bus reset detection, which ensures the device properly handles the reset.

There are other approaches that could be considered to allow the host to reset the device's state. For instance, a control request to endpoint 0 could serve this purpose.

A simple receive/send loop lends itself to potential synchronisation issues. More advanced techniques, such as waiting for a new request while concurrently handling response transmission, can help avoid these problems. Alternatively, using two separate threads for each direction of communication—decoupled from one another—could also be a viable solution, depending on the application's needs.

It's worth noting that some devices may not encounter these issues at all. For example, an endpoint continuously sending data to the host or two independent endpoints running in separate threads would face fewer synchronization concerns.

## 3 Hardware setup

To run the demo the following hardware is required:

- *xcore.ai* Evaluation Kit (*XK-EVK-XU316*)
- 2 x Micro-B USB cable

**Note:** The *XK-EVK-XU316* has an integrated XTAG debug device, no additional debug adaptor is required

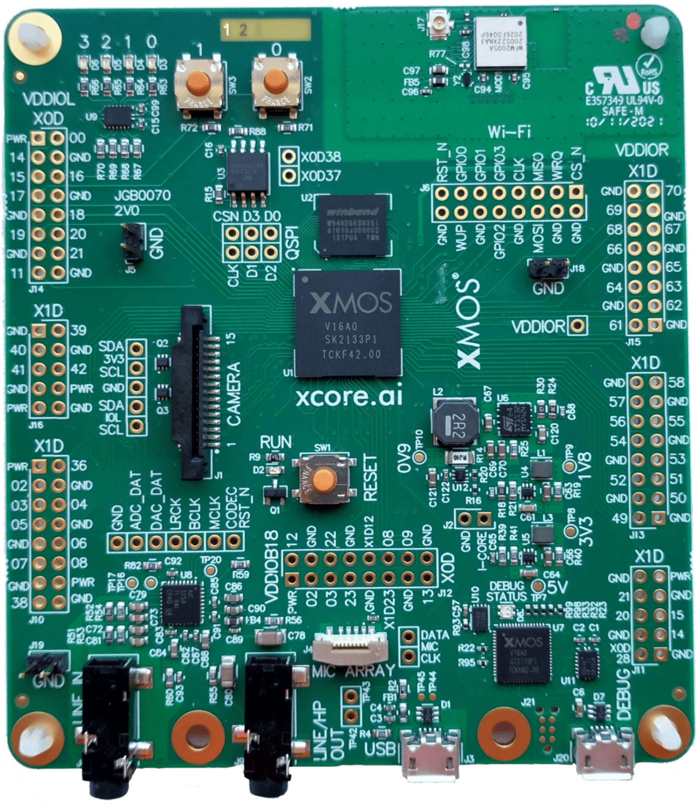

Fig. 3: *XMOS xcore.ai* Evaluation Kit

The hardware should be configured as follows:

- Connect the **USB** receptacle of the *XK-EVK-XU316* to the host machine using a USB cable
- Connect the **DEBUG** receptacle *XK-EVK-XU316* to the host machine using a USB cable

# 4 Host detail

## 4.1 Test application

The provided example host program, *bulktest*, demonstrates bulk transfer between the host and the *xcore* device.

The application simply transfers a data buffer to the device and back. The device performs a simple manipulation of the data (increments the data values by 1) before returning the new values to the host. The host program then increments the values and sends them again. This loop continues for a set number of iterations (nominally 1000).

Pre-compiled host binaries and, where required, setup scripts are provided for each sample platform in a suitably named directory in the *host* directory in the supplied software download.

Since the USB vendor class is specified by the developer, there is no native support with any operating system. The developer provide their own driver and host application. The provided examples use *libusb* to facilitate this.

*libusb* is a cross-platform, user-space library that provides access to USB devices without needing kernel-level drivers. It allows developers to communicate with USB hardware through a consistent API across different operating systems, supporting various USB transfer types (control, bulk, interrupt, isochronous). *libusb* is commonly used for device communication, firmware updates, and creating USB utilities. It's lightweight, portable, and simplifies USB programming by abstracting OS-specific details.

*libusb* is written in C and licensed under the LGPL-2.1 (Lesser General Public License v2.1).

## 4.2 Async vs sync API

*libusb* provides two main modes of operation for handling USB transfers, one synchronous and one asynchronous.

1. **Synchronous API:**

   - Description: In synchronous mode, the program issues a USB transfer request and waits for the transfer to complete before proceeding. The function calls block until the data transfer is finished.
   - Use Case: Synchronous operations are straightforward and easier to implement but can cause delays in your application if the transfers take a long time.

2. **Asynchronous API:**

   - Description: In asynchronous mode, the program issues a USB transfer request and can continue executing other tasks while waiting for the transfer to complete. Callbacks are used to notify the program when the transfer is done.
   - Use Case: Asynchronous operations are useful for high-performance applications where you want to avoid blocking and handle multiple transfers concurrently.

For the sake of simplicity, *bulktest* uses the synchronous scheme. A more advanced example is also provided, see *Bulk read benchmark example*.

## 4.3 Windows driver

For hosts running Windows a driver needs to be installed to support the vendor specific USB device. This is provided in the driver directory within the Win32 directory. When starting the device for the first time you will need to point Windows at this directory when it requests a driver to install for the device.

For more information on driver installation, including the option of a using a Automated Driver Installer GUI application, see the relevant libusb documentation

### 4.4 Compilation instructions

Pre-compiled binaries of *bulktest* are supplied. Example commands below demonstrate how it might be recompiled for the various platforms. Ensure the relevant compilation chain is installed on the machine.

Win32:

```
cl -o bulktest ..\bulktest.cpp -I ..\libusb\Win32 ..\libusb\Win32\libusb.lib
```

macOS:

```
g++ -o bulktest ../bulktest.cpp -I ../libusb/macOS ../libusb/macOS/libusb-1.0.0.dylib -m32
```

macOSARM64:

```
g++ -o bulktest ../bulktest.cpp -I ../libusb/macOSARM64 ../libusb/macOSARM64/libusb-1.0.0.dylib
```

Linux32:

```
g++ -o bulktest ../bulktest.cpp -I ../libusb/Linux32 ../libusb/Linux32/libusb-1.0.a -lpthread -lrt
```

Linux64:

```
g++ -o bulktest ../bulktest.cpp -I ../libusb/Linux64 ../libusb/Linux64/libusb-1.0.a -lpthread -lrt
```

## 5 Running the example

This section assumes you have downloaded and installed the XMOS XTC tools (see *README* for required version). Installation instructions can be found here.

Be sure to pay attention to the section Installation of required third-party tools.

### 5.1 Building the *xcore* app

The application uses the xcommon-cmake build system as bundled with the XTC tools.

The `an00136` software zip-file should be downloaded and unzipped to a chosen directory.

The file *CMakeLists.txt* contains build configurations named `BULKTEST` and `BENCHMARK`.

Initially this document concerns itself wth the `BULKTEST` configuration. See *Bulk read benchmark example* for information relating to the `BENCHMARK` configuration.

To configure the build run the following from an XTC command prompt:

```
cd an00136
cd app_an00136
cmake -G "Unix Makefiles" -B build
```

All required dependencies are included in the software download, however, if any are missing it is at this configure step that they will be downloaded by the build system.

Finally, the application binaries can be built using `xmake`:

```
xmake -j -C build
```

This command will cause two binaries (.xe files) to be generated in relevant subdirectories of the *app_an00136/bin* directory, one for each of the build configurations previously mentioned.

Blocks of code are optionally compiled based on a BENCHMARK define, set by the relevant build configuration, for example:

```
#ifdef BENCHMARK
    // Some code
#endif
```

This accounts for the differences in the functionality between the two binaries.

## 5.2 Launching the *xcore* app

From a XTC command prompt run the following command:

```
xrun ./bin/BULKTEST/app_an02003_BULKTEST.xe
```

Once this command has executed the device should have enumerated on the host machine.

## 5.3 Running the host app

Source the appropriate provided *setup* script for the platform and then execute the 'bulk-test' application from the command line.

This will connect to the USB device and transfer data buffers back and forth.

The output should be similar the following:

```
XMOS Bulk USB device opened .....
Timing write/read of 1000 512-byte buffers.....
125 ms (7.81 MB/s)
XMOS Bulk USB device data processed correctly .....
XMOS Bulk USB device closed .....
```

This application is intended as a simple demonstration application and has not been programmed for efficient data transfer. The performance reported for this simple application will vary depending on the capabilities of your USB host, operating system and other bus activity.

# 6 Advanced example

## 6.1 Bulk read benchmark example

Included with the example host code is a bulk read benchmark demo. This demonstrates high performance data throughput from the device to the host. The main difference is in the host code which uses asynchronous, non blocking *libusb* calls to utilise the USB bus more effectively. The rest of this section lists the steps required to run this benchmark application.

**Note:** Currently the optimized bulk read benchmark is only supported on macOS and Linux, Windows is not supported at this time.

### Device code changes

An alternative implementation for `bulk_endpoint()` is used in the BENCHMARK build configuration.

This function is a simplified version of the one used in the BULKTEST configuration. It only deals with transmitting packets to USB host. It also doesn't concern itself with receiving bus state updates.

This alternative implementation of `bulk_endpoint()` is listed below:

```
/* An optimised endpoint for the read benchmark test */
void bulk_endpoint(chanend chan_ep_to_host)
{
    char host_transfer_buf[BUFFER_SIZE_WORDS*4];
    unsigned host_transfer_length = 512;

    XUD_ep ep_to_host = XUD_InitEp(chan_ep_to_host);

    while(1)
    {
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
        XUD_SetBuffer(ep_to_host, host_transfer_buf, host_transfer_length);
    }
}
```

Notable differences include:

- The host transfer length is set to 512 bytes to match the host application
- The while loop has been unrolled to contain 8 calls to `XUD_SetBuffer()`
- The "from host" channel-end is not used

### Host code changes

A separate host application is provided to work with the BENCHMARK build configuration. See the file *bulk_read_benchmark.cpp*.

Steps for building follow that already provided for *bulktest*, replacing the file *bulktest.cpp* with *bulk_read_benchmark.cpp*.

This example runs forever and will need to be terminated with a ctrl-C when required.

The output will look as follows, with the performance depending on host platform, USB hardware and other bus traffic:

```
XMOS Bulk USB device opened .....
Read transfer rate 32.19 MB/s
Read transfer rate 34.94 MB/s
```

(continues on next page)

(continued from previous page)

```
Read transfer rate 39.56 MB/s
Read transfer rate 39.62 MB/s
Read transfer rate 39.56 MB/s
Read transfer rate 39.56 MB/s
Read transfer rate 39.56 MB/s
Read transfer rate 39.56 MB/s
Read transfer rate 39.56 MB/s
```

## 7 Further reading

- *XMOS* XTC Tools Installation Guide
  https://xmos.com/xtc-install-guide
- *XMOS* XTC Tools User Guide
  https://www.xmos.com/view/Tools-15-Documentation
- USB 2.0 Specification
  https://www.usb.org/sites/default/files/usb_20_20240604.zip
- *XMOS* application build and dependency management system; *xcommon-cmake*
  https://www.xmos.com/file/xcommon-cmake-documentation/?version=latest