



AN00127: USB Video Class Device

Publication Date: 2025/4/23

Document Number: XM-007149-AN v3.0.0

IN THIS DOCUMENT

1	Introduction	1
2	Overview	1
3	USB Video Class application note	2
4	Further reading	21

1 Introduction

This application note shows how to create a USB device compliant to the standard USB Video Class (UVC) on an xCORE device.

The code associated with this application note provides an example of using the XMOS USB Device Library ([lib_xud](#)) and associated USB class descriptors to provide a framework for the creation of USB video devices like webcam, video player, camcorders etc.

This example USB video class implementation provides a video camera device running over high speed USB. It supports standard requests associated with the class. The application doesn't connect a camera sensor device but emulates it by creating simple video data which is streamed to the host PC. Any host software that supports viewing UVC compliant video capture devices can be used to view the video streamed out of the XMOS device. This demonstrates the simple way in which USB video devices can easily be deployed using an xCORE-USB device.

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

This application note is designed to run on *XMOS xcore-200* or *xcore.ai* series devices.

The example code provided with the application has been implemented and tested on the *XK-EVK-XU316* board but there is no dependency on this board and it can be modified to run on any development board which uses an *xcore-200* or *xcore.ai* series device.

- 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 [Further reading](#).
- For the full API listing of the XMOS USB Device (XUD) Library please see the document *XMOS USB Device (XUD) Library*¹.

2 Overview

USB Video Class (UVC) is a standard class specification that standardizes video streaming functionality on the USB. It enables devices like webcams, digital camcorders, analog

¹ https://www.xmos.com/file/lib_xud



video converters, analog and digital television tuners etc to connect seamlessly with host machines.

UVC supports streaming multiple video formats including YUV, MJPEG, MPEG-2 TS, H.264, DV etc. It provides structures for describing the functionalities of the video device to the host and defines USB requests to control different parameters of the device and characteristics of the video stream. It also provides flexibility for a video device to support multiple video resolutions, formats and frame rates, which highly influence the bandwidth negotiation between the device and the host.

Many OS platforms have native support for UVC drivers which greatly reduces the time required for developers to create USB video devices.

This application note provides a detailed explanation of the UVC implementation for the *xcore* device, enabling developers to build their own USB video devices. The demo example doesn't interface a camera sensor but can be easily extended to add a camera.

The standard USB Video Class specification can be found on the USB-IF website.

Note: This application note addresses v1.1 of the specification

(<https://www.usb.org/document-library/video-class-v11-document-set>)

2.1 Block diagram

Fig. 1 shows the block diagram of USB video class application

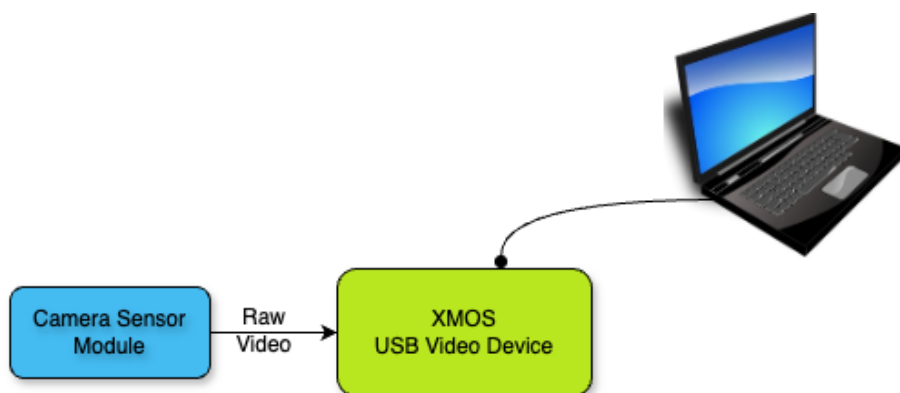


Fig. 1: Block diagram of USB video class application

The 'Camera Sensor' shown in the above figure is not interfaced in the demo example but it is emulated by creating color video frames inside the device.

3 USB Video Class application note

The example in this application note uses the XMOS USB device library (*lib_xud*) and shows a simple program that enumerates a USB Video Class device in a host machine and streams uncompressed video frames in YUV format with thirty frames per second to a video capture host software.

For this USB Video device application example, the system comprises three tasks running on separate threads of an *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 both standard and video class-specific USB requests.
- ▶ A task implementing the application code to send video data over streaming endpoints.

These tasks communicate via the use of *xCONNECT* channels which allow data to be passed between application code running on separate logical cores.

Fig. 2 shows the task and communication structure for this USB video class application example.

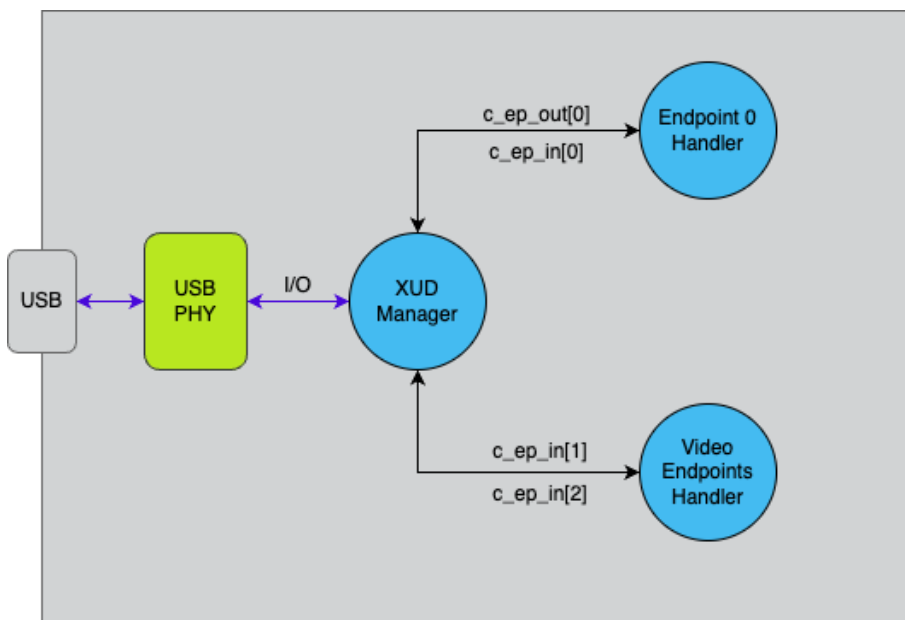


Fig. 2: Task diagram of the USB video device example

3.1 CMakeLists.txt additions for this application

To start using the USB library, you need to add **lib_xud** to your *CMakeLists.txt*:

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

You can then access the USB functions in your source code via the **xud_device.h** header file:

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

3.2 Source code files

The example application consists of multiple source code files and the following list provides an overview of how the source code is organised.

- ▶ **usb_video.xc, usb_video.h** - Contains the USB video class descriptors and endpoint handler tasks (functions).
- ▶ **uvc_req.c, uvc_req.h** - Contains functions and data structures to handle class-specific USB requests.
- ▶ **uvc_defs.h** - This header file has defines that are used for USB descriptors, class-specific requests and video details like resolution, payload size and frame rate etc.
- ▶ **main.xc** - Contains **main()** function and some USB related defines.

3.3 Setting up the USB components

main.xc has some arrays in it that are used to configure the endpoints for the the *XMOS* USB device library. These are displayed below.

```
/* Endpoint count defines */
#define EP_COUNT_OUT 1 // 1 OUT EP0
#define EP_COUNT_IN 3 // (1 IN EP0 + 1 INTERRUPT IN EP + 1 ISO IN EP)

/* Endpoint type tables - informs XUD what the transfer types for each Endpoint in use and also
 * if the endpoint wishes to be informed of USB bus resets
 */
XUD_EpType epTypeTableOut[EP_COUNT_OUT] = {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE};
XUD_EpType epTypeTableIn[EP_COUNT_IN] = {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE, XUD_EPTYPE_INT, XUD_EPTYPE_ISO};
```

The tables above describe the endpoint configurations for this device. This example has bi-directional communication with the host machine via the standard endpoint 0 and two other *IN* endpoints for implementing the part of our video class.

These tables are passed to the function for the USB library which is called from **main()**.

3.4 The application main() function

Below is the source code for the main function of this application, which is taken from the source file **main.xc**

```
int main() {
    chan c_ep_out[EP_COUNT_OUT], c_ep_in[EP_COUNT_IN];

    /* 'Par' statement to run the following tasks in parallel */
    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]);

        on USB_TILE: VideoEndpointsHandler(c_ep_in[1], c_ep_in[2]);
    }
    return 0;
}
```

Looking at this in more detail, the following can be observed:

- ▶ The par statement starts three separate tasks in parallel.
- ▶ There is a task to configure and execute the USB library: **XUD_Main()**. This library call runs in an infinite loop and handles all the underlying USB communications and provides abstraction at the endpoints level.
- ▶ There is a task to start and run the Endpoint 0 code: **Endpoint0()**. It handles the control endpoint zero and must be run in a separate logical core in order to provide timely response to control requests from the host.
- ▶ There is a task to handle two other endpoints required for the Video class: **VideoEndpointsHandler()**. This function handles one Isochronous IN endpoint for video streaming and one interrupt IN endpoint for sending notifications to host.
- ▶ The define **USB_TILE** describes the tile on which the individual tasks will run.
- ▶ In this example all tasks run on the same tile as the USB PHY although this is only a requirement of **XUD_Main()**.
- ▶ The xCONNECT communication channels used for inter-task communication are setup at the beginning of **main()** and passed on to respective tasks.
- ▶ The USB defines discussed earlier are passed into the function **XUD_Main()**.

3.5 Configuring the USB device ID

The USB ID values used for vendor ID, product ID and device version number are defined in the file **uvc_defs.h**. These are used by the host machine to determine the vendor of the device (in this case **XMOS**) and the product plus the firmware version.

```
/* USB Video device product defines */
#define BCD_DEVICE 0x0100
#define VENDOR_ID 0x20B1
#define PRODUCT_ID 0x1DE0
```

3.6 Video device topology

This section provides a brief overview of the representation of video device in a topology. It introduces the terms used in the video class specification, which help the reader to understand the further sections of this application note.

A video device is represented as an interconnection of multiple addressable entities. Each entity represents a functionality and has properties which are controlled by the USB host. The following are the different entities:

- ▶ **Units**

► Selector Unit

- Processing Unit
- Extension Unit

► Terminals

- Input Terminal
- Output Terminal
- Special Terminals (extends the I/O terminal)
 - Media Transport Terminal
 - Camera Terminal

These entities are interconnected by means of *Input Pins* and *Output Pins*. A Unit has one or more Input Pins and a single Output Pin, where each Pin represents logical data streams inside the video device. A Terminal has either a single Input Pin or a single Output Pin. An *Input Terminal(IT)* represents a starting point for data streams of the video device. An *Output Terminal(OT)* represents an ending point for data streams.

The functionality of a Unit or Terminal is further described through Video Controls. A Control typically provides access to a specific video property. Video properties include brightness, contrast, sharpness, digital zoom etc. Each Control has a set of attributes that can be manipulated or that provide additional information, they are:

- Current setting
- Minimum setting
- Maximum setting
- Resolution
- Size
- Default

For example, the brightness of the video stream can be controlled by the USB host by changing the current setting of the Brightness Control inside a Processing unit.

Fig. 3 shows the topology of the demo application

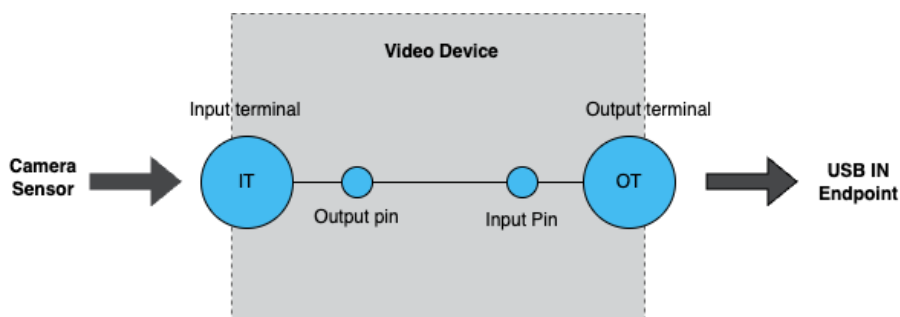


Fig. 3: Topology of the UVC example

No Units are involved in the demo application example. More information on Units can be found from the USB video class specification documents.

This video device topology is communicated to the host through USB descriptors which is discussed in the following section.

3.7 USB Descriptors

USB Video class device has to support class-specific descriptors apart from the standard descriptors defined in the USB specifications. The class specific descriptors are customised according to the need of the USB Video device.

Fig. 4 shows the descriptors used in the example code.

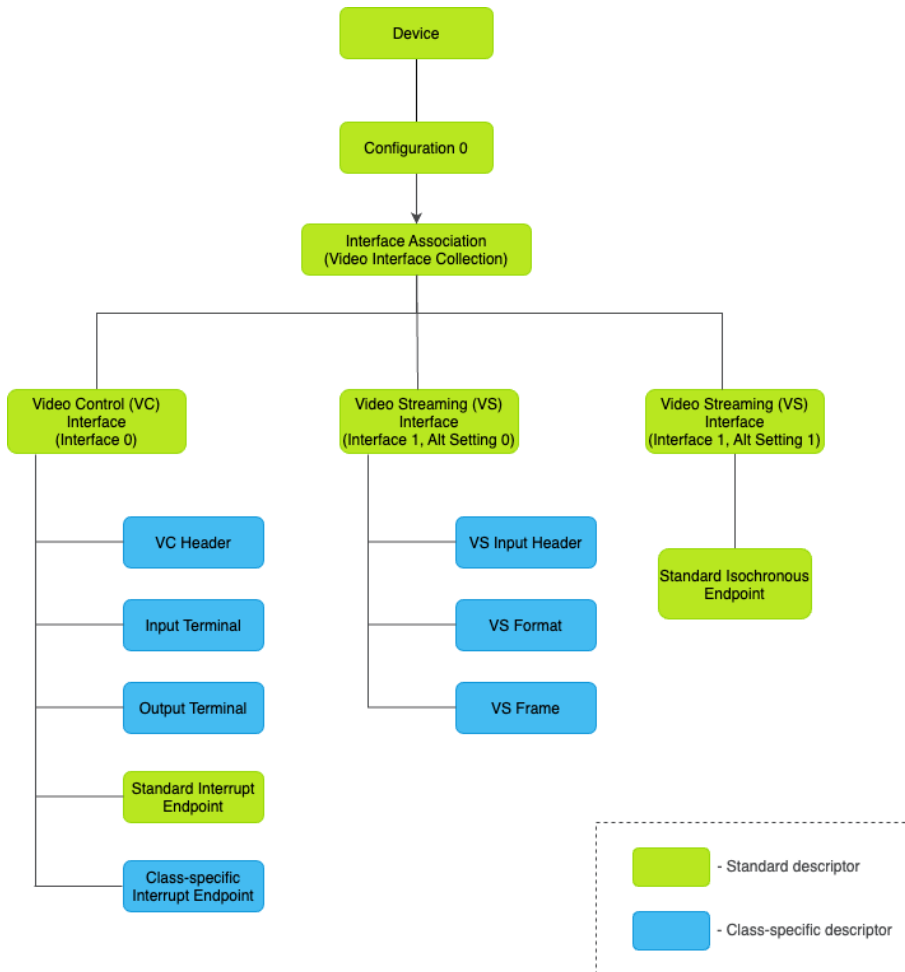


Fig. 4: Hierarchical structure of USB descriptors of UVC example

The above figure is discussed in detail in the following sections.

USB Device Descriptor

`usb_video.xc` is where the standard USB device descriptor is declared for the Video class device. Below is the structure which contains this descriptor. This will be requested by the host when the device is enumerated on the USB bus.

```
/* USB Device Descriptor */
static unsigned char devDesc[] =
{
    0x12,          /* 0 bLength */
    USB_DSCTYPE_DEVICE, /* 1 bdescriptorType - Device*/
    0x00,          /* 2 bcdUSB version */
    0x02,          /* 3 bcdUSB version */
    0xEF,          /* 4 bDeviceClass - USB Miscellaneous Class */
    0x02,          /* 5 bDeviceSubClass - Common Class */
    0x01,          /* 6 bDeviceProtocol - Interface Association Descriptor */
    0x40,          /* 7 bMaxPacketSize for EP0 - max = 64*/
    (VENDOR_ID & 0xFF), /* 8 idVendor */
    (VENDOR_ID >> 8), /* 9 idVendor */
    (PRODUCT_ID & 0xFF), /* 10 idProduct */
    (PRODUCT_ID >> 8), /* 11 idProduct */
    (BCD_DEVICE & 0xFF), /* 12 bcdDevice */
    (BCD_DEVICE >> 8), /* 13 bcdDevice */
    0x01,          /* 14 iManufacturer - index of string*/
    0x02,          /* 15 iProduct - index of string*/
    0x00,          /* 16 iSerialNumber - index of string*/
    0x01           /* 17 bNumConfigurations */
};
```

From this descriptor you can see that product, vendor and device firmware revision are all coded into this structure. This will allow the host machine to recognise the video device when it is connected to the USB bus.

For Video class device, it is mandatory to set the **bDeviceClass**, **bDeviceSubClass** and **bDeviceProtocol** fields to *0xEF*, *0x02* and *0x01* respectively.

USB Configuration Descriptor

The USB configuration descriptor is used to configure the device in terms of the device class and the endpoints setup. The hierarchy of descriptors under a configuration includes interface association descriptor, interfaces descriptors, class-specific descriptors and endpoints descriptors.

When a host requests a configuration descriptor, the entire configuration hierarchy including all the related descriptors are returned to the host. The following code shows the configuration hierarchy of the demo application.

```
/* USB Configuration Descriptor */
static unsigned char cfgDesc[] = {

    0x09,          /* 0 bLength */
    USB_DESCRIPTOR_CONFIGURATION, /* 1 bDescriptorType - Configuration */
    0x0AE, 0x00,    /* 2 wTotalLength */
    0x02,           /* 4 bNumInterfaces */
    0x01,           /* 5 bConfigurationValue */
    0x03,           /* 6 iConfiguration - index of string */
    0x80,           /* 7 bmAttributes - Bus powered */
    0xFA,           /* 8 bMaxPower (in 2mA units) - 500mA */

    /* Interface Association Descriptor */
    0x08,          /* 0 bLength */
    USB_DESCRIPTOR_INTERFACE_ASSOCIATION, /* 1 bDescriptorType - Interface Association */
    0x00,          /* 2 bFirstInterface - VideoControl i/f */
    0x02,          /* 3 bInterfaceCount - 2 Interfaces */
    USB_CLASS_VIDEO, /* 4 bFunctionClass - Video Class */
    USB_VIDEO_INTERFACE_COLLECTION, /* 5 bFunctionSubClass - Video Interface Collection */
    0x00,          /* 6 bFunctionProtocol - No protocol */
    0x02,          /* 7 iFunction - index of string */

    /* Video Control (VC) Interface Descriptor */
    0x09,          /* 0 bLength */
    USB_DESCRIPTOR_INTERFACE, /* 1 bDescriptorType - Interface */
    0x00,          /* 2 bInterfaceNumber - Interface 0 */
    0x00,          /* 3 bAlternateSetting */
    0x01,          /* 4 bNumEndpoints */
    USB_CLASS_VIDEO, /* 5 bInterfaceClass - Video Class */
    USB_VIDEO_CONTROL, /* 6 bInterfaceSubClass - VideoControl Interface */
    0x00,          /* 7 bInterfaceProtocol - No protocol */
    0x02,          /* 8 iInterface - Index of string (same as iFunction of IAD) */

    /* Class-specific VC Interface Header Descriptor */
    0x0D,          /* 0 bLength */
    USB_DESCRIPTOR_CS_INTERFACE, /* 1 bDescriptorType - Class-specific Interface */
    USB_VC_HEADER, /* 2 bDescriptorSubType - HEADER */
    0x10, 0x01,    /* 3 bcdUVC - Video class revision 1.1 */
    0x28, 0x00,    /* 5 wTotalLength - till output terminal */
    WORD_CHARS(100000000), /* 7 dwClockFrequency - 100MHz (Deprecated) */
    0x01,          /* 11 bInCollection - One Streaming Interface */
    0x01,          /* 12 baInterfaceNr - Number of the Streaming interface */

    /* Input Terminal (Camera) Descriptor - Represents the CCD sensor (Simulated here in this demo) */
    0x12,          /* 0 bLength */
    USB_DESCRIPTOR_CS_INTERFACE, /* 1 bDescriptorType - Class-specific Interface */
    USB_VC_INPUT_TERMINAL, /* 2 bDescriptorSubType - INPUT TERMINAL */
    0x01,          /* 3 bTerminalID */
    0x01, 0x02,    /* 4 wTerminalType - ITT_CAMERA type (CCD Sensor) */
    0x00,          /* 6 bAssocTerminal - No association */
    0x00,          /* 7 iTerminal - Unused */
    0x00, 0x00,    /* 8 wObjectiveFocalLengthMin - No optical zoom supported */
    0x00, 0x00,    /* 10 wObjectiveFocalLengthMax - No optical zoom supported */
    0x00, 0x00,    /* 12 wOcularFocalLength - No optical zoom supported */
    0x03,          /* 14 bControlSize - 3 bytes */
    0x00, 0x00, 0x00, /* 15 bmControls - No controls are supported */

    /* Output Terminal Descriptor */
    0x09,          /* 0 bLength */
    USB_DESCRIPTOR_CS_INTERFACE, /* 1 bDescriptorType - Class-specific Interface */
    USB_VC_OUTPUT_TERMINAL, /* 2 bDescriptorSubType - OUTPUT TERMINAL */
    0x02,          /* 3 bTerminalID */
    0x01, 0x01,    /* 4 wTerminalType - TT_STREAMING type */
    0x00,          /* 6 bAssocTerminal - No association */
    0x01,          /* 7 bSourceID - Source is Input terminal 1 */
    0x00,          /* 8 iTerminal - Unused */

    /* Standard Interrupt Endpoint Descriptor */
    0x07,          /* 0 bLength */
    USB_DESCRIPTOR_ENDPOINT, /* 1 bDescriptorType */
    (VIDEO_STATUS_EP_NUM | 0x80), /* 2 bEndpointAddress - IN endpoint */
    0x03,          /* 3 bmAttributes - Interrupt transfer */
    0x40, 0x00,    /* 4 wMaxPacketSize - 64 bytes */
    0x09,          /* 6 bInterval - 2^(9-1) microframes = 32ms */

    /* Class-specific Interrupt Endpoint Descriptor */
    0x05,          /* 0 bLength */
    USB_DESCRIPTOR_CS_ENDPOINT, /* 1 bDescriptorType - Class-specific Endpoint */

```

(continues on next page)



(continued from previous page)

```

0x03,          /* 2 bDescriptorSubType - Interrupt Endpoint */
0x40, 0x00,    /* 3 wMaxTransferSize - 64 bytes */

/* Video Streaming Interface Descriptor */
/* Zero-bandwidth Alternate Setting 0 */
0x09,          /* 0 length */
USB_DESCRIPTOR_INTERFACE, /* 1 bDescriptorType - Interface */
0x01,          /* 2 bInterfaceNumber - Interface 1 */
0x00,          /* 3 bAlternateSetting - 0 */
0x00,          /* 4 bNumEndpoints - No bandwidth used */
USB_CLASS_VIDEO, /* 5 bInterfaceClass - Video Class */
USB_VIDEO_STREAMING, /* 6 bInterfaceSubClass - VideoStreaming Interface */
0x00,          /* 7 bInterfaceProtocol - No protocol */
0x00,          /* 8 iInterface - Unused */

/* Class-specific VS Interface Input Header Descriptor */
0x0E,          /* 0 length */
USB_DESCRIPTOR_CS_INTERFACE, /* 1 bDescriptorType - Class-specific Interface */
USB_VS_INPUT_HEADER, /* 2 bDescriptorSubType - INPUT HEADER */
0x01,          /* 3 bNumFormats - One format supported */
0x47, 0x00,    /* 4 wTotalLength - Size of class-specific VS descriptors */
(VIDEO_DATA_EP_NUM | 0x80), /* 6 bEndpointAddress - Iso EP for video streaming */
0x00,          /* 7 bmInfo - No dynamic format change */
0x02,          /* 8 bTerminalLink - Denotes the Output Terminal */
0x01,          /* 9 bStillCaptureMethod - Method 1 supported */
0x00,          /* 10 bTriggerSupport - No Hardware Trigger */
0x00,          /* 11 bTriggerUsage */
0x01,          /* 12 bControlSize - 1 byte */
0x00,          /* 13 bmaControls - No Controls supported */

/* Class-specific VS Format Descriptor */
0x1B,          /* 0 length */
USB_DESCRIPTOR_CS_INTERFACE, /* 1 bDescriptorType - Class-specific Interface */
USB_VS_FORMAT_UNCOMPRESSED, /* 2 bDescriptorSubType - FORMAT UNCOMPRESSED */
0x01,          /* 3 bFormatIndex */
0x01,          /* 4 bNumFrameDescriptors - 1 Frame descriptor followed */
0x59, 0x55, 0x59, 0x32, /* 5 guidFormat - YUY2 Video format */
0x00, 0x00, 0x10, 0x00, /* 21 bBitsPerPixel - 16 bits */
0x80, 0x00, 0x00, 0xAA, /* 22 bDefaultFrameIndex */
0x00, 0x38, 0x9B, 0x71, /* 23 bAspectRatioX */
BITS_PER_PIXEL, /* 24 bAspectRatioY */
0x01,          /* 25 bmInterlaceFlags - No interlaced mode */
0x00,          /* 26 bCopyProtect - No restrictions on duplication */

/* Class-specific VS Frame Descriptor */
0x1E,          /* 0 length */
USB_DESCRIPTOR_CS_INTERFACE, /* 1 bDescriptorType - Class-specific Interface */
USB_VS_FRAME_UNCOMPRESSED, /* 2 bDescriptorSubType */
0x01,          /* 3 bFrameIndex */
0x01,          /* 4 bmCapabilities - Still image capture method 1 */
SHORT_CHARS(WIDTH), /* 5 wWidth */
SHORT_CHARS(HEIGHT), /* 7 wHeight */
WORD_CHARS(MIN_BIT_RATE), /* 9 dwMinBitRate */
WORD_CHARS(MAX_BIT_RATE), /* 13 dwMaxBitRate */
WORD_CHARS(MAX_FRAME_SIZE), /* 17 dwMaxVideoFrameBufSize */
WORD_CHARS(FRAME_INTERVAL), /* 21 dwDefaultFrameInterval (in 100ns units) */
0x01,          /* 25 bFrameIntervalType */
WORD_CHARS(FRAME_INTERVAL), /* 26 dwFrameInterval (in 100ns units) */

/* Video Streaming Interface Descriptor */
/* Alternate Setting 1 */
0x09,          /* 0 length */
USB_DESCRIPTOR_INTERFACE, /* 1 bDescriptorType - Interface */
0x01,          /* 2 bInterfaceNumber - Interface 1 */
0x01,          /* 3 bAlternateSetting - 1 */
0x01,          /* 4 bNumEndpoints */
USB_CLASS_VIDEO, /* 5 bInterfaceClass - Video Class */
USB_VIDEO_STREAMING, /* 6 bInterfaceSubClass - VideoStreaming Interface */
0x00,          /* 7 bInterfaceProtocol - No protocol */
0x00,          /* 8 iInterface - Unused */

/* Standard VS Isochronous Video Data Endpoint Descriptor */
0x07,          /* 0 length */
USB_DESCRIPTOR_ENDPOINT, /* 1 bDescriptorType */
(VIDEO_DATA_EP_NUM | 0x80), /* 2 bEndpointAddress - IN Endpoint */
0x05,          /* 3 bmAttributes - Isochronous EP (Asynchronous) */
0x00, 0x04,    /* 4 wMaxPacketSize 1x 1024 bytes */
0x01,          /* 6 bInterval */
};

```

The *configuration descriptor* tells host about the power requirements of the device and the number of interfaces it supports.

Multiple interfaces together provides the video functionality. This group of interfaces is called Video Interface Collection. The Video Interface Collection is described by an *interface association descriptor* (IAD). In the example application, the IAD defines that



the interface zero and one group to form the USB Video device. These two interfaces are:

- Video Control Interface (VC Interface)
- Video Streaming Interface (VS Interface)

Note: A video function must have one VideoControl interface and zero or more VideoStreaming interfaces.

VideoControl Interface

This interface controls the functional behavior of the video device. It is described by both standard and class-specific descriptors.

The *Standard VC interface descriptor* identifies the interface number and class and provides the number of endpoints that belongs to this interface. The default *Endpoint 0* is used by this interface for control purpose through class-specific requests. Another optional endpoint called *Status Interrupt Endpoint* is used to send asynchronous status notifications to the host. This interrupt endpoint is described by both standard and class-specific endpoint descriptors.

The *Class-Specific VC interface descriptor* describes the whole topology of the video device. It includes *Unit descriptors* and *Terminal descriptors*. The example application doesn't include any Units and hence only Terminal descriptors can be found in the descriptors hierarchy structure.

The Class-Specific descriptors starts with a header called VC Interface Header descriptor. This descriptor mentions the version of UVC specification followed and the collection of streaming interfaces to which this VideoControl interface belongs.

The *Input Terminal descriptor* provides information on the functional aspects of the input source of the video device. Following code shows the fields of this descriptor:

```
USB_VC_INPUT_TERMINAL,    /* 2 bDescriptorSubType - INPUT TERMINAL */
0x01,                    /* 3 bTerminalID */
0x01, 0x02,              /* 4 wTerminalType - ITT_CAMERA type (CCD Sensor) */
0x00,                    /* 6 bAssocTerminal - No association */
```

In the above code, the **bTerminalID** is a unique identifier of this terminal and **bTerminalType** declares camera as the input type.

The *Output Terminal descriptor* is shown in the following code.

```
USB_VC_OUTPUT_TERMINAL,  /* 2 bDescriptorSubType - OUTPUT TERMINAL */
0x02,                    /* 3 bTerminalID */
0x01, 0x01,              /* 4 wTerminalType - IT_STREAMING type */
0x00,                    /* 6 bAssocTerminal - No association */
0x01,                    /* 7 bSourceID - Source is Input terminal 1 */
0x00,                    /* 8 iTerminal - Unused */
```

The above descriptor shows that the **bSourceID** is defined as 0x01 which is the **bTerminalID** of the input terminal. This information shows the interconnection between the entities, which the host uses to identify the topology of the video device.

VideoStreaming Interface

VideoStreaming interfaces are used to interchange video data streams between the Host and the Video device. Each interface can have one isochronous or bulk data endpoint. Interfaces supporting isochronous video transfer must have alternate settings which enables host to change the bandwidth requirements imposed by an active isochronous pipe. It is also mandatory to provide a zero-bandwidth alternate setting as the default alternate setting (alternate setting zero) that provides the host software the option to temporarily relinquish USB bandwidth by switching to this alternate setting.

In the UVC example, the zero-bandwidth alternate setting of the VideoStreaming interface is described by standard interface descriptor and class-specific VS interface descriptors.

The *Standard VS interface descriptor* provides the interface number, the number of endpoints that belongs to this interface etc. In case of zero-bandwidth alternate setting the number of endpoints is set to zero.

The *Class-Specific VS interface descriptors* are used to describe the supported video stream formats, video frame details, still image frame details, color profile of video data etc. The following is the list of these class-specific descriptors:

- ▶ Input Header descriptor
- ▶ Output Header descriptor
- ▶ Payload Format descriptor
- ▶ Video Frame descriptor
- ▶ Still Image frame descriptor
- ▶ Color Matching descriptor

The *Input Header descriptor* is meant for interfaces that contain IN endpoint and *Output Header* is for interfaces that contain OUT endpoint.

The following code shows the fields of Input Header descriptor:

```
USB_VS_INPUT_HEADER,      /* 2 bDescriptorSubType - INPUT HEADER */
0x01,                    /* 3 bNumFormats - One format supported */
0x47, 0x00,              /* 4 wTotalLength - Size of class-specific VS descriptors */
(VIDEO_DATA_EP_NUM | 0x00), /* 6 bEndpointAddress - Iso EP for video streaming */
0x00,                    /* 7 bmInfo - No dynamic format change */
0x02,                    /* 8 bTerminalLink - Denotes the Output Terminal */
```

The above code shows the number of formats supported, the address of endpoint which streams video data and the output terminal ID which links to this streaming interface.

The *Payload Format descriptor* describes the video format. The fields of this descriptor is shown below:

```
USB_VS_FORMAT_UNCOMPRESSED, /* 2 bDescriptorSubType - FORMAT UNCOMPRESSED */
0x01,                       /* 3 bFormatIndex */
0x01,                       /* 4 bNumFrameDescriptors - 1 Frame descriptor followed */
0x59, 0x55, 0x59, 0x32,    /* 5 guidFormat - YUY2 Video format */
0x00, 0x00, 0x10, 0x00,    /* 21 bBitsPerPixel - 16 bits */
0x00, 0x00, 0x00, 0xAA,    /* 22 bDefaultFrameIndex */
0x00, 0x38, 0x9B, 0x71,
BITS_PER_PIXEL,
0x01,
```

The above code shows that the video stream is of uncompressed YUY2 format and uses 16-bits per pixel.

The *Video Frame descriptor* mentions the frame resolution, frame rate, video buffer size etc. The following code shows the fields of this descriptor:

```
USB_VS_FRAME_UNCOMPRESSED, /* 2 bDescriptorSubType */
0x01,                       /* 3 bFrameIndex */
0x01,                       /* 4 bmCapabilities - Still image capture method 1 */
SHORT_CHARS(WIDTH),         /* 5 wWidth */
SHORT_CHARS(HEIGHT),        /* 7 wHeight */
WORD_CHARS(MIN_BIT_RATE),   /* 9 dwMinBitRate */
WORD_CHARS(MAX_BIT_RATE),   /* 13 dwMaxBitRate */
WORD_CHARS(MAX_FRAME_SIZE), /* 17 dwMaxVideoFrameBufSize */
WORD_CHARS(FRAME_INTERVAL), /* 21 dwDefaultFrameInterval (in 100ns units) */
0x01,                       /* 25 bFrameIntervalType */
WORD_CHARS(FRAME_INTERVAL), /* 26 dwFrameInterval (in 100ns units) */
```

The defines used in the above code are present in the *uvc_defs.h* file and they are shown below:

```
/* USB Video resolution */
#define BITS_PER_PIXEL 16
#define WIDTH 480
#define HEIGHT 270

/* Frame rate */
#define FPS 30
```

(continues on next page)



(continued from previous page)

```

#define MAX_FRAME_SIZE (WIDTH * HEIGHT * BITS_PER_PIXEL / 8)
#define MIN_BIT_RATE (MAX_FRAME_SIZE * FPS * 8)
#define MAX_BIT_RATE (MIN_BIT_RATE)
#define LINE_SIZE_BYTES (WIDTH * BITS_PER_PIXEL / 8)
#define HEADER_BYTES (12)
#define PAYLOAD_SIZE (LINE_SIZE_BYTES + HEADER_BYTES)

/* Interval defined in 100ns units */
#define FRAME_INTERVAL (1000000/FPS)

```

The other alternate setting of this interface has the data streaming isochronous endpoint and it is the operational alternate setting. The class-specific descriptors are not repeated in this alternate setting.

The *Standard VS Isochronous Endpoint descriptor* of the alternate setting 1 of the UVC example is shown below:

```

/* Standard VS Isochronous Video Data Endpoint Descriptor */
0x07, /* 0 bLength */
USB_DESCRIPTOR_ENDPOINT, /* 1 bDescriptorType */
(VIDEO_DATA_EP_NUM | 0x80), /* 2 bEndpointAddress - IN Endpoint */
0x05, /* 3 bmAttributes - Isochronous EP (Asynchronous) */
0x00, 0x04, /* 4 wMaxPacketSize 1x 1024 bytes */
0x01, /* 6 bInterval */

```

The above code shows that the maximum packet size of the endpoint is 1024 bytes and the 'bInterval' of 0x01 requests host to poll the endpoint every microframe (125 us).

In general, USB video devices supports a set of video parameter combinations (including video format, frame size and frame rate) and multiple alternate settings with different maximum packet size endpoints. This enables the host to select the appropriate alternate setting that provides only the required bandwidth for a given video parameter combination.

USB String Descriptors

String descriptors provide human readable information for the device and can be configured with specific USB product information. The descriptors are placed in an array as shown below:

```

/* String table - unsafe as accessed via shared memory */
static char * unsafe_stringDescriptors[] =
{
    "\x09\x04", /* Language ID string (US English) */
    "XMOS", /* iManufacturer */
    "XMOS USB Video Device", /* iProduct */
    "Config", /* iConfiguration string */
};

```

The XMOS USB library will take care of encoding the strings into Unicode and structure the content into USB string descriptor format.

3.8 USB Standard and Class-Specific requests

In *usb_video.xc* there is a function *Endpoint0()* which handles all the USB control requests sent by host to the control endpoint 0. USB control requests includes both standard USB requests and the UVC class-specific requests.

In *Endpoint0()* function, a USB request is received as a setup packet by calling *USB_GetSetupPacket()* library function. The setup packet structure is then examined to distinguish between standard and class-specific requests.

The XMOS USB library provides a function *USB_StandardRequests()* to handle the standard USB requests. This function is called with setup packet and descriptors structures as shown below

```
/* Returns XUD_RES_OKAY if handled okay,
 * XUD_RES_ERR if request was not handled (STALLED),
 * XUD_RES_RST for USB Reset */
unsafe{
    result = USB_StandardRequests(ep0_out, ep0_in, devDesc,
    sizeof(devDesc), cfgDesc, sizeof(cfgDesc),
    null, 0, null, 0, stringDescriptors, sizeof(stringDescriptors)/
    ← sizeof(stringDescriptors[0]),
    sp, usbBusSpeed);
}
```

The video class interfaces use endpoint 0 as the control element and receives all class-specific requests on it. The class-specific requests are used to set and get video related controls. These request are divided into:

- ▶ VideoControl requests
- ▶ VideoStreaming requests

The function *UVC_InterfaceClassRequests()* present in *uvc_req.c* handles the class-specific requests. The defines corresponding to the class-specific request codes are present in *uvc_defs.h* as shown below.

```
/* Video Class-specific Request codes */
#define SET_CUR 0x01
#define GET_CUR 0x81
#define GET_MIN 0x82
#define GET_MAX 0x83
#define GET_RES 0x84
#define GET_LEN 0x85
#define GET_INFO 0x86
#define GET_DEF 0x87
```

In the UVC example, the SET and GET requests for Video Probe and Commit Controls are handled. The Video Probe and Commit Controls are involved in the negotiation of streaming parameters between the host and the device. The following code shows the structure of the streaming parameters that are negotiated with those Controls.

```
/* Video Probe and Commit Controls (Table 4-47 , UVC 1.1) */
typedef struct
{
    unsigned short bmHint;
    unsigned char bFormatIndex;
    unsigned char bFrameIndex;
    unsigned int dwFrameInterval;
    unsigned short wKeyFrameRate;
    unsigned short wPFrameRate;
    unsigned short wCompQuality;
    unsigned short wCompWindowSize;
    unsigned short wDelay;
    unsigned int dwMaxVideoFrameSize;
    unsigned int dwMaxPayloadTransferSize;
    unsigned int dwClockFrequency;
    unsigned char bmFramingInfo;
    unsigned char bPreferredVersion;
    unsigned char bMinVersion;
    unsigned char bMaxVersion;
} __attribute__((packed)) UVC_ProbeCommit_Ctrl_t;
```

The demo application doesn't have multiple set of streaming parameters and therefore the GET_DEF, GET_MIN, GET_MAX and GET_CUR requests are handled similarly and return same values to the host.

This source code can be easily extended to support more class-specific requests.

3.9 Video data streaming

Streaming video data between device and host takes place through the streaming endpoint of the VideoStreaming interface. The video is streamed by continuously transmitting the video samples at a particular rate. A video sample refers to an encoded block of video data that the format-specific decoder is able to accept and interpret in a single transmission.

In the UVC example, the video data is packed in 4:2:2 YUV format (YUY2) and a video sample corresponds to a single video frame of 480x270 pixels. Each video sample is split into multiple class-defined Payload Transfers. A Payload Transfer is composed of the class-defined payload header followed by the video payload data. The payload format is as shown below:

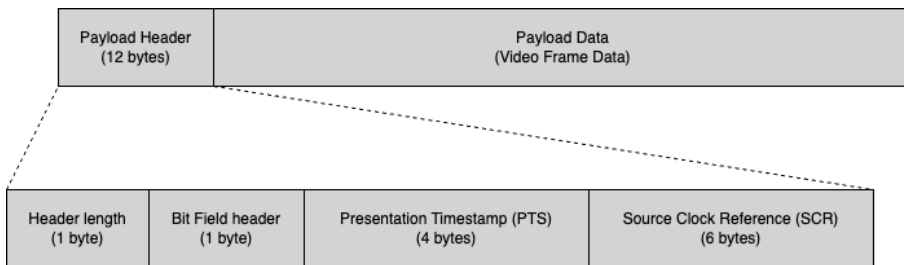


Fig. 5: Payload format for uncompressed streams

For an isochronous endpoint, each (micro)frame will contain a single payload transfer. Each payload transfer consists of the payload header followed by the payload data.

The maximum packet size of the isochronous endpoint in the example code is 1024 bytes, therefore excluding the 12 bytes of payload header, 1012 bytes are available for the video data in a single payload transfer. The video payload data in each payload transfer consists of a single scan line of the image.

The function *VideoEndpointsHandler()* present in *usb_video.xc* handles the isochronous video data endpoint. Each payload transfer is carried out by using the *XUD_SetBuffer()* API of the USB library.

For demonstration, a predefined image of size 480x270 pixels in YUV2 format is stored in *app_an00127/imgs/img.h* file. Each time a full frame is transmitted over USB, the starting position of each row shifts by one pixel, creating a motion effect that simulates video.

The code below fills the payload buffer with the payload header:

```

/* Fill the payload buffer with payload header */
/* Make the Payload header */
payload_header_ptr[0] = HEADER_BYTES;
payload_header_ptr[1] = frame;
memcpy(&payload_header_ptr[2], &pts, sizeof(pts));
memcpy(&payload_header_ptr[6], &pts, sizeof(pts));
short sof_count_short = (sofCounts>3) & 2047;
memcpy(&payload_header_ptr[10], &sof_count_short, sizeof(sof_count_short));

```

In the above code, the **pts** is Presentation timestamp and it is obtained from a timer running in the xCORE device at 100MHz. The **pts** and **sofCounts** (count of USB SOF) are used to arrive at the Source Clock reference field.

This is followed by filling the payload data in the the payload buffer with one line of the image. The line start pixel for a give frame transfer is governed by **increment** which is incremented by 1 every frame transfer to simulate a rolling effect.

```

/* Fill the payload buffer with payload data (one line of the image) */
unsigned start_of_line = line_count * (LINE_SIZE_BYTES / sizeof(int));
unsigned middle_of_line = start_of_line + increment;
unsigned line_size = LINE_SIZE_BYTES / sizeof(int);

// Ensure increment does not exceed line_size
if (increment > line_size) {
    increment = line_size;
}

// Copy data from middle_of_line to the end of the line
for (unsigned i = 0; i < line_size - increment; i++) {
    payload_data_ptr[i] = img_ptr[middle_of_line++];
}

// Copy data from start_of_line to increment
for (unsigned j = 0; j < increment; j++) {
    payload_data_ptr[line_size - increment + j] = img_ptr[start_of_line++];
}

```

The payload transfer is carried out by calling `XUD_SetBuffer()` as shown below:

```

/* Payload transfer */
result = XUD_SetBuffer(episo_in, (gVideoBuffer, unsigned char[]), PAYLOAD_SIZE);

```

When the complete image is transferred, `line_count` rolls back to 0 and `increment` is incremented by one to move the line start pixel for the next transfer.

```

if (line_count >= HEIGHT)
{
    line_count = 0;
    frame = frame ^ 1; /* Toggle FID bit */
    sofCounts += HEIGHT;
    increment++;
    if (increment >= LINE_SIZE_BYTES / sizeof(int)) {
        increment = 0;
    }
}

```



```
cd app_an00127
cmake -G "Unix Makefiles" -B build
```

If any dependencies are missing it is at this configure step that they will be downloaded by the build system.

Finally, the application binary can be built using **xmake**:

```
xmake -C build
```

This command will cause a binary *app_an00127.xe* to be generated in the *app_an00127/bin* directory,

3.12 Launching the demo device

Once the demo example has been built the application can be executed on the XK-EVK-XU316.

Once built, the binary *app_an00127.xe* will be generated in the *app_an00127/bin* directory.

Launching from the command line

From the command line the **xrun** tool is used to download and run the code on the xCORE device.

In a terminal with XTC tools sourced, from the **app_an00127/bin** directory, run:

```
xrun --xscope app_an00127.xe
```

Once this command has executed the application will be running on the xCORE device and the XMOS USB video device should have enumerated on the host machine.

3.13 Running the demo

The demo can be run on any OS that has support for USB Video class driver. Windows, Linux and macOS have native support for UVC driver. The following sections describe in detail on how to run the demo on those OS platforms.

Running on Windows

- In Microsoft Windows, When the USB Video device enumerates the host driver will be installed to get the device ready for operation. Fig. 7 shows the dialog that completes installation of driver for the *XMOS USB Video Device*.

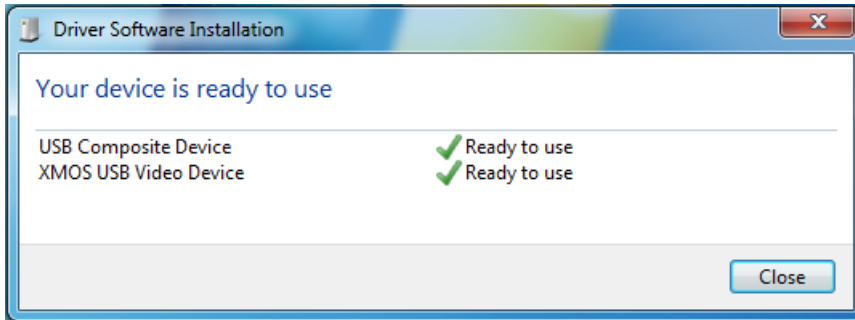


Fig. 7: Driver installation for the Video device

- After the driver is installed properly, use any video capture softwares like VLC Media player, AmCap etc to open the *XMOS USB Video Device*.
- Open VLC Media player, select *Media* menu and click *Open Capture Device....* This will open a dialog window on which you can select the Video device as shown in Fig. 8

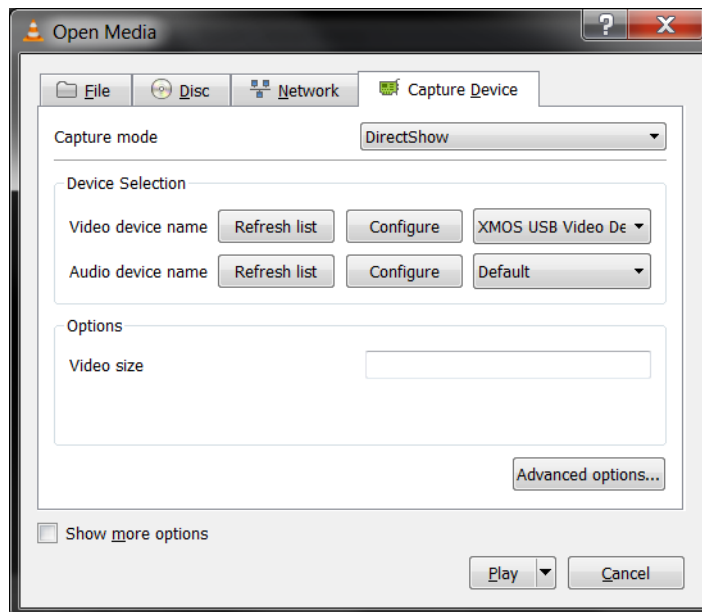


Fig. 8: Open Video device in VLC Media player

- Click *Play* to see the demo video streamed out of the Video device. The video is a rolling image of a skyline. [Fig. 9](#) shows a snapshot of the video.

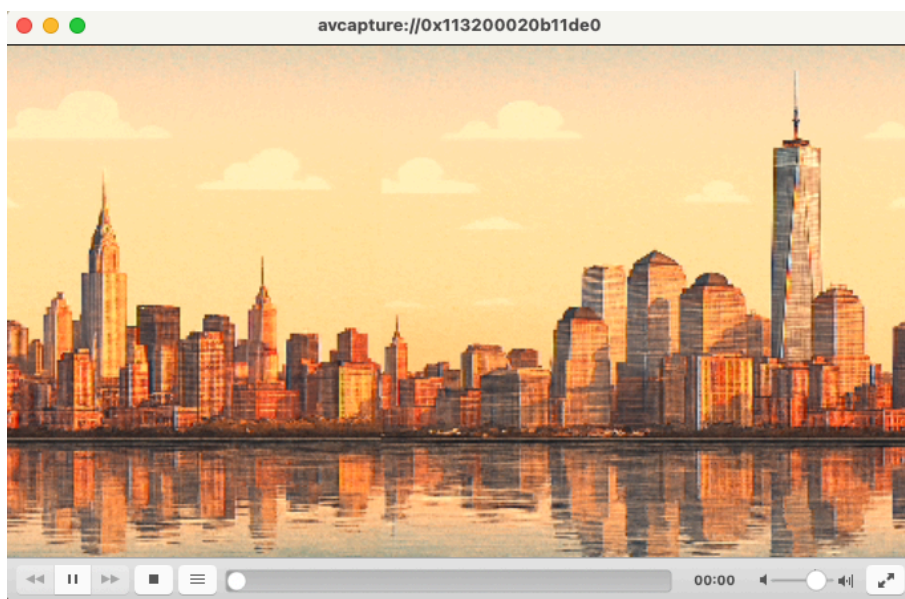


Fig. 9: Video streamed from the XMOS USB Video device

Running on macOS

- On macOS, once the USB Video device is enumerated the UVC driver will be loaded by the host to get the device ready for operation. The device will have enumerated as *XMOS USB Video Device*.
- The Video device can be opened using any video capture software. *Photo Booth* is one such application that comes by default with Mac. Open *Photo Booth* application, click on 'Camera' menu and then select *XMOS USB Video Device*. The application will then show the video streamed out of the USB device. The demo video shows the skyline of a city moving horizontally in a continuous loop.

Running on Linux

- Under Linux, when the device enumerates the native UVC driver will be loaded and the device will be mounted as */dev/videoX* where 'X' is a number.
- Use any video capture software like VLC Media player, Cheese, luvview etc to open the Video device.
- Open VLC Media player, select *Media* menu and click *Open Capture Device....*. This will open a dialog window on which the video device can be selected as shown in [Fig. 10](#)
- Click *Play* to see the demo video streamed out of the *XMOS USB Video Device*. The demo video shows the skyline moving horizontally in a continuous loop

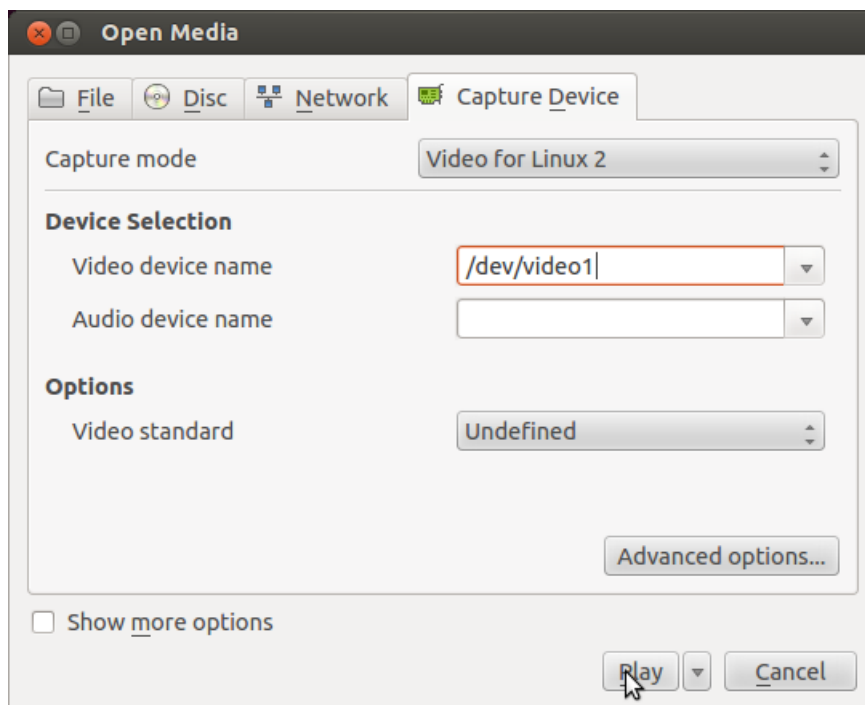


Fig. 10: Open Video device in VLC Media player

4 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>
- ▶ XMOS application build and dependency management system; *xcommon-cmake*
<https://www.xmos.com/file/xcommon-cmake-documentation/?version=latest>
- ▶ USB 2.0 Specification
<https://www.usb.org/document-library/usb-20-specification>
- ▶ USB Video Class Specification v1.1, USB.org:
<https://www.usb.org/document-library/video-class-v11-document-set>
- ▶ YUV Video Format
<http://en.wikipedia.org/wiki/YUV>
<https://www.kernel.org/doc/html/v4.8/media/uapi/v4l/pixfmt-yuyv.html>



Copyright © 2025, All Rights Reserved.

Xmos Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. Xmos Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, xCore, xcore.ai, and the XMOS logo are registered trademarks of XMOS Ltd in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.

