Implementing a USB video and audio composite device using EZ-USB™ FX20/FX10/FX5N/FX5 controller

About this document

Scope and purpose

This application note provides implementation details for the USB Video Class (UVC) and USB Audio Class (UAC), as well as guidance on developing firmware for UVC and UAC composite device. Conforming to these classes enables a camera and audio device to operate with built-in PC drivers and host applications, such as Windows Camera app, Webcamoid, and VLC media player. This application note also offers insight into using the Infineon product EZ-USB™ FX20 as a composite device to interface with a camera (image sensor)/HDMI frame grabber and microphone/audio to stream uncompressed data into a PC.

This application note describes the synchronous slave FIFO interface of the EZ-USB™ FX20 with the Efinix Ti180 FPGA and provides common trouble shooting methods.

Intended audience

This document is primarily intended for individuals who wish to design and implement a composite device (UVC + UAC) using EZ-USB™ FX20 slave FIFO interface and PDM interface with an FPGA.

Table of contents

1 Introduction

The EZ-USB™ FX20/FX10/FX5N/FX5 is a family of USB peripheral controllers that comply with the USB 3.2 Gen 2x2 specification and are backward compatible with the USB 3.1 and USB 2.0 specification. EZ-USB™ FX20 consist of dual Arm® Cortex®-M4 and M0+ core CPUs, a 512 KB flash, 128 KB SRAM, 128 KB ROM, seven Serial Communication Blocks (SCBs), a cryptography accelerator, and a high bandwidth data subsystem providing DMA data transfers between LVDS/LVCMOS and USB ports at speeds up to 20 Gbps. An additional 1 MB SRAM is included in the high bandwidth data subsystem to provide buffering for USB data. EZ-USB™ FX20 also supports USB Type-C plug orientation detection and flip-mux function without the need for external logic.

For a detailed understanding of the EZ-USB™ FX device architecture and features, see Figure 1, which shows the block diagram of the device. The architectural reference manual provides a comprehensive description of the device's architecture and functionality, including the operation of each resource in all modes. Additionally, the manual offers specific guidance on the use of associated registers, helping to ensure effective implementation and configuration of the device.

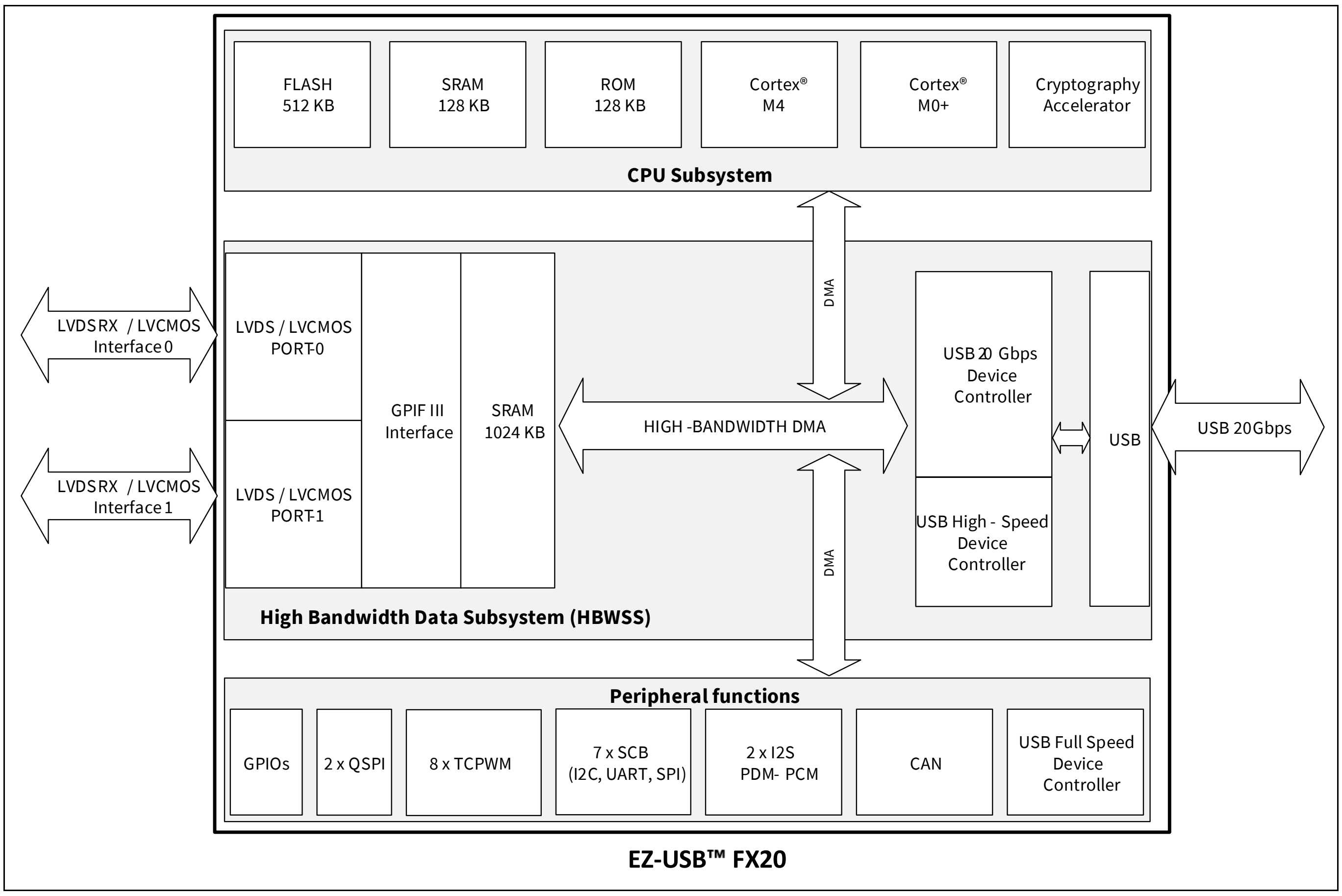


Figure 1 **EZ-USB™ FX block diagram**

EZ-USB™ FX20 controller is typically used in cameras, video, imaging, and data acquisition. Figure 2 shows EZ-USB™ FX20 controller used in a video and audio application. This Application note provides guidance for implementing a UVC and UAC composite device with EZ-USB™ FX20 controller.

Figure 2 shows an EZ-USB™ FX20 controller connected to a PC equipped with a USB 3.2 Gen 2x2 port. EZ-USB™ FX20 is also connected to an FPGA using the LVDS/LVCMOS interface. The FPGA decodes the image sensor/ISP stream received over MIPI CSI-2/LVDS and sends the video data to EZ-USB™ FX20 over the

LVDS/LVCMOS interface. The microphone is directly interfaced through Pulse Density Modulation (PDM) port of the EZ-USB™ FX20 controller.

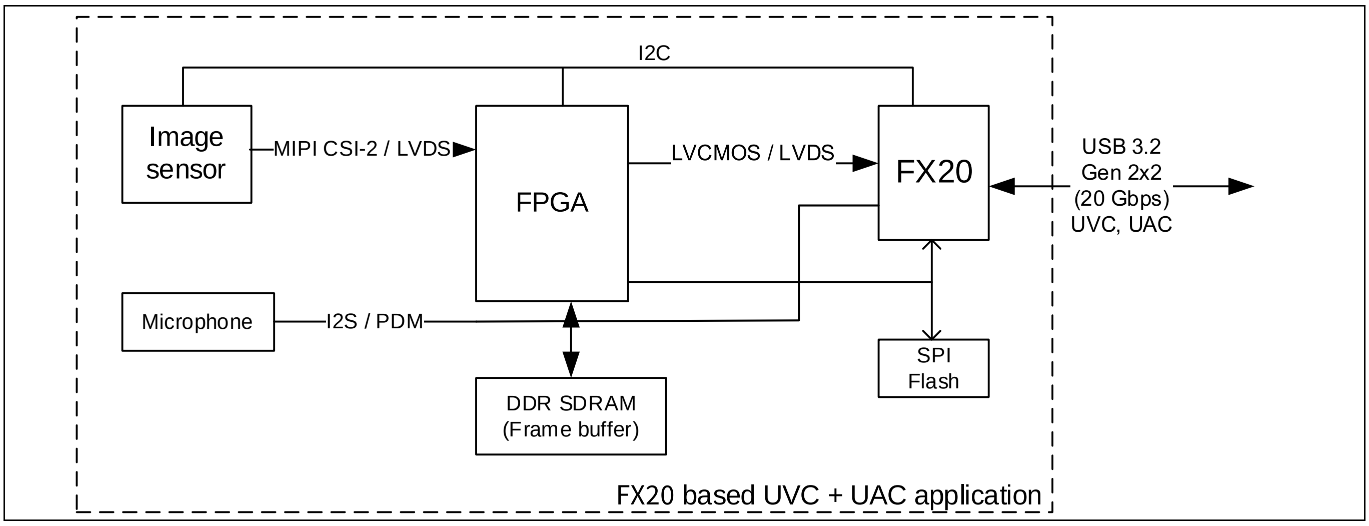


Figure 2 EZ-USB™ FX20 based video and audio application

2 USB Video Class (UVC) standard

USB Video Class (UVC) is a standard introduced by the USB Implementers Forum (USB-IF) that defines the protocol for video streaming functionality over a USB connection. It enables devices, such as webcams, digital camcorders, and other video capturing devices to be quickly recognized and used by computers without the need for special drivers. UVC is an extension of the USB device class specification designed to ensure cross-platform compatibility and plug-and-play functionality. Its standardization simplifies the process of connecting and using video devices, making it a popular choice for manufacturers and end-users alike.

2.1 Key features and benefits of USB Video Class (UVC)

- **Plug-and-Play:** One of the main advantages of UVC is its plug-and-play capability. When you connect a UVC-compliant device to a computer, the operating system automatically detects and installs the necessary drivers to support the device, eliminating the need for additional software installation
- **Driverless Operation:** UVC devices do not require specific drivers for each operating system, as modern operating system including Windows, macOS, Linux supports native driver for UVC
- **Stream Control:** UVC defines a standard set of controls for managing video streaming parameters, such as resolution, frame rate, exposure, focus, and white balance. This allows applications and operating systems to adjust the settings of the UVC-compliant camera
- **Interoperability:** With UVC, video streaming devices from different manufacturers can interoperate seamlessly. As long as they adhere to the UVC standard, they should work together without any conflicts

2.2 UVC version support

The UVC specification version has gone through several revisions to improve and expand its capabilities. Each version of UVC introduced new features and enhancements, making it easier to integrate video devices with various host systems without the need for specialized drivers.

- UVC 1.0, The initial release of UVC, was introduced around 2005
- UVC 1.1, released in 2006, brought improvements to UVC 1.0. The size of the processing unit was reduced by one-byte as it does not support analog video standards
- UVC 1.1 includes a new 16-byte GUID field, the length of the MPEG2TS format descriptor was increased to 23 bytes from 7 bytes in UVC 1.0
- UVC 1.5, also known as UVC H.264, was introduced in 2009. UVC 1.1 does not support H.264 compressed video format, while UVC 1.5 supports it and introduced new controls in the camera terminal, processing unit, and encoding unit

Note: The code example attached with this application note supports UVC specification 1.1 and 1.5.

2.3 UVC implementation details

Conforming to the UVC specification requires two EZ-USB™ FX20 code modules:

- Enumeration using UVC descriptors
- Handling UVC requests

2.3.1 Enumeration using UVC descriptors

A USB device's capabilities or characteristics are described using descriptors. The code attached to this application note includes a file named *fx3g2_descriptors.c* (explained in Section 2.4.1), which contains the UVC enumeration data. The USB specification that defines the format for UVC descriptors is available in the video class section at usb.org. This section gives a high-level overview of the descriptors.

A UVC device comprises four logical elements:

1. Input camera terminal (IT)
2. Output terminal (OT)
3. Processing unit (PU)
4. Extension unit (EU)

The elements are connected in the descriptors, as shown in Figure 3. Connections are made between elements by associating terminal numbers in the descriptors. For example, the Input (camera) terminal descriptor declares its ID to be 1, and the processing unit descriptor specifies its input connection to have the ID of 1, logically connecting it to the input terminal. The output terminal descriptor specifies which USB endpoint to use; in this case, BULK-IN endpoint 1.

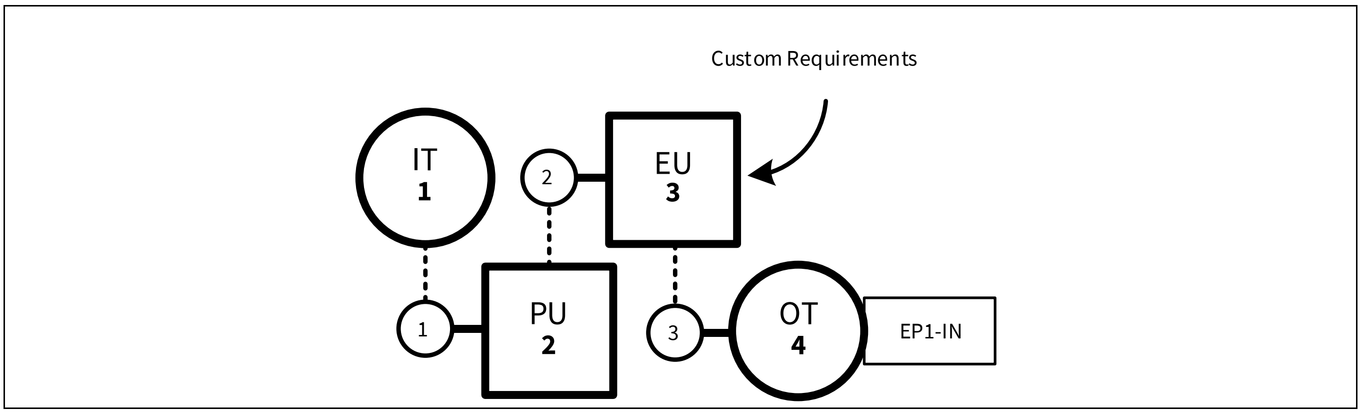


Figure 3 UVC diagram of the camera architecture

The descriptors also include video properties, such as width, height, frame rate, frame size, and bit depth, and control properties, such as brightness, exposure, gain, contrast, pan, tilt, and zoom (PTZ). Apart from the standard video controls, the extension unit provides additional controls based on user requirements.

After successful enumeration, the device will be listed under **Camera** in the **Windows Device Manager**.

Note: The maximum bandwidth achievable using the ISOC-IN endpoint under UVC for USB 3.2 in Windows 7/8 machine is 24 Mbps because the UVC driver limits the BURST to 1 for the ISOC endpoint. But with the Windows 10 machine, BURST can be set to 15 to achieve maximum ISOC bandwidth (10 Gbps). There is no such limitation for BULK endpoint. Therefore, the ISOC-IN endpoint is not used in this project. See this forum for more information.

2.3.2 Handling UVC requests

After the host enumerates the camera, the UVC driver sends a series of requests to the camera to determine operational characteristics. This is called the capability request phase. It precedes the streaming phase, in which the host application starts streaming video. The EZ-USB™FX20 firmware responds to the requests that arrive over the USB control endpoint (EP0).

For example, suppose a UVC device indicates that it supports a brightness control in one of its USB descriptors. During the capability request phase, the UVC driver queries the device to discover the relevant brightness parameters.

When a host application makes a request to change the brightness value, the UVC driver issues a SET control request to change the brightness value (SET_CUR).

Similarly, when the host application chooses to stream a supported video format/frame rate/frame size, it issues streaming requests. There are two types: PROBE and COMMIT. PROBE requests are used to determine if the UVC device is ready to accept changes to the streaming mode. A streaming mode is a combination of image format, frame size, and frame rate. The COMMIT control is used to configure the hardware with the streaming parameters from the Probe control.

2.4 USB video class requirements

The firmware project for this application note is in the folder named *cyfxuvc_uac_an38689*. This section explains how the UVC requirements are satisfied by the example project. UVC requires a device to:

- Enumerate with the UVC-specific USB descriptors.
- Handle SET/GET UVC-specific requests for the UVC control and stream capabilities reported in the USB descriptors.
- Stream video data in a UVC-conformant color format.
- Add a UVC conformant header for every image payload.

Details of these requirements can be found in the UVC Specification v1.5.

2.4.1 USB descriptors for UVC

The *fx3g2_descriptors.c* file contains the USB descriptor tables. The byte arrays *CyFxUSBHSConfigDscr* (Hi-Speed) and *CyFxUSBSSConfigDscr* (SuperSpeed and SuperSpeed Plus) contain the UVC-specific descriptors. These descriptors implement the following tree of sub-descriptors:

Configuration descriptor

- Interface association descriptor
- Video control (VC) interface descriptor
  - VC interface header descriptor
    - Input (camera) terminal descriptor
    - Processing unit descriptor
    - Extension unit descriptor
    - Output terminal descriptor
  - VC status interrupt endpoint descriptor
- Video streaming (VS) interface descriptor
  - VS interface input header descriptor

    - VS format descriptor
    - VS frame descriptor
- BULK-IN video endpoint descriptor

The configuration descriptor is a standard USB descriptor that defines the functionality of the USB device in its sub-descriptors. The interface association descriptor is used to indicate to the host that the device conforms to a standard USB class. Here, this descriptor reports a UVC-conformant device with two interfaces: video control (VC) interface and video streaming (VS) Interface. Having two separate interfaces makes the UVC device a USB composite device.

2.4.1.1 Video control (VC) interface

The VC interface descriptor and its sub-descriptors report all the control interface-related capabilities. Examples include brightness, contrast, hue, exposure, and PTZ controls.

The VC interface header descriptor is a UVC-specific interface descriptor that points to the VS interfaces to which this VC Interface belongs.

The input (camera) terminal descriptor, the processing unit descriptor, the extension unit descriptor, and the output terminal descriptor contain bitfields that describe features supported by the respective terminal or unit.

The camera terminal controls mechanical (or equivalent digital) features, such as exposure and the PTZ of the device that transmits the video stream.

The processing unit controls image attributes, such as brightness, contrast, and hue of the video being streamed through it.

The extension unit allows vendor-specific features to be added, much like standard USB vendor requests. In this design, the extension unit is empty, but a sample design is implemented, and it can be enabled to get or set the device firmware version. If the extension unit is utilized, the standard host application will not see its features unless the host application is designed to recognize them. A sample host application is provided to get or set the device firmware version. You can design your own extension controls and host application to query these controls. Some features that can be supported include getting the device firmware version and hardware IDs, writing to sensor/image signal processor (ISP) registers, and so on. The device can support more than one extension unit. For more details on the UVC extension unit with the associated firmware project, see 2.4.2.

The output terminal is used to describe an interface between these units (IT, PU, EU) and the host. The VC status interrupt endpoint descriptor is a standard USB descriptor for an Interrupt endpoint. This endpoint can be used to communicate UVC-specific status information. The functionality of this endpoint is outside the scope of this application note.

The UVC specification divides these functionalities so that you can easily structure the implementation of the class-specific control requests. However, the implementation of these functionalities is application-specific. The supported control capabilities are reported in the bitfield bmControls (*fx3g2_descriptors.c*) of the respective terminal/unit descriptor by setting the corresponding capability bits to '1'. The UVC device driver polls for details about the control on enumeration. The polling for details is carried out over EP0 requests. All such requests, including the video streaming requests, are handled by the `Cy_USB_AppSetupCallback` function in the *cy_usb_app.c* file.

2.4.1.2 Video streaming (VS) interface

The video streaming interface descriptor and its sub-descriptors report the various frame formats (e.g., uncompressed, MPEG, H.264), frame resolutions (width, height, and bit depth), and frame rates. Based on the values reported, the host application can choose to switch streaming modes by selecting supported combinations of frame formats, frame resolutions, and frame rates.

The VS interface input header descriptor specifies the number of VS format descriptors that follow.

The VS format descriptor contains the images' aspect ratio and the color format, such as uncompressed or compressed.

The VS frame descriptor contains image resolution and all supported frame rates for that resolution. If the camera supports different resolutions, multiple VS frame descriptors follow the VS format descriptor.

The BULK-IN video endpoint descriptor is a standard USB endpoint descriptor that contains information about the bulk endpoint used for streaming video.

This example uses four resolutions with multiple frame rates. Its image characteristics are contained in two descriptor tables and the probe/commit control structure table, as shown in the following tables (only relevant byte offsets are shown).

Table 1 VS format descriptor value

VS format descriptor byte offset	Characteristic	SuperSpeed Plus value	Hi-Speed value
4	Number of frame descriptors with the described format	4	1
5-20	GUID of streaming-encoding format	'Y', 'U', 'Y', '2',0X00, 0X00, 0X10, 0X00, 0X80, 0X00, 0X00, 0XAA,0X00, 0X38, 0X9B, 0X71	'Y', 'U', 'Y', '2',0X00, 0X00, 0X10, 0X00, 0X80, 0X00, 0X00, 0XAA,0X00, 0X38, 0X9B, 0X71
21	Number of bits per pixel	16	16
23-24	Width to height ratio	16:9	4:3

Table 2 VS frame descriptor value

VS frame descriptor byte offset	Characteristic	SuperSpeed value	Hi-Speed value
3	Frame descriptor index	1	1
5-6	Width in pixel	3840	640
7-8	Height in pixel	2160	480
13-16	Maximum bit rate at the shortest frame interval in unit of bps	3840x2160x16x30 fps (16 bits/pixel)	640x480x16x15 fps (16 bits/pixel)
21-24, also 26-29	Frame interval in 100 ns units	0x51615 (30 fps)	0xA2C2A (15 fps

Note that multiple-byte values are listed LSB first (little-endian), so, for example, the frame rate is 0x00051615, which is 33.33 milliseconds, or 30 fps.

Table 3 Probe/commit structure values

Probe/commit structure byte offset	Characteristic	SuperSpeed value	Hi-Speed value
2	Format index	1	1
3	Frame index	1	1
4-7	Frame interval in 100 ns units	0x51615 (30 fps)	0xA2C2A (15 fps)
18-21	Maximum image size in bytes	3840x2160x2 (2 bytes/pixel)	640x480x2 (2 bytes/pixel)

This design can be adapted to support different image resolutions, such as 1080p, 720p, etc. by modifying the entries in these three tables.

2.4.2 UVC-specific requests

The UVC specification uses USB control endpoint EP0 to communicate control and streaming requests to the UVC device. These requests are used to discover and change the attributes of the video-related controls. The UVC specification defines these video-related controls as capabilities. These capabilities allow you to change image properties or to stream video. A capability (first item) can be a video control property, such as brightness, contrast, and hue, or video stream mode properties, such as the color format, frame size, and frame rate. Capabilities are reported via the UVC-specific section of the USB Configuration descriptor. Each of the capabilities has attributes. The attributes of a capability are as follows:

- The minimum value
- The maximum value
- The number of values between the minimum and the maximum
- The default values
- The current value

SET and GET are the two types of UVC-specific requests. SET is used to change the current value of an attribute, while GET is used to read an attribute.

Here is a list of UVC-specific requests:

- SET_CUR is the only type of SET request
- GET_CUR reads the current value
- GET_MIN reads the minimum supported value
- GET_MAX reads the maximum supported value
- GET_RES reads the resolution (step value to indicate the supported values between min and max)
- GET_DEF reads the default value (return the appropriate value to set defaults for any parameter, e.g., default frame resolution, etc.)
- GET_LEN reads the size of the attribute in bytes
- GET_INFO queries the status/support for specific capability

The UVC specification defines these requests as either mandatory or optional for a given capability. For example, if the SET_CUR request is optional for a particular capability, its presence is determined through the GET_INFO request. If the camera does not support a certain request for a capability, it must indicate this by stalling the control endpoint when the request is issued from the host to the camera.

There are byte fields in these requests that qualify their target capability. These byte fields have a hierarchy, which follows the same structure as the UVC-specific descriptors described in Section 2.3.1. The first level identifies the interface (video control or video streaming).

If the first level identifies the interface as video control, the second level identifies the terminal or unit, and the third level identifies the capability of that terminal or unit. For example, if the target capability is the brightness control, then:

- First level = video control
- Second level = processing unit
- Third level = brightness control

If the first level identifies the interface as video streaming, the second level would be PROBE or COMMIT. There is no third level. When the host wants the UVC device to start streaming or to change the streaming mode, the host first determines if the device supports the new streaming mode. To determine this, the host sends a series of SET and GET requests with the second level set to PROBE. The device either accepts or rejects the change to the streaming mode. If the device accepts the change request, the host confirms it by sending the SET_CUR request with the second level set to COMMIT. This interaction between the host and the device is illustrated in Figure 4. The following three flowcharts show how the host interacts with a UVC device.

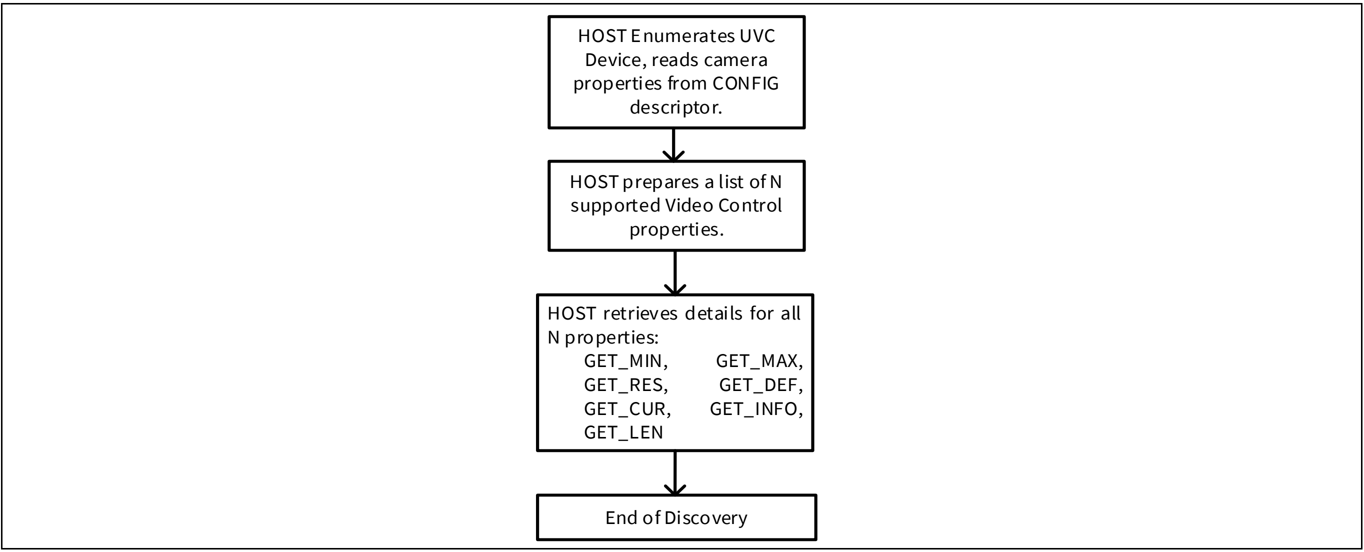

HOST Enumerates UVC Device, reads camera properties from CONFIG descriptor.
HOST prepares a list of N supported Video Control properties.
HOST retrieves details for all N properties: GET_MIN, GET_MAX, GET_RES, GET_DEF, GET_CUR, GET_INFO, GET_LEN
End of Discovery

Figure 4 **The UVC enumeration and discovery flow figure**

When the UVC device is plugged into USB, the host enumerates it and discovers details about the properties supported by the camera (Figure 5).

During a video operation, a camera operator may change a camera property, such as brightness, in a display dialog presented by the host application. Figure 6 shows this interaction.

This kind of implementation is a synchronous control transfer. As per the UVC specification, a device can support two kinds of control requests: synchronous control transfer and asynchronous control transfer.

Standard UVC requests like brightness, pan, tilt, and zoom (PTZ) requests take lesser time to complete (less than 10 ms). In this case, UVC device writes to a sensor/ISP register and doesn't wait for any acknowledgment from the sensor/ISP. When host drivers send a SET_CUR request, the device responds within 10 ms with a success (acknowledge the request) or failure (stall the request). As per UVC specs, this request is synchronous in nature. All standard UVC video control requests can be supported through the **synchronous control transfers**.

Now assume that a host driver sets an exposure value. As per design implementation, the sensor/ISP responds with an acknowledgement in about 100mS. The device always responds to the SET_CUR request with a success for the SET_CUR request and after receiving the acknowledgement from the sensor/ISP, the device loads the control status Interrupt endpoint with a success or failure. This kind of request is an **asynchronous control request**. Note that the asynchronous control request transfers are not handled on this project as all the requests are synchronous in nature.

Note: *For more details on how to use the control status interrupt of the video control interface, see Section 2.4.2.2 "Status Interrupt Endpoint" in the UVC Specification v1.5.*

Note: *Status interrupt endpoint can be used to handle status interrupt endpoint for hardware-triggered still capture. For asynchronous video controls, the status interrupt endpoint response can be populated based on Table 2-2: Status Packet Format (VideoControl Interface as the Originator) of the UVC Specification, with appropriate Control selector value (bSelector) based on the queried video control.*

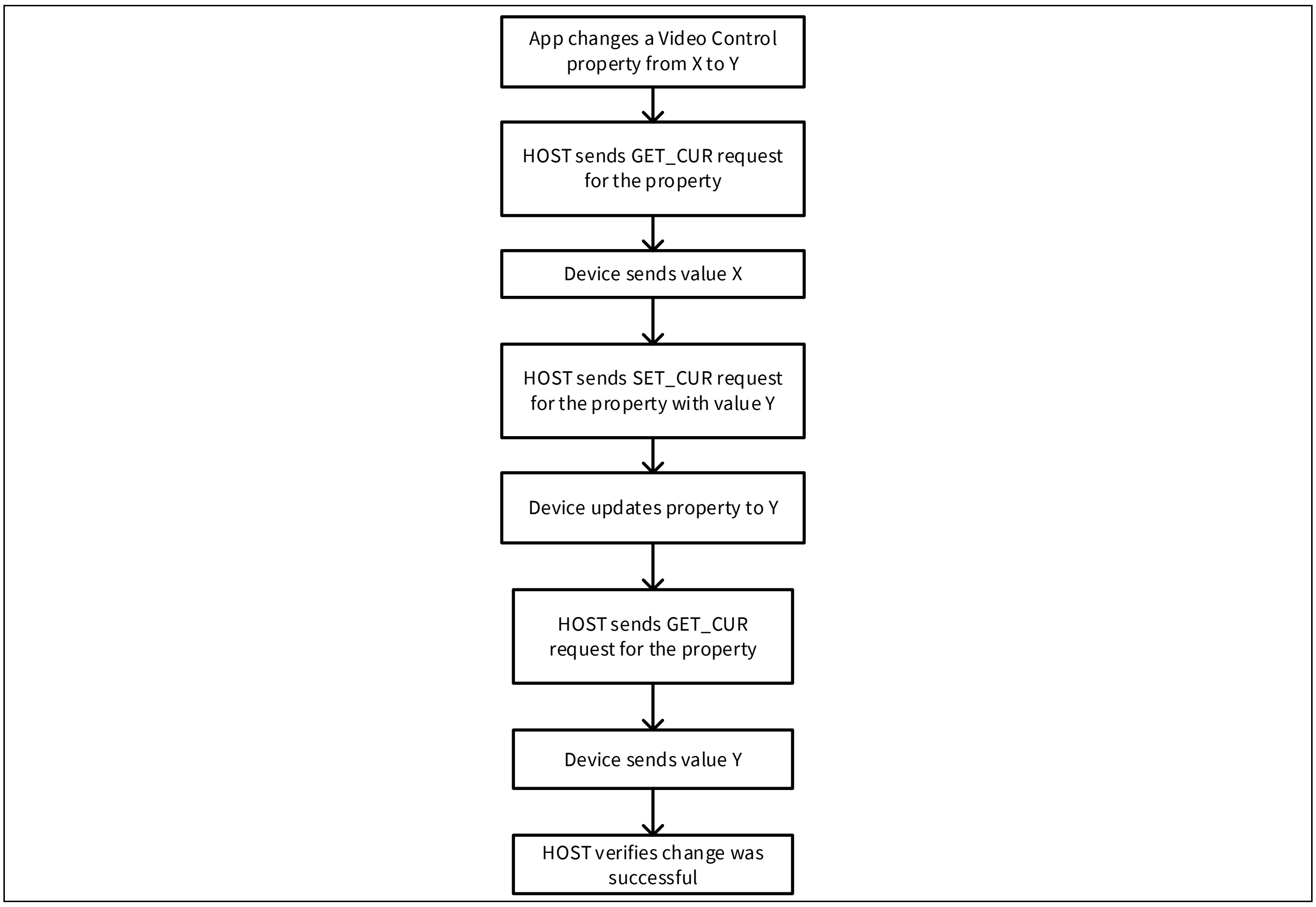

App changes a Video Control property from X to Y
HOST sends GET_CUR request for the property
Device sends value X
HOST sends SET_CUR request for the property with value Y
Device updates property to Y
HOST sends GET_CUR request for the property
Device sends value Y
HOST verifies change was successful

Figure 5 A host application changes a camera setting

Before starting to stream, the host application issues a set of probe requests to discover the possible streaming modes. After the default streaming mode is decided, the UVC driver issues a COMMIT request. This process is shown in Figure 6. At this point, the UVC driver is ready to stream video from the UVC device.

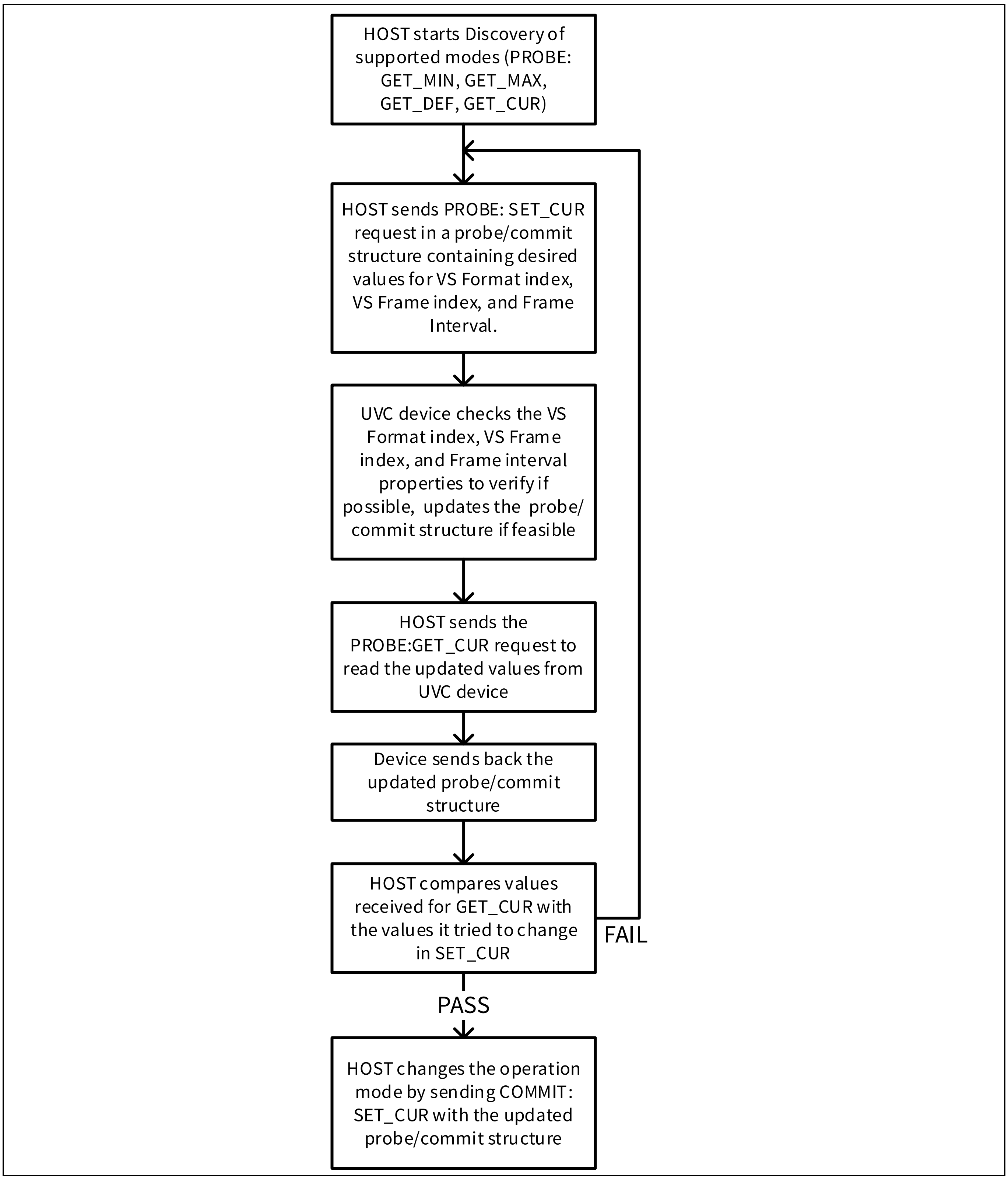

HOST starts Discovery of supported modes (PROBE: GET_MIN, GET_MAX, GET_DEF, GET_CUR)
HOST sends PROBE: SET_CUR request in a probe/commit structure containing desired values for VS Format index, VS Frame index, and Frame Interval.
UVC device checks the VS Format index, VS Frame index, and Frame interval properties to verify if possible, updates the probe/ commit structure if feasible
HOST sends the PROBE:GET_CUR request to read the updated values from UVC device
Device sends back the updated probe/commit structure
HOST compares values received for GET_CUR with the values it tried to change in SET_CUR
FAIL
PASS
HOST changes the operation mode by sending COMMIT: SET_CUR with the updated probe/commit structure

Figure 6 **Host-camera pre-streaming dialog**

2.4.2.1 Control requests – Brightness, PTZ control, and extension unit control

Brightness and PTZ controls are implemented in the associated project. PTZ can be turned on by setting the macro value `UVC_PTZ_EN` to '1' in the *cy_usb_uvc_device.h* file. It is important to note that this control may not be supported by the image sensor, and if that is the case, specific hardware must be designed accordingly. Regardless, the firmware implementation for these controls on the USB side remains the same. However, the image sensor implementation might differ. Therefore, only placeholder functions are provided that implement the USB side of these controls, however, so you will need to write code for the image sensor specific PTZ implementation.

Note: The functions described for UVC controls are implemented in the cy_usb_videocontrol.c file. This file is part of the project stored in the cyfxuvc_an38689 folder, which can be found in the zip file of the source code attached to this application note.

The host application sends video control requests (over EP0) that are targeted to the processing unit for brightness control. All setup requests are handled via the `Cy_USB_AppSetupCallback` function. This function detects whether the host has sent a UVC-specific request (control or stream) and then handles the specific request in the `Cy_USB_AppSetupCallback` function and sets an event flag for only SET_CUR request, labeled as "CY_USB_UVC_DEVICE_SET_CUR_RQT". This flag is processed by the `Cy_USB_UvcDeviceTaskHandler( )` function.

Control requests will trigger the video control request event CY_USB_UVC_DEVICE_VIDEO_CNTRL_RQT flag. The `Cy_USB_UvcDeviceTaskHandler` function looking for any of these flags to be raised, and then it calls the `HandleVCInterfaceRequest` function in order to decode the video control request. Based on the type of control UVC that is being requested, the appropriate function is called.

Modify the function under *CY_UVC_VC_PROCESSING_UNIT_ID* to implement processing unit-related controls (brightness, contrast, hue, and so on). Modify the function under *CY_UVC_VC_CAMERA_TERMINAL_ID* to implement camera terminal controls.

The UVC specification includes details on camera terminal, processing unit, and extension unit USB descriptors.

An extension unit control – to Get or Set device firmware version can be implemented in the EZ-USB™ FX20. For more details on how to design a UVC extension unit in the firmware, see Implementing extension unit control in AN75779 example project. The get or set firmware version control allows you to retrieve and update the firmware version. This can be achieved by using the uvc_extension_app_x86.exe (for 32-bit Windows) or uvc_extension_app_x64 (for 64-bit Windows). Guidelines for running the host application is provided in the readme.txt file. The host application is based on Microsoft's Media foundation class and DirectShow APIs. For additional information on the host application, see the forum discussion.

Note: At this moment, EZ-USB™ FX20 UVC firmware does not support Extension unit control request, it will be added later.

2.4.2.2 Streaming requests – Probe and commit control

The `Cy_USB_AppSetupCallback` function is responsible for handling streaming-related requests. When the UVC driver needs to stream video from a UVC device, the first step is to negotiate with the device. In this phase, the driver sends a set of parameters to the device using PROBE requests, which includes GET_MIN, GET_MAX, GET_RES, and GET_DEF. In response, the EZ-USB™ FX20 firmware returns a PROBE structure that, contains the USB descriptor indices of video format and video frame, frame rate, maximum frame size, and payload size (the number of bytes that the UVC driver can fetch in one transfer).

The switch case for *CY_USB_UVC_VS_PROBE_CONTROL* handles the negotiation phase of the streaming process for either a USB 2.0 or USB 3.2 connection (the properties of the supported video in these modes differ). Note that the reported values for GET_MIN, GET_MAX, GET_DEF, and GET_CUR are the same because the same streaming mode is supported in either USB 2.0 or USB 3.2. These values would differ if multiple streaming modes needed to be supported.

The switch case for *CY_USB_UVC_VS_COMMIT_CONTROL* handles the start of the streaming phase. When the host sends a SET_CUR request for COMMIT control, it indicates that it will start streaming video next. Therefore, when SET_CUR for COMMIT control is received, it triggers the *CY_USB_UVC_DEVICE_SET_CUR_RQT* event. This event, in turn, causes the main application thread's `Cy_USB_UvcDeviceTaskHandler` to call the `Cy_USB_UvcSetCurRqtHandler` function. This function informs the video source about selected video parameters and starts the GPIF III state machine for video streaming.

2.4.3 Video data format: YUY2

The UVC specification supports only a subset of color formats for video data. Therefore, you should choose an image sensor/ISP that streams images in a color format that conforms to the UVC specification. This application note covers an uncompressed color format called YUY2, which is supported by most, but not all, image sensors. The YUY2 color format is a 4:2:2 down-sampled version of the YUV color format. Luminance values Y are sampled for every pixel, but chrominance values U and V are sampled only for even pixels. This creates "macro pixels", each of which describes two image pixels using a total of four bytes. Notice that every other byte is a Y value, and the U and V values represent only even pixels:

- Y0, U0, Y1, V0 (first 2 pixels)
- Y2, U2, Y3, V2 (next 2 pixels)
- Y4, U4, Y5, V4 (next 2 pixels)

See Wikipedia for additional information on color formats.

Note: *If you want to change the video format, the GUID must also be changed in the VS format descriptor. See the Media types for information on the GUIDs for supported formats.*

Although a monochrome image is not supported as a part of the UVC specification, an 8-bit monochrome image can be represented in the YUY2 format by sending the monochrome image data as Y values and setting all the U and V values to 0x80.

2.4.4 UVC video data header

As per the UVC specification, each video payload sent to the host requires a header. This header contains the information about the properties of the image data being transferred. For instance, it contains a "new frame" bit that the image sensor / ISP controller (EZ-USB™ FX20) toggles for every frame. Additionally, the EZ-USB™ FX20 can include Presentation Time Stamp and Source Clock reference in the header and set an error bit to indicate any issues encountered during the streaming of the current frame. This UVC data header is required for every USB transfer. For more information, see the UVC specification. You can see the format of the UVC video data header in Table 4. The code example used in this application note uses a 32-byte header.

Table 4 UVC video data header format

Byte offset	Field name	Description
0	HLF	bHeaderLength - Header length field specifies the length of the header in bytes

Byte offset	Field name	Description
1	BFH	bmHeaderInfo - Bitfield header indicates the type of the image data, status of the video stream and the presence or absence of other fields
2-5	PTS	dwPresentationTime - Presentation time stamp indicates the source clock time in native device clock units
6-11	SCR	scrSourceClock - Source clock reference indicates system time clock and USB start-of-frame (SOF) token counter
12-31	PAD	To ensure 32-byte alignment is required for DMA channel creation, zero padding is required

Note: *See USB_Video_Payload_Frame_Based_1.5.pdf section 2.1 Payload Header available in Video Class v1.5 document at www.usb.org.*

The Header Length Field (HLF) value is always 32 in the example code provided with this application note. Although the Presentation Time Stamp (PTS) and Source Clock Reference (SCR) fields are optional, the firmware example (Firmware link should be added here) populates these fields with zeros.

Table 5 shows the format of the BFH, which is a part of the UVC video data header.

Table 5 Bitfield header (BFH) format

Bit offset	Field name	Description
0	FID	Frame identifier bit toggles at each image frame start boundary and stays constant for the rest of the image frame
1	EOF	End of frame bit indicates the end of a video and is set only in the last USB transfer belonging to an image frame
2	PTS	Presentation time stamp bit indicates the presence of a PTS field in the UVC video data header (1=present)
3	SCR	Source clock reference bit indicates the presence of an SCR field in the UVC video data header (1=present)
4	RES	Reserved, set to 0
5	STI	Still image bit indicates if the video sample belongs to a still image
6	ERR	Error bit indicates an error in the device streaming
7	EOH	End of header bit, when set, indicates the end of the BFH fields

The bitfield header (BFH) keeps changing value at the end of a frame.

UVC header insertion can be done in any of the following methods:

- Adding UVC header by firmware
- Adding UVC header by FPGA
- Adding UVC header using insert metadata feature of EZ-USB™ FX20

Method 1: In this method, the CPU is involved in adding the UVC header. The DMA channel should be created in manual mode by enabling producer and consumer interrupts.

Method 2: In this method, FPGA is preconfigured to insert the header. The FPGA adds the 32-byte UVC header to every payload, which is then sent to the USB host. The DMA channel can be created in Auto mode, and this method can help to get maximum throughput as there is no CPU interaction.

Method 3: In this mode, the 32-byte UVC header is stored in EZ-USB™ FX20 Metadata Memory, and the DMA Hardware inserts the UVC header to each DMA buffer on receiving the Insert metadata command (INMD) over the control lane and commit it to the USB host without CPU interaction. The DMA channel can be created in Auto mode. See the EZ-USB™ FX20 architecture reference manual to know more details about metadata insertion.

In this project, the UVC header can be inserted by FPGA or EZ-USB™ FX20 by setting the appropriate macros in *"cy_usb_app.h"* header file.

- Set 'MANUAL_DMA_CHANNEL' to value '1' and ensure that 'UVC_HEADER_BY_FX20' is updated with value '1'. This will enable the manual DMA channels configuration and UVC header insertion by EZ-USB™ FX20 CPU
- Set 'MANUAL_DMA_CHANNEL' to value '0' and ensure that 'UVC_HEADER_BY_FPGA' is updated with value '1'. This will enable the Auto DMA channel configuration and UVC header insertion by the EFINIX Ti180 FPGA

Figure 7 illustrates how the headers are added to the video data. The 32-byte header is added for every USB bulk transfer, and each transfer includes a total of 63 bulk packets. The USB 3.2 bulk packet size is 1024 bytes.

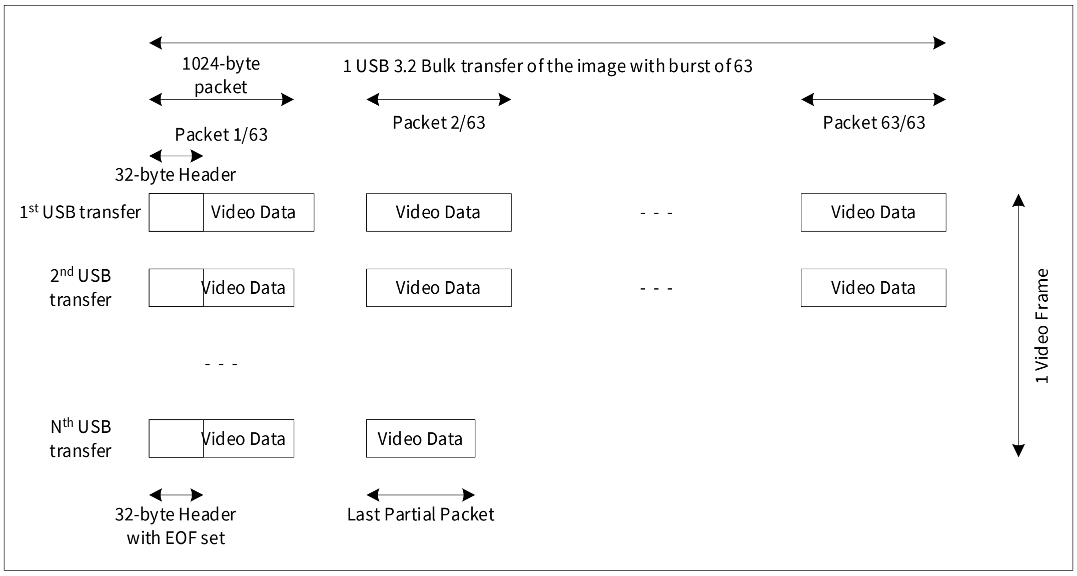


Figure 7 **UVC video data transfers**

3 USB Audio Class (UAC) standard

The USB Audio Class (UAC) was introduced to simplify the process by defining a common set of rules and requirements that audio devices must follow to communicate with USB host devices effectively. This standardization ensures that USB audio devices, such as microphones, speakers, headphones, sound cards, and audio interfaces, can be easily connected and used without the need for specialized drivers or software installations.

The USB Audio class (UAC) uses the isochronous transfer type (Endpoint) to stream audio across a USB link. Isochronous transfers are continuous, real-time transfers that have a pre-negotiated bandwidth. Because isochronous transfers do not have an error recovery mechanism or handshaking, they must support streams of error-tolerant data.

Conforming to the UAC requires two EZ-USB™ FX20 code modules:

- Enumeration using UAC descriptors
- Handling UAC requests. Stream Audio data

3.1 UAC version support

The USB Implementers Forum (USB-IF) is responsible for defining and updating the USB Audio specifications. Below is the list of USB audio class version released by USB-IF.

- **USB Audio 1.0:** This was the initial version of the USB Audio specification and was released in 1998. Supports audio data rate up to 96 kHz. Maximum bit depth is 24 bits
  - Supported low sample rates and bit depths
  - USB Audio Version 1.0 is implemented in the firmware example provided with this application.
- **USB Audio 2.0:** This version was released in 2006 and brought significant improvements over the previous versions. Supports audio data rate up to 192 kHz and supports some feature
  - Support for multiple audio channels
  - Maximum bit depth is 32 bits
  - Support for higher sample rates and bit depths
  - Support for asynchronous data transfers
  - Support for digital signal processing (DSP)
  - Supported more advanced audio devices like external sound cards and audio interfaces
- **USB Audio 3.0:** This version was a proposed update to the specification to support SuperSpeed USB (USB 3.0) capability and was released in 2016. Supports audio data rate up to 384 kHz
  - Number of channel supports is 32. Maximum bit depth is 32 bits
  - Support USB Type-C
- **USB Audio 4.0:** This version is released in April 2023

3.2 Enumeration using UAC descriptor

The code attached to this application note includes a file named *fx3g2_descriptors.c*, which is explained in Section 2.3.1 contains the UAC enumeration data. The USB specification that defines the format for UAC descriptors is available in the Audio class section at usb.org. This section gives a high-level overview of the descriptors.

The endpoint descriptors store stream-related parameters, while the Terminal descriptor stores control-related parameters. The audio functionality is divided into two entities are called Unit and Terminals. The standard UAC specification describes seven different types of standard Unit and Terminals that represent most of the audio functions. In this application note, interfacing the microphone as a UAC device, and it requires two logical elements.

- Input camera terminal (IT)
- Output terminal (OT)

The elements are connected in the descriptors, as shown in Figure 8. Connections are made between elements by associating terminal numbers in the descriptors. For example, the input (Audio) terminal descriptor describes how signals carried over USB are handled, in this case Microphone or Digital audio. The IT declares its ID to be 1, and the Feature unit descriptor specifies that its input connection to have the ID of 1, logically connecting it to the input terminal. The output terminal descriptor specifies which USB endpoint to use, in this case ISOCHRONOUS-IN endpoint 4.

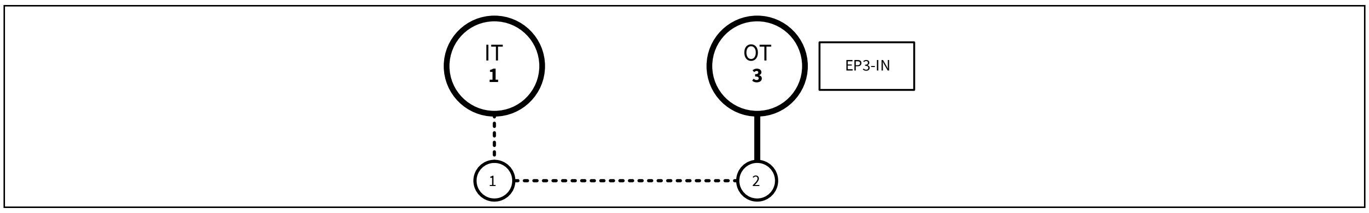


Figure 8 **UAC diagram**

The descriptor also specifies the type of input audio, such as Microphone or Digital audio.

After successful enumeration, the device will be listed under **Audio input and output** in the **Windows Device Manager**.

3.3 Operational code

When the USB device connects to a USB port, the host retrieves information about the device's capabilities through the control transfers during enumeration. After receiving the information during the enumeration process, the host determine how to communicate with the device. To control the functional behavior of an audio function, the host can manipulate the Units and Terminals inside the audio function via the Audio control interface. The device precedes to streaming phase, during which the host application starts streaming Audio. The EZ-USB™ FX20 firmware commits the audio data to the host over the Audio streaming endpoint (EP4).

For example, suppose a UAC device indicates that it supports a volume control in Feature Unit descriptors. The USB Audio Class defines a standard set of control requests that can be used to control various aspects of the audio function, including volume.

When a host application makes a request to change the audio volume, the host issues a SET control request (SET_CUR), which contains the parameters that the host wants to set, including the volume level. The audio function responds with a status message indicating whether the request was successful or not to alter the volume.

Similarly, when the host application chooses to control supported audio control (Volume/Mute/Tone Control), it issues a SET control request (SET_CUR) and GET control request (GET_CUR) to alter the volume level.

3.4 USB audio class requirements

The firmware project for this application note is in the folder named *cyfxuvc_uac_an38689*. This section explains how the UAC requirements are satisfied by the example project. The UAC requires a device to:

- Enumerate with the UAC-specific USB descriptors.
- Stream audio data in a UAC-conformant audio format.

Details of these requirements are found in the UAC Specification.

3.4.1 USB descriptors for UAC

The *fx3g2_descriptors.c* file contains the USB descriptor tables. The byte arrays *CyFxUSBHSConfigDscr* (Hi-Speed) and *CyFxUSBSSConfigDscr* (SuperSpeed and Super Speed Plus) contain the UAC-specific descriptors. These descriptors implement the following tree of sub-descriptors:

Configuration descriptor

- Interface association descriptor
- Audio control (AC) interface descriptor
  - Input (Audio) terminal descriptor
  - Output terminal descriptor
- Audio streaming (AS) interface descriptor
  - AS format descriptor
- ISOCHRONOUS-IN video endpoint descriptor

3.4.1.1 Audio control interface

Audio functions are addressed through their audio interfaces. Each audio function has a single Audio Control (AC) interface and can have several Audio Streaming (AS) interfaces. The Audio Control (AC) interface descriptor and its sub-descriptors report all the control interface-related capabilities. Examples include Volume, Mute, Automatic gain control, equalizer, bass boost, and tone controls.

The input (Audio) terminal descriptor, the feature unit descriptor, and the output terminal descriptor contain bitfields that describe features supported by the respective terminal or unit.

The Input Terminal (IT) is used to interface between the audio function's 'outside world' and other Units in the audio function. It serves as a medium for audio information flowing into the audio function and it represents the source of incoming audio data.

The Output Terminal (OT) represents an ending point for audio channels, it is used to describe an interface between IT, FU, and the host. It serves as an outlet for audio information. Its function is to represent a sink of outgoing audio data, for example a speaker built into an audio device or a Line Out connector.

An interrupt endpoint can be used for status returns. This endpoint is optional. All descriptors related to the internals of the audio function are part of the class-specific Audio Control- (AC) interface descriptor. The Audio Control (AC) interface of an audio function may support multiple alternate settings. The functionality of this endpoint is outside the scope of this application note.

The UAC device driver polls for details about the control on enumeration. The polling for details is carried out over EP0 requests. All such requests, including the Audio streaming requests, are handled by the `Cy_USB_PDMDeviceTaskHandler` function in the *cy_usb_uac.c* file.

3.4.1.2 Audio streaming (AS) interface

Audio Streaming interfaces are used to interchange digital audio data streams between the Host and the audio function. The Audio streaming interface descriptor and its sub-descriptors report the various Audio stream and Audio data format (e.g., Mono 8-bit PCM to MPEG2 7.1), see USB Audio Data Formats document from USB-IF to know about supported audio data format. The audio device can support multiple configurations. Within each configuration can be multiple interfaces, each interface can possibly having alternate settings.

Table 6 Class specific audio streaming interface

AS format descriptor byte offset	Characteristic	SuperSpeed Plus value	Hi-Speed value
3	Terminal link ID	2	2
4	Interface delay in ms	0x01	0x01
5-6	Format - PCM	0x0001	0x0001

Table 7 Class specific audio streaming format descriptor

AS format descriptor byte offset	Characteristic	SuperSpeed Plus value	Hi-Speed value
3	Audio format type - PCM	1	1
4	Number of channels	0x02 (Stereo)	0x02 (Stereo)
5	Frame size in bytes	2	2
6	Bit Resolution	0x10	0x10
8-10	Sampling frequency	48000 Hz	48000 Hz

3.5 UAC-specific requests

The UAC specification uses USB control endpoint EP0 to communicate control and streaming requests to the UAC device. These requests are used to discover and change the attributes of the audio-related controls. The UAC specification defines these audio-related controls as capabilities. These capabilities allow you to change audio properties, such as Volume, Mute, and equalizer. Capabilities are reported via the UVC-specific section of the USB Configuration descriptor. Each of the capabilities has attributes. The attributes of a capability are as follows:

- The minimum value
- The maximum value
- The number of values between the minimum and the maximum
- The default values
- The current value

SET and GET are the two types of UAC-specific requests. SET is used to change the current value of an attribute, while GET is used to read an attribute.

Note: *In this application note, used PDM microphones as input to (IT) and (OT) is USB host. Feature Unit (FU) is not needed for this example.*

Note: *The Product ID in the Standard Device Descriptor must be changed to support stereo and mono channel.*

4 UVC + UAC composite device application

A UVC + UAC composite device is an application that enables a device to function as both video and audio device simultaneously, using a single USB connection.

This type of application is often used in devices such as webcams, conference cameras, and other video conferencing equipment. By combining both video and audio capabilities into a single device, it simplifies the setup process and reduces the need for multiple cables and connections.

Figure 9 illustrates how to form a UVC + UAC composite device by combining UVC descriptor and UAC descriptor using the EZ-USB™ FX20.

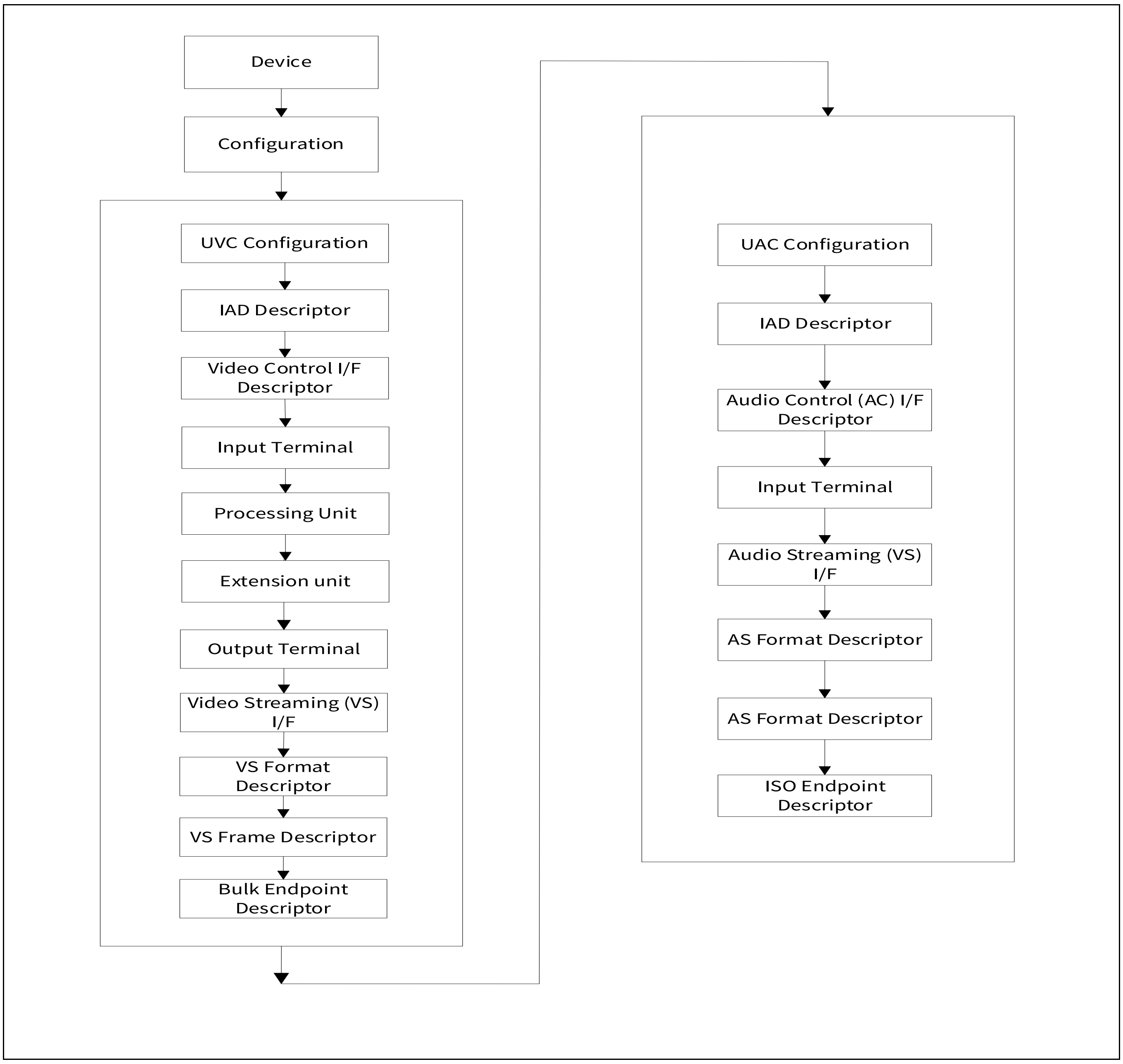


Figure 9 UVC + UAC composite device descriptor

5 LVDS/LVCMOS interface

The high bandwidth data subsystem provides DMA data transfers between LVDS/LVCMOS I/O subsystem and the USB 3.2 Gen 2x2 device at speeds up to 20 Gbps. The 1024 KB SRAM is included in this subsystem to provide sufficient buffering for data. This subsystem is also interfaced with the peripheral and system interconnect for data transfers to other low-bandwidth peripherals.

High bandwidth data subsystem I/Os can be configured as LVDS or LVCMOS interfaces.

The LVDS/LVCMOS interface is used to connect to external data sources. This interface enables data transfer over EZ-USB™ FX20's LVDS or LVCMOS I/Os to other peripherals using DMA.

The LVDS/LVCMOS interface is internally split into a physical layer (LVDS/LVCMOS ports), a link layer (GPIF III links), and a high-bandwidth DMA layer.

The data flow path between the ports and the GPIF III interface is referred to as a link, which is a connection that enables data transfer between these components.

The EZ-USB™ FX20 contains two links, both the ports can be configured as independent data paths (or links). Alternatively, two ports can be merged to form a single link, routing data from both ports to the GPIF III interface as a single link (wide link). Figure 11 to Figure 16 shows the interconnect diagram between FPGA and EZ-USB™ FX20 in various configuration mode. In this application we are using WideLink mode as a default configuration, however user can change the configuration by setting / clearing appropriate pre-processor macros in the firmware. See the EZ-USB™ FX20 datasheet section 2.2 and EZ-USB™ FX20 architecture reference manual section 25 to learn more about the features supported, control byte encoding and the configuration supported by the LVCMOS / LVDS interface.

The maximum bandwidth supported by EZ-USB™ FX20 in LVCMOS DDR WideLink receive mode (Input data) is 10 Gbps, 5 Gbps of output data (Transmit mode) throughput and the maximum bandwidth supported in LVDS WideLink receive mode is 20Gbps.

5.1 Slave FIFO access sequence and interface timing (LVDS/LVCMOS)

The Slave FIFO interfacing timings depend on the different configurations supported by EZ-USB™ FX20, i.e., LVCMOS (SDR/DDR). See Section 6 “Timing and electrical diagrams” in the EZ-USB™ FX20 datasheet for more details about the timing diagrams for LVCMOS configurations.

Figure 10 shows the write cycle timing diagram in LVDS receiver (RX) mode. Note that only DMA_READY (FLAGA) is used to indicate whether the DMA buffer is available or not. Unlike in the EZ-USB™ FX3, PARTIAL_FLAG(FLAGB) is not used in this application. Master FPGA starts writing the data only when DMA_READY(FLAGA) is asserted (HIGH).

5.1.1 LVDS RX write sequence

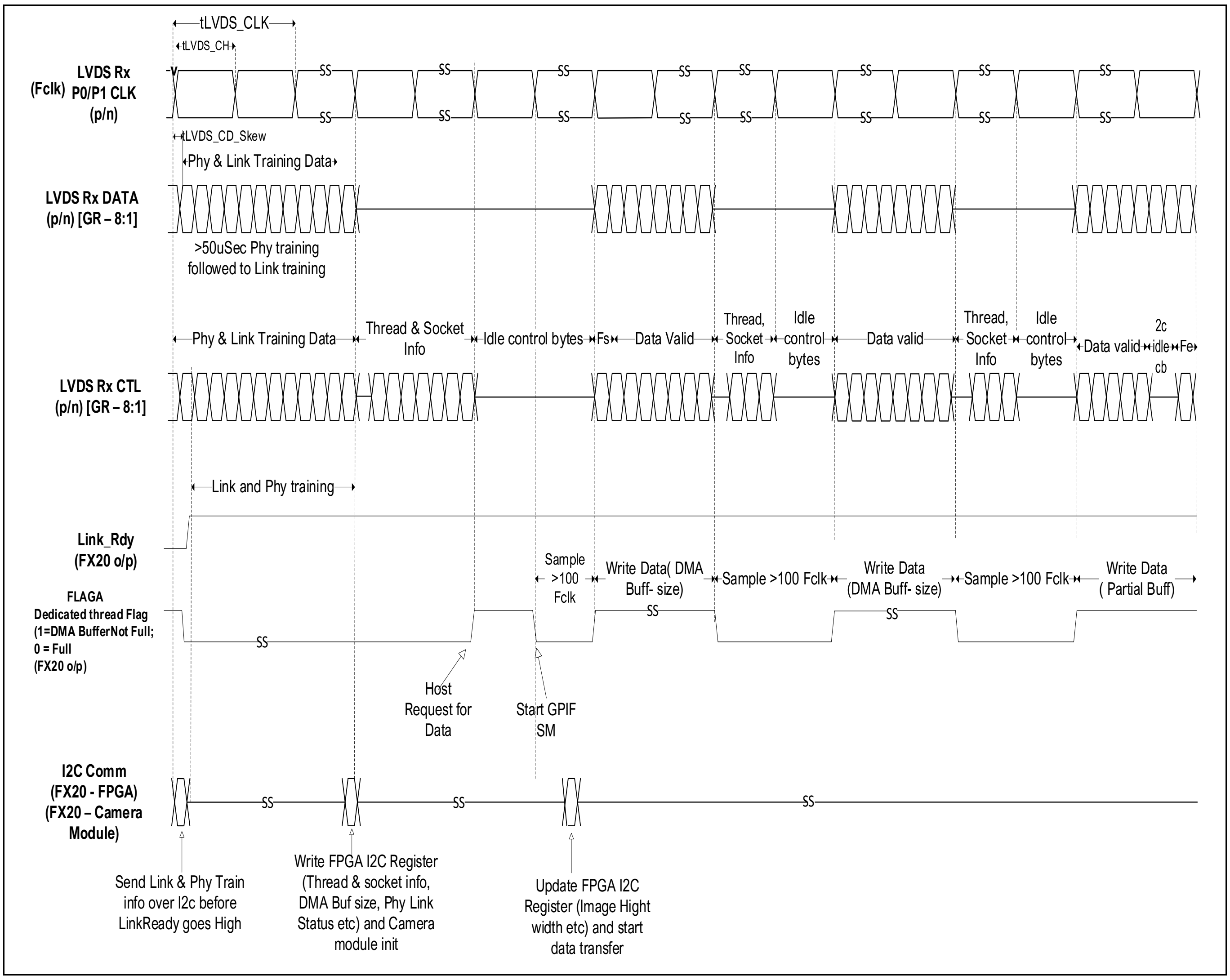

tLVDS_CLK
tLVDS_CH
LVDS Rx (Fclk) P0/P1 CLK (p/n)
tLVDS_CD_Skew
Phy & Link Training Data
LVDS Rx DATA (p/n) [GR – 8:1]
>50uSec Phy training followed to Link training
Phy & Link Training Data
Thread & Socket Info
Idle control bytes
Fs
Data Valid
Thread, Socket Info
Idle control bytes
Data valid
Thread, Socket Info
Idle control bytes
Data valid
2c idle cb
Fe
LVDS Rx CTL (p/n) [GR – 8:1]
Link and Phy training
Link_Rdy (FX20 o/p)
FLAGA Dedicated thread Flag (1=DMA BufferNot Full; 0 = Full (FX20 o/p)
Sample >100 Fclk
Write Data(DMA Buff- size)
Sample >100 Fclk
Write Data (DMA Buff- size)
Sample >100 Fclk
Write Data (Partial Buff)
Host Request for Data
Start GPIF SM
I2C Comm (FX20 - FPGA) (FX20 – Camera Module)
Send Link & Phy Train info over I2c before LinkReady goes High
Write FPGA I2C Register (Thread & socket info, DMA Buf size, Phy Link Status etc) and Camera module init
Update FPGA I2C Register (Image Hight width etc) and start data transfer

Figure 10 LVDS RX write cycle timing diagram

The FPGA LVDS TX should be configured before establishing LVDS communication.

EZ-USB™ FX20 should assert the LINK_READY signal to start the training. Once the LINK_READY signal is asserted (HIGH), the FPGA will send the PHY training bytes for 50 µs, followed by the LINK training pattern on all data lanes and control lanes for four clock cycles.

The FPGA will receive the PHY and LINK training status, thread and socket information, DMA buffer size, and other video streaming information via I2C communication.

The FPGA uses the received information to configure the EZ-USB™ FX20 thread and socket by sending commands over the LVDS control lane. For more information on LVDS control byte encoding, see Section 25.4 of the EZ-USB™ FX20 architecture manual.

When the host application requests data, the EZ-USB™ FX20 starts the GPIF state machine and enables the DMA channel. This step asserts the DMA_RDY (FLAGA) signal, indicating that the DMA buffer is available to the FPGA.

The EZ-USB™ FX20 also provides video streaming parameters, including ‘image height’, ‘width’, ‘frame rate’, and ‘Start/Stop stream’ parameters, over I2C.

After receiving the 'Start/Stop stream' signal (HIGH), the FPGA waits (sampling more than 100 FCLK cycle) for the DMA_RDY (FLAGA) signal. Once the DMA_RDY (FLAGA) signal is available, the FPGA starts transmitting the DATA_CMD command on the control lane and video data bytes on the data lane, which is equal to the DMA buffer size, and waits for the DMA_RDY (FLAGA) signal to de-assert.

Once the DMA_RDY signal is de-asserted, the FPGA will send the thread and socket number over the control lane and waits (sampling more than 100 FCLK cycle) for the DMA_RDY (FLAGA) signal. The received thread and socket number will be used by the FX20 to allocate the thread and sockets.

The above steps for transmitting data are continued until the last buffer. Finally, to terminate the ongoing DMA transfer, the FPGA sends the EOP_CMD command on the control lane, which allows the EZ-USB™ FX20 to wrap up the DMA buffer and commits the partial data to the host.

5.2 GPIF III interface in LVCMOS/LVDS mode

Figure 11 to Figure 16 show the interconnect diagram between the FPGA and EZ-USB™ FX20 using LVCMOS and LVDS interface with narrow and wide link modes. In this application, the EZ-USB™ FX20 is configured to operate in receive (RX) mode, allowing it to receive colorbar data generated by the FPGA or video stream from a camera module (either THine camera module with built-in ISP or a Sony IMX715 image sensor board). For details on how the firmware is configured to stream video from different sources, see Section 7.2. Note that the FPGA is always in transmit (TX) mode. To understand how FPGA FSM is designed to send video data over the LVCMOS and LVDS interface, see the FPGA Tx implementation Section 6 and section 7.

The number of GPIF threads supported by EZ-USB™ FX20 in WideLink mode is four, and number of GPIF threads supported in narrow link mode is two.

Based on the FPGA TX and EZ-USB™ FX20 RX interface configurations, below configurations have been tested for UVC and UAC composite device implementation:

- LVCMOS
  - LVCMOS WideLink: Port0 and Port1 combined
  - LVCMOS Narrow Link: Port0
  - LVCMOS Narrow Link: Port1
- LVDS
  - LVDS WideLink: Port0 and Port1 combined
  - LVDS Narrow Link: Port0
  - LVDS Narrow Link: Port1

5.2.1 LVCMOS WideLink: Port0 and Port1 combined

Figure 11 shows the 2-bit SlaveFIFO configuration in LVCMOS WideLink mode, which combines Port0 and Port1 (GPIF Link0 and GPIF Link1). Up to four GPIF threads (Thread0, Thread1, Thread2, Thread3) can be used in this mode.

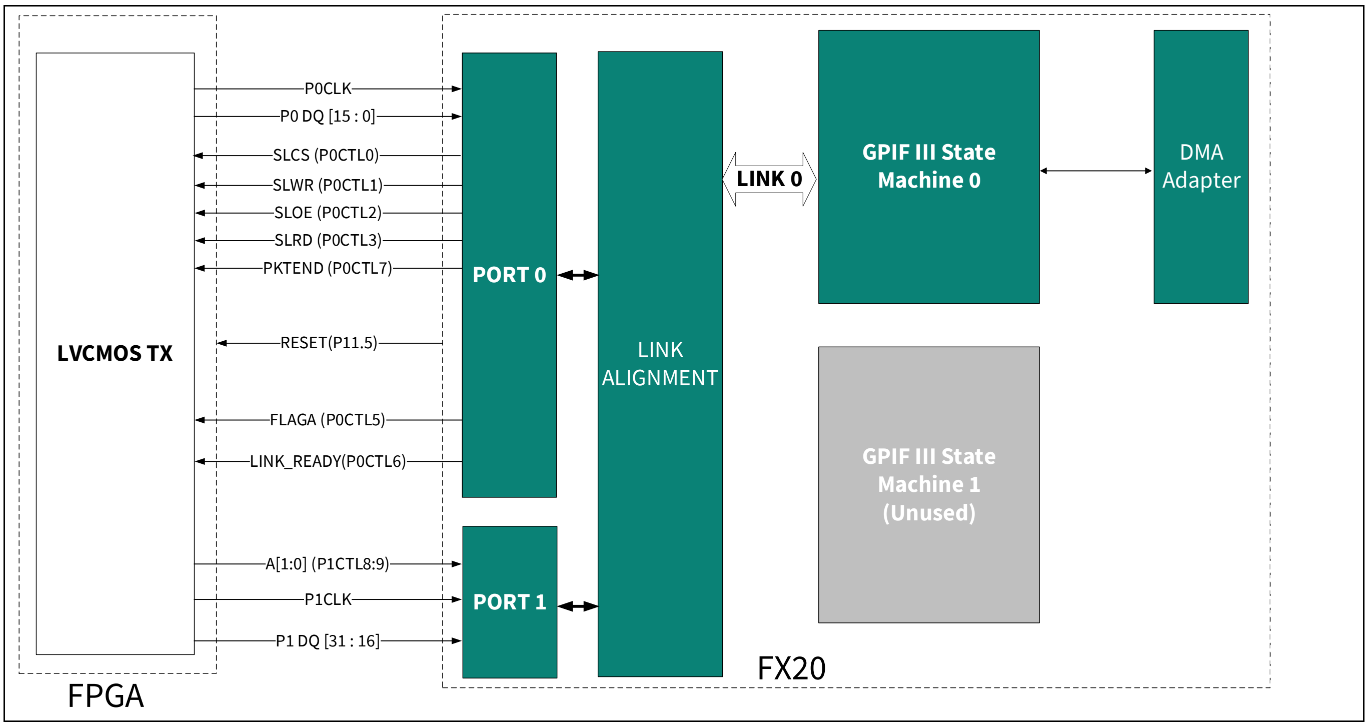


Figure 11 Interconnect diagram between FPGA and EZ-USB™ FX20 in LVCMOS DDR WideLink mode

5.2.2 LVCMOS Narrow Link: Port0

Figure 12 shows the 2-bit SlaveFIFO configuration in LVCMOS Port0 Narrow Link, which uses only Port0 (GPIF Link0). Up to two GPIF threads (Thread0, Thread1) can be used in this mode.

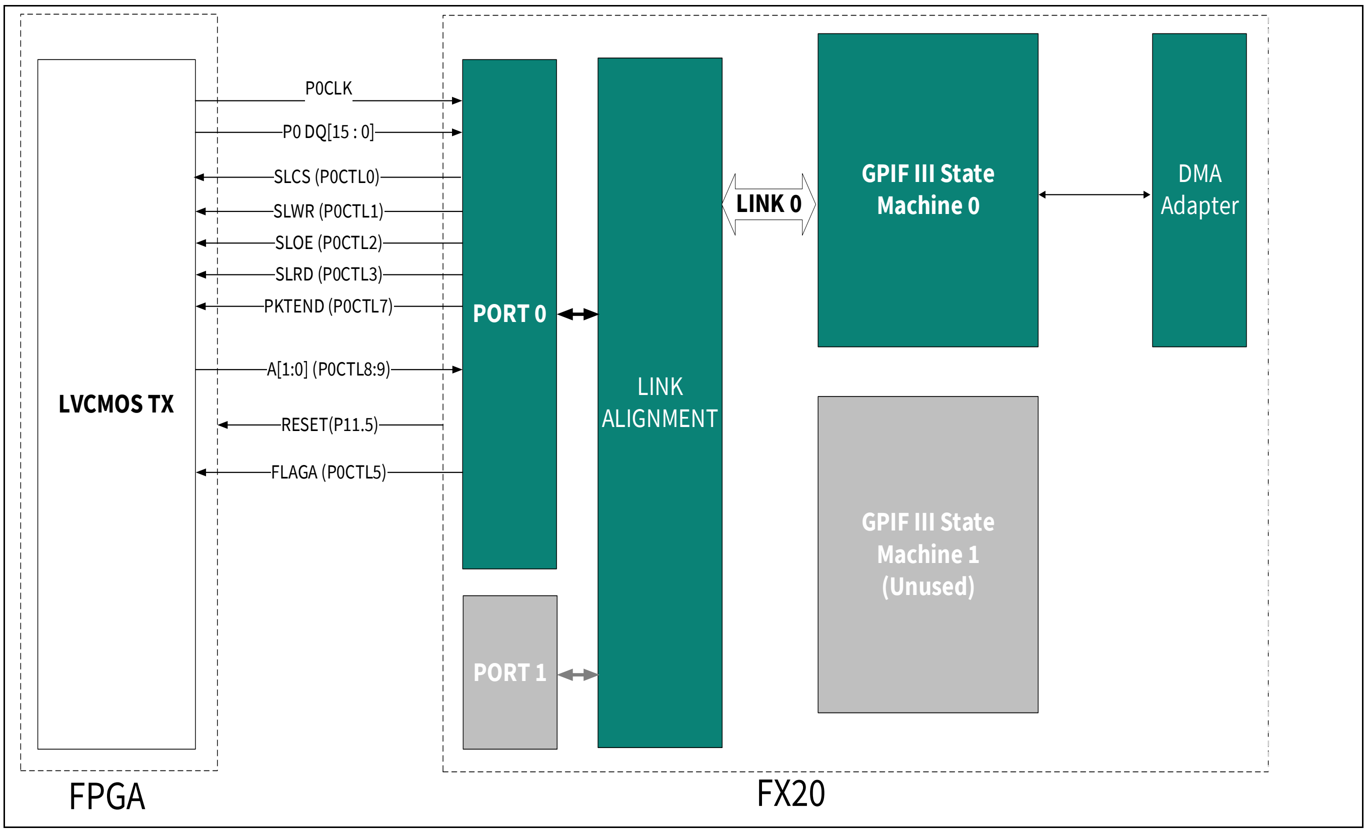


Figure 12 **Interconnect diagram between FPGA and EZ-USB™ FX20 in LVCMOS SDR/DDR NarrowLink0 mode**

5.2.3 LVCMOS Narrow Link: Port1

Figure 13 shows the 2-bit SlaveFIFO configuration in LVCMOS Port1 Narrow Link, which uses only Port1 (GPIF Link1). Up to two GPIF threads (Thread2, Thread3) can be used in this mode.

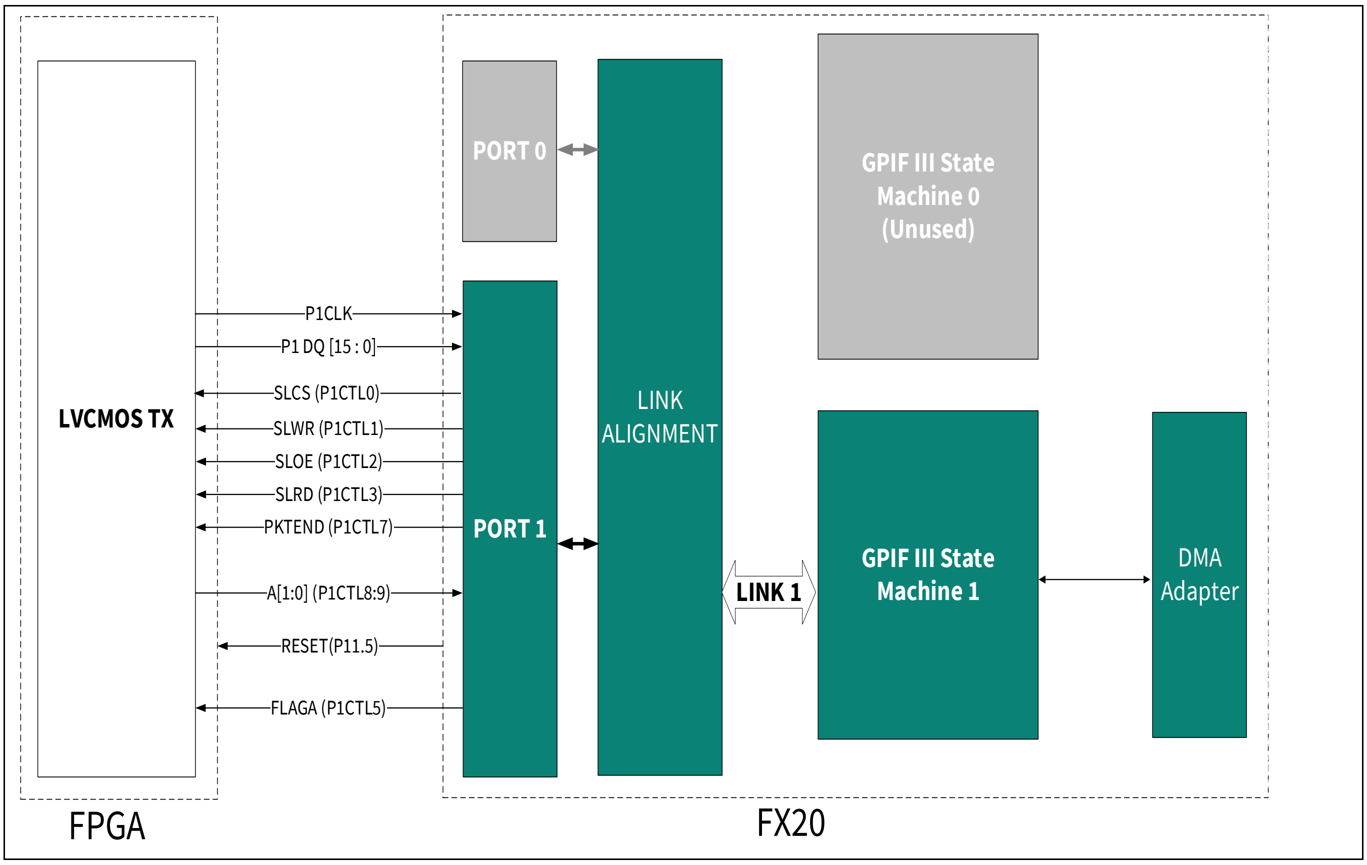


Figure 13 Interconnect diagram between FPGA and EZ-USB™ FX20 in LVCMOS SDR / DDR NarrowLink1 mode

Note: The RESET (P11.5) signal mentioned in the figure 11 to figure 16 is not part of the GPIF state machine, it is handled using the GPIO, externally to reset the FPGA logic/FSMs

It is recommended that the master FPGA master clock be routed to the P0CLK and P1CLK pins of the EZ-USB™ FX20 when configured in LVCMOS WideLink mode.

The control and address signals used in the LVCMOS DDR interface is summarized in the table 8.

Table 8 Control signal usage in LVCMOS DDR receiver state machine

EZ-USB™ FX20 pin	Function	Description
P0CTL0 (K4)	SLCS	Active-low Chip Selection signal. Should be asserted low by the master FPGA when communication with EZ-USB™ FX20
P0CTL1 (J4)	SLWR	Active-low Write Enable signal. Should be asserted low by the master FPGA when sending any data to the EZ-USB™ FX20 upon DMA_READY (FLAGA) is asserted (High). The data present on the data lanes will be sampled and stored into the DMA buffer when this signal

EZ-USB™ FX20 pin	Function	Description
		is asserted along with SLCS. Can be combined with PKTEND signal to indicate that this data ends the transfer and the DMA operation should be terminated
P0CTL2 (K4)	SLOE	Active-low Output Enable signal. Not used in this application as data is only being received by EZ-USB™ FX20
P0CTL3 (G4)	SLRD	Active-low Read Enable signal. Not used in this application as data is only being received by EZ-USB™ FX20
P0CTL5 (E4)	DMA_READY (FLAGA)	Active-High DMA Ready indication for current Thread.
P0CTL7 (F2)	PKTEND	Active-low Packet End signal. Should be asserted low when the FPGA master wants to terminate the ongoing DMA transfer. Should be asserted along with SLWR and the last cycle of data to complete transfers with non-empty data. Can be asserted with only SLCS# low and SLWR# high to complete the DMA transfer with zero bytes of data
P1CTL9 (B13)	A0	LS bit of 2-bit address bus used to select thread
P1CTL8 (A13)	A1	MS bit of 2-bit address bus used to select thread

5.2.4 LVDS WideLink: Port0 and Port1 combined

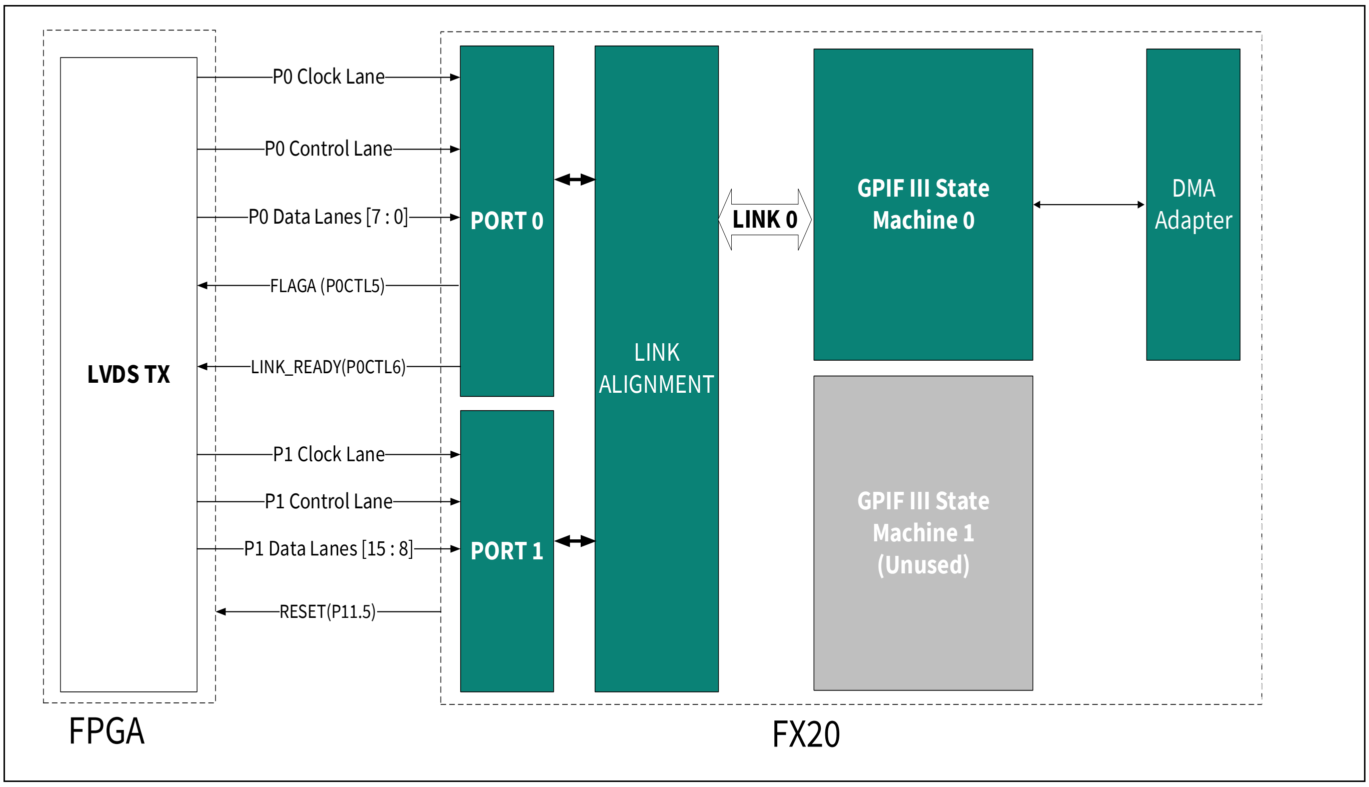


Figure 14 Interconnect diagram between FPGA and EZ-USB™ FX20 in LVDS WideLink mode

It is important to note that, 'P0 Clock Lane' and 'P1 Clock Lane' have to be connected when FX20 is configured to LVDS WideLink mode and same clock as to route to both the LVDS clock lanes.

5.2.5 LVDS Narrow Link: Port0

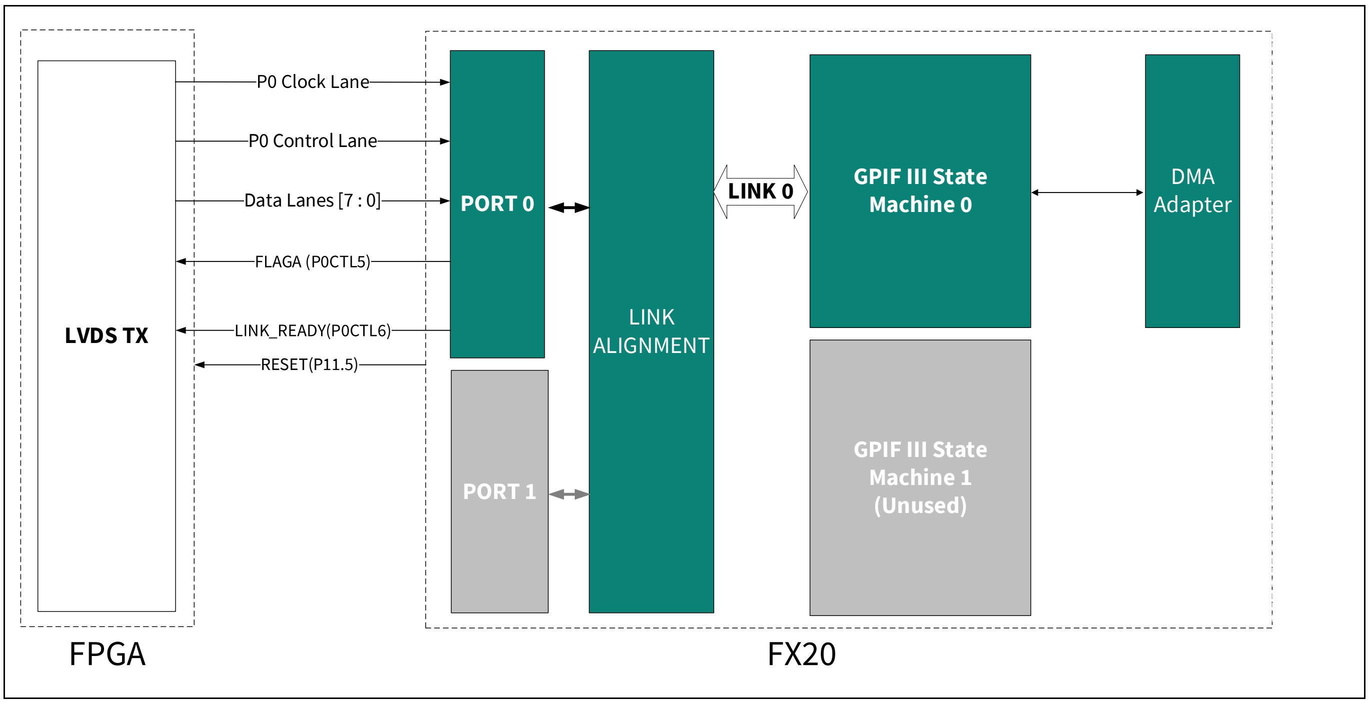


Figure 15 **Interconnect diagram between FPGA and EZ-USB™ FX20 in LVDS NarrwLink0 mode**

5.2.6 LVDS Narrow Link: Port1

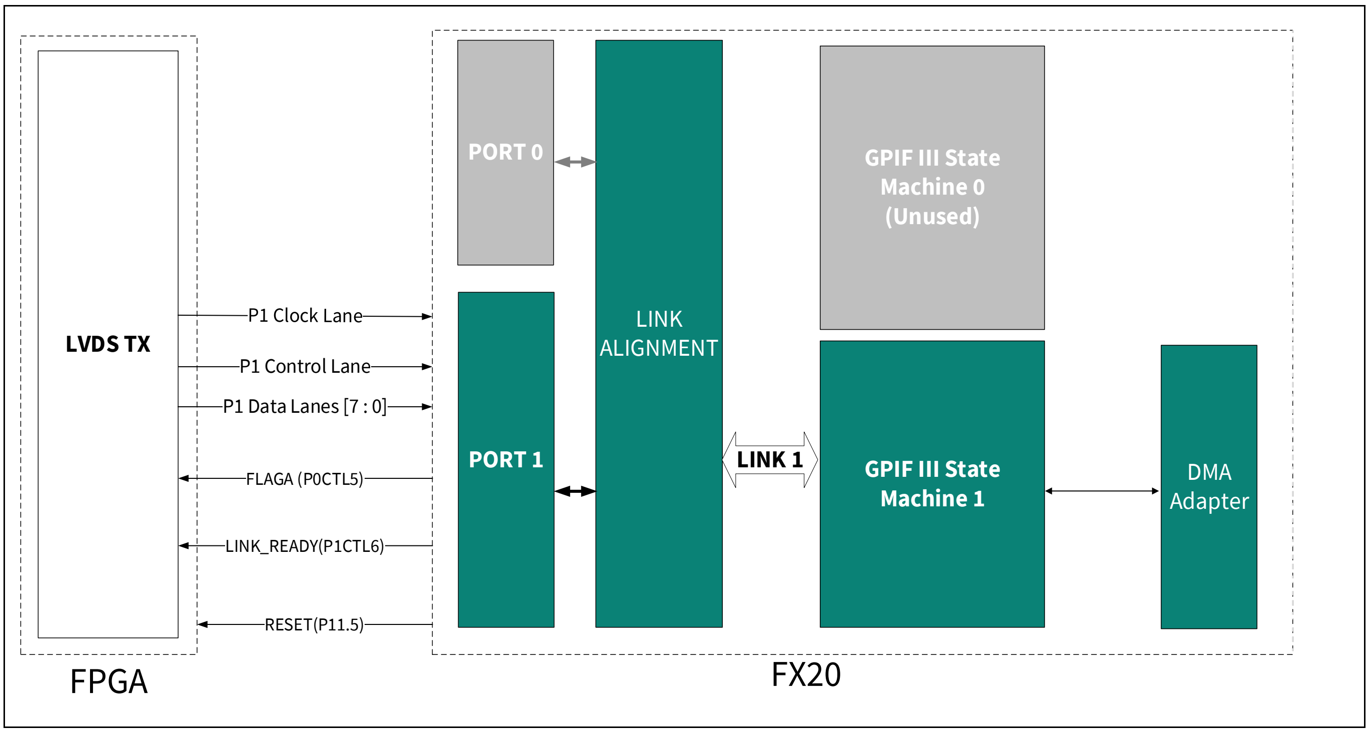


Figure 16 **Interconnect diagram between FPGA and EZ-USB™ FX20 in LVDS NarrwLink1 mode**

In this application, the P0CTL6 control signal serves as the LINK_READY signal, it initiates the transmission of the PHY and LINK training patterns from FPGA in LVDS mode, and only the LINK training pattern for LVCMOS mode.

Alternatively, the LINK_READY signal can be communicated to the FPGA via I2C, SPI, QSPI, UART, GPIO, or GPIF control signals it is not part of the GPIF III state machine.

5.3 Ping-pong DMA buffers

The application firmware uses DMA descriptors, and the DMA buffer connections manage data flow. Single / two GPIF threads are employed to fill DMA buffers. These GPIF threads use separate GPIF sockets (acting as producer sockets) and DMA descriptor chains (descriptor chain 1 and descriptor chain 2). The USB socket (acting as a consumer socket) uses a different DMA descriptor chain (descriptor chain 3) to read the data out in the correct order. This example uses singles GPIF thread by default and can be changed to two GPIF threads by setting the *PORT0_THREAD_INTLV* flag to '1' in *Makefile.*

The GPIF III state machine can use either internal control signals or external inputs to select the active GPIF thread. In this example, the switching between GPIF threads is controlled by the external control signals (A0, A1) driven by the FPGA.

When the FPGA choose the address line and start writing data via the LVDS / LVCMOS interface, the EZ-USB™ FX20 switches the buffer to receive the data. Switching the active GPIF thread changes the active socket for the data transfer, thereby changing the DMA buffer used for data transfers. Note that the GPIF thread switch has no latency. The internal architecture of EZ-USB™ FX20 for thread, socket, and buffer switching is similar to that of EZ-USB™ FX3 architecture. For more information, see AN75779 section 3.3 and 3.4.

5.3.1 LVCMOS 2-bit SlaveFIFO GPIF III state machine

The GPIF III is a programmable state machine that provides flexibility for implementing either an industry standard or a proprietary interface. A programmable state machine engine controls the LVDS / LVCMOS operations and control signals. The state machine operations can be controlled by the external processor using control signals that are configured as input to the EZ-USB™ FX20.

Figure 17 shows the state machine used by the LVCMOS receiver implementation. This state machine is based on the 2-bit Slave FIFO interface and supports both read and write operations from the FPGA to the EZ-USB™ FX20 device. However, only the write operation from FPGA to EZ-USB™ FX20 is used in this application.

The state machine and configuration used in the LVCMOS interface can be found in the cy_gpif_header_lvcmos.h file, which is included in the firmware provided with this application.

Note: *The GPIF III designer tool is currently not available to customer, but it will be released in the future. A Pre-configured 2-bit SlaveFIFO and LVDS receiver interface GPIF state machine is used in this application.*

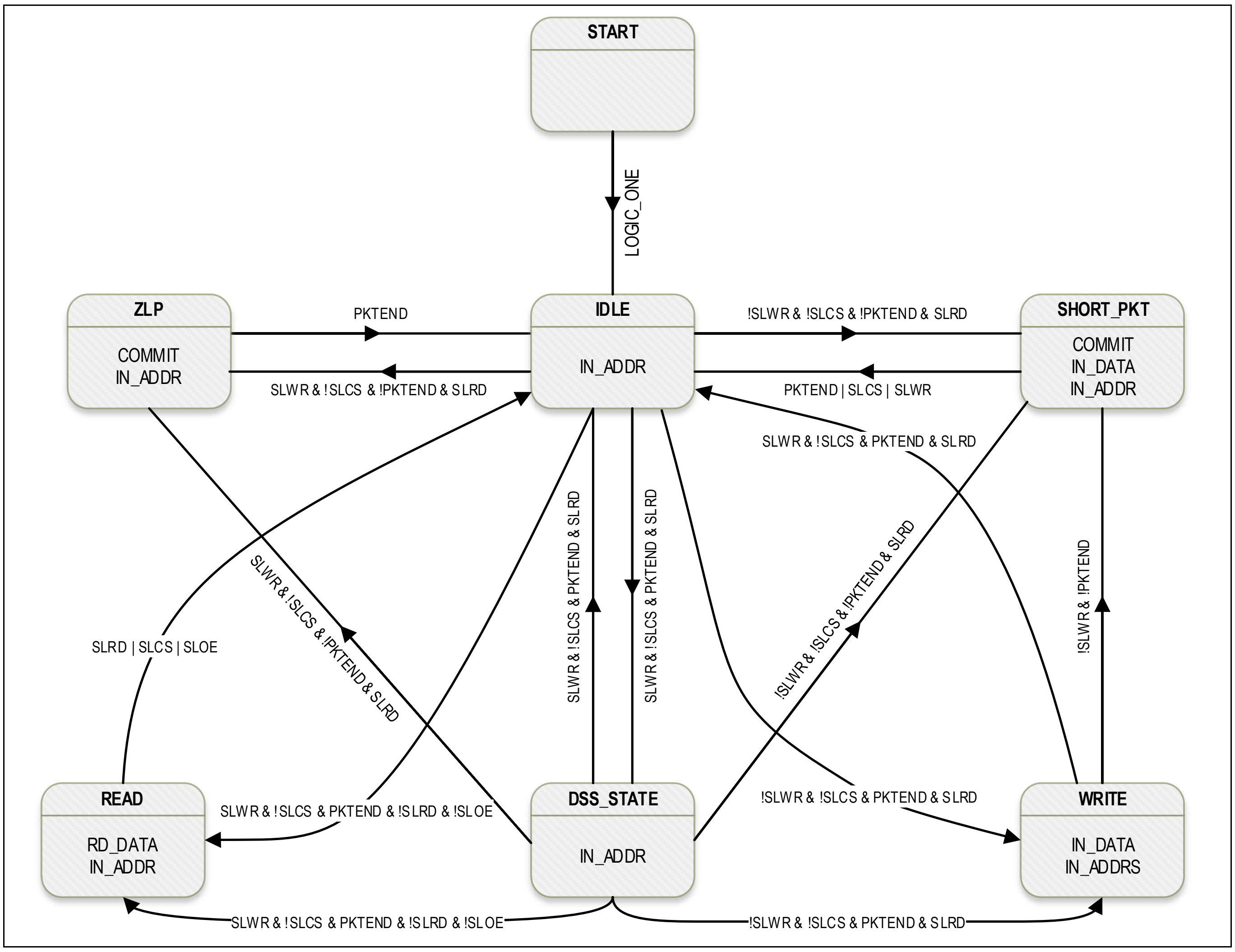

START
LOGIC_ONE
ZLP
COMMIT
IN_ADDR
PKTEND
SLWR & !SLCS & !PKTEND & SLRD
IDLE
IN_ADDR
!SLWR & !SLCS & !PKTEND & SLRD
PKTEND | SLCS | SLWR
SHORT_PKT
COMMIT
IN_DATA
IN_ADDR
SLWR & !SLCS & PKTEND & SLRD
SLRD | SLCS | SLOE
SLWR & !SLCS & !PKTEND & SLRD
SLWR & !SLCS & PKTEND & SLRD
SLWR & !SLCS & PKTEND & SLRD
!SLWR & !SLCS & !PKTEND & SLRD
!SLWR & !PKTEND
READ
RD_DATA
IN_ADDR
SLWR & !SLCS & PKTEND & !SLRD & !SLOE
DSS_STATE
IN_ADDR
!SLWR & !SLCS & PKTEND & SLRD
WRITE
IN_DATA
IN_ADDRS
SLWR & !SLCS & PKTEND & !SLRD & !SLOE
!SLWR & !SLCS & PKTEND & SLRD

Figure 17 GPIF III slave FIFO state machine

5.3.2 LVDS RX GPIF III state machine

When GPIF III interface is configured to LVDS mode, the control information is exchanged through the dedicated LVDS control lane, eliminating the need for LVCMOS control signals at the protocol level. As a result, a simplified GPIF state machine is used to control the link operation in LVDS mode.

For detailed information on LVDS control byte encoding, see Section 25.4 of the EZ-USB™ FX20 architecture manual.

Figure 18 shows the state machine used by the LVDS receiver implementation. In this state machine, data is sample on the LVDS data lanes and written to the DMA buffer whenever the opcode sent on the control lane matches a Data command (DATA_CMD) and the DMA buffers are ready (DMA_RDY_CT) to accept data.

Thread and socket selection in this application are performed using the STAD and SSAD commands received on the control lane. Before transmitting date to any thread, the master is expected to verify that the corresponding DMA_RDY signal is asserted (High) by the EZ-USB™ FX20 device.

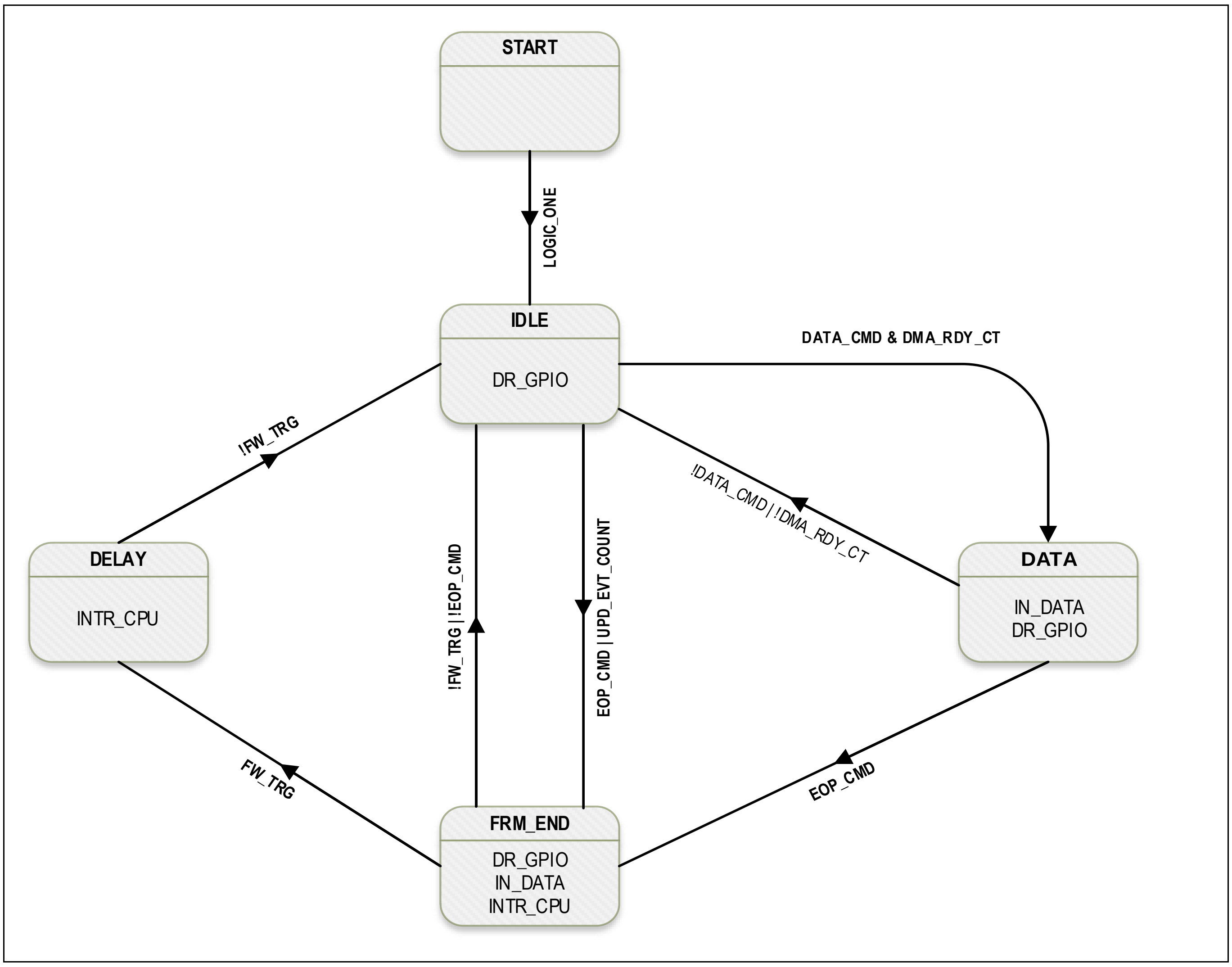


Figure 18 **LVDS Receiver state machine**

The state machine and configuration used in the LVDS interface can be found in the *cy_gpif_header_lvds.h* file, which is included in the firmware provided with this application.

6 LVCMOS transmitter implementation in FPGA

This chapter provides an overview of the LVCMOS transmitter implementation in the FPGA, enabling the FPGA to write data to the EZ-USB™ FX20 over LVCMOS interface with maximum speed of 10 Gbps. To achieve more than 10 Gbps, use LVDS interface that can support up to 20 Gbps.

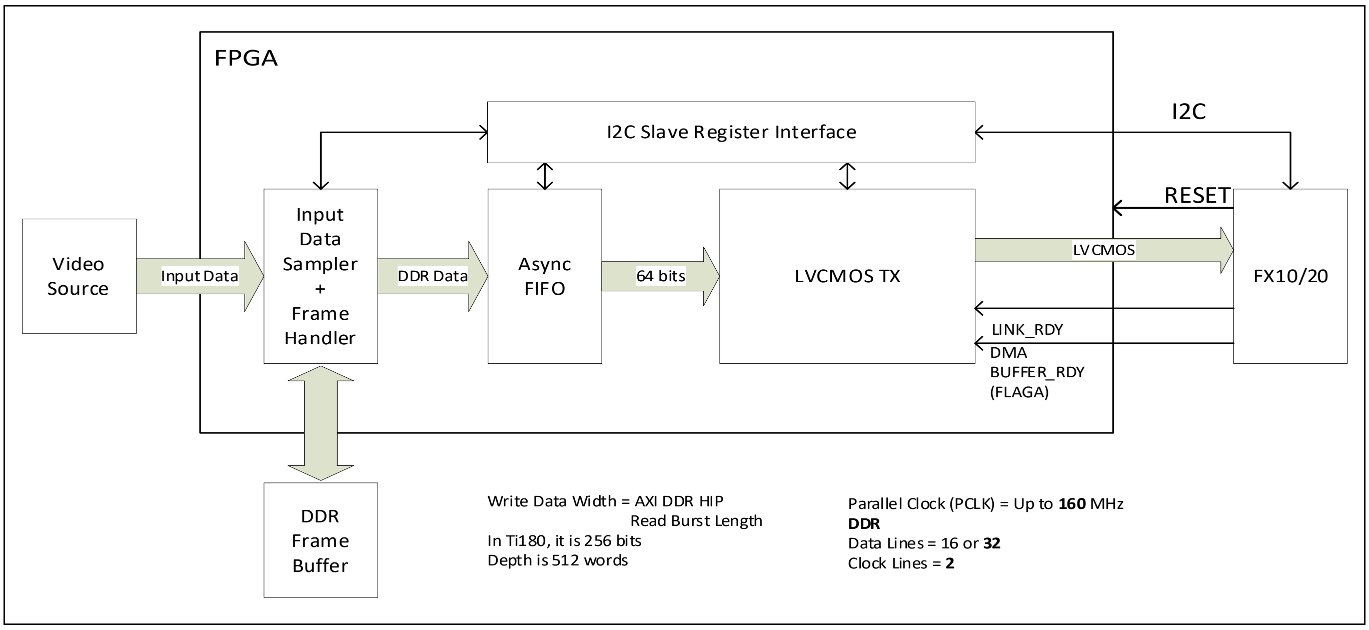


Figure 19 LVCMOS transmitter high level block diagram

As shown in Figure 19, the input data is sampled and stored in DDR frame buffer. The same data is then read from DDR frame buffer and stored in an asynchronous FIFO, from which it is transmitted to the EZ-USB™ FX20. This process is managed by the LVCMOS TX block. As we are using LVCMOS wide-link, the configuration consists of 32 data lines. The LVCMOS (TX) Transmitter block incorporates multiple Finite State Machines (FSMs) that facilitate the transmission of frame data to the EZ-USB™ FX20, which are described in more detail in subsequent sections.

6.1 Functional description of FPGA LVCMOS TX

In this section, the working of each FSMs to send a data over LVCMOS interface will be discussed in detail. The FPGA master is designed to handle up to eight streams / devices.

6.1.1 Device selection FSM

See Section 7.2.1 to know how FPGA FSM is designed to serve the activated devices.

6.1.2 LINK training FSM

See Section 7.2.1 to know more details on how FPGA handles to train the EZ-USB™ FX20 LINK.

PHY training is not needed in LVCMOS mode (Narrow/Wide), and LINK Training is also not needed if EZ-USB™ FX20 is configured to LVCMOS DDR Narrow link or LVCMOS SDR mode. Table 9 shows the PHY and LINK training requirements based on the functional operation mode configuration.

Table 9 PHY and LINK training requirement

Mode	Link configuration	PHY training required?	LINK training required?
LVCMOS SDR	Narrow Link	No	No
	Wide Link	No	No
LVCMOS DDR	Narrow Link	No	No
	Wide Link	No	Yes
LVDS	Narrow Link	Yes	Yes
	Wide Link	Yes	Yes

6.1.3 LVCMOS data packet transmitter

The Data Packet Transmitter FSM manages the transmission of USB Video Class (UVC) data packets. The Data Packet Transmitter FSM sends packet information (UVC Header) before each data packet if the FPGA is required to add UVC header. See the "Frame Header Info" register in Section 7.2 to know header insertion method supported by the FPGA.

After the last data packet, the Data Packet Transmitter FSM asserts PKTEND signal to indicate EZ USB™ FX20 to terminate the ongoing DMA transfer. Subsequently, it also switches to a new thread and socket address if it is configured with multi thread.

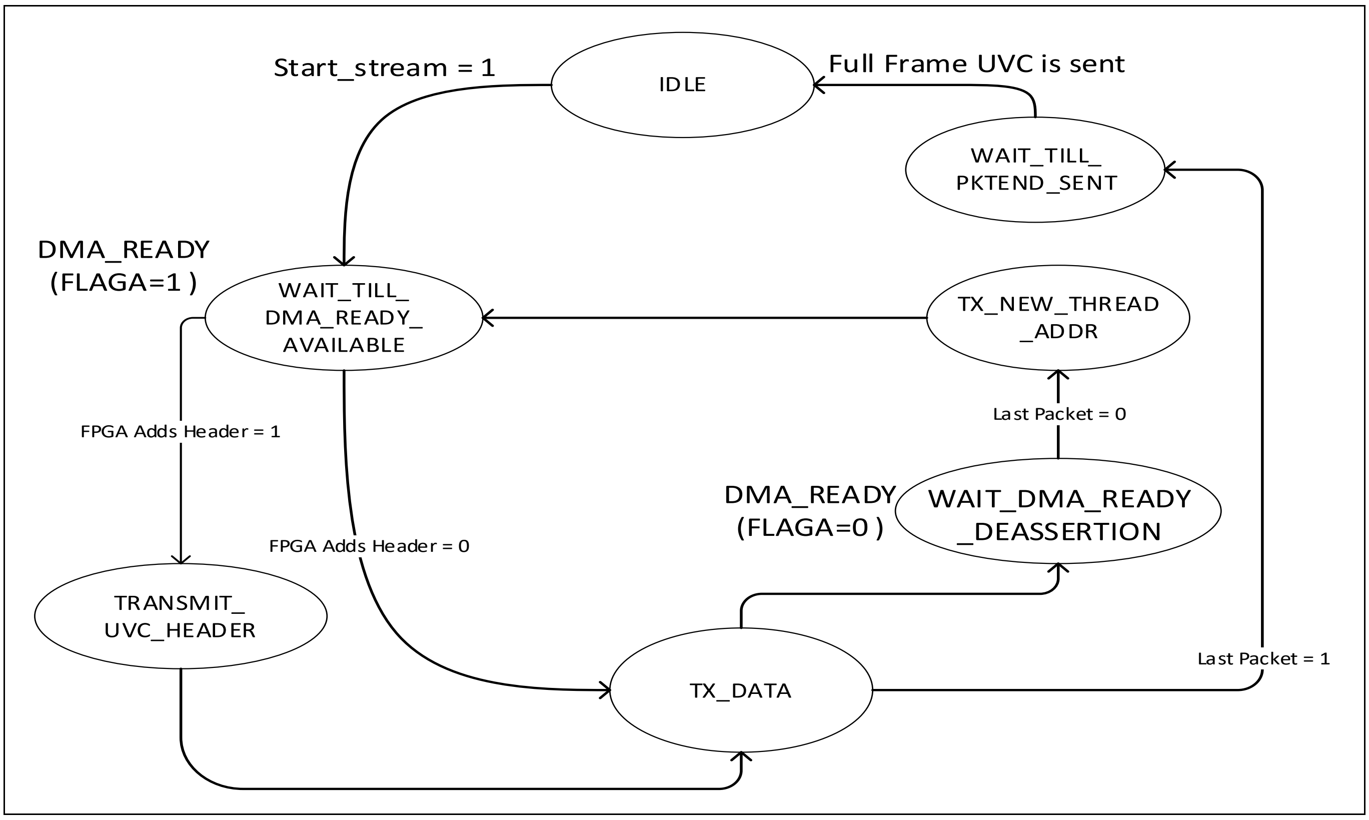


Figure 20 LVCMOS data packet transmitter

7 LVDS transmitter implementation in FPGA

7.1 FPGA LVDS transmitter high level block diagram

This chapter provides an overview of the LVDS Transmitter implementation in the FPGA, enabling the FPGA to write data to the EZ-USB™ FX20. This section focuses on how the FPGA should transmit data over the LVDS interface with maximum speed of 20 Gbps. LVCMOS mode can be used to supports up to 10 Gbps.

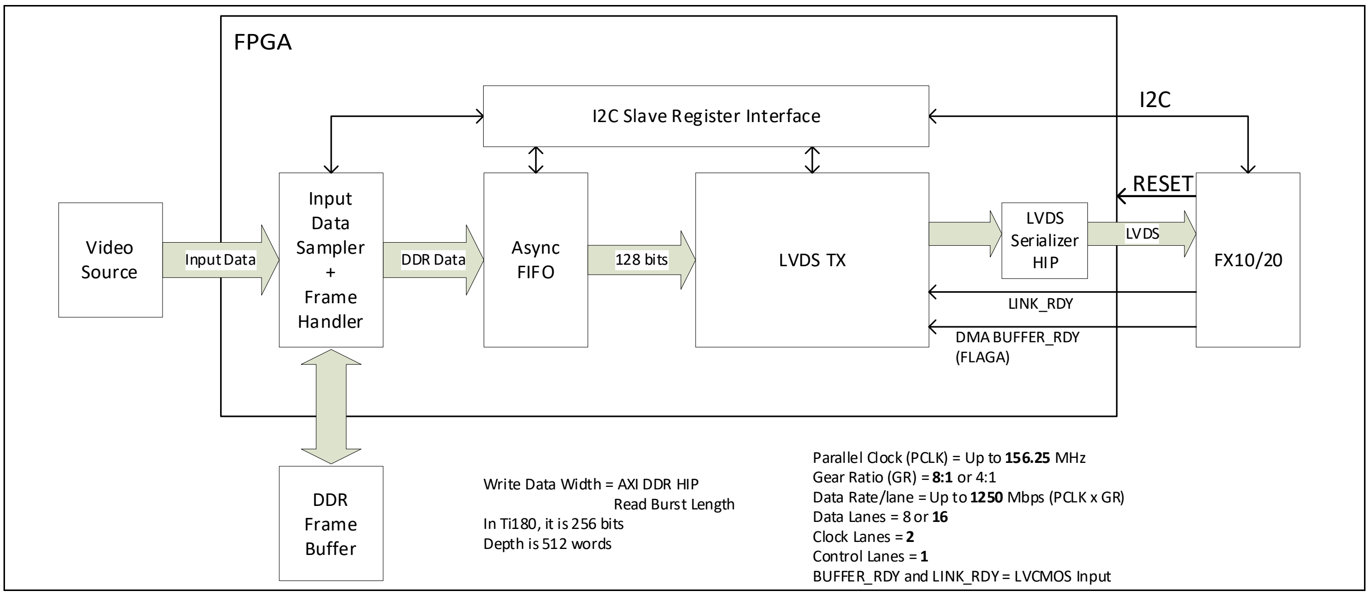

FPGA
I2C Slave Register Interface
I2C
RESET
Video Source
Input Data
Input Data Sampler + Frame Handler
DDR Data
Async FIFO
128 bits
LVDS TX
LVDS Serializer HIP
LVDS
FX10/20
LINK_RDY
DMA BUFFER_RDY (FLAGA)
DDR Frame Buffer
Write Data Width = AXI DDR HIP Read Burst Length
In Ti180, it is 256 bits
Depth is 512 words
Parallel Clock (PCLK) = Up to 156.25 MHz
Gear Ratio (GR) = 8:1 or 4:1
Data Rate/lane = Up to 1250 Mbps (PCLK x GR)
Data Lanes = 8 or 16
Clock Lanes = 2
Control Lanes = 1
BUFFER_RDY and LINK_RDY = LVCMOS Input

Figure 21 LVDS transmitter High level block diagram

As shown in Figure 21, the input data is sampled and stored in DDR frame buffer. The same data is then read from DDR frame buffer and stored in an asynchronous FIFO, from which it is transmitted to the EZ-USB™ FX20. This process is managed by the LVDS TX block. As we are using LVDS wide-link, the configuration consists of 16 data lanes. The LVDS Transmitter block incorporates multiple Finite State Machines (FSMs) that facilitate the transmission of frame data to the EZ-USB™ FX20, which are described in more detail in subsequent sections.

Note: *The master FPGA (LVCMOS / LVDS) is designed to handle up to 8 streams / devices. In this application note, focusing on streaming from one device.*

7.2 I2C slave register interface

I2C (Inter-Integrated Circuit) is a widely used communication protocol that enables data exchange between master and slave devices. The FPGA requires a range of configuration data, including PHY and LINK training patterns, image height and width, frames per second (FPS), bits per pixel, thread and socket information, and other relevant parameters. The EZ-USB™ FX20 acts as a I2C master device, while the FPGA acts as the I2C slave device. Upon configuring or soft resetting FPGA, the EZ-USB™ FX20 transmits all necessary I2C slave register parameters to the FPGA.

The example code provided with this application note uses the following pin assignments for SCL and SDA signals

SCL: P10_0 (SCB0)

SDA: P10_1 (SCB0)

7.2.1 FPGA I2C register details

FPGA I2C registers are divided mainly into two categories (Common control and status registers, Device specific registers).

7.2.1.1 Common control and status registers

Table 10 Common control and status register details

Sl.#	Register name	Register offset address	Type	Description
1	RTL Version	0x00	RO	RTL Version. It needs to be updated after every change in RTL Bit 7-4: Major Version Bit 3-0: Minor Version
2	Streaming Mode	0x01	RW	Mode of USB Streaming. It can be either "UVC " or "U3V" Bit 7-1: Not used Bit 0: 1 for UVC, 0 for U3V
3	Frame Header Info	0x02	RW	Whether frame header should be sent by FPGA or not. If 0 means, FPGA should not send frame header, it will be taken care by FX10/20. If 1, FPGA should send header 0: FPGA should not add header data. FX10/20 will add it 1: FPGA should add header data as a payload 2: FPGA should add header data as a metadata. It is valid only for LVDS mode
4	LVDS PHY Training Mode Config	0x03	RW	LVDS PHY Training Sequence Pattern Byte. It is sent during PHY training. It is same as FX10/20 Register PHY Training Mode Config Register
5	LVDS Training Block Byte P0	0x04	RW	It is mandatory to align lanes and for that, Training Block (P0 to P3) needs to be sent through all the active data lanes. It is same as FX10/20 Register LVDS Training Block Sequence Byte
6	LVDS Training Block Byte P1	0x05	RW	
7	LVDS Training Block Byte P2	0x06	RW	
8	LVDS Training Block Byte P3	0x07	RW	

Sl.#	Register name	Register offset address	Type	Description
9	Active Device Mask	0x08	RW	Each bit corresponds to the connected source device. Here, device is referred to a peripheral from which FPGA receives a data and transmits to FX10/20. It could be either various types of cameras or internal color bar generator which generates data. Maximum 8 source devices can be connected. Bit 0 is for device 1 and bit 7 is for device 8. If corresponding bit is high, it means, device is active and mapped to threads. Hence, LVDS TX FSM will serve this device. If corresponding bit is 0, it will be ignored by FSM. FSM serves devices in round robin manner.
10	Low Power Mode Support	0x09	RW	Low Power Mode support. Bit 7-2: Not used Bit 1: L3 Power Mode Support. Set means, enter into L3 mode and reset means, exit L3 mode. Bit 0: L1 Power Mode Support. Set means, enter into L1 mode and reset means, exit L1 mode.
11	PHY and Link Training Control and Status	0x0A	RW	PHY and Link training control register. If particular bit is high, then only, training will be carried out by FPGA. Bit 7: FX10/20 Port 1 training is completed. If it is low after reset, FPGA starts training Bit 6: FX10/20 Port 0 training is completed. If it is low after reset, FPGA starts training Bit 5-2: Not used Bit 1: Link Training is required Bit 0: PHY Training is required
12	External Controller Status Info	0x0B	RO	It gives the status of DDR HIP Controller. Bit 7-5: Not used Bit 4: Datapath is idle or not Bit 3: Command queue full status Bit 2: DDR Controller busy status Bit 1: DDR configuration status Bit 0: DMA Ready flag status

7.2.1.2 Device specific registers

The master FPGA is designed to support 8 streams / devices, necessitating 8 set of Device Specific I2C Registers, with one set allocated to each stream / device. The details of these device-specific registers are provided in Table 11, and the register offset addresses can be found in Table 12.

Table 11 **Device specific register details**

Sl.#	Register	Register offset address	Type	Description
1	Stream Enable	0x00	RW	This register is for enabling/disabling or restarting the streaming. FX10/20 FW must set start/stop stream bit once all relevant registers are written Bit 7-3: Not used Bit 2: For UVC, no need to use this bit Bit 1: Start/Stop stream. 1 = Start and 0 = Stop. When this bit is set, FPGA starts sending frame data to FX10/20. If it is reset, FPGA stops immediately Bit 0: Soft reset. It is DMA Reset to restart streaming. It is not being used as of now
2	Mode Enable	0x01	RW	This register is for enabling/disabling different modes Bit 7-4: Not used Bit 3: Enable YUV422-420 Conversion Bit 2: Enable Mono8 Conversion Bit 1: Enable Still Capture Bit 0: Enable Interlaced streaming
3	Image Height Info LB	0x02	RW	Image height info register Bit 7-0: Height lower byte
4	Image Height Info UB	0x03	RW	Image height info register Bit 7-0: Height upper byte
5	Image Width Info LB	0x04	RW	Image width info register Bit 7-0: Width lower byte
6	Image Width Info UB	0x05	RW	Image width info register Bit 7-0: Width upper byte
7	Image Frame Rate Info	0x06	RW	Image frame rate info register Bit 7-0: frame rate
8	Image Pixel Width Info	0x07	RW	Image pixel width in bits Bit 7-0: Pixel width in bits

Sl.#	Register	Register offset address	Type	Description
9	Source Type Info	0x08	RW	This register defines the source of data. Whether it is received from HDMI, MIPI, test pattern generator module in RTL or some other Bit 7-2: Not used Bit 1:0: 0 - Test Pattern Generated from RTL, i.e., internal test pattern or color bar 1 - Over HDMI 2 - Over MIPI 3 - Other
10	Status Info	0x09	RO	It gives the status of slave FIFO, intermediate FIFO and flag status Bit 7-6: Not used Bit 5: DDR read status (Full frame read complete) Bit 4: DDR write status (Full frame write complete) Bit 3: Not Used Bit 2: Y data FIFO full status Bit 1: Y data FIFO empty status Bit 0: Slave FIFO almost empty status
11	MIPI Status	0x0A	RO	It gives the status of MIPI error. It is valid only if source type is MIPI
12	HDMI/MIPI Source Info	0x0B	RW	This register gives information regarding channel information and source connection if HDMI source is connected. If MIPI camera is connected, it gives information regarding ISP and image cropping algorithm should be enabled or not Bit 7-6: Not used Bit 5: Enable Crop Algorithm (Valid for only MIPI source) Bit 4: Enable CIS ISP IP (Valid for only MIPI source) Bit 3: Not used Bit 2: 0 for YUV and 1 for RGB Bit 1: 0 for single channel and 1 for dual channel Bit 0: Source connection status
13	Reserved	0x0C	RW	–
14	Reserved	0x0D	RW	–
15	Reserved	0x0E	RW	–

Sl.#	Register	Register offset address	Type	Description
16	Device X Active Thread Info	0x0F	RW	Number of active threads for a device. If its value is 1 = Only single thread is assigned to device 2 = Two threads are assigned to device Maximum 2 threads can be mapped. Device X Thread 1 Info and Thread 2 Info registers inform that which thread is mapped
17	Device X Thread 1 Info	0x10	RW	Thread number which is mapped to a device
18	Device X Thread 2 Info	0x11	RW	Thread number which is mapped to a device
19	Device X Thread 1 Socket Info	0x12	RW	It defines the socket number assigned to thread defined in respective Thread Info registers
20	Device X Thread 2 Socket Info	0x13	RW	It defines the socket number assigned to thread defined in respective Thread Info registers
21	Device X Flag Info	0x14	RW	Defines which flags are active for this device. Usually, there are four flags for each thread but here it is assumed that flag configuration will be same for both the flags. Whichever bit is high, that flag is active for device Bit 7-4: Not used Bit 3: FX10/20 is ready to receive data Bit 2: Not used Bit 1: New UVC packet starts Bit 0: New frame starts
22	Device X Counter and CRC Info	0x15	RW	Defines the event counters, payload counters, DCRC and MCRC for device. There are 2 event counters, 1 payload counter, Data CRC and Metadata CRC available for each thread but here, it is assumed it is same for all threads Bit 7-5: Not used Bit 4: MCRC Bit 3: DCRC Bit 2: Payload Counter Bit 1: Not used Bit 0: Frame Count
23	Device X Buffer Size Info LB	0x16	RW	Device X buffer size. It is same for all threads mapped to this Device. Default value is 49104 bytes which is same as FX3. However, in FX10/20, it can be up to 65535 bytes
24	Device X Buffer Size Info UB	0x17	RW	

7.2.1.3 Base offset for device specific register

Table 12 Base offset for common register and device specific register details

Sl.#	Register	Register offset address	Description
1	Common Register Base Address	0x00	Offset of common control and status register
2	Device 0 Base Address	0x20	Device 0 Buffer Size Info address is 0x36 and 0x37.
3	Device 1 Base Address	0x3C	Device 1 Buffer Size Info address is 0x52 and 0x53.
4	Device 2 Base Address	0x58	Device 2 Buffer Size Info address is 0x6E and 0x6F.
5	Device 3 Base Address	0x74	Device 3 Buffer Size Info address is 0x8A and 0x8B.
6	Device 4 Base Address	0x90	Device 4 Buffer Size Info address is 0xA6 and 0xA7.
7	Device 5 Base Address	0xAC	Device 5 Buffer Size Info address is 0xC2 and 0xC3.
8	Device 6 Base Address	0xC8	Device 6 Buffer Size Info address is 0xDE and 0xDF.
9	Device 7 Base Address	0xE4	Device 7 Buffer Size Info address is 0xFA and 0xFB.

7.3 Functional description of FPGA LVDS TX

In this section, the working of each FSMs to send data over LVDS interface will be discussed in detail. The FPGA master is designed to handle up to eight streams/devices

7.3.1 Device selection FSM

Figure 22 shows the Device Selection FSM, and it is the main FSM. Upon power up or logical reset, the Device Selection FSM will be in the IDLE state. The FPGA checks the PHY and Link Training Control and Status I2C slave Register to determine the type of training required for the EZ-USB™ FX20. It also checks whether the training is completed or not.

If training is not completed, the Device Selection FSM instructs the PHY and Link Training FSM to start the training and waits until completion.

After training, the Device Selection FSM starts serving the devices in a round-robin manner. The FPGA LVDS TX IP supports a maximum of 8 devices, which can be either video source or an audio source.

If the device is a video source, it can be either an image sensor (such as the THine camera module with built-in ISP or the Sony IMX715 image sensor board) connected to the FPGA, or an internal colorbar generated by the FPGA.

The Device Selection FSM starts serving the first device. If the device has data to send, it instructs the Frame Data Transmitter FSM to transmit a frame and waits for acknowledgement. If the device does not have data to send, it moves on to the next device. Once acknowledgement is received, the Device Selection FSM moves on to the next device and follows the same steps until all active devices (up to a maximum of 8) have been served.

The Device Selection FSM also moves to low power mode upon receiving an L1 Entry or L3 Entry signal from the EZ-USB™ FX10 / 20. In response, it moves to the L1_ENTER or L3_ENTER state and instructs the LVDS IP to power down the LVDS lanes according to the mode. The Device Selection FSM moves to the L1_EXIT or L3_EXIT state when the L1 or L3 Entry signal is de-asserted by the EZ-USB™ FX10/20.

PHY and Link training is mandatory after exiting low power mode.

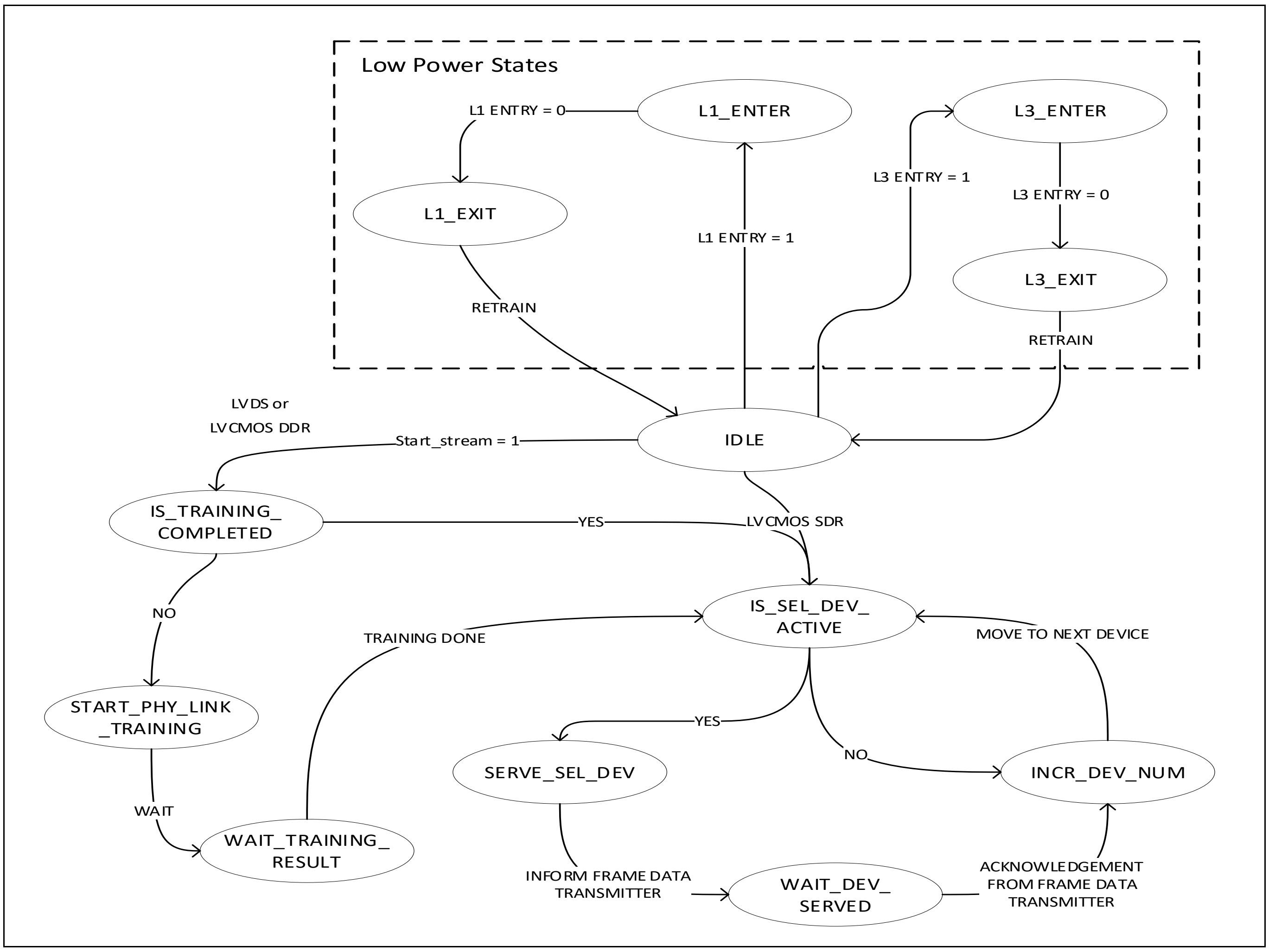


Figure 22 **Device selection FSM**

7.3.2 IDLE

If the transfer mode is not selected, the FPGA master will remain in this state.
Based on the number of active devices communicated to the FPGA via I2C, the FPGA will proceed to serve these devices.

7.3.3 PHY and LINK training FSM

This FSM sends the training pattern required to train the LVDS Rx PHY and LINK of EZ-USB™ FX20 in LVDS mode and sends the LINK training pattern for LVCMOS mode. The PHY and LINK training is required for LVDS receiver (RX) as soon as EZ-USB™ FX20 is powered up or LVDS low power modes (L1 or L3) exit. Upon power up, EZ-USB™ FX20 initializes its PHY and asserts LINK_RDY. Once LINK_RDY is asserted, the PHY and LINK training FSM transmits PHY test pattern (PHY_TS_Pattern) byte for PHY_TS_Time (50us) followed by LINK training block consist of a 4-bytes (P3, P2, P1, P0) sequence for 4 clock cycles for all the data lane and control lane. The Link training is to compensate skew on all lanes and align data bytes of all lanes.

The FPGA receives training requirement information, including the PHY_TS_Pattern (one byte) and LINK training bytes (P0 to P3), via I2C. Details about the FPGA I2C slave register can be found in Section 7.2.1.

Upon receiving the acknowledgement from EZ-USB™ FX20, it triggers to main FSM that training is completed.

Note: *In case of LVCMOS interface, there is no serial data stream, as lvcmos interface is a parallel interface. Hence LVCMOS doesn't need PHY level training.*

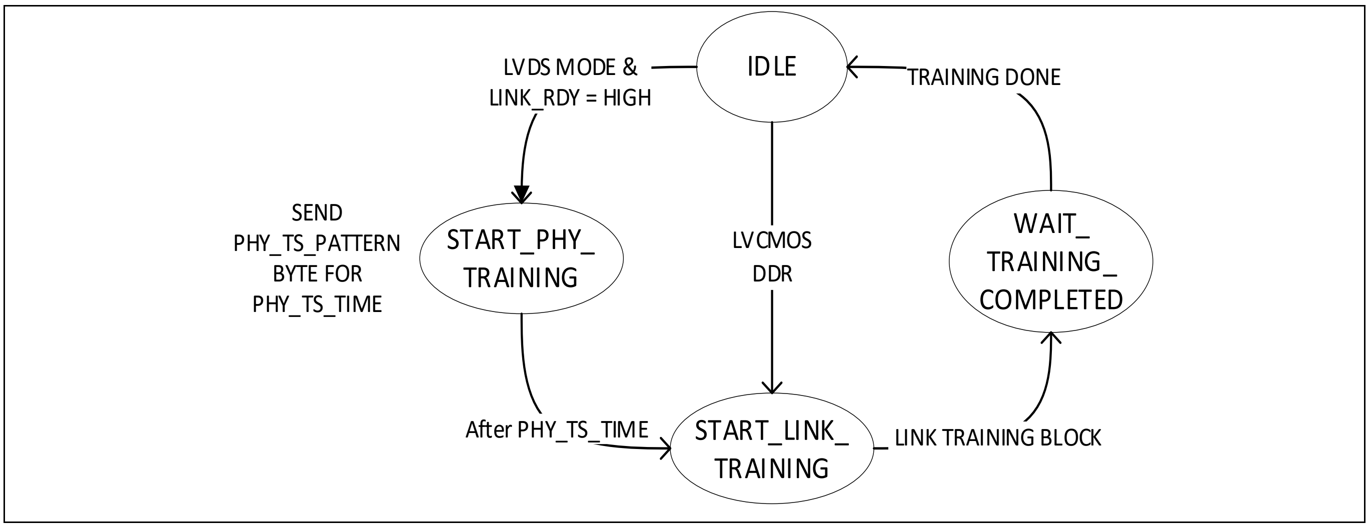


Figure 23 LVDS PHY and LINK training FSM

7.3.4 Frame data transmitter FSM

Frame Data Transmitter FSM is responsible for frame data transmission to the EZ-USB™ FX20. Once PHY and LINK training is completed, and the selected device has data ready to transmit, the Device Selection FSM instructs the Frame Data Transmitter FSM to transmit a frame.

The Frame Data Transmitter FSM utilizes sub-FSMs to manage the specific tasks, including the transmission of initial socket/thread information, frame header information, data, and frame end information. The details of these sub-FSMs are discussed from Section 7.3.4.1.

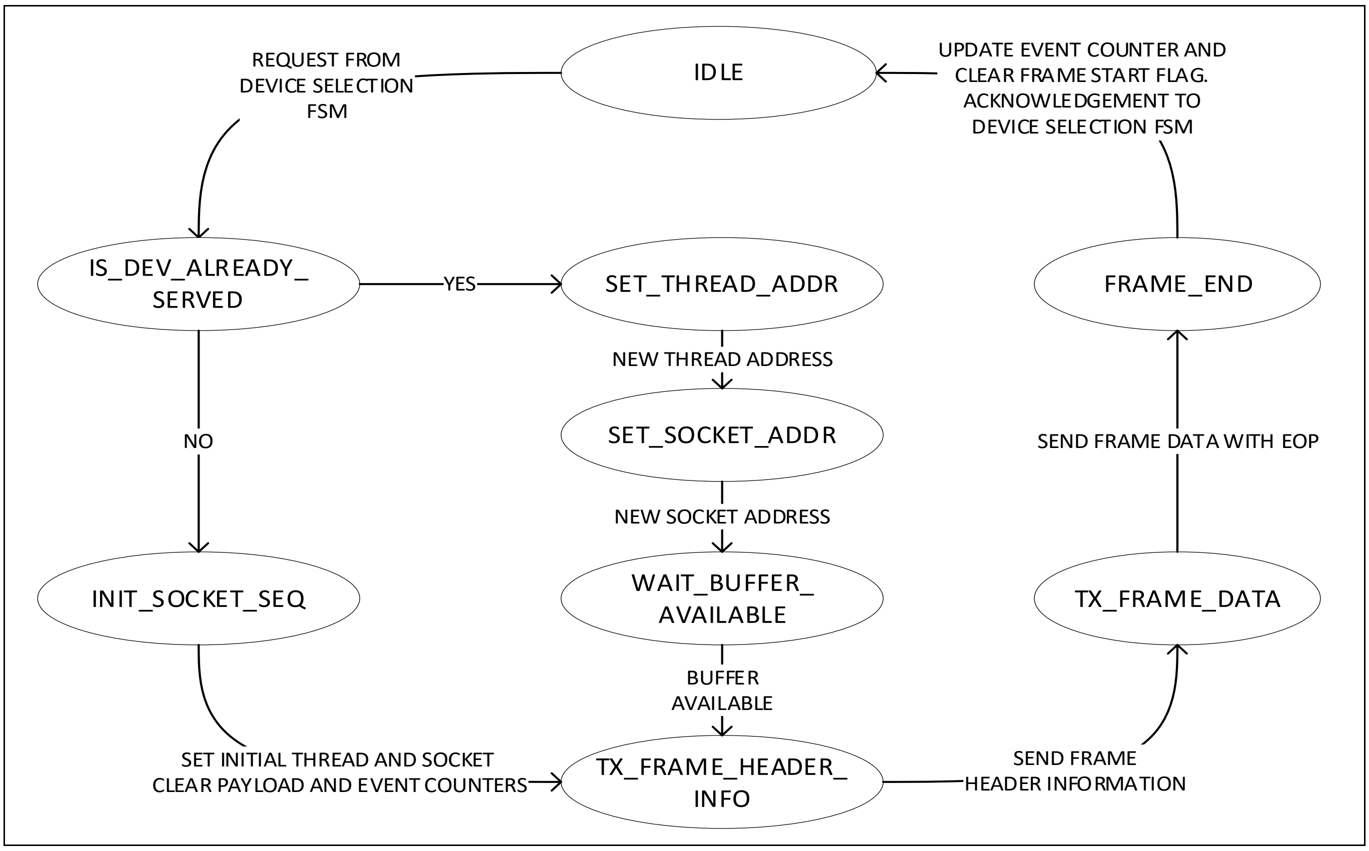


Figure 24 **Frame data transmitter FSM**

7.3.4.1 Initial socket sequence FSM

The Initial Socket Sequence FSM handles the initialization sequence. It assigns the initial thread and socket address. Additionally, it initializes the payload and event counters if they are activated through FPGA I2C slave registers. Besides these tasks, it also sets the lower byte of the metadata variable word with the length of the frame header.

At the end, it acknowledges to the Frame Data Transmitter FSM indicating that the initial thread and socket information has been sent.

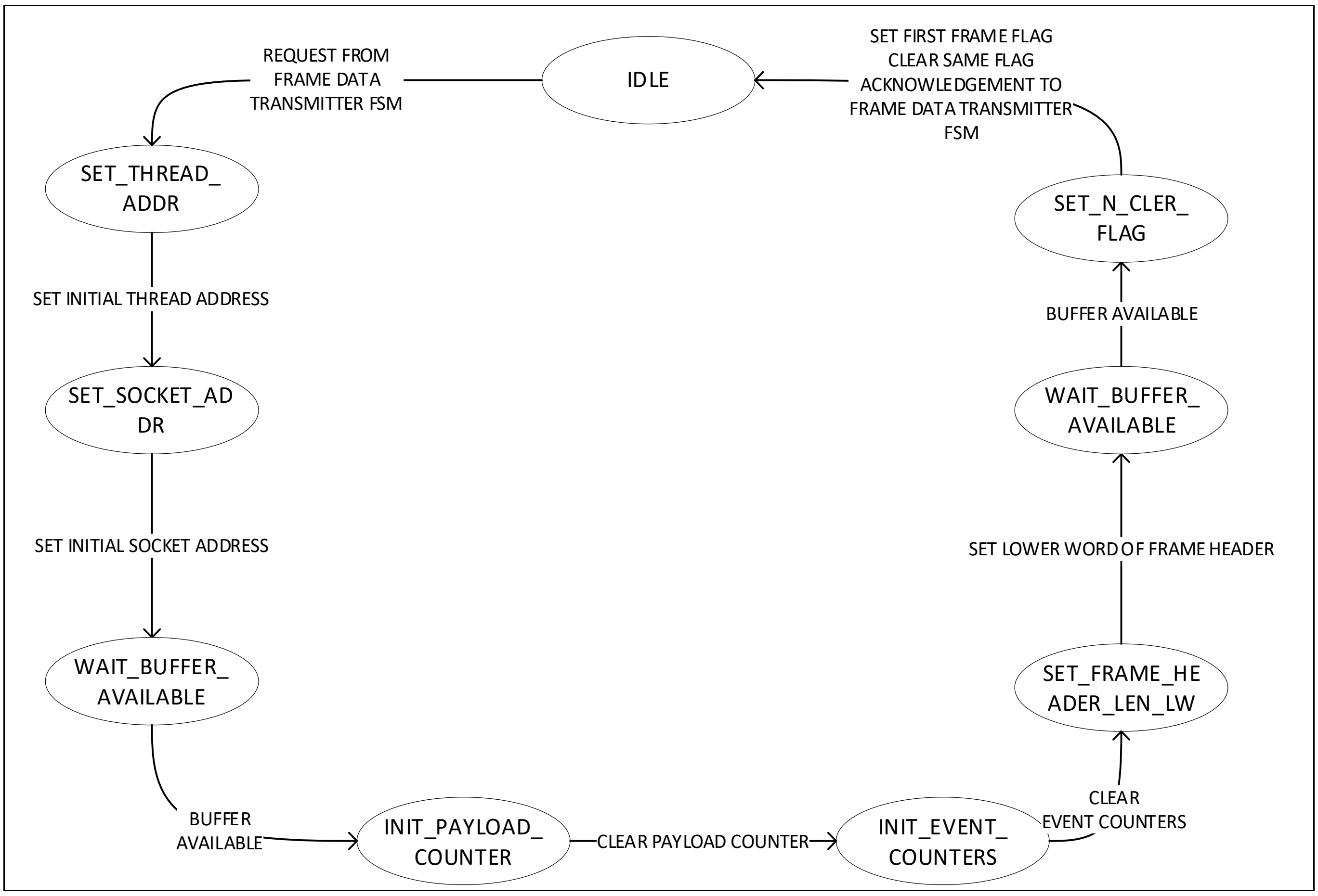


Figure 25 Initial socket sequence FSM

7.3.4.2 Frame header information transmitter FSM

The Frame Header Information Transmitter FSM is responsible for transmitting information related to UVC frame header. It sends IDLE control byte until the imaging device (THine camera module with built-in ISP or the Sony IMX715 image sensor board or internal colorbar pattern generator) begins capturing data. As soon as the selected device begins capturing data, the PTS control byte is sent, followed by IDLE control bytes until the camera device starts sending data to the FPGA.

Once the FPGA receives data, the Frame Header Information Transmitter FSM transmits the SCR control byte. These control bytes are used to manage Precision Time Stamp (PTS) values when multiple devices are transmitting data.

Once a buffer is available on the EZ-USB™ FX20 side, it transmits the SET FLAG control byte to indicate that the first frame data will be transmitted soon if FLAGs are enabled through FPGA I2C slave register.

At the end, it acknowledges to the Frame Data Transmitter FSM, confirming that the header information has been sent.

Each GPIF thread has 4 flags, flags are visible to GPIF and CPU. A Maskable interrupt will be raised to CPU, whenever these flags get set or cleared by FPGA. No counters are associated with flags, for detailed information on LVDS control byte, see Section 25.4 of the EZ-USB™ FX20 Architecture reference manual.

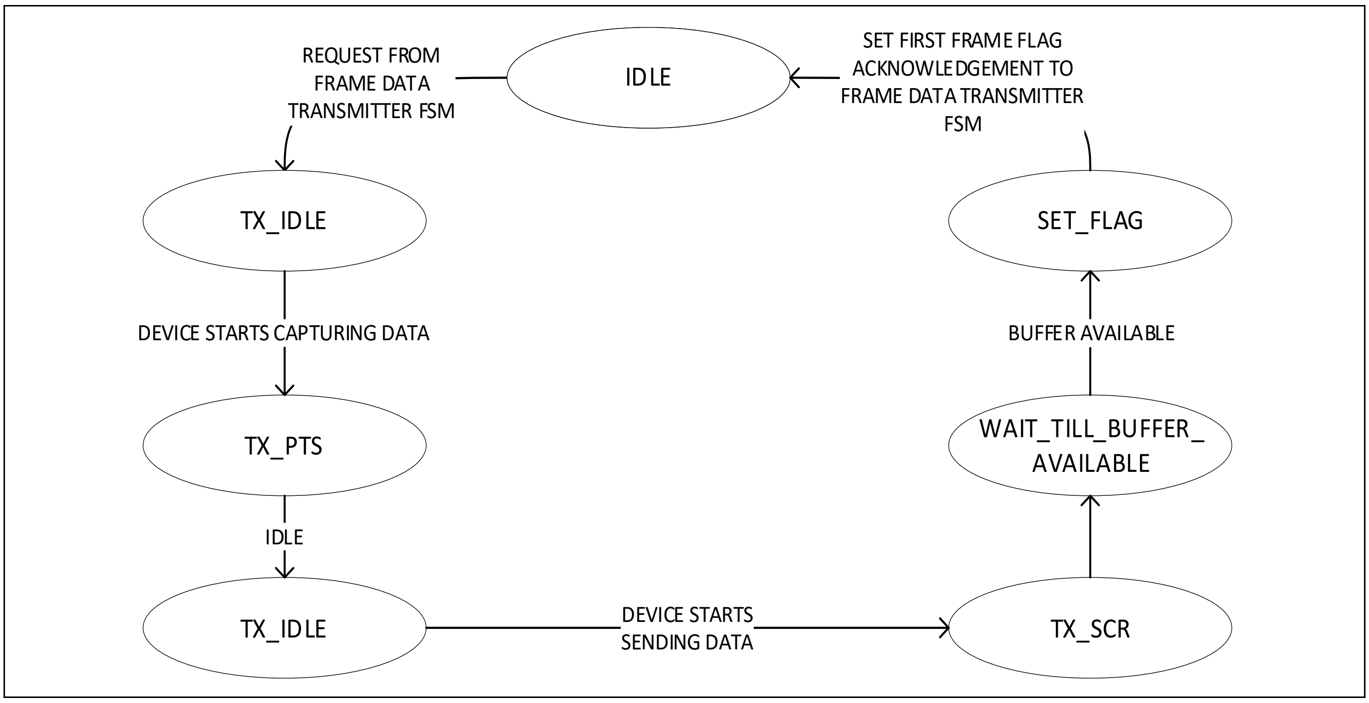

IDLE
REQUEST FROM FRAME DATA TRANSMITTER FSM
SET FIRST FRAME FLAG ACKNOWLEDGEMENT TO FRAME DATA TRANSMITTER FSM
TX_IDLE
SET_FLAG
DEVICE STARTS CAPTURING DATA
BUFFER AVAILABLE
TX_PTS
WAIT_TILL_BUFFER_ AVAILABLE
IDLE
TX_IDLE
DEVICE STARTS SENDING DATA
TX_SCR

Figure 26 **Frame header transmitter FSM**

7.3.4.3 LVDS data packet transmitter FSM

The Data Packet Transmitter FSM manages the transmission of USB Video Class (UVC) data packets.

The Data Packet Transmitter FSM sends packet information before each data packet if the FPGA is required to transmit packet details. Additionally, it transmits a control byte to enable the payload counter and flags to inform EZ-USB™ FX20 that a new packet is coming, provided the corresponding bits are set in the FPGA I2C register. It also manages to send data packets and control bytes, such as clearing flags to indicate the completion of the current data packet and hold payload count if respective bits are configured in the FPGA I2C register.

After the last data packet, the Data Packet Transmitter FSM transmits an EOP (End of Packet) control byte to commit the buffer to EZ-USB™ FX20. Subsequently, it also switches to a new thread and socket address if it is configured with multi thread.

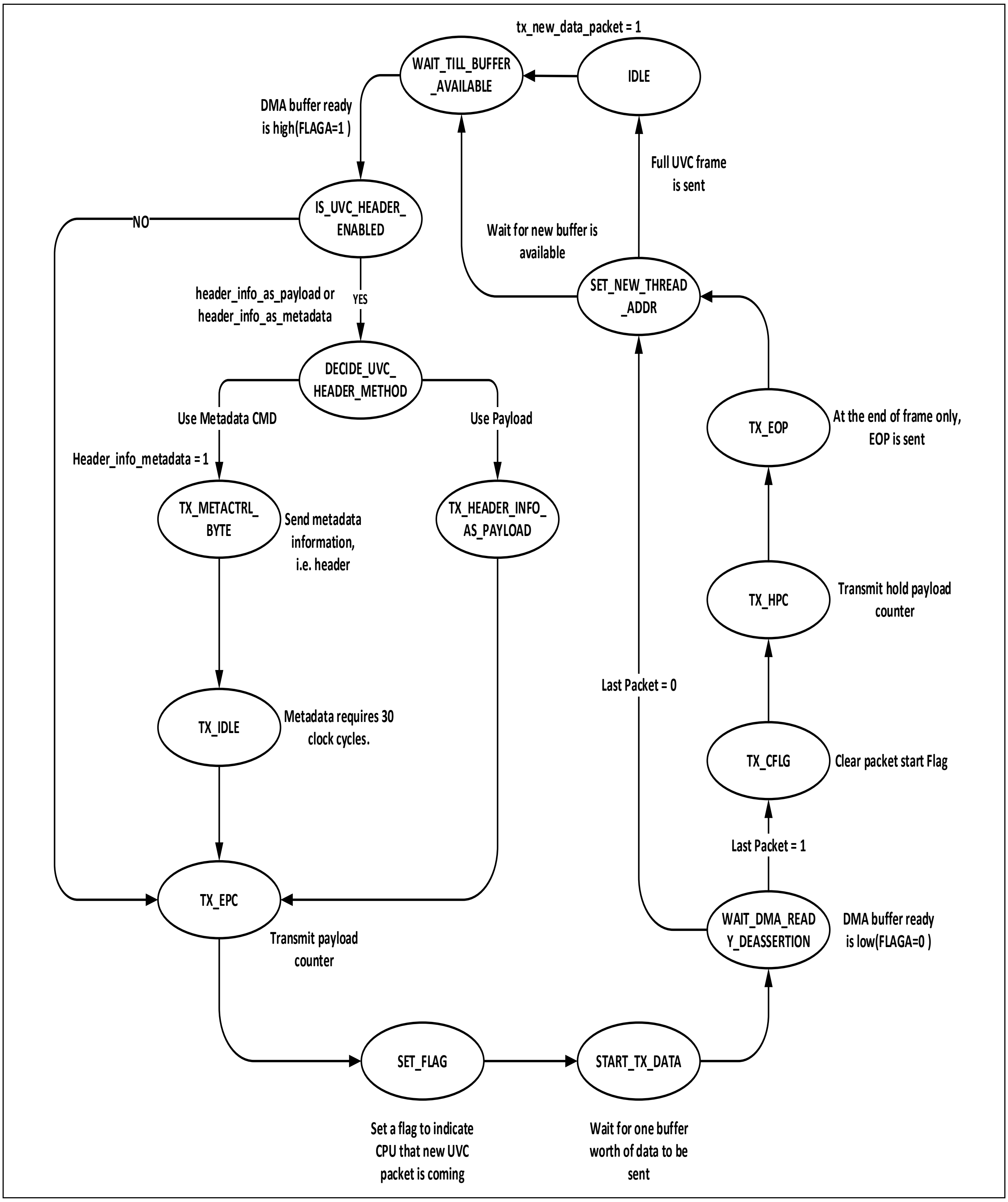

tx_new_data_packet = 1
WAIT_TILL_BUFFER_AVAILABLE
IDLE
DMA buffer ready is high(FLAGA=1)
Full UVC frame is sent
IS_UVC_HEADER_ENABLED
NO
Wait for new buffer is available
SET_NEW_THREAD_ADDR
header_info_as_payload or header_info_as_metadata
YES
DECIDE_UVC_HEADER_METHOD
Use Metadata CMD
Use Payload
TX_EOP
At the end of frame only, EOP is sent
Header_info_metadata = 1
TX_METACTRL_BYTE
Send metadata information, i.e. header
TX_HEADER_INFO_AS_PAYLOAD
TX_HPC
Transmit hold payload counter
Last Packet = 0
TX_IDLE
Metadata requires 30 clock cycles.
TX_CFLG
Clear packet start Flag
Last Packet = 1
TX_EPC
WAIT_DMA_READY_DEASSERTION
DMA buffer ready is low(FLAGA=0)
Transmit payload counter
SET_FLAG
START_TX_DATA
Set a flag to indicate CPU that new UVC packet is coming
Wait for one buffer worth of data to be sent

Figure 27 Data packet transmitter FSM

7.3.4.4 Frame end transmitter FSM

The Frame End Transmitter Finite State Machine (FSM) is responsible for sending a sequence of control bytes to indicate the end of a frame (EOP). It first transmits a control byte to clear a flag (CFLAG), which indicated that all bytes in a frame have been transferred. Subsequently, it updates the event counter to count the current frame. This update occurs if the respective bits are set in the FPGA I2C slave register by the EZ-USB™ FX20.

At the end, it informs Frame Data Transmitter that current frame is completed. Upon receiving acknowledgement, the Frame Data Transmitter informs the Device Selection FSM that the selected device is served. Hence, it moves to next activated device. Thus, it serves all the activated devices one by one in a round robin manner.

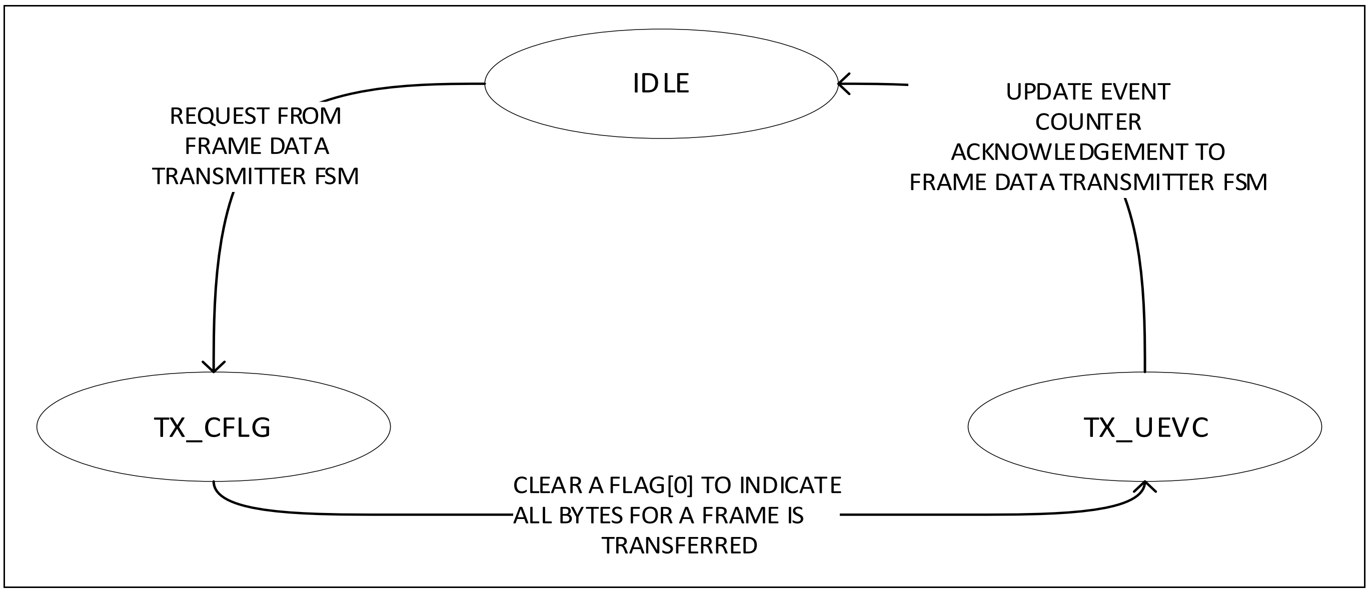

IDLE
REQUEST FROM FRAME DATA TRANSMITTER FSM
UPDATE EVENT COUNTER ACKNOWLEDGEMENT TO FRAME DATA TRANSMITTER FSM
TX_CFLG
TX_UEVC
CLEAR A FLAG[0] TO INDICATE ALL BYTES FOR A FRAME IS TRANSFERRED

Figure 28 Frame end transmitter FSM

8 Audio interface

The EZ-USB™ FX20 supports Audio device interface via Inter-IC sound bus (I2S) serial bus standard, as well as a microphone interface using a PDM to PCM converter. In addition to the standard I2S format, the I2S block also supports the Left Justified (L J) format and the Time Division Multiplexed (TDM) format. In this example, interfacing PDM microphone with EZ-USB™ FX20 GPIO accessory board.

8.1 Direct I2S Rx/Tx support in EZ-USB™ FX20

The I2S block consists of two sub-blocks: the I2S Transmitter (Tx) and I2S Receiver (Rx). Each sub-block can be configured independently for the digital audio interface format and master/slave mode When in master mode, the I2S block in the EZ-USB™ FX20 MCU generates the word select (WS) and serial data clock (SCK). However, In the slave mode, the WS, and SCK signals are inputs signals to the EZ-USB™ FX20 MCU and are generated by the external master device.

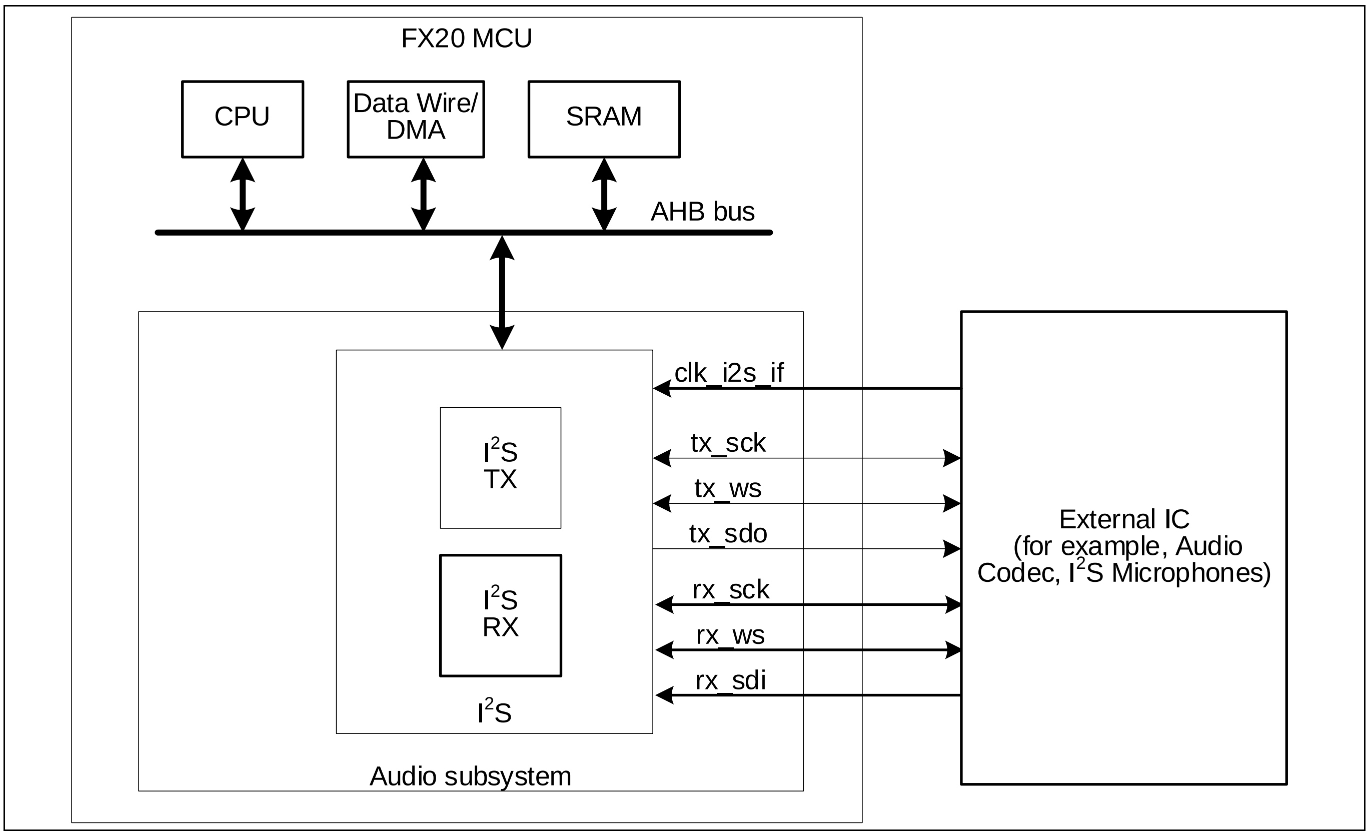


Figure 29 EZ-USB™ FX20 I2S internal design

8.2 Built-in PDM to PCM converter support

The PDM-PCM unit accepts a stereo or mono serial data stream (pulse modulated 1-bit stream) coming from external digital PDM microphones. The entire PDM-PCM conversion process is handled in hardware; the PCM output data streaming can be done using the DMA controller, thus freeing up the CPU bandwidth from performing periodic audio streaming activities.

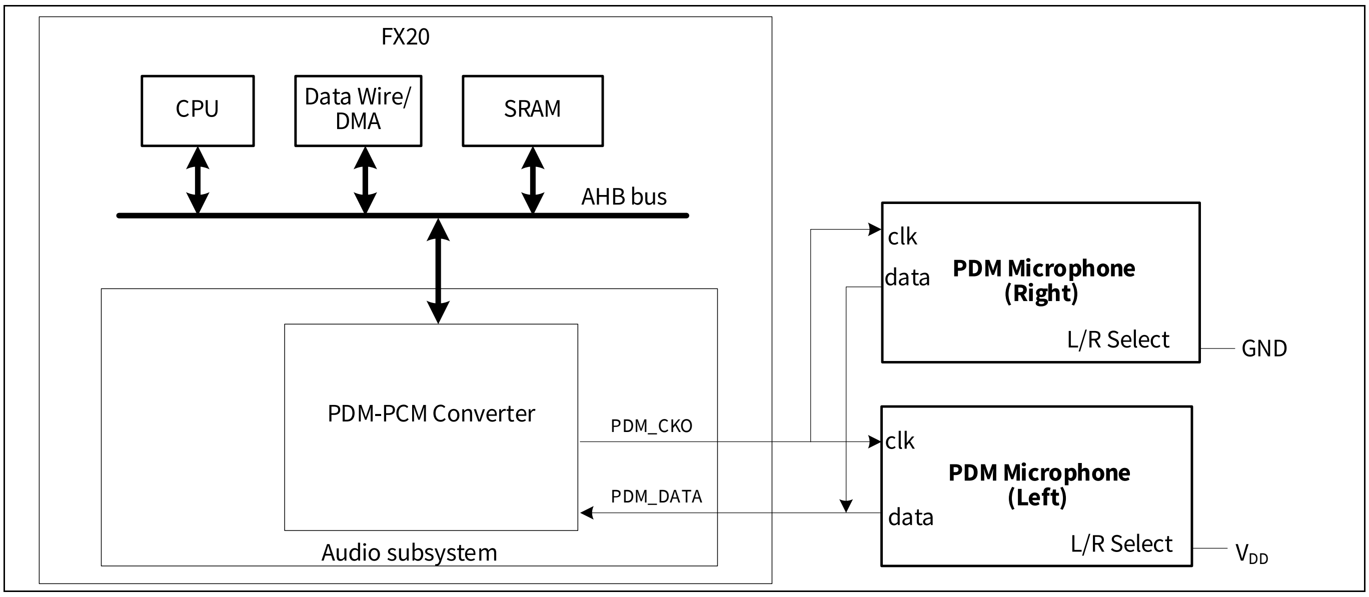


Figure 30 PDM to PCM internal architecture

The PDM-PCM converter can be powered ON or OFF by using the ENABLED bit in the PDM_CTL register.

Mono/stereo microphone support

The PDM-PCM converter supports mono-left, mono-right, stereo, and swapped stereo modes of operation. The operation mode is controlled by the PDM_CH_SET and SWAP_LR bits in the PDM_MODE_CTL register. See EZ-USB™ FX20 controller architecture reference manual section 33 for more information about PDM to PCM converter.

8.3 Pin mapping of audio (microphone)

For PDM interface, see the “Pin definitions” section 3.1 in the EZ-USB™ FX20 datasheet to know the PDM Clock and PDM Data pins.

9 Firmware

See the example firmware needed for this application note in the *cyfx3g2uvcuac_an38689* folder from the GitHub. This example firmware can enumerate as UVC + UAC composite device and stream video from a camera module (either THine camera module with built-in ISP or a Sony IMX715 image sensor board) and stream audio from the PDM microphones.

Note: Check the Makefile and cy_usb_app.h file to set the appropriate flag to stream video from either THine camera module with built-in ISP or a Sony IMX715 image sensor board and PDM microphone.

Table 13 summarizes the code modules and the functions implemented in each module.

Table 13 Example project source files

File	Description
cy_imagesensor.c	Contains Sony IMX715 sensor configuration details such as I2C register and register value for 3840 x 2160 60 fps, 1920 x 1080 60 fps. When the EZ-USB™ FX20 device powered, it configures the sensor with 3840 x 2160 resolution by calling the function `cy_configureImagesensor`. The function `cyImagesensorSetResolution` takes image width and height as a parameter and configure the Image sensor according to it. In case a wrong resolution parameter is received then the function will print an error message and return the error code. The function defined in this source file calls I2C read and write function to send sensor configuration via I2C
cy_imagesensor.h	Contains the constants used for the image sensor (its I2C slave address). You must define the I2C slave address of the image sensor here. Contains the declarations of all the functions defined in *cy_isp.c*
cy_usb_videocontrol.c	The function `HandleVCInterfaceRequest` is called from *cy_usb_app.c* when EZ-USB™ FX20 receives a VC request from the host camera application. This function mainly checks the type of control request received (Camera Terminal Control or Processing Unit Control) by examining the parameter wIndex, and then handles the appropriate controls based on the parameter wValue. *cy_usb_videocontrol.c* also contains Min, Max, Default, Current, and Set values for all video controls. The above-mentioned values can be changed as per the sensor limits
cy_usb_videocontrol.h	Contains the constants used for Camera Terminal Controls and Process Unit Control. Contains the declarations of all the functions defined in *cy_usb_videocontrol.c*
fx3g2_descriptors.c	Contains the USB enumeration descriptors for the UVC and UAC application. This file needs to be changed if the frame rate, image resolution, bit depth, or supported video controls need to be changed. The UVC specification has all the required details

File	Description
main.c	Contains startup code for the CM0+ controller. As soon as EZ-USB™ FX20 powered, CM0+ core gets initialized then it hands over to CM4 core. Therefore, the CM0+ boot code is kept at the beginning of the flash, observe cyusb4xxx_cm4.ld for more info. The CM4 core started execution from the function main, then start initializing peripheral driver library (PDL), device clock (Initialize FLL and PLL to generate clock for clk_fast, clk_peri), I2C, USB-FS for debug, and registering ISR for USB block, Hb_DMA, and initializes the USBD layer. Once the initialization is done, then firmware check whether FPGA is configured or not by checking the C_DONE pin of FPGA. If FPGA is configured, then firmware call's function `Cy_USB_AppInit`. This function is responsible for initializing FPGA register via I2C, registering callback functions, creating FREERTOS threads, queue, and timer for application. FreeRTOS systick timer is initialized in the function `vPortSetupTimerInterrupt` The function *InitLvdsInterface* is defined to create a GPIF threads, callback function when GPIF encounters an error and starting the GPIF state machine.
Makefile	It is a script file and it contains a set of directives used to build the EZ-USB™ FX20 binary. It specifies how this project should be built and can include rules for compiling source code, linking object files and managing dependencies. By default, the firmware is configured for SONY_IMX715 sensor and GPIF III configured as LVCMOS-WideLink. Modifying the below flag can change the firmware configuration • Set the flag SONY_IMX715_EN=1 if you are interfacing with a Sony sensor • Set WL_EN=1 and PORT1_EN=0 for WideLink • Set WL_EN=0 and PORT1_EN=0 for Port0 Narrow link • Set WL_EN=0 and PORT1_EN=1 for Port1 Narrow link • Set AUDIO_IF_EN=1 for enabling audio composite device, this flag can be set to '0' if you are not using AUDIO • Set PORT0_THREAD_INTLV=1 for configuring two GPIF threads (this gives better performance when streaming video from a single device) • Set SONY_IMX715_EN=1 to stream video from the Sony IMX715 image sensor • Set THINE_EN =1 to stream video from THine camera module with built in ISP

File	Description
cy_usb_i2c.c	This file contains functions used to initialize the I2C, write data, and read data via I2C. The function `Init_i2c` is called from the *main.c* file when the EZ-USB™ FX20 is powered. This function initializes the I2C peripheral in interrupt mode and sets the SCL frequency to 100 kHz of SCB0 The function `Cy_APP_GetFPGAVersion` is used to read and print the FPGA version. The function *cyUSBI2cWrite* receives parameters such as slave address. RegisterAddress, data, and length of the address and data from the calling function. It extracts the register address and calls the API *cyi2c_mster_write* to write data via I2C Similarly, the function *cyUSBI2cRead* also receives the parameter such as slave address, register address, a pointer to store read data and length of the address and data from the calling function. It extracts the register address and calls the API *cyi2c_master_write* to send the address to the I2C slave. Subsequently, it calls the API *cyi2_master_read* to read the I2C slave register data and store it to the provided data pointer
cy_usb_i2c.h	This file contains a constant for FPGA I2C address. Change this value in case FPGA having different address This file also contains function declaration for all the functions defined in the *cy_usb_i2c.c* file
cy_uac_app.c	This file handles the USB Audio functionalities. The function *MxPdmInit*, initialize the PDM module including configuring IO pins, setting the PDM clock frequency and initialize single channel for MONO and dual channel for STEREO as required for the USB Audio class interface. The DMA channel configured and Task creation for Audio is done in *cy_usb_app.c* file. The task handler function `cy_USB_PDMDEviceTaskHandler` running in a super loop and looking for an event CY_USB_PDM_MSG_WRITE_COMPLETE or CY_USB_PDM_MSG_READ_COMPLETE. These events are asserted when EZ-USB™ FX20 receives a DMA event when reading and committing the data to the USB host.

File	Description
cy_usb_app.c	*cy_usb_app.c* is the main application source file for UVC and UAC. Changes are needed when modifying the code to support different video resolution (probe control info), creating DMA channel configuration and initializing FPGA register This source file contains the following functions: Upon the invocation of the FreeRTOS scheduler by the firmware, the application firmware starts executing the Task Cy_USB_UvcDeviceTaskHandler. The first job of this task is to initialize the LVDS IP and GPIF state machine by calling the function `InitLvdsInterface`. Subsequently, it checks for VBUS, enabling the USB connection based on the detection of VBUS. Next, the Image sensors are configured, and the status of the image sensor configuration is printed on the debug terminal. The One second software timer is started as it is required for calculating frames received in one second and reading FPGA status. Once all the configurations are done, the firmware enters a super loop and servicing various kinds of events. • The function `Cy_AppPrintStartupLogs` is used to print the HW board used, the mode of GPIF configuration and software versions of firmware and FPGA • The function `Cy_USB_UvcAddHeader` is used to add a UVC header to every DMA buffer when DMA is configured in Manual mode. This function is called when FX20 executes the DMA call back function • The function `Cy_usb_StreamStartStop` is used to inform FPGA whether to send frame data or not. This information is delivered via I2C by setting and clearing the appropriate FPGA register • The operation of the function `CyUVCAppStop` is to stop committing the video data to the USB host. This can be achieved by forming FPGA not to send data, followed by resetting the DMA channel, flushing the endpoint, and resetting the endpoint • The function `CyUVCAppStop` is called by the callback function `Cy_USB_AppSetupCallback` when FX20 receives CLEAR_FEATURE • The function `Cy_USB_UvcSetCurRqtHandler` and `CyUVCAppStart` are called when FX20 receives UVC PROBE and COMMIT control • The function `Cy_USB_UvcSetCurRqtHandler` is used to read the commit control data received form USB host and update the same to image sensor and FPGA. On the other hand, UVCAppStart performs Enabling DMA channel, updating the video resolution and restarting the GPIF state machine • The `Cy_USB_AppSetupEndpDmaParamsSs` perform high bandwidth DMA initialization associated with a USB endpoint. User can modify this function to change the DMA configuration as per the requirement • The function `Cy_USB_AppConfigureEndp` is used to configure all the endpoint except the endp0 after set configuration

File	Description
cy_usb_app.h	The file contains macro definition for HW board selection, LVDS/LVCMOS configuration, GPIO port and pin definitions, constants for FPGA register address and data values. Additionally, it includes an application data structure which is a bridge between USB system and device functionality. This file also contains function declaration for all the functions defined in the *cy_usb_app.c* file The following shows macro setup for interface mode selection **LVCMOS SDR mode** #define LVCMOS_EN (1) **LVCMOS DDR mode** #define LVCMOS_EN (1) #define LVCMOS_DDR_EN (1) **LVDS mode** #define LVCMOS_EN (0) #define LVCMOS_DDR_EN (0)
cy_usb_uvc_device.h	Header file contains constant values for different events used in this project, Interface number, bitfield value as per the UVC specification
cyusb4xxx_cm4.ld	Linker file for Cortex® M4 processor
cy_gpif_header_lvcmos.h	Header file for GPIF III register writes. No changes are required if you are using it for Slave FIFO with LVCMOS interface This file contains the GPIF III register address, register data, and contains GPIF configuration information. These details are passed to API calls in the *cy_usb_app.c* file to start and run the GPIF III state machine
cy_gpif_header_lvds.h	Header file for GPIF III register writes. No changes are required if you are using it for Slave FIFO with LVDS interface. This file contains the GPIF III register address, register data, and contains GPIF configuration information. These details are passed to API calls in the `cy_usb_app.c` file to start and run the GPIF III state machine

9.1 Application thread (RTOS task)

The application firmware is developed as a composite device, it contains CDC class for debug, UAC class for audio (microphone) recording and UVC class for video streaming. The application contains total of three threads (RTOS Task), for handling video, audio streaming and printing debug message. These application thread enables the concurrent functionality.

9.2 Firmware flowchart

Figure 31 shows how the firmware gets started and runs in EZ-USB™ FX20 for UVC application.

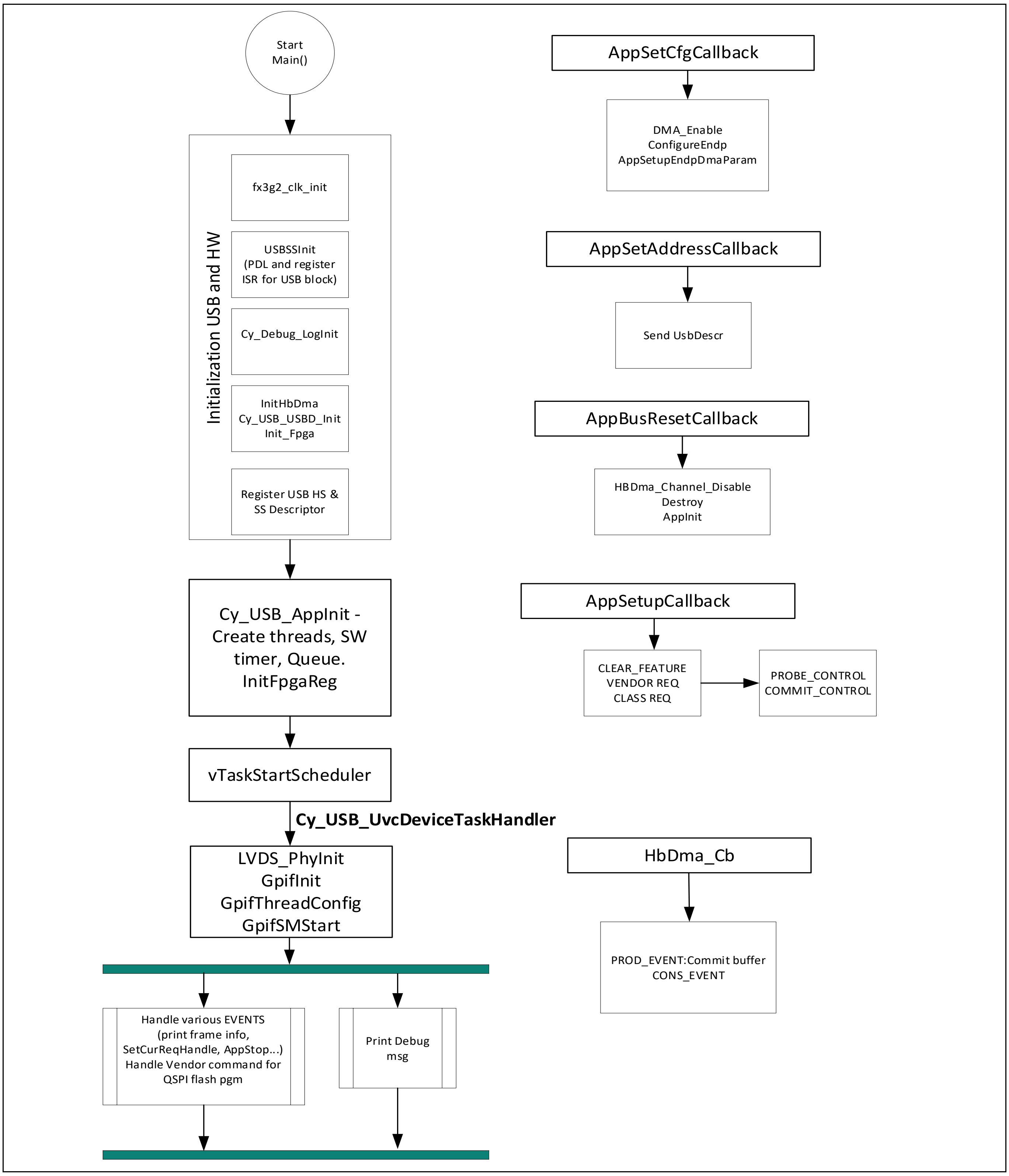

Start Main()
Initialization USB and HW
fx3g2_clk_init
USBSSInit (PDL and register ISR for USB block)
Cy_Debug_LogInit
InitHbDma Cy_USB_USBD_Init Init_Fpga
Register USB HS & SS Descriptor
Cy_USB_AppInit - Create threads, SW timer, Queue. InitFpgaReg
vTaskStartScheduler
Cy_USB_UvcDeviceTaskHandler
LVDS_PhyInit GpifInit GpifThreadConfig GpifSMStart
Handle various EVENTS (print frame info, SetCurReqHandle, AppStop...) Handle Vendor command for QSPI flash pgm
Print Debug msg
AppSetCfgCallback
DMA_Enable ConfigureEndp AppSetupEndpDmaParam
AppSetAddressCallback
Send UsbDescr
AppBusResetCallback
HBDma_Channel_Disable Destroy AppInit
AppSetupCallback
CLEAR_FEATURE VENDOR REQ CLASS REQ
PROBE_CONTROL COMMIT_CONTROL
HbDma_Cb
PROD_EVENT:Commit buffer CONS_EVENT

Figure 31 **UVC firmware flowchart**

9.3 Audio (PDM) data flow

Audio data flow is configured on receiving the SET configuration event in setup call-back. Audio data is received from the PDM peripheral which is outside of the HBWSS. The CPU subsystem's Data Wire (CPUSS_DW) interface is used to move the data present in PDM Receiver's FIFO to the HBWSS SRAM buffers. The Data Wire (DW) is configured to trigger an interrupt to notify the CPU on completion of data transfer from PDM RX FIFO to HBWSS SRAM. This interrupt is used by the application to trigger the transfer from HBWSS SRAM to the USB endpoint. See Figure 27 for a diagrammatic representation of audio data flow mechanism.

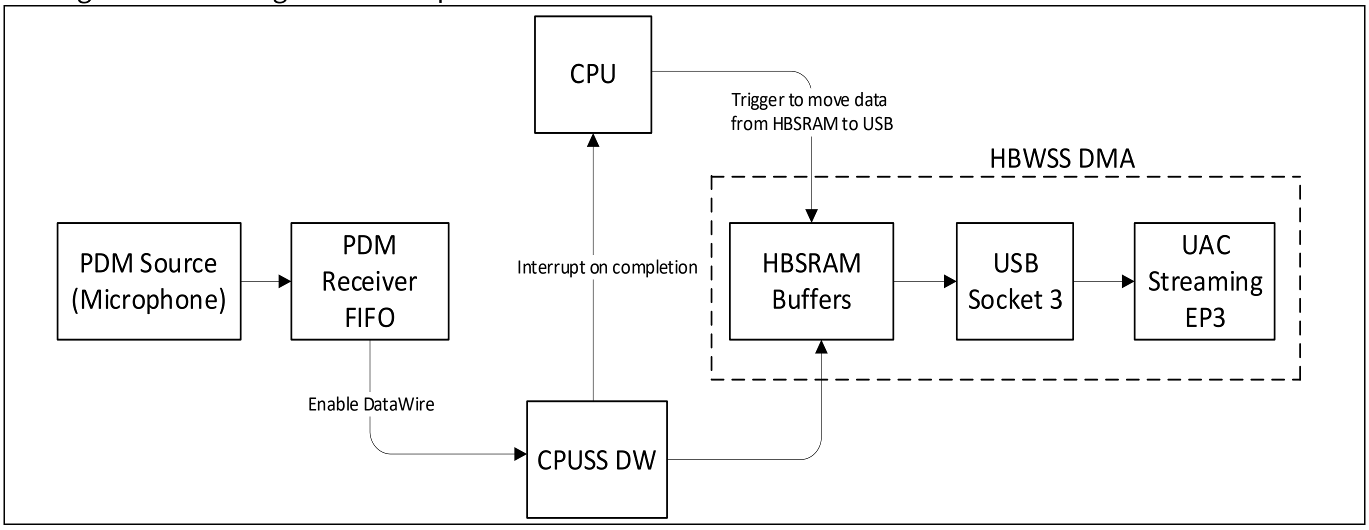


Figure 32 Audio streaming data flow

9.4 Enumeration

In the `Cy_USB_AppInit` function, the callback functions are registered `Cy_USB_AppRegisterCallback` with USBD layer. These callbacks will be called based on the appropriate event, when a USB address has been assigned to the device trigger the event and call the function `Cy_USB_AppSetAddressCallback`. In the `Cy_USB_AppSetAddressCallback` function, check the type of USB connection and register appropriate descriptors defined in *fx3g2_descriptors.c* file to ensure that EZ-USB™ FX20 enumerates as a composite device (UVC + UAC + CDC). These descriptors are defined for the FPGA to send the data in 16 bits per pixel using the uncompressed YUY2 format as image sensor SONY IMX 715 is sending RAW 12 bits per pixel at 60 fps. See Section 2.3.1 if you need to change these settings.

9.5 Configuring the video source through the I2C interface

The image sensor is configured using the I2C master block SCB0 of EZ-USB™ FX20. The function `Cy_ConfigureImageSensor` and `CyImageSensorSetResolution` defined in *cy_imagesensor.c* is used to write image sensor configuration over I2C. The functions `ConfigureImageSensor` and `CyImageSensorSetResolution` call upper layer function `CyUSBSendI2cTable` and this function call the standard API `CyUSBI2cWrite` to write data to the image sensor.

9.6 Starting the video streaming

The USB host application, such as the VLC media player or Windows camera app, or Webcamoid uses the UVC driver to display the video. The application will send a PROBE/COMMIT control request to select the appropriate USB interface and the USB alternate setting combination to stream video (usually Interface 0 Alternate setting 1). It is an indication by the host that it will shortly begin to stream video data. On a stream event, the USB host application starts requesting image data from EZ-USB™ FX20; EZ-USB™ FX20 is supposed to start sending the image data from the image sensor to the USB host. In the firmware, the `Cy_USB_UvcDeviceTaskHandler` function is an infinite loop. While there is no streaming, the main application thread waits in this loop until there is a stream event.

Note: *If there is no stream event, EZ-USB™ FX20 does not need to transfer any data. Therefore, the GPIF III state machine is disabled or function CyUVCAppStop should be called. The function `CyUVCAppStop` reset the HBDma and flushes Endpoint by calling `Cy_HBDma_Channel_Reset`, `Cy_USBD_FlushEp`. This leaves all the DMA buffers associated with the channel in the empty state and place the channel into the disabled state. Otherwise, all the DMA buffers will be full before the host application starts to pull data out of the DMA buffers, and EZ-USB™ FX20 would transmit a bad frame.*

When EZ-USB™ FX20 receives a stream event, the callback function `Cy_USB_UvcSetCurRqtHandler` calls the `CyUVCAppStart` function to enable the HBDma channel and to start the GPIF III state machine. Inside this function, the firmware switches from any state to the start state (**START**) and restarts the GPIF state machine. Pass the start state name and start condition as arguments to the `Cy_LVDS_GpifSMSwitch` function. The from state and end state can be any invalid state (255 in the attached project) as this will ensure that a GPIF III state machine switches from any state to the start state. The start state (**START**) and the start condition (**ALPHA_START**) are defined in *cy_gpif_header_lvcmos.h* file.

9.7 Setting up DMA buffers

The UVC spec requires adding a header to each USB transfer (In this example, adding 32 bytes) to each DMA buffer. However, the EZ-USB™ FX20 architecture requires that any DMA buffer associated with a DMA descriptor must have a size that is a multiple of 32 bytes. The maximum DMA buffer size must be less than 64KB.

As explained in Section 2.4.4 in this application note, UVC header insertion can be added in three ways.

In case EZ-USB™ FX20 CPU needs to add UVC header, then manual DMA should be created and assigns 32 bytes to the DMA header size parameter and the call-back function should be provided to handle UVC header insertion by CPU. When a header is inserted by the FPGA or using the metadata feature, the DMA header size parameter should be updated to the value of '0', and no callback function should be provided.

The following code snippet shows typical Manual DMA configuration for single thread, WideLink mode.

Code listing 1

```
dmaConfig.chType = CY_HBDMA_TYPE_IP_TO_IP;     /* DMA Channel type: from
LVDS to USB EG IP. */
dmaConfig.consSckCount  = 1;                    /* Only one consumer socket
per channel. */
dmaConfig.consSck[0] = (cy_hbdma_socket_id_t)(CY_HBDMA_USBEG_SOCKET_00 +
endpNumber);
dmaConfig.consSck[1] = (cy_hbdma_socket_id_t)0;
dmaConfig.eventEnable = 0;                          /* Manual channel: Disable
event signalling between sockets. */
dmaConfig.intrEnable = VDSSS_LVDS_ADAPTER_DMA_SCK_INTR_PRODUCE_EVENT_Msk |
LVDSSS_LVDS_ADAPTER_DMA_SCK_INTR_CONSUME_EVENT_Msk;
dmaConfig.cb  = HbDma_Cb;   /* HB-DMA callback */
dmaConfig.size = 64512;              /* DMA Buffer size in bytes */
dmaConfig.prodHdrSize = 32;
dmaConfig.prodBufSize = 64512-32;  use macro
dmaConfig.count = 4;                          /* DMA Buffer Count */
dmaConfig.bufferMode = true;                      /* DMA buffer mode enabled */
dmaConfig.userCtx = (void *)(pUsbApp);             /* Pass the application
context as user context. */
dmaConfig.prodSckCount = 1; /* No. of producer sockets */
dmaConfig.prodSck[0] = CY_HBDMA_LVDS_SOCKET_00;
dmaConfig.prodSck[1] = (cy_hbdma_socket_id_t)0; /* Producer Socket ID: None
*/
```

Note: *See the `Cy_USB_AppSetupEndpDmaParamsSs` function in cy_usb_app.c to know how to configure auto / manual DMA channel and configuration required for USB HS.*

9.8 Handling the DMA buffers during video streaming

The function `Cy_USB_AppSetupEndpDmaParams` get called when it receives a set config event from the USBD layer, this function creates a DMA channel with callback notification for producer and consumer event. This notification is used to track the amount of data sent by the FPGA and the amount of data read by the host. A DMA buffer becomes available to the EZ-USB™ FX20 CPU when the GPIF III produces a DMA buffer and EZ-USB™ FX20 gets a pointer to this buffer via Cy_HBDma_Channel_GetBuffer. This buffer is committed to the USB using Cy_HBDma_Channel_CommitBuffer API.

The UVC header carries information about the frame identifier and an end-of-frame marker. At the end of a frame, the EZ-USB™ FX20 firmware sets bit 1 and toggles bit 0 of the second UVC header byte (see *Cy_USB_UvcAddHeader*). Toggle the UVC header FID bit only after the frame ends.

A slower USB host will result in commit buffer failures and resetting of DMA to restart the video stream. If the Sensor/ISP fails to send video on time, the video will eventually freeze or show some jitters. Frame timer is an essential entity to avoid this. The frame timer starts when the video streaming starts. It is reset when EZ-USB™ FX20 receives first buffer from the Sensor and restarts to ensure that each producer buffers arrive on time.

9.9 Terminating the video streaming

There are three ways image streaming can be terminated:

- The camera may be disconnected from the host, the USB host program may close
- The USB host may issue a reset
- Suspend request to EZ-USB™ FX20

This action does not always happen when there is no data in the EZ-USB™ FX20 FIFO. When EZ-USB™ FX20 receives a SUSPEND event, then firmware sends ACK and waits for the next streaming event to occur. When the application closes, it issues a clear feature request on a Windows platform or a set interface with alternate setting = 0 request on a Mac platform. Streaming stops when this request is received. These requests are handled in the `Cy_USB_UvcSetCurRqtHandler` function under the conditional case CY_USB_SC_SET_FEATURE and CY_USB_SC_CLEAR_FEATURE.

9.10 Enabling debug interface using USB FS

The EZ-USB™ FX20 SDK supports a built-in debug interface. Debug prints can be routed to UART in any SCBs, or to USB. The EZ-USB™ FX20 DVK contains a USB FS port on a J3 connector that can be used as a debug port. The following code changes need to be made to use the debug prints feature on USB FS.

Code listing 2

```
dbgCfg.pBuffer    = (uint8_t *)LogDataBuffer; /*pointing to buffer*/
dbgCfg.traceLvl   = 3U;                       /*debug trace level*/
dbgCfg.bufSize    = LOGBUF_RAM_SZ;            /*debug buffer size*/
dbgCfg.dbgIntfce = CY_DEBUG_INTFCE_USBFS_CDC;
dbgCfg.printNow   = true;                    /* Printing messages
immediately for now. */
Cy_Debug_LogInit(&dbgCfg);                    /*function call to initialize
debug*/
```

Note: The EZ-USB™ Code builder tool can be used to generate the UVC and UAC application firmware in LVCMOS / LVDS interface mode. Additionally, the tool can generate firmware for multiple UVC (up to 4 UVC function) endpoints.

10 UVC host driver and application

10.1 UVC host driver

The USB Video Class (UVC) driver is typically included with the operating system and is automatically loaded when the video device is connected.

10.2 UVC host application

The UVC application is the software that controls the video capture process and can be used to adjust various settings, such as resolution, frame rate, and exposure. EZ-USB™ FX20 supports all standard USB Video Class-compatible host applications. Some of the UVC applications are listed below.

- Windows: Windows Camera, VLC media player, MPC-HC, e-CAMView, Webcamoid, etc.
- Linux: Cheese player, Webcamoid
- macOS: PhotoBooth, Webcamoid

11 Tools

11.1 USB 3.2 Protocol Analyzer (LeCroy, Wireshark)

A USB protocol analyzer is a debugging tool that allows you to analyze the traffic on the USB between EZ-USB™ FX20 and the USB host. The Software tools included with each analyzer then decodes the data into USB transfer packets. By analyzing this data, issues can be easily identified, and performance can be maximized.

Several USB analyzers are available in the market today. Although Infineon does not recommend any specific analyzer, however, providing you with a few options:

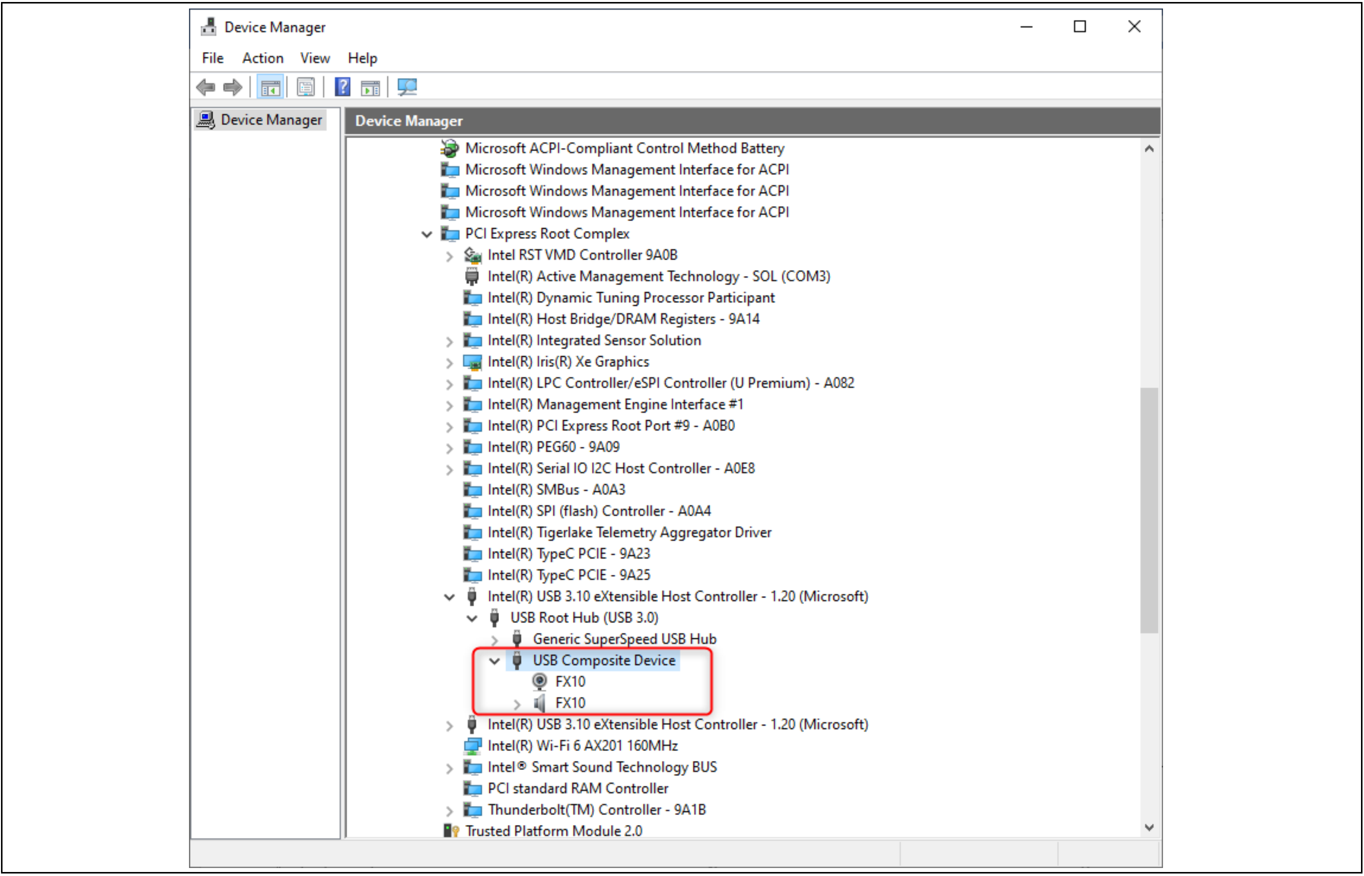
- Standalone USB 10 Gbps protocol analyzer
  - LeCroy USB Voyager M310
- PC software USB 3.0 protocol analyzer
  - Wireshark

See AN237841 – Getting started with EZ-USB™ FX20 to know more about the tools and troubleshooting guide.

11.2 Device Manager

Open the Device Manager and check the device FX20 is listed under Camera.

If the device is not listed, then check and verify the descriptor information provided under *SuperSpeedConfigDescr* and *HighSpeedConfigDescr* in the *FX20_descriptors.c* file.



Microsoft ACPI-Compliant Control Method Battery
Microsoft Windows Management Interface for ACPI
Microsoft Windows Management Interface for ACPI
Microsoft Windows Management Interface for ACPI
PCI Express Root Complex
Intel RST VMD Controller 9A0B
Intel(R) Active Management Technology - SOL (COM3)
Intel(R) Dynamic Tuning Processor Participant
Intel(R) Host Bridge/DRAM Registers - 9A14
Intel(R) Integrated Sensor Solution
Intel(R) Iris(R) Xe Graphics
Intel(R) LPC Controller/eSPI Controller (U Premium) - A082
Intel(R) Management Engine Interface #1
Intel(R) PCI Express Root Port #9 - A0B0
Intel(R) PEG60 - 9A09
Intel(R) Serial IO I2C Host Controller - A0E8
Intel(R) SMBus - A0A3
Intel(R) SPI (flash) Controller - A0A4
Intel(R) Tigerlake Telemetry Aggregator Driver
Intel(R) TypeC PCIE - 9A23
Intel(R) TypeC PCIE - 9A25
Intel(R) USB 3.10 eXtensible Host Controller - 1.20 (Microsoft)
USB Root Hub (USB 3.0)
Generic SuperSpeed USB Hub
USB Composite Device
FX10
FX10
Intel(R) USB 3.10 eXtensible Host Controller - 1.20 (Microsoft)
Intel(R) Wi-Fi 6 AX201 160MHz
Intel® Smart Sound Technology BUS
PCI standard RAM Controller
Thunderbolt(TM) Controller - 9A1B
Trusted Platform Module 2.0

Figure 33 Device Manager view

11.3 Serial terminal (Tera Term, Cool Term, etc.)

EZ-USB™ FX20's USB Communication Device Class (CDC) interface can be accessed using the following serial terminal tools:

- **Windows**: Tera Term, PuTTY. etc.
- **Linux**: screen, CuteCom, etc.
- **macOS**: Cool Term

11.4 EZ-USB™ FX Control Center

The EZ-USB™ FX Control Center tool is a GUI-based application used to load firmware and interact with the EZ-USB™ FX20 device. On the Windows platform, the tool uses the WinUSB driver from Microsoft, while on Linux and macOS platforms, it uses the libusb library. EZ-USB™ Control Center supports various features, including descriptor information, data transfer to different endpoints, data loopback, and performance measurement.

See the EZ-USB™ Control Center help document to learn about the features supported by the tool and how to use them.

11.5 EZ-USB™ FX20 to FPGA interface

This section provides a design example in which, an EFINIX Ti180 FPGA is connected to the EZ-USB™ FX20 through the synchronous slave FIFO or LVDS interface using FMC connector. The hardware and software tools used to stream video and audio over the UVC and UAC interface are described in detail below.

11.6 Hardware and software requirement

Hardware setup consists of the following:

- EZ-USB™ FX20 DVK
- Efinix Ti180J484 FPGA development kit
- EZ-USB™ FX20 GPIO Accessory Board
- Sony IMX715 image sensor accessory board (DEMO_MIPI_4K_CAM01)
- PDM Microphones
- USB 3.2 Gen 2x2 cable (supplied with FX20 DVK)

Host PC Minimum requirement.

- Intel core i5 or i7 CPU (11th Gen or higher)
- Ram should be minimum 2 x 8 GB or more (2 slot, 128 bit)
- USB 3.2 Gen 2x2 (20Gbps) port or higher

The following software should be installed in the PC:

- EZ-USB™ FX Control Center
- Efinity FPGA Programmer
- Serial terminal (Tera Term)
- Audacity

11.7 Hardware kit

Make sure you have collected all the necessary hardware listed in Section 11.6.

First, connect the EZ-USB™ FX20 DVK board to the EFINIX Ti180 FPGA DVK board. Next, attach the GPIO accessory board to the EZ-USB™ FX20 DVK board. Additionally, connect the Sony IMX715 camera board to the Efinix Ti180 DVK. See Figure 34 for a visual representation of these connections.

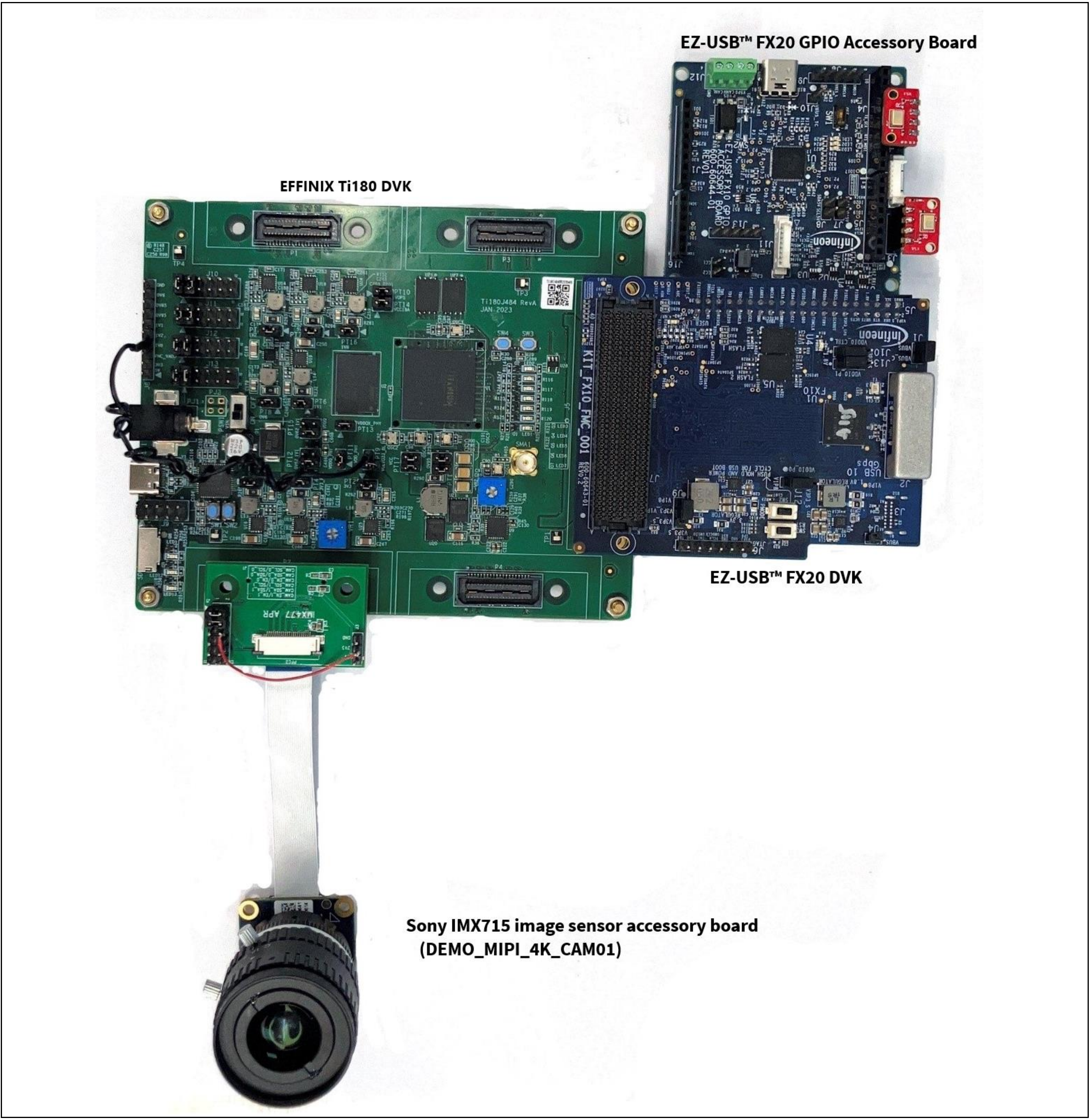

EZ-USB™ FX20 GPIO Accessory Board
EFINIX Ti180 DVK
EZ-USB™ FX20 DVK
Sony IMX715 image sensor accessory board
(DEMO_MIPI_4K_CAM01)

Figure 34 Hardware setup

Note: Before powering the setup, ensure that the required modifications are done on the EFINIX Ti180 FPGA DVK board.

Note that in this application note, using the EFINIX Ti180 FPGA to interface with EZ-USB™ FX20, user can use any FPGA as per the requirement.

11.8 EFINIX Ti180 FPGA board setup and hardware modification

1. The I2C signals from the EZ-USB™ FX20 DVK board are connected to the BL, BR, TL, and TR banks of the FPGA. These banks are powered by a 3.3 V supply on the FPGA board, while the EZ-USB™ FX20 DVK board is set to operate at 1.8 V
2. Remove PT9 jumper from EFINIX Ti180 FPGA DVK and connect the pin 2 of PT9 to 1.8 V supply as shown in Figure 34
3. This modification will set the FPGA bank power at 1.8 V
4. I2C lines EZ-USB™ FX20 DVK board is pulled to 1.8 V and I2C lines on EFINIX Ti180 FPGA DVK are pulled up to 3.3 V. So, remove R228 and R229 Pull up resistors on the EFINIX Ti180 FPGA DVK
5. To configure FPGA from EZ-USB™ FX20, remove R191 and R192 resistors and place on R193 and R194 respectively.
6. To control EFFINIX Ti180 FPGA reset from EZ-USB™ FX20, attach a pull down resistor to CRESET_N. Connect one end of resistor (10K-Ohm) to R192 resister pad and other end to Ground (GND).
7. To enable the camera connected to IMX477 daughter board, connect the 3.3 V supply pin of jumper J3 to pin6(EN_0) of jumper J1 as shown in Figure 35 and short pin1 – pin2 and pin3 – pin4 of Header J1, which connects the i2c signals from FPGA to camera sensor

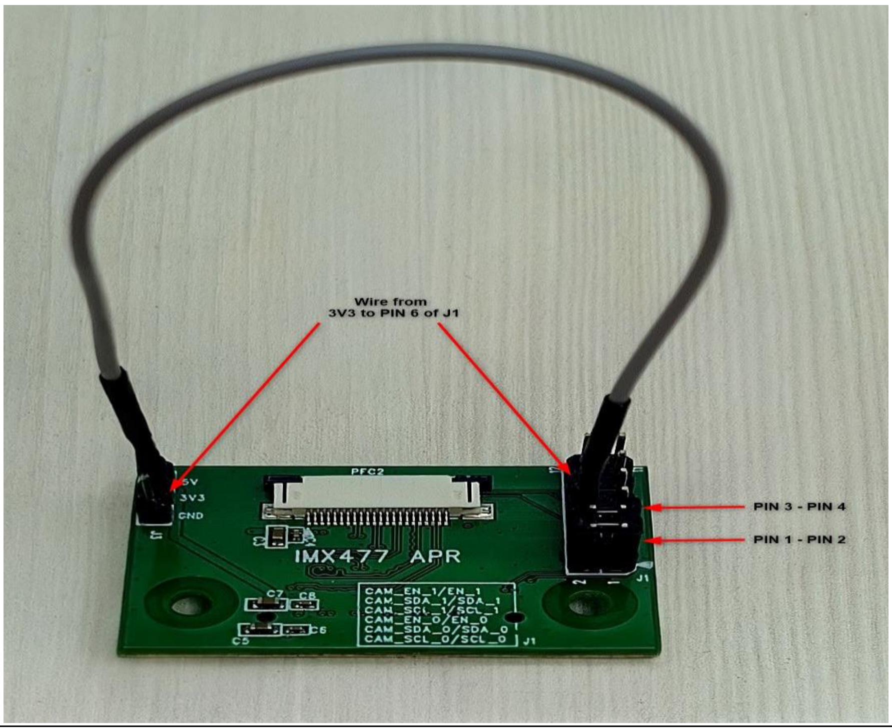


Figure 35 **IMX477 MIPI daughter board**

Note: *Follow the procedures provided in the EZ-USB™ FX20 DVK user manual to setup the EZ-USB™ FX20 DVK board. The daughter board mentioned in Figure 35 is supplied with EFINIX Ti180 DVK.*

Table 14 Jumper configuration in EZ-USB™ FX20 DVK

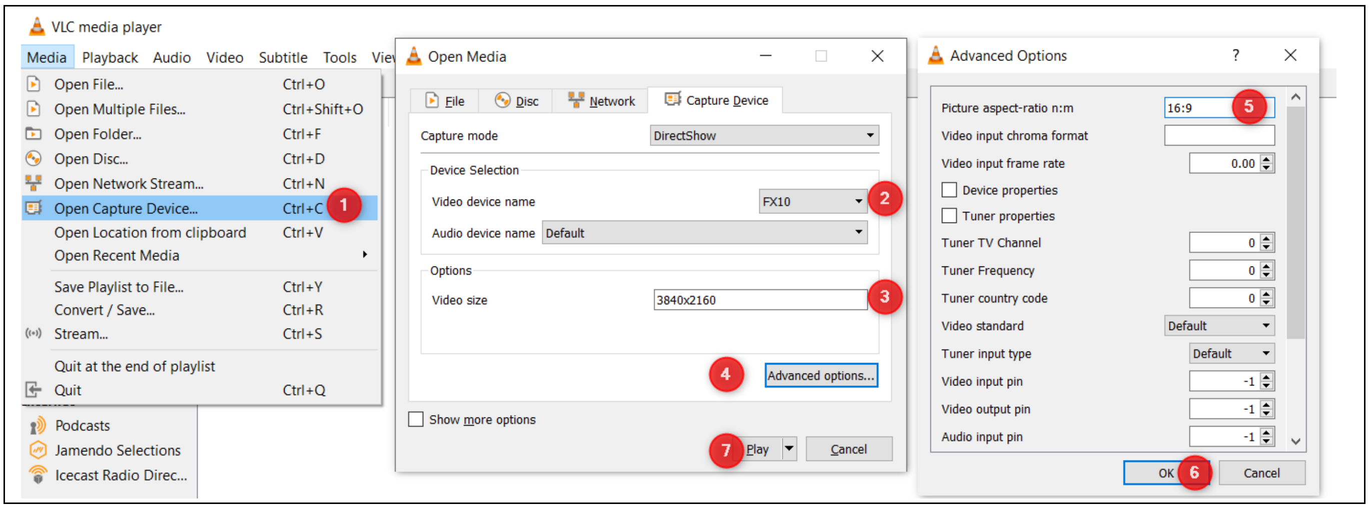
Jumper	LVCMOS	LVDS
J13 (VDDIO_P1)	1 – 2 (1.8 V)	2 - 3 (3.3 V)
J12 (VDDIO_P0)	1 – 2 (1.8 V)	2 – 3 (3.3 V)
J10 (VDDIO_CTRL)	1 -2 (1.8 V)	2 – 3 (3.3 V)

1. Make a board setup as explained in Section 11.7 with respect to Figure 34
2. Plug the USB 3.2 Gen 2x2 C - C cable provided with the EZ-USB™ FX20 DVK kit from host into EZ-USB™ FX20 DVK USB-C port
3. Load the firmware into the board using the EZ-USB™ Control Center application provided as part of the EZ-USB™ FX20 SDK. The firmware source and pre-built image is provided in *AN38689.zip*
4. Connect Sony IMX715 sensor to P2 connector of FPGA board and connect PDM microphones to J3 (MIC_RIGHT) and J2 (MIC_LEFT) in GPIO accessory board
5. Power the FPGA DVK board using the provided 12 V DC adapter and load the FPGA configuration file provided in the AN38689.zip using the Efinity FPGA programmer tool
6. Reset EZ-USB™ FX20 using Reset button (SW3) and observe the UVC and UAC device enumeration using Device Manager

11.8.1 Streaming video using VLC media player

Open the VLC media player application.

1. Go to **Media** > **Open Capture Device…**
2. Select **FX20** as Video device in Video device name
3. Provide resolution as **3840x2160** in Video size field box
4. Click on **Advance options…** and
5. Provide Picture aspect -ratio n:m as **16:9**
6. Click **OK** and then **Play**

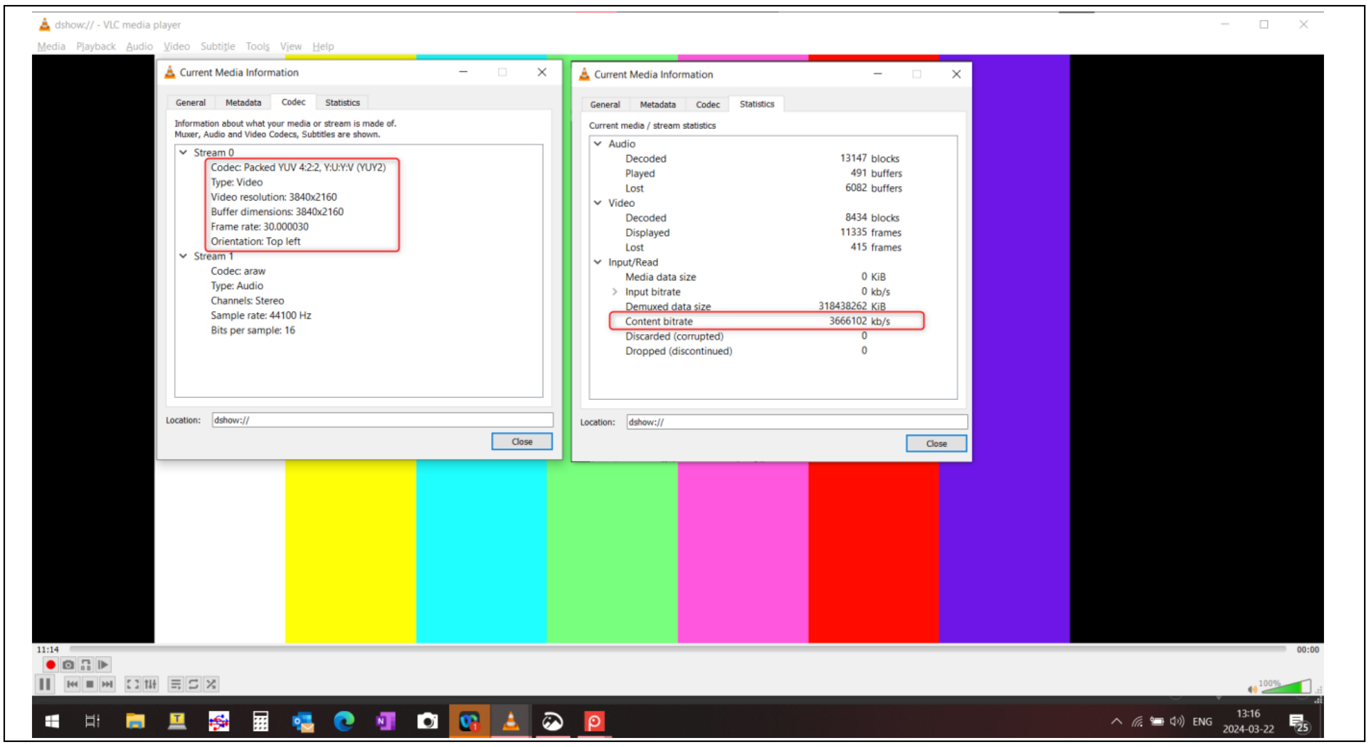


Capture mode: DirectShow
Video device name: FX10
Audio device name: Default
Video size: 3840x2160

Picture aspect-ratio n:m: 16:9
Video input chroma format:
Video input frame rate: 0.00
Tuner TV Channel: 0
Tuner Frequency: 0
Tuner country code: 0
Video standard: Default
Tuner input type: Default
Video input pin: -1
Video output pin: -1
Audio input pin: -1

Figure 36 VLC media player settings

7. Observe the video streaming, shown in Figure 37
8. To Display statistics (for example, Rendering Frame Rate) of a streaming image Go to **Tool** > **Codec Information**


dshow:// - VLC media player
Media Playback Audio Video Subtitle Tools View Help
Current Media Information
General Metadata Codec Statistics
Location: dshow://
Close
11:14
00:00
100%
ENG
13:16
2024-03-22

Information about what your media or stream is made of.
Muxer, Audio and Video Codecs, Subtitles are shown.

```
Stream 0
    Codec: Packed YUV 4:2:2, Y:U:Y:V (YUY2)
    Type: Video
    Video resolution: 3840x2160
    Buffer dimensions: 3840x2160
    Frame rate: 30.000030
    Orientation: Top left
Stream 1
    Codec: araw
    Type: Audio
    Channels: Stereo
    Sample rate: 44100 Hz
    Bits per sample: 16
```

Current media / stream statistics

```
Audio
    Decoded                      13147 blocks
    Played                         491 buffers
    Lost                          6082 buffers
Video
    Decoded                       8434 blocks
    Displayed                    11335 frames
    Lost                           415 frames
Input/Read
    Media data size                  0 KiB
  > Input bitrate                    0 kb/s
    Demuxed data size        318438262 KiB
    Content bitrate            3666102 kb/s
    Discarded (corrupted)            0
    Dropped (discontinued)           0
```

Figure 37 Enable display statistics

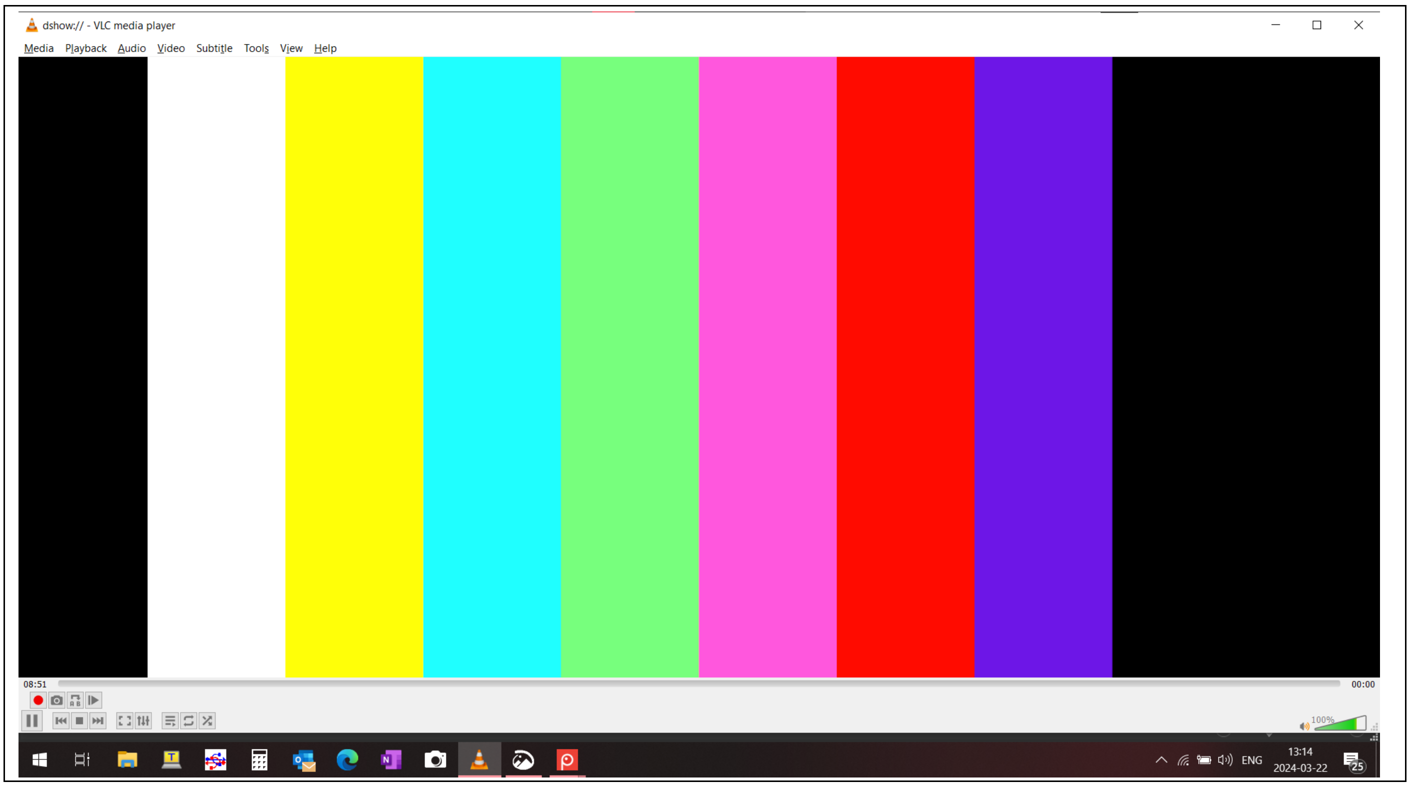

dshow:// - VLC media player
Media Playback Audio Video Subtitle Tools View Help
08:51
00:00
100%
ENG
13:14
2024-03-22

Figure 38 Streaming colorbar generated in EFINIX Ti180 FPGA

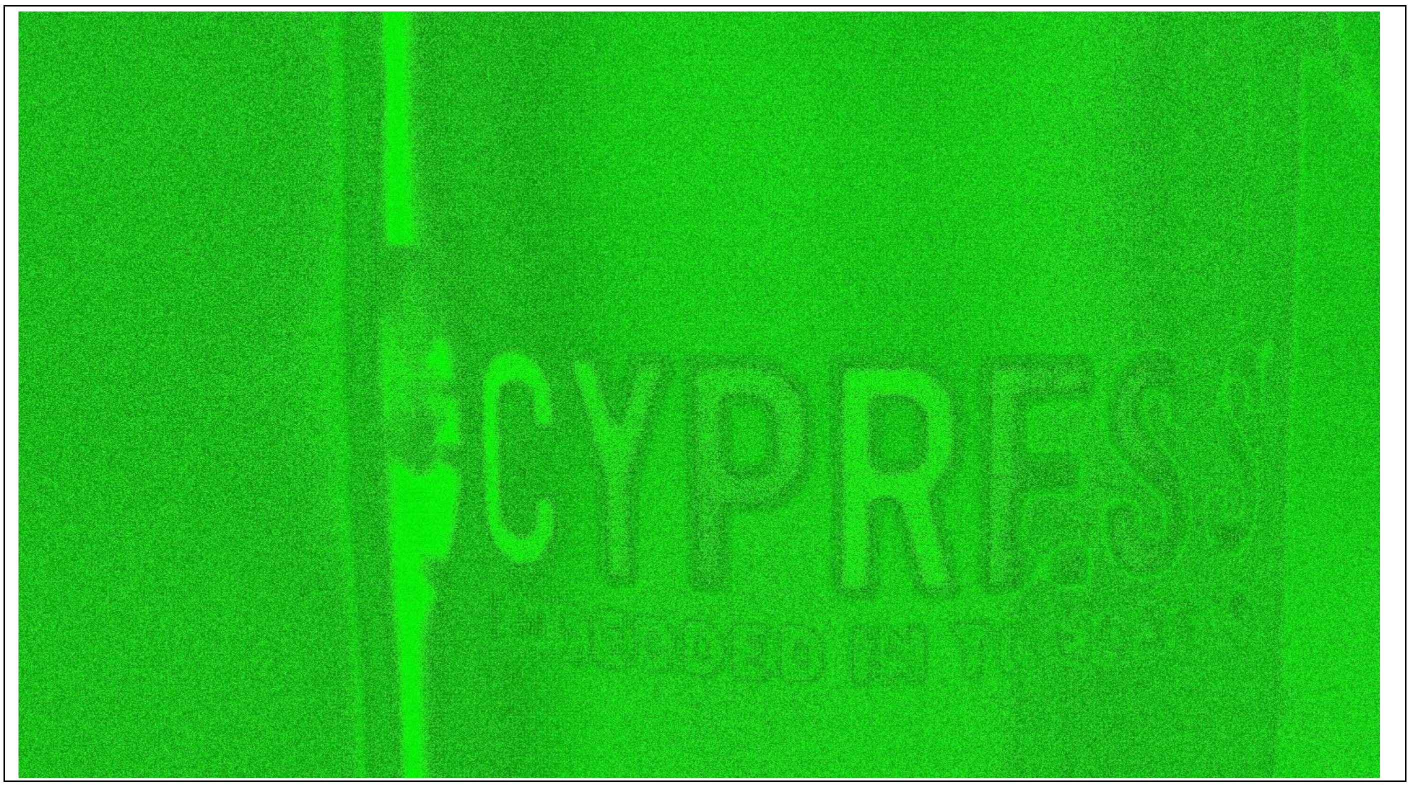
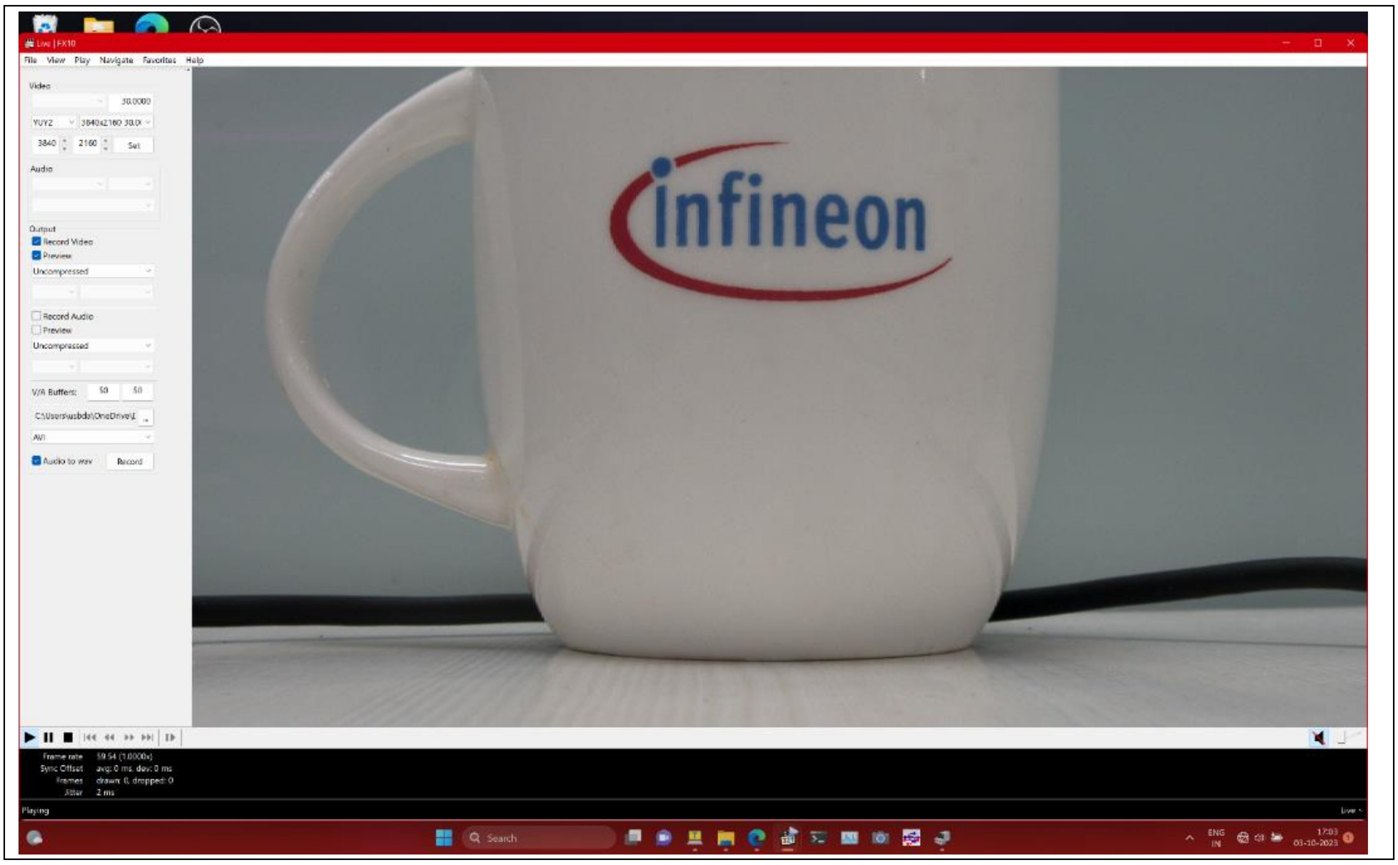


Figure 39 Streaming RAW image (Without ISP) from Sony IMX715 sensor



Video
30.0000
YUY2 3840x2160 30.0
3840 2160 Set
Audio
Output
[✓] Record Video
[✓] Preview
Uncompressed
[ ] Record Audio
[ ] Preview
Uncompressed
V/A Buffers: 50 50
C:\Users\usbdx\OneDrive\
AVI
[✓] Audio to wav Record

```
Frame rate    59.54 (1.0000x)
Sync Offset   avg: 0 ms, dev: 0 ms
     Frames   drawn: 0, dropped: 0
     Jitter   2 ms
Playing
```

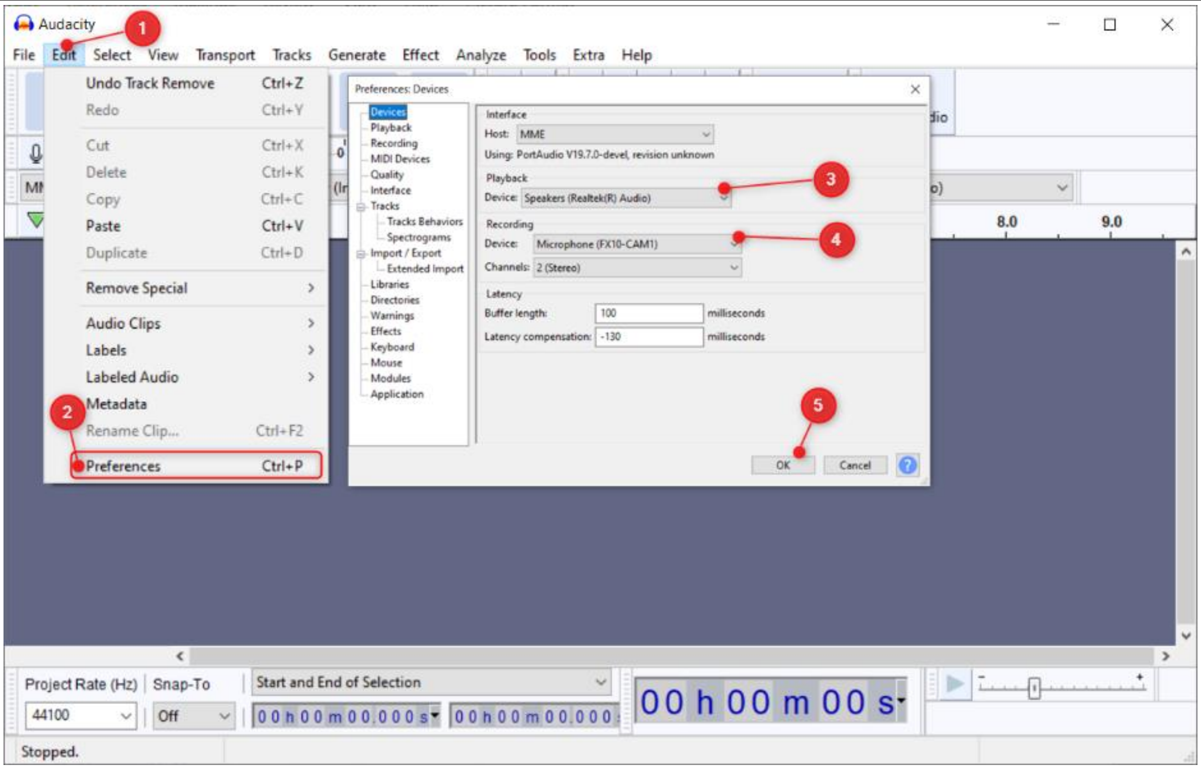
Figure 40 Streaming Processed Image (with ISP) from Sony IMX715 sensor

Note: *CIS ISP can be implemented on EFINIX Ti180 FPGA. Contact CIS Corp Japan for more details.*

11.8.2 PDM microphone add-on board support

To understand the functionality of UAC and Record the audio using the Audacity application, follow these instructions:

1. Open Audacity application
2. Go to **Transport** > **Rescan Audio Device**. This will help to rescan and list the devices if not listed
3. Go to **Edit** -> Click on **Preferences**
4. In the **Device** option, select the playback Device as **Speaker** and then select the Recording Device as **FX20** and click **OK**



Interface
Host: MME
Using: PortAudio V19.7.0-devel, revision unknown
Playback
Device: Speakers (Realtek(R) Audio)
Recording
Device: Microphone (FX10-CAM1)
Channels: 2 (Stereo)
Latency
Buffer length: 100 milliseconds
Latency compensation: -130 milliseconds

Project Rate (Hz): 44100
Snap-To: Off
00 h 00 m 00.000 s
00 h 00 m 00 s

Figure 41 **Device selection in Audacity**

5. Click **Start** button to start recording, it will record the audio from the microphone connected to EZ-USB™ FX20 GPIO accessory board
6. To play recorded audio, first **Stop** the recording then click on the **Play** button

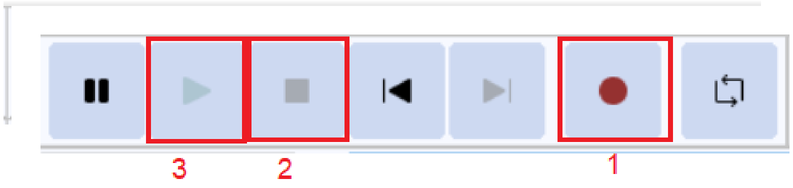


Figure 42 **Start, stop, and play record**

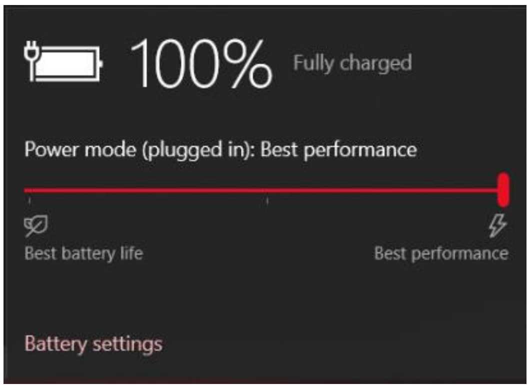
12 Troubleshooting

12.1 Black screen viewed in host application

- Connect the FS USB cable to J3 of the EZ-USB™ FX20 DVK and open the debug terminal (Tera Term). Check the resolution selected in the host camera application and verify it with the debug prints
- Switch to a different resolution and verify the I2C communication between EZ-USB™ FX20 and FPGA with the help of debug prints
- Use Wireshark software USB analyzer and check whether the USB data is being transmitted to the host.
- Check Link and PHY training status printed in the debug terminal

12.2 Unable to get 4K video streaming at 60 fps (FX10) or 120 fps (FX20)

- Check whether EZ-USB™ FX20 is enumerating as USB 3.2 Gen 2x2 (Speed 6), if it enumerates as USB 3.2 Gen1 or Gen 2x1 then it will not stream 4K at 120 fps as USB 3.2 Gen 2x1 is not capable of handling 20 Gbps
- Check whether EZ-USB™ FX10 is enumerating as USB 3.2 Gen 2 (Speed 5), if it enumerates as USB 3.2 Gen1 then it will not stream 4K at 60 fps as USB 3.2 Gen1 is not capable of handling 10 Gbps
- If the system is running multiple processes or programs that require a lot of resources, the VLC media player may not be able to keep up with the input fps
- Some UVC host application may be unable to display full bandwidth data, resulting in reduced frame rate, close unwanted applications, and use VLC media player to improve the frame rate performance
- The Host PC may not be powerful enough to handle the video input stream at a high frame rate. This can be due to outdated hardware or insufficient resources such as RAM, CPU, or GPU
- See section 5.1.1 minimal PC requirement
- If the system is not properly ventilated or has a cooling issue, it can overheat, leading to performance problems such as reduced fps due to thermal throttling
- Connect the AC adapter to your laptop and choose the ‘best performance’ option


100% Fully charged

Power mode (plugged in): Best performance

Figure 43 Choose best performance

- If the PC supports an additional Graphic card, choosing the High performance can help you achieve 4K resolution at 60 fps.
  To do this, Go to **Settings** then navigate to **System, Display**, and scroll down to select **Graphic Settings**. Click on **Browse**, choose the player application (VLC media player) and click on **Add**. Once the application is added, click on **Option,** and select **‘High Performance’** as shown in Figure 44

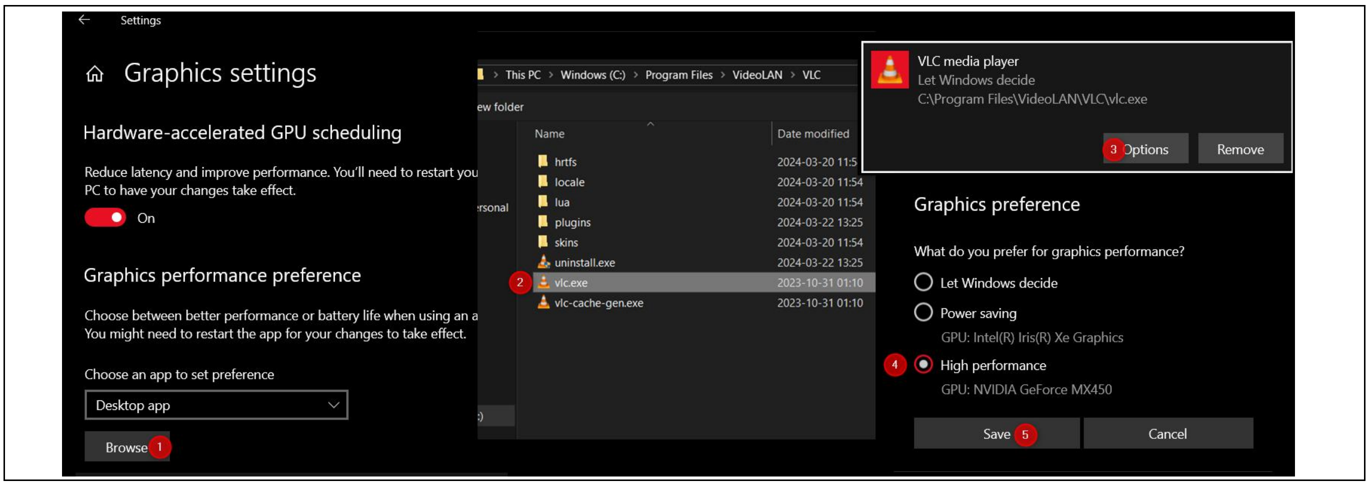

Settings
Graphics settings
Hardware-accelerated GPU scheduling
Reduce latency and improve performance. You'll need to restart your PC to have your changes take effect.
On
Graphics performance preference
Choose between better performance or battery life when using an app. You might need to restart the app for your changes to take effect.
Choose an app to set preference
Desktop app
Browse 1
This PC > Windows (C:) > Program Files > VideoLAN > VLC
Name
Date modified
hrtfs
locale
lua
plugins
skins
uninstall.exe
2 vlc.exe
vlc-cache-gen.exe
VLC media player
Let Windows decide
C:\Program Files\VideoLAN\VLC\vlc.exe
3 Options
Remove
Graphics preference
What do you prefer for graphics performance?
Let Windows decide
Power saving
GPU: Intel(R) Iris(R) Xe Graphics
4 High performance
GPU: NVIDIA GeForce MX450
Save 5
Cancel

Figure 44 Choosing High performance video streaming application (VLC media player)

12.3 Video freeze during streaming

- Open the debug terminal and observe what is printing, if it is printing DMA Reset Timeout that mean FPGA is not pushing the data to EZ-USB™ FX20
- Try streaming again by closing and re-opening the UVC camera application
- Try resetting the EZ-USB™ FX20 using the reset button

12.4 No audio output

- Check A**UDIO_IF_EN is set to "1" or not in Makefile.**
- **Stereo audio input from two PDM microphones can be enabled by setting STEREO_ENABLE=1 to CFLAGS in Makefile.**
- **In case, PDM microphones are connected to EZ-USB™ FX20, and microphone as a loopback to the speaker then follow the steps mentioned below to configure.**
- **STEP 1:** Open the Windows **Sounds** app by right-clicking on the speaker icon on the taskbar and click on **Sounds**

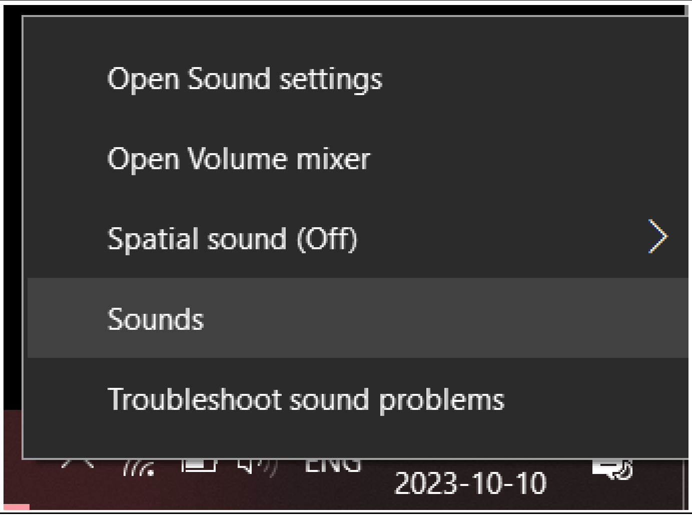
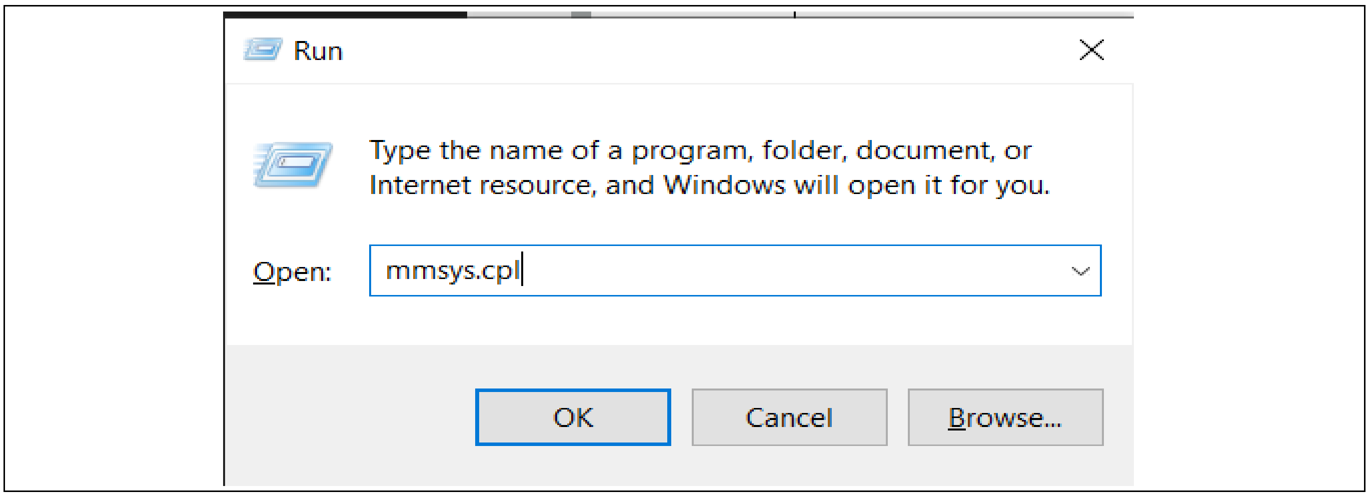


Figure 45 Right click on the speaker icon and select Sounds

You can also open the Sounds app by using the Run window. Open Run Window using Windows Key + R and run **mmsys.cpl**.

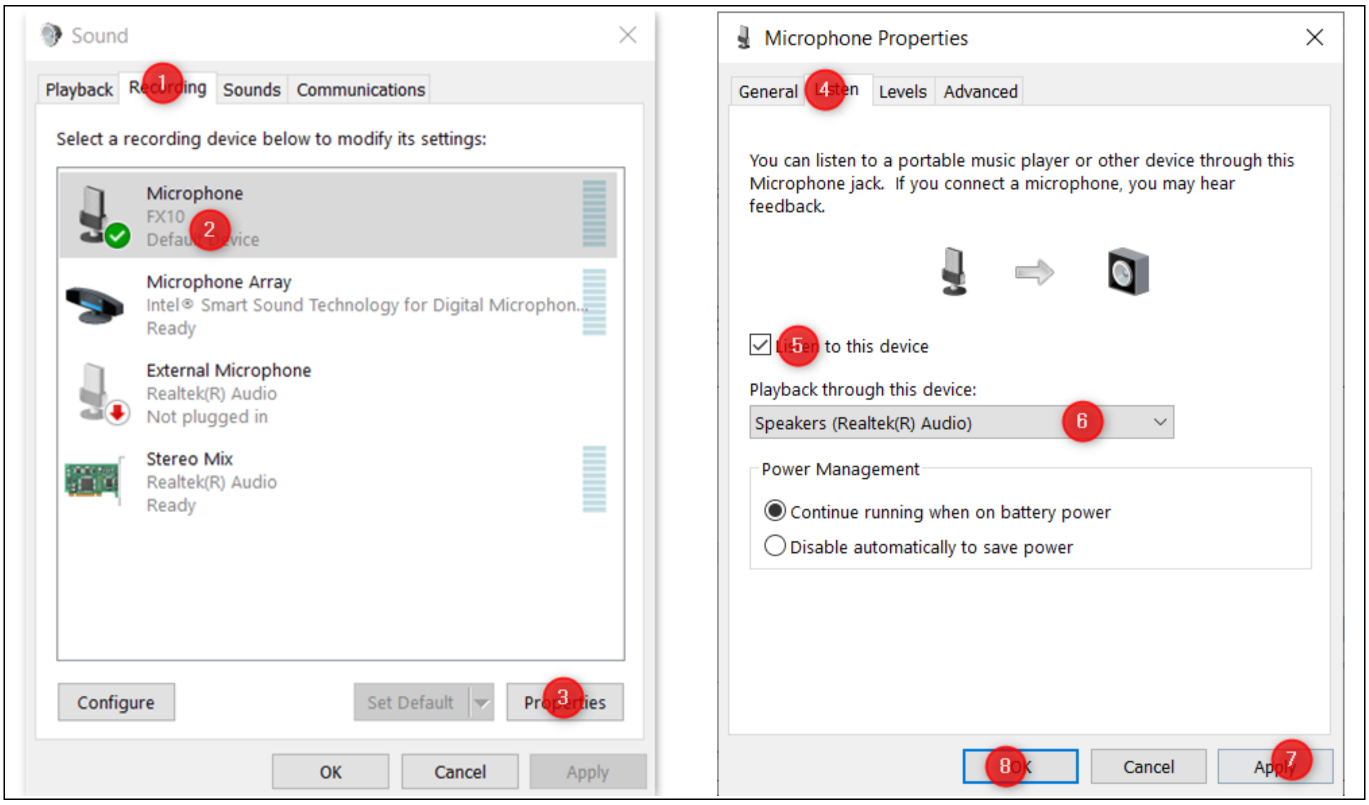

Run

Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.

Open: mmsys.cpl

Figure 46 **Run mmsys.cpl**

STEP 2: Follow the steps as shown in Figure 47.


Sound

Select a recording device below to modify its settings:

Microphone
FX10
Default Device

Microphone Array
Intel® Smart Sound Technology for Digital Microphon...
Ready

External Microphone
Realtek(R) Audio
Not plugged in

Stereo Mix
Realtek(R) Audio
Ready

Microphone Properties

You can listen to a portable music player or other device through this Microphone jack. If you connect a microphone, you may hear feedback.

[✓] Listen to this device

Playback through this device: Speakers (Realtek(R) Audio)

Power Management
[✓ Continue running when on battery power]
Disable automatically to save power

Figure 47 **Listen to Device option to stream audio**

- Use the **Listen to Device** option to stream audio from the EZ-USB™ FX20 Demo Device

12.5 Discontinuous audio output

- If you are sending the audio data over the GPIF, it is recommended to use I2S if the data is from an audio source. If the input is from a microphone, suggesting using the PDM interface
- If the frame rate (fps) is less than 30 fps, you may experience discontinuous audio in case audio and video data are sent over GPIF.
- Use stereo interface for better audio experience.

13 Associated project files

13.1 EZ-USB™ FX20 firmware

EZ-USB™ FX20 UVC + UAC composite firmware used in this application note is available on GitHub.

13.2 FPGA projects

The source code and FPGA project referenced in this application note can be found in the *fpga* folder. The FPGA project supports LVCMOS SDR WideLink, LVCMOS DDR WideLink, and LVDS WideLink mode available on the GitHub.

14 Summary

You are now able to develop a UVC and UAC composite USB device using EZ-USB™ FX20 and interface a camera/image sensor with an FPGA and stream video and audio data over the USB 3.2 Gen 2x2 port with PC.

References

For a complete and updated list of EZ-USB™ FX20 code examples, visit our GitHub repository. For more EZ-USB™ FX20 related documents, visit the EZ-USB™ FX20 webpage.

Application note

[1] Infineon Technologies AG: AN237841 – Getting started with EZ-USB™ FX20/FX10/FX5N/FX5

[2] Infineon Technologies AG: *AN75779 - How to implement an image sensor interface using EZ-USB™ FX3 in a USB Video Class (UVC) framework;* Available online

Datasheet

[3] Infineon Technologies AG: 002-33684: CYUSB402x EZ-USB™ FX20 USB 20 Gbps peripheral controller

[4] Infineon Technologies AG: 002-40837: CYUSB401x EZ-USB™ FX10 USB 10 Gbps peripheral controller

[5] Infineon Technologies AG: 002-41019: CYUSB328x EZ-USB™ FX5N USB 10 Gbps peripheral controller

[6] Infineon Technologies AG: 002-40850: CYUSB308x EZ-USB™ FX5 USB 5 Gbps peripheral controller

Reference manual

[7] Infineon Technologies AG: 002-36077: EZ-USB™ FX20 USB 3.2 Gen 2x2 device controller architecture reference manual

Programming specifications

[8] Infineon Technologies AG: 002-37666: CYUSB401x, CYUSB308x EZ-USB™ FX20 programming specifications

Development kits

[9] Infineon Technologies AG: KIT_FX20_FMC_001: EZ-USB™ FX20 Development Kit

[10] Infineon Technologies AG: KIT_FX10_FMC_001: EZ-USB™ FX10 Development Kit

[11] Infineon Technologies AG: KIT_FX5N_FMC_001: EZ-USB™ FX5N Development Kit

[12] Infineon Technologies AG: KIT_FX5_FMC_001: EZ-USB™ FX5 Development Kit

[13] Infineon Technologies AG: DEMO_FX20_MIPI_001: EZ-USB™ FX20 MIPI Camera Demo Kit

Tools

[14] Infineon Technologies AG: *Eclipse IDE for ModusToolbox™*; Available online

[15] Infineon Technologies AG: *EZ-USB™ FX Control Center*; Available online

Product webpages

[16] Infineon Technologies AG: *EZ-USB™ FX20 webpage*; Available online

[17] Infineon Technologies AG: *EZ-USB™ FX10/FX5N webpage*; Available online

[18] Infineon Technologies AG: *EZ-USB™ FX5 webpage*; Available online

Revision history

Document revision	Date	Description of changes
**	2025-04-08	Initial release

Edition 2025-04-08

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

002-38689 Rev. **