AN75779

How to Implement an Image Sensor Interface Using EZ-USB® FX3™ in a USB Video Class (UVC) Framework

Author: Karnik Shah

Associated Project: Yes
Software Version: FX3 SDK1.2.3
Related Application Notes: AN75705

The high bandwidth provided by USB 3.0 puts high demands on ICs that connect peripherals to USB. This application note focuses on a popular USB 3.0 application: a camera (image sensor interfaced with EZ-USB® FX3™) streaming uncompressed data into a PC. The application note highlights the FX3 features specifically designed to maximize data throughput without sacrificing interface flexibility. This application note also gives implementation details about the USB Video Class (UVC). Conforming to this class allows the camera device to operate using built-in PC drivers and Host applications, such as AMCap and VLC Media Player. Finally, the application note shows you how to use the flexible image sensor interface in FX3 to connect two image sensors in order to implement 3-D imaging and motion-tracking applications.

Table of Contents

1. Introduction

The high bandwidth provided by USB 3.0 puts high demands on ICs that connect peripherals to USB. A popular example is a camera streaming uncompressed data into a PC. In this application note, a converter that connects to the image sensor on one side and to a USB 3.0 Host PC on the other side is implemented using the Cypress EZ-USB® FX3™ chip. FX3 uses its General Programmable Interface, Gen 2 (GPIF II), to provide the image sensor interface, and its SuperSpeed USB unit to connect to the PC. The FX3 firmware converts the data coming from the image sensor into a format compatible with the USB Video Class (UVC). Conforming to this class allows the camera to operate using built-in OS drivers, making the camera compatible with Host applications, such as AMCap and VLC Media Player.

Figure 1. The Camera Application

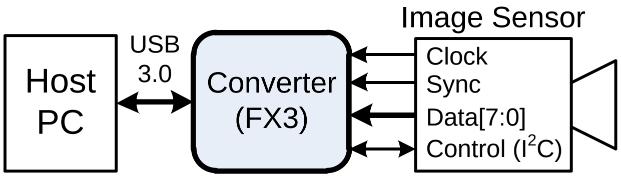


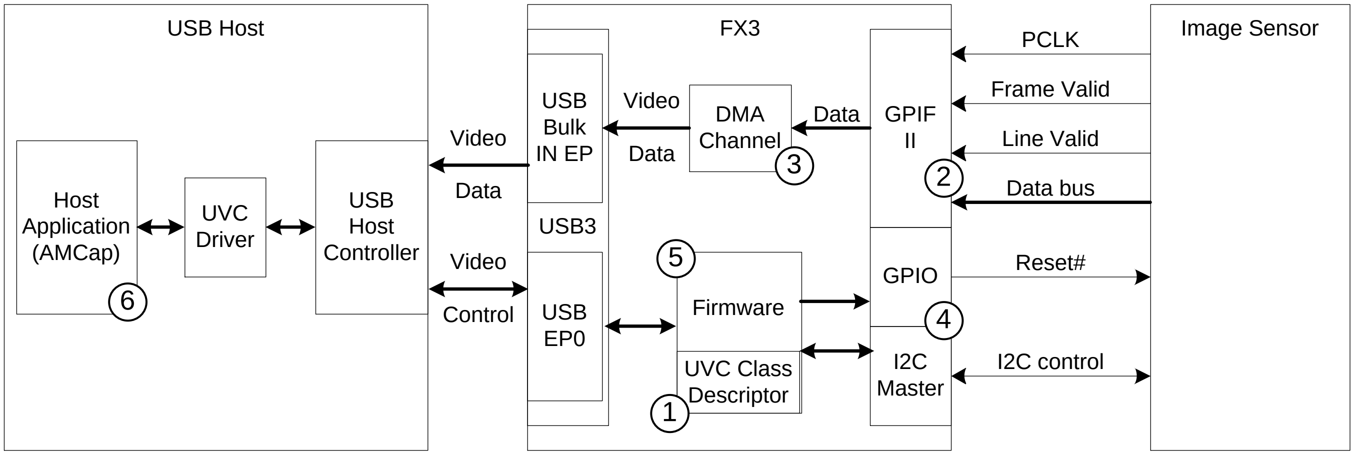
Figure 1 illustrates the camera application. On the left side is a PC equipped with a SuperSpeed USB 3.0 port. On the right side is an image sensor with the following features:

- 8-bit synchronous parallel data interface
- 16 bits per pixel
- YUY2 color space
- 1280 x 720-pixel resolution (720p)
- 30 frames per second
- Active high frame/line valid signals
- Positive clock edge polarity

Although this application note discusses a specific image sensor interface, these features are common to many image sensor interfaces with slight variations, such as data bus width and signal polarities. As a result of the programmable nature of the GPIF II block, these variations can easily be accommodated. In addition, FX3 uses its I^{2}C interface to implement a control bus for image sensor configuration.

Figure 2 is a more detailed system block diagram. The main sub-blocks of the block diagram are numbered, and the tasks that are executed by each sub-block are described below:

Figure 2. System Block Diagram



1. Provide the proper USB Descriptors so that the Host recognizes the peripheral as a device conforming to the UVC. For details, refer to Section 2.
2. Implement a parallel-bus interface to the image sensor. This is done using the FX3 GPIF II interface. A Cypress tool called GPIF II Designer allows custom waveforms to be designed in a graphical state machine editor. Because the interface is programmable, minor edits can customize it for a different image sensor, for example one with a 16-bit data bus. For details, refer to Section 3.
3. Construct a DMA channel that moves the image sensor data from the GPIF II block to the USB Interface Block (UIB). In this application, header data must be added to the image sensor's video data to conform to the UVC specification. As such, the DMA is configured to enable the CPU to add the required header to the DMA buffers. This channel must be designed so that maximum bandwidth can be used to

stream video from the image sensor to the PC. For details, refer to Section 4.

4. Use the FX3 I^{2}C master to implement a control bus to the image sensor. The I^{2}C and GPIO units are programmed using standard Cypress library calls, and they are mentioned in Section 5.4.
5. The FX3 firmware initializes the hardware blocks of FX3 (Section 5.2), enumerates as a UVC camera (Section 2.3.1), handles UVC-specific requests (Section 2.3.2), translates video control settings (such as brightness) to the image sensor over the I^{2}C interface (Section 5.4), adds a UVC header to the video data stream (Section 2.3.4), commits the video data with headers to USB (Section 5.7), and maintains the GPIF II state machine (Section 5.8).
6. A Host application, such as the AMCap or VLC Media Player, accesses the UVC driver to configure the image sensor over the UVC control interface and to receive video data over the UVC streaming interface. For details on UVC-based Host applications, refer to Section 7.

If the camera is plugged into a USB 2.0 port, the FX3 firmware uses the I2C control bus to reduce the frame rate from 30 FPS to 15 FPS and the frame size from 1280 x 720 pixels to 640 x 480 pixels to accommodate the lower USB bandwidth. The PC Host optionally can use the control interface to send brightness, pan, tilt, and zoom adjustments to the camera. Pan, tilt, and zoom are usually implemented together and are referred to as PTZ.

2. The USB Video Class (UVC)

Conforming to the UVC requires two FX3 code modules:

1. Enumeration Data
2. Operational Code

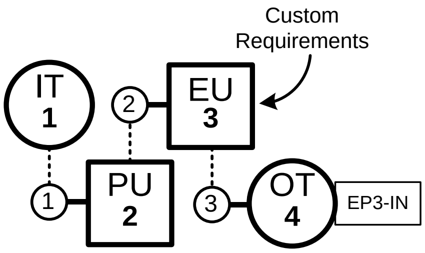
2.1 Enumeration Data

The code attached to this application note includes a file, named **cyfxuvcdscr.c** (explained in section 2.3.1), that contains the UVC enumeration data. The USB specification, which defines the format for UVC Descriptors, is available at usb.org. This section gives a high-level view of the Descriptors. A UVC device has four logical elements:

1. Input Camera Terminal (IT)
2. Output Terminal (OT)
3. Processing Unit (PU)
4. Extension Unit (EU)

The elements connect 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 3.

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, and PTZ.

2.2 Operational Code

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 FX3 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.

2.3 USB Video Class Requirements

The firmware project for this application note is in the folder named **USBVideoClass**. This section explains how the UVC requirements are satisfied by the example project. The 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 are found in the UVC specification.

2.3.1 USB Descriptors for UVC

The **cyfxuvcdscr.c** file contains the USB Descriptor tables. The byte arrays "CyFxUSBHSConfigDscr" (Hi-Speed) and "CyFxUSBSSConfigDscr" (SuperSpeed) 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.

2.3.1.1 Video Control Interface

The VC Interface Descriptor and its sub-Descriptors report all of 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 bit fields 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 the Descriptor is included as a placeholder for custom features. Note that if the Extension Unit is utilized, the standard Host application will not see its features unless the Host application is modified to recognize them.

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 bit field "bmControls" (**cyfxuvcdscr.c**) of the respective terminal/unit descriptor by setting 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 **UVCAppEP0Thread_Entry** function in the **uvc.c** file.

2.3.1.2 Video Streaming 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 a single resolution and frame rate. Its image characteristics are contained in three Descriptors, as shown in the following three tables (only relevant byte offsets are shown).

Table 1. VS Format Descriptor Values

VS Format Descriptor Byte Offset	Characteristic	SuperSpeed Value	Hi-Speed Value
23-24	Width to Height ratio	16:9	4:3

Table 2. VS Frame Descriptor Values

VS Frame Descriptor Byte Offset	Characteristic	SuperSpeed Value	Hi-Speed Value
5-8	Resolution (W,H)	1280x720	640x480
17-20	Maximum image size in bytes	1280x720x2 (2 bytes/ pixel)	640x480x2 (2 bytes/ 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 100ns units	0x51615 (30 FPS)	0xA2C2A (15 FPS)
18-21	Maximum image size in bytes	1280x720x2 (2 bytes/ pixel)	640x480x2 (2 bytes/ pixel)

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

2.3.2 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 value
- 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
- 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.

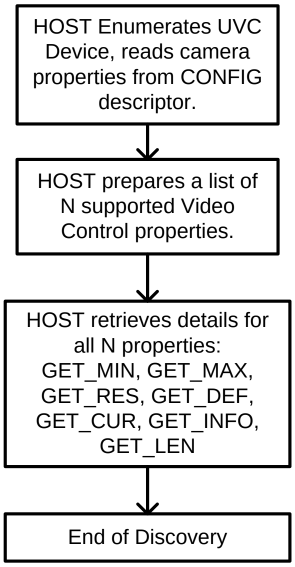
There are byte fields in these requests that qualify their target capability. These byte fields have a hierarchy, which follows the same structure as the UVC-specific Descriptors described in Section 2.3.1. The first level identifies the interface (video control or video streaming).

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:

- the first level = video control
- the second level = processing unit
- the 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 6. The following three flowcharts show how the Host interacts with a UVC device.

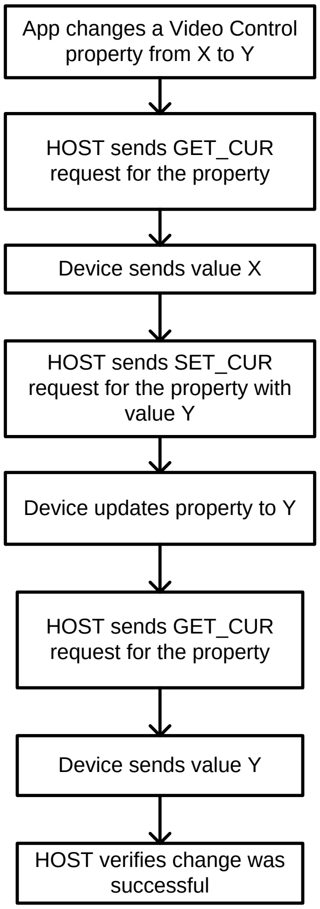
Figure 4. The UVC Enumeration and Discovery Flow


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

When the UVC device is plugged into USB, the Host enumerates it and discovers details about the properties supported by the camera (Figure 4).

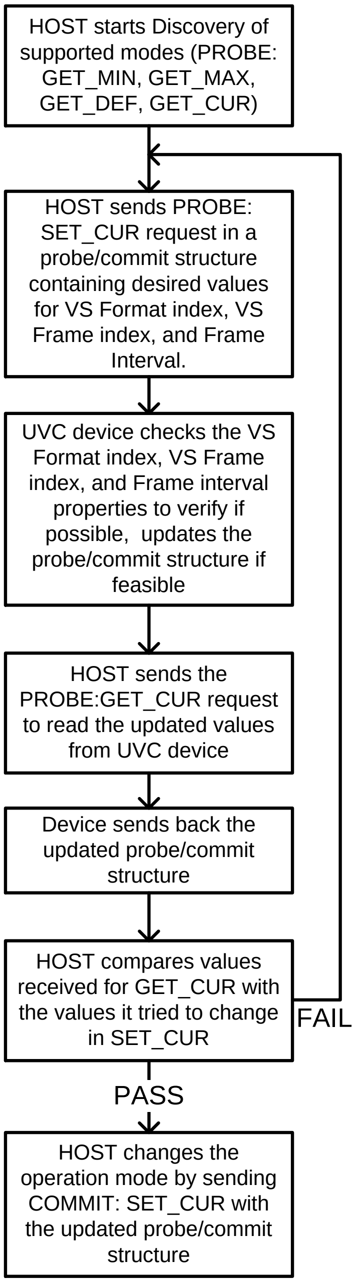
During a video operation, a camera operator may change a camera property, such as brightness, in a display dialog presented by the Host application. Figure 5 shows this interaction.

Figure 5. A Host Application Changes a Camera Setting


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

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. EndpointAt this point, the UVC driver is ready to stream video from the UVC device.

Figure 6. Host-Camera Pre-Streaming Dialog


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

2.3.2.1 Control Requests – Brightness and PTZ Control

Brightness and PTZ controls are implemented in the associated project. PTZ can be turned on by defining "UVC_PTZ_SUPPORT" in the **uvc.h** file. These capabilities may or may not be supported by the image sensor. If they are not supported, specific hardware has to be designed to implement them. In any case, the firmware implementation on the USB side of the controls for these capabilities remains the same. However, the image sensor implementation would differ. Therefore, only placeholder functions are provided that implement the USB side of these controls. You have to write the code for the image sensor-specific PTZ implementation.

Note All functions described hereafter in the application note are implemented in the **uvc.c** file unless specified otherwise. This file is a part of the project stored under the **USBVideoClass** folder in the zip 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 **CyFxUVCApplnUSBSetupCB** function. This function detects whether the Host has sent a UVC-specific request (control or stream) and then sets an event flag:

"CY_FX_UVC_VIDEO_CONTROL_REQUEST_EVENT" or "CY_FX_UVC_VIDEO_STREAM_REQUEST_EVENT", respectively. Then, the flag is processed by function **UVCAppEP0Thread_Entry** (EP0 application thread).

Brightness control requests will trigger the video control request event flag. The EP0 application thread, which is waiting for any of these flags to trigger, decodes the video control request and calls the appropriate function. The EP0 application thread calls the **UVCHandleProcessingUnitRqts** function to handle brightness control requests.

Modify the **UVCHandleProcessingUnitRqts** function to implement processing unit-related controls (brightness, contrast, hue, and so on). Modify the **UVCHandleCameraTerminalRqts** function to implement camera terminal controls. Modify the **UVCHandleExtensionUnitRqts** function to implement any vendor-specific requests. To enable support for any of these controls, you must set corresponding bits in the USB Descriptors. The UVC specification includes details on Camera Terminal, Processing Unit, and Extension Unit USB Descriptors.

2.3.2.2 Streaming Requests – Probe and Commit Control

UVCHandleVideoStreamingRqts handles streaming-related requests. When the UVC driver needs to stream video from a UVC device, the first step is negotiation. In that phase, the driver sends PROBE requests, such as GET_MIN, GET_MAX, GET_RES, and GET_DEF. In response, the FX3 firmware returns a PROBE structure. The structure 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_FX_UVC_PROBE_CTRL" handles the negotiation phase of the streaming for either a USB 2.0 or USB 3.0 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 3.0. These values would differ if multiple streaming modes need to be supported.

The switch case for "CY_FX_UVC_COMMIT_CTRL" handles the start of the streaming phase. The SET_CUR request for COMMIT control indicates that the Host will start streaming video next. Therefore, SET_CUR for COMMIT control sets the "CY_FX_UVC_STREAM_EVENT" event, which indicates the main application thread **UVCAppThread_Entry** to start the GPIF II state machine for video streaming.

2.3.3 Video Data Format: YUY2

The UVC specification supports only a subset of color formats for video data. Therefore, you should choose an image sensor 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)

Refer to Wikipedia for additional information on color formats.

Note: The RGB format is not supported. 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.

2.3.4 UVC Video Data Header

The UVC class requires a 12-byte header for uncompressed video payloads. The header describes the properties of the image data being transferred. For example, it contains a "new frame" bit that the image sensor controller (FX3) toggles for every frame. The FX3 code also can set an error bit in the header to indicate a problem in streaming the current frame. This UVC data header is required for every USB transfer. Refer to the UVC specification for additional details.

Table 4 shows the format of the UVC video data header.

Table 4. UVC Video Data Header Format

Byte Offset	Field Name	Description
0	HLF	Header Length Field specifies the length of the header in bytes
1	BFH	Bit Field Header indicates type of the image data, status of the video stream and presence or absence of other fields
2-5	PTS	Presentation Time Stamp indicates the source clock time in native device clock units
6-11	SCR	Source Clock Reference indicates system time clock and USB Start-Of-Frame (SOF) token counter

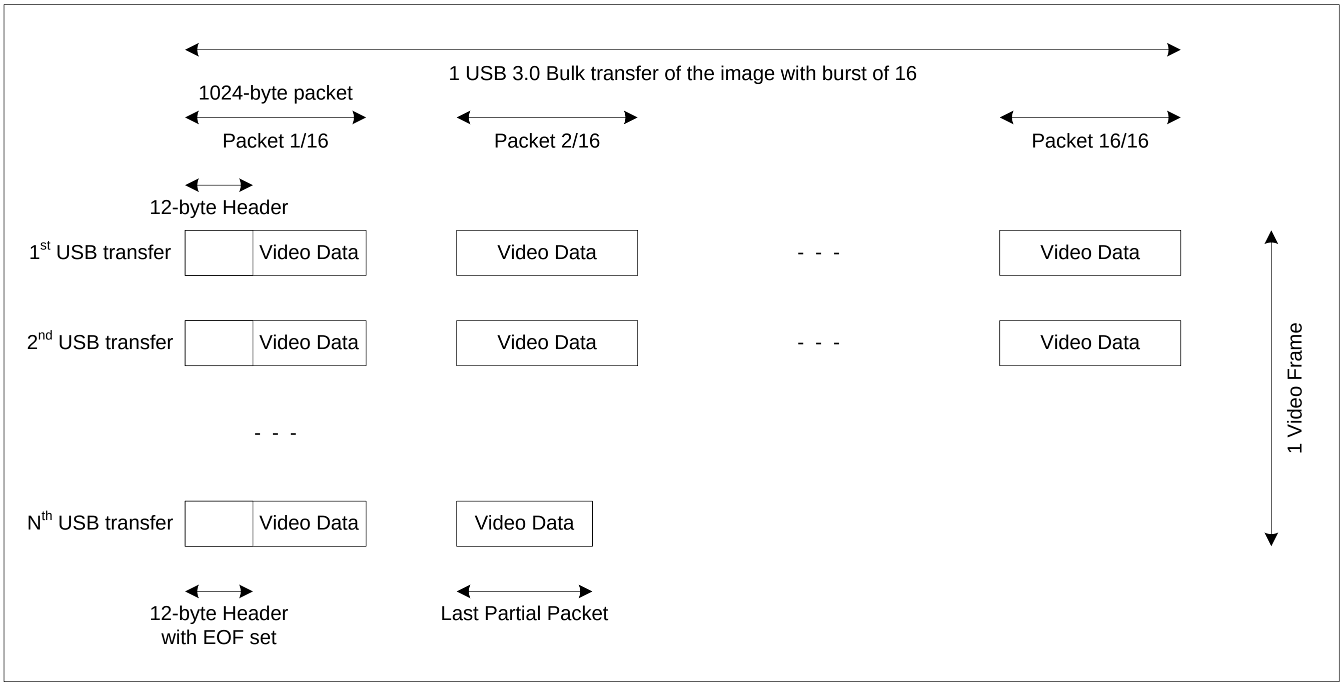
The value for the HLF is always 12. The PTS and the SCR fields are optional. The firmware example populates zeros in these fields. The Bit Field Header (BFH) keeps changing value at the end of a frame. Table 5 shows the format of the BFH that is a part of the UVC video data header.

Table 5. Bit Field 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

Figure 7 shows how these headers are added to the video data in this application. The 12-byte header is added for every USB bulk transfer. Here, each USB transfer has a total of 16 bulk packets. The USB 3.0 bulk packet size is 1024 bytes.

Figure 7. UVC Video Data Transfers



3. GPIF II Image Sensor Interface

The GPIF II block of FX3 is a flexible state machine that can be customized to drive the FX3 pins to interface with external hardware, such as an image sensor. To design a state machine, you need to understand the interface requirements and FX3's DMA capabilities.

3.1 Image Sensor Interface

A typical image sensor interface is shown in Figure 2. Usually the image sensor requires a reset signal from the FX3 controller. This can be handled by using an FX3 General-Purpose Input/Output (GPIO).

Image sensors typically use an I^{2}C connection to allow a controller to read and write the image sensor parameters. Image sensors may also use the Serial Peripheral Interface (SPI) or the Universal Asynchronous Receiver/Transmitter (UART) connection for the same purpose. The I^{2}C, SPI, and UART blocks of FX3 can provide this function. This application uses I^{2}C to configure the image sensor.

To transfer images, the image sensor supplies the following signals:

1. FV: Frame Valid (indicates start and stop of a frame)
2. LV: Line Valid (indicates start and stop of a line)
3. PCLK: Pixel clock (clock for the synchronous interface)
4. Data: Eight data lines for image data

Figure 8 shows the timing diagrams for the FV, LV, PCLK, and Data signals. The image sensor asserts the FV signal to indicate the start of a frame. Then, the image data is transferred line by line. The LV signal is asserted during each line transfer as the image sensor drives 8-bit pixel data in the YUY2 format, which comprises four bytes for every two pixels. Byte data is clocked into the GPIF II unit on each rising edge of PCLK.

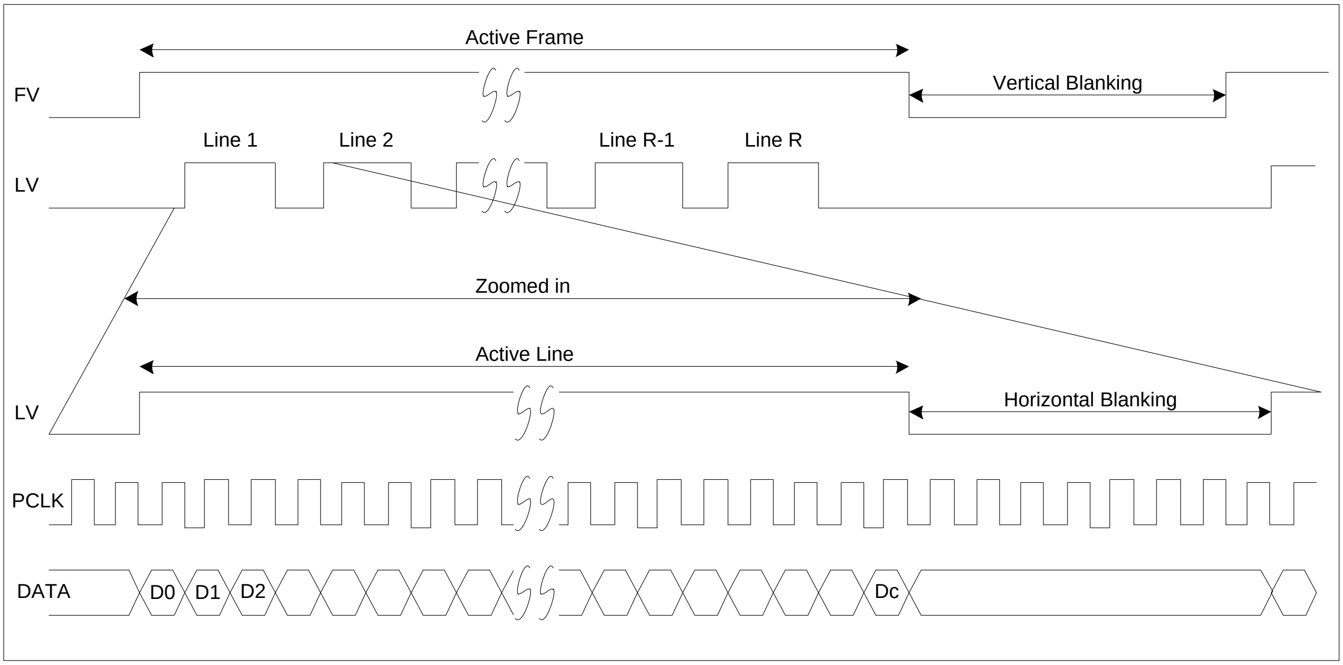
The FX3 GPIF II bus can be configured for 8-, 16-, or 32-bit data buses. This application uses the image sensor's 8-bit bus. If an image sensor supplies a non-byte-aligned bus, for example a 12-bit bus, use the next size up and either pad or discard the unused bits.

3.1.1 GPIF II Interface Requirements

Based on the timing diagram (Figure 8), the GPIF II state machine has the following requirements:

- The GPIF II block must transfer data from the data pins only when LV and FV signals are asserted.
- The image sensor does not provide flow control. Therefore, the interface must transfer a full line of video without interruption. In this design there are 1280 pixels per line, and each pixel requires 2 bytes, so 2560 bytes transfer per line.
- The CPU must be notified at the end of every frame so it can update the header bits accordingly.

Figure 8. Image Sensor Interface Timing Diagram



3.2 Pin Mapping of Image Sensor Interface

The GPIF II to image sensor pin mapping is shown in Table 6.

Table 6. Pin Mapping for Parallel Image Sensor Interface

EZ-USB FX3 Pin	Synchronous Parallel Image Sensor Interface with 8-bit Data Bus
GPIO[28]	LV
GPIO[29]	FV
GPIO[0:7]	DQ[0:7]
GPIO[16]	PCLK
I^{2}C_GPIO[58]	I^{2}C SCL
I^{2}C_GPIO[59]	I^{2}C SDA

Use the pin mapping shown in Table 7 for image sensors that use UART or SPI as an interface and if they use a 16-bit data bus.

Table 7. Additional Pin Mapping for Image Sensors

EZ-USB FX3 Pin	Image Sensor Interface (Additional Pins)
GPIO[8:15]	DQ[8:15]
GPIO[46]	GPIO/UART_RTS
GPIO[47]	GPIO/UART_CTS
GPIO[48]	GPIO/UART_TX
GPIO[49]	GPIO/UART_RX
GPIO[53]	GPIO/SPI_SCK /UART_RTS
GPIO[54]	GPIO/SPI_SSN/UART_CTS
GPIO[55]	GPIO/SPI_MISO/UART_TX
GPIO[56]	GPIO/SPI_MOSI/UART_RX

Note For the complete pin mapping of EZ-USB FX3, please refer to the datasheet “EZ-USB FX3 SuperSpeed USB Controller.”

3.3 Ping-Pong DMA Buffers

To understand the data transfer in and out of FX3, it is important to know the following terminology:

a. Socket
b. DMA Descriptor
c. DMA buffer
d. GPIF thread

A socket is a point of connection between a peripheral hardware block and the FX3 RAM. Each peripheral hardware block on FX3, such as USB, GPIF, UART, and SPI, has a fixed number of sockets associated with it. The number of independent data flows through a peripheral is equal to the number of its sockets. The socket implementation includes a set of registers that point to the active DMA descriptor and enable or flag interrupts associated with the socket.

A DMA Descriptor is a set of registers allocated in the FX3 RAM. It holds information about the address and size of a DMA buffer as well as pointers to the next DMA Descriptor. These pointers create DMA Descriptor chains.

A DMA buffer is a section of RAM used for intermediate storage of data transferred through the FX3 device. DMA buffers are allocated from the RAM by the FX3 firmware, and their addresses are stored as part of DMA Descriptors.

A GPIF thread is a dedicated data path in the GPIF II block that connects the external data pins to a socket.

Sockets can directly signal each other through events or they can signal the FX3 CPU via interrupts. This signaling is configured by firmware. Take, for example, a data stream from the GPIF II block to the USB block. The GPIF socket can tell the USB socket that it has filled data in a DMA buffer, and the USB socket can tell the GPIF socket that a DMA buffer has been emptied. This implementation is called an automatic DMA channel. The automatic DMA channel implementation is typically used when the FX3 CPU does not have to modify any data in a data stream.

Alternatively, the GPIF socket can send an interrupt to the FX3 CPU to notify it that the GPIF socket filled a DMA buffer. The FX3 CPU can relay this information to the USB socket. The USB socket can send an interrupt to the FX3 CPU to notify it that the USB socket emptied a DMA buffer. Then the FX3 CPU can relay this information back to the GPIF socket. This implementation is called a manual DMA channel. The manual DMA channel implementation is typically used when the FX3 CPU has to add, remove, or modify data in a data stream. The firmware example of this application note uses the manual DMA channel implementation because the firmware needs to add UVC video data header.

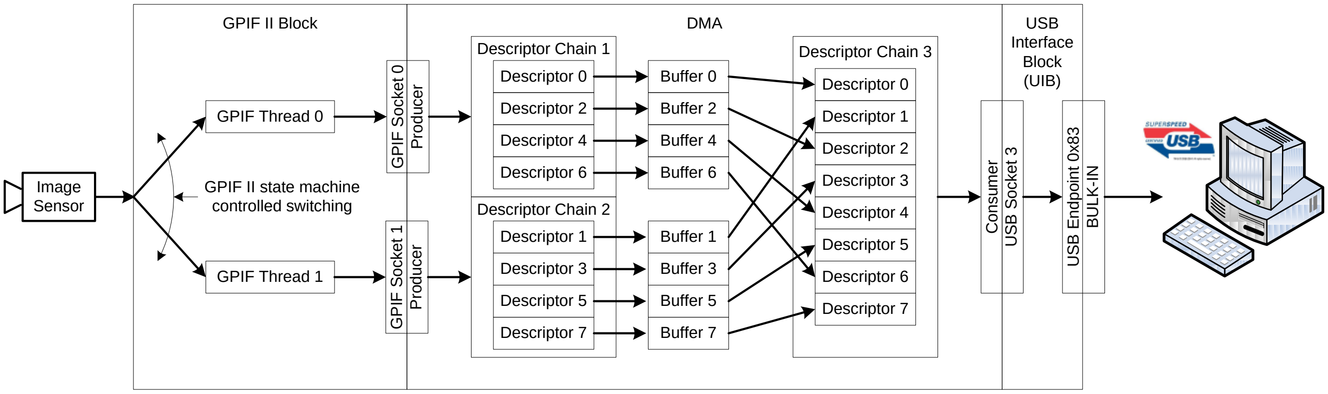
A socket that writes data to a DMA buffer is called a producer socket. A socket that reads data from a DMA buffer is called a consumer socket. A socket uses the values of the DMA buffer address, DMA buffer size and DMA Descriptor chain stored in a DMA Descriptor for data management (Section 4). A socket takes a finite amount of time (up to a few microseconds) to switch from one DMA Descriptor to another after it fills or empties a DMA buffer. The socket will not be able to transfer any data while this switch is in progress. This latency can be a problem for interfaces that have no flow control. One such example is an image sensor interface.

This issue is addressed in the GPIF II block through the use of multiple GPIF threads. The GPIF II block implements four GPIF threads. Only one GPIF thread can transfer data at a time. The GPIF II state machine must select an active GPIF thread to transfer data.

The GPIF thread selection mechanism is like a MUX. The GPIF II state machine uses internal control signals or external inputs to select the active GPIF thread. In this example, this switching between GPIF threads is controlled by the internal control signals. Switching the active GPIF thread will switch the active socket for the data transfer, thereby changing the DMA buffer used for data transfers. The GPIF thread switch has no latency. The GPIF II state machine can implement this switch at the DMA buffer boundary, masking the latency of the GPIF socket switching to a new DMA Descriptor. This allows the GPIF II block to take in data from the sensor, without any loss, when the DMA buffer is filled up.

Figure 9 shows the sockets, the DMA Descriptors, and the DMA buffer connections used in this application along with the data flow. Two GPIF threads are used to fill in alternate 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.

Figure 9. FX3 Data Transfer Architecture



3.4 Design Strategy

Before building the detailed GPIF II state machine, it is useful to consider the basic transfer strategy.

Figure 10. Data Transfer Within a Video Line

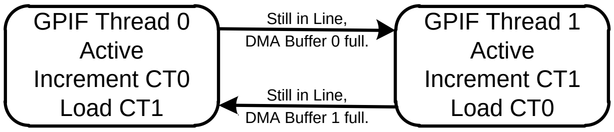
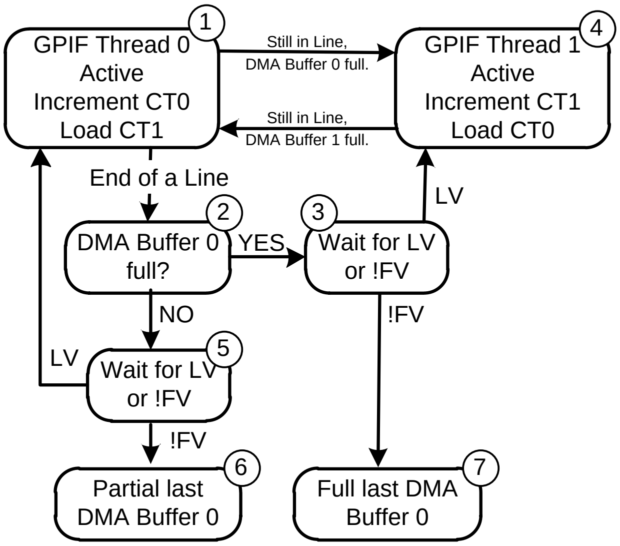


Figure 10 shows the basic DMA buffer ping-pong action when a DMA buffer fills during an active horizontal line. The GPIF II state machine has three independent counters. In this example, the GPIF II unit maintains two counters, CT0 and CT1, to count the number of bytes written to a DMA buffer. When the count reaches the DMA buffer limit, the “DMA buffer full” branch is taken and the GPIF II switches GPIF threads. As bytes are transferred in one GPIF thread, the counter for the other GPIF thread is initialized to prepare the counter to count bytes after the next GPIF thread switch occurs.

When the image sensor deasserts LV, it has reached the end of a video line. At this moment, the state machine has several options, as shown in Figure 11, where LV is the Line Valid signal and FV is the Frame Valid signal. The choices depend on whether a DMA buffer has just filled and if the frame has just completed.

Figure 11. Data Transfer Decisions at the End of a Video Line



Note A mirror image decision tree starting with “End of a Line” also emanates from state 4; this was omitted from the diagram for the sake of clarity.

At the end of a line (LV=0), the state machine transitions from state 1 to state 2 where it checks the DMA buffer byte counter to determine if the DMA buffer is full. If it is, the state machine proceeds to state 3. In state 3, if FV is low, a full frame has transferred and state 7 is entered. In state 7, the CPU is alerted via an interrupt request that signifies a full last DMA buffer. Using this information, the CPU can set up the GPIF II for the next frame.

If FV=1 in state 3, the image sensor is still transferring a frame, and it will soon reassert LV=1 to indicate the next video line. When LV=1, the state machine transitions from state 3 to state 4, performing the same GPIF thread switch as it does in the state 1 to state 4 transition. Thus, the paths 1-4 and 1-2-3-4 both result in a GPIF thread switch.

The difference between these paths is that the second path takes an extra cycle because it has an intervening end-of-line pause.

If the DMA buffer is not full, the state machine moves from state 2 to state 5. In state 5, as in state 3, it waits either for the end of the frame or for reassertion of LV (onset of the next video line). If LV=1, it continues to fill DMA buffer 0 in state 1. If FV=0, the image sensor has finished sending a video frame and the DMA buffer 0 is only partially filled. In state 6, the state machine raises a different interrupt to the CPU to allow it to account for sending a short DMA buffer over USB.

3.5 GPIF II State Machine

The FX3 GPIF II is a programmable state machine that can have up to 256 states. Each state can perform actions, including the following:

- drive multiple control lines
- send or receive data and/or address
- send interrupts to the internal CPU

State transitions can depend on internal or external signals, such as DMA Ready, and the image sensor's Frame/Line Valid.

To begin the GPIF II state machine design, choose a point in the image sensor waveform for the state machine to start. The start of a frame, indicated by a positive FV transition, is a logical starting point. GPIF II detects this edge by first waiting for FV=0 (first state) and then waiting for FV=1 (second state). The second state also initializes a transfer counter to correspond to one DMA buffer full of video data. The state machine tests the counter value and switches GPIF threads (DMA buffers) when the counter limit is reached. The counter limit is reached when a DMA buffer fills.

The state machine uses two GPIF II internal counters to count DMA buffer bytes: the GPIF II address counter ADDR and the data counter DATA. Whenever the GPIF II state machine switches GPIF threads, it initializes the appropriate counter for the other GPIF thread. Because loading a counter limit takes one clock cycle, the loaded value is one less than the terminal count.

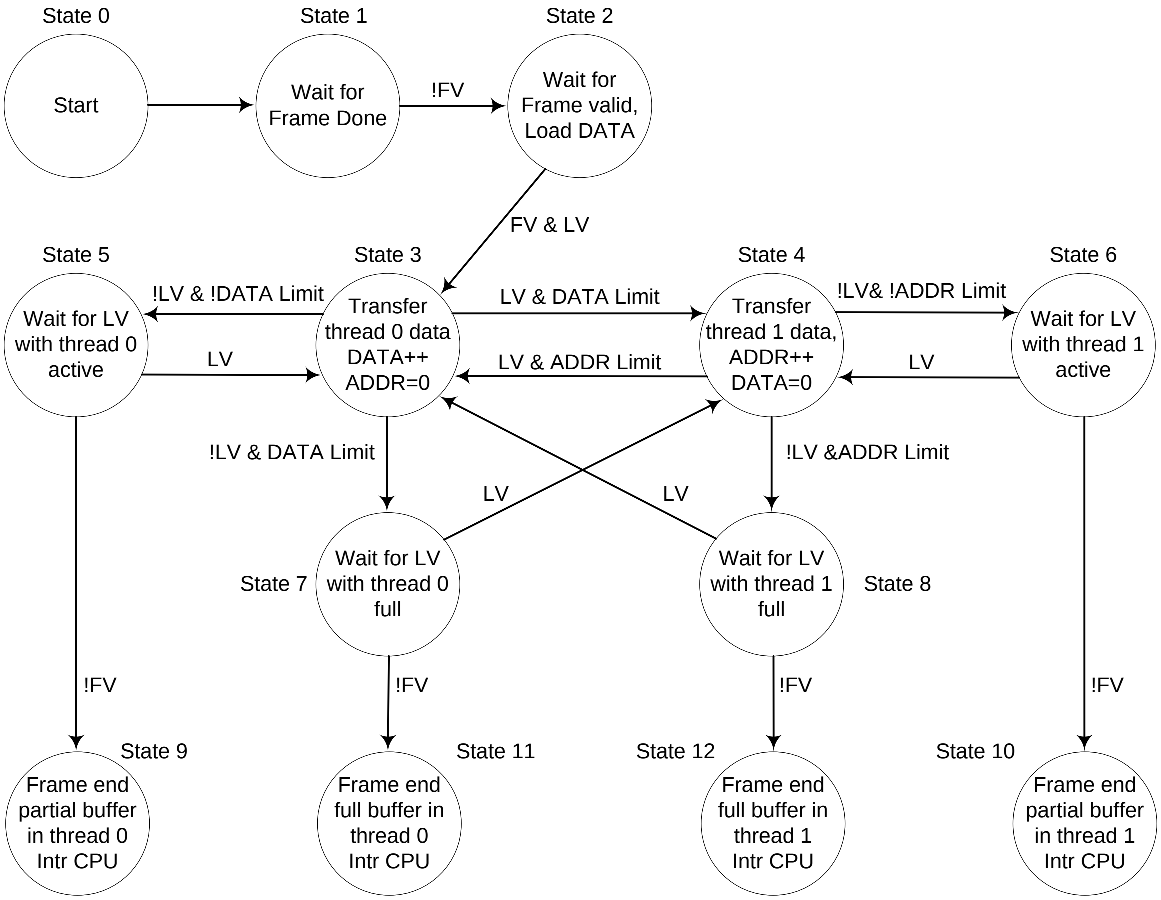
The transfer counters increment by one every clock cycle. Therefore, depending on the data bus width of the interface, the value of the counter limit would change. For this example the data bus width is 8 bits and the DMA buffer size is 16,368 bytes, as explained in Section 5.6, so the programmed limit should be 16,367. In general, the DMA buffer count limit is:

$$count = \left(\frac{producer\_buffer\_size(L)}{data\_bus\_width} \right) - 1$$

Therefore, for a 16-bit interface, the DMA buffer size is 8184 16-bit words, so the programmed limit would be 8183. For a 32-bit interface, the DMA buffer size is 4092 32-bit words, so the programmed limit would be 4091.

The detailed GPIF II state machine is shown in Figure 12. The two DMA buffer byte counters are the GPIF II DATA and ADDR counters. These counters correspond to CT0 and CT1, which are shown in Figure 11.

Figure 12. GPIF II State Machine Diagram

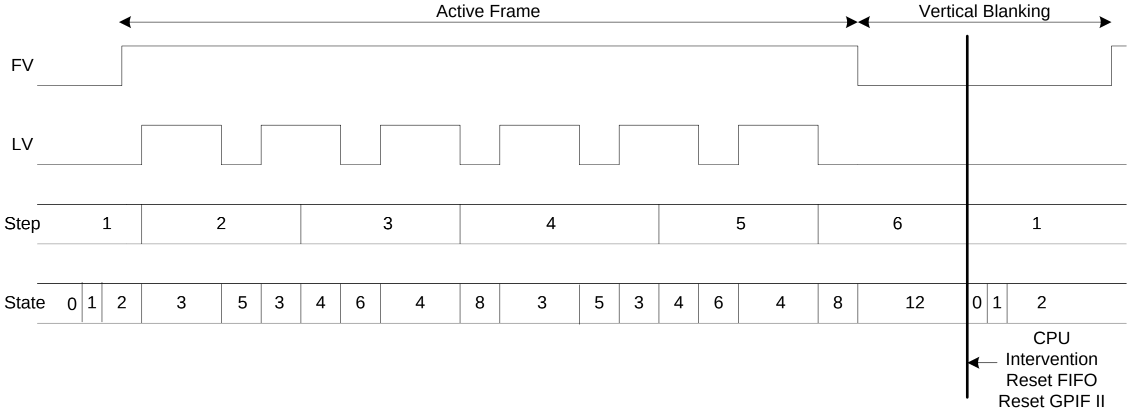

State 0
Start
State 1
Wait for Frame Done
!FV
State 2
Wait for Frame valid, Load DATA
FV & LV
State 5
Wait for LV with thread 0 active
!LV & !DATA Limit
LV
State 3
Transfer thread 0 data DATA++ ADDR=0
LV & DATA Limit
LV & ADDR Limit
State 4
Transfer thread 1 data, ADDR++ DATA=0
!LV& !ADDR Limit
LV
State 6
Wait for LV with thread 1 active
!LV & DATA Limit
LV
LV
!LV &ADDR Limit
State 7
Wait for LV with thread 0 full
Wait for LV with thread 1 full
State 8
!FV
!FV
!FV
!FV
State 9
Frame end partial buffer in thread 0 Intr CPU
State 11
Frame end full buffer in thread 0 Intr CPU
State 12
Frame end full buffer in thread 1 Intr CPU
State 10
Frame end partial buffer in thread 1 Intr CPU

At the end of every frame, the CPU receives one of four possible interrupt requests, indicating the DMA buffer number and fullness (states 9-12). These requests can be used to implement callback functions, which enable the CPU to handle several tasks, including the following:

1. Commit the last DMA buffer for USB transfer if there is a partial DMA buffer at the end of a frame (states 9 and 10). If the DMA buffer was full at the end of the frame, the GPIF II automatically committed it for USB transfer (states 11 and 12).
2. Wait for the consumer socket (USB) to transmit the last of the DMA buffer data; then reset the channel and the GPIF II state machine to state 0 to prepare it for the start of the next frame.
3. Handle any special application-specific tasks to indicate the frame advance. The UVC requires indicating the change-of-frame event by toggling a bit in the 12-byte header.

Figure 13 connects image sensor waveforms to GPIF II states for a small portion of a horizontal line. The “Step” line deals with the DMA system, which is described in Section 4 with Figure 45.

Figure 13. Image Sensor Interface, Data Path Execution, and State Machine Correlation



3.6 Implementing Image Sensor Interface Using GPIF II Designer

This section describes the design of the image sensor interface using the GPIF II Designer. For reference, the completed project is available in the “**GPIF II Designer/ImageSensorInterface.cydsn**” directory path inside the zip file of the attached source.

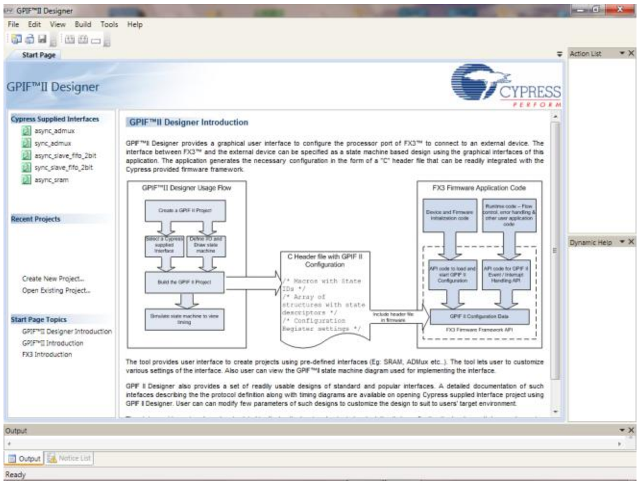
The design process involves three steps:

1. Create a project using the GPIF II Designer
2. Choose the Interface Definition
3. Draw the state machine on the canvas

3.6.1 Create the Project

Start GPIF II Designer. The GUI appears as Figure 14. Follow the step-by-step instructions as indicated in the following figures. Each figure includes sub-steps. For advance information about any of the steps, refer to the GPIF II Designer User Guide.

Figure 14. Start GPIF II Designer


GPIF™II Designer
File Edit View Build Tools Help
Start Page
GPIF™II Designer
CYPRESS
PERFORM
Action List
Dynamic Help
Cypress Supplied Interfaces
async_admux
sync_admux
async_slave_fifo_2bit
sync_slave_fifo_2bit
async_sram
Recent Projects
Create New Project...
Open Existing Project...
Start Page Topics
GPIF™II Designer Introduction
GPIF™II Introduction
FX3 Introduction
GPIF™II Designer Introduction
GPIF™II Designer Usage Flow
Create a GPIF II Project
Select a Cypress supplied Interface
Define I/O and Draw state machine
Build the GPIF II Project
Simulate state machine to view timing
C Header file with GPIF II Configuration
Include header file in firmware
FX3 Firmware Application Code
Device and Firmware Initialization code
Runtime code – Flow control, error handling & other user application code
API code to load and start GPIF II Configuration
API code for GPIF II Event / Interrupt Handling API
GPIF II Configuration Data
FX3 Firmware Framework API
Output
Output
Notice List
Ready

GPIF™II Designer provides a graphical user interface to configure the processor port of FX3™ to connect to an external device. The interface between FX3™ and the external device can be specified as a state machine based design using the graphical interfaces of this application. The application generates the necessary configuration in the form of a "C" header file that can be readily integrated with the Cypress provided firmware framework.

```
/* Macros with State
IDs */
/* Array of
structures with state
descriptors */
/* Configuration
Register settings */
```

The tool provides user interface to create projects using pre-defined interfaces (Eg: SRAM, ADMux etc..). The tool lets user to customize various settings of the interface. Also user can view the GPIF™II state machine diagram used for implementing the interface.

GPIF II Designer also provides a set of readily usable designs of standard and popular interfaces. A detailed documentation of such intefaces describing the the protocol definition along with timing diagrams are available on opening Cypress supplied Interface project using GPIF II Designer. User can can modify few parameters of such designs to customize the design to suit to users' target environment.

Figure 15. Open File Menu and Select New Project

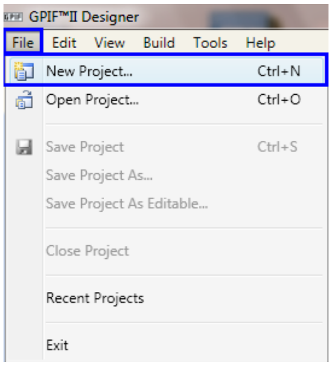
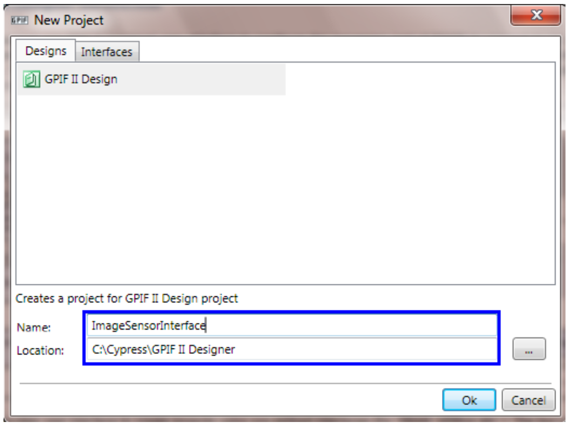


Figure 16. Enter Project Name and Location



Creates a project for GPIF II Design project

Name: ImageSensorInterface

Location: C:\Cypress\GPIF II Designer

The project creation is now complete, and the GPIF II Designer opens up access to the interface definition and state machine tabs. In the Interface Definition tab, the FX3 is on the left and the image sensor (labeled “Application Processor”) is on the right. The next step is to set the interface between the two.

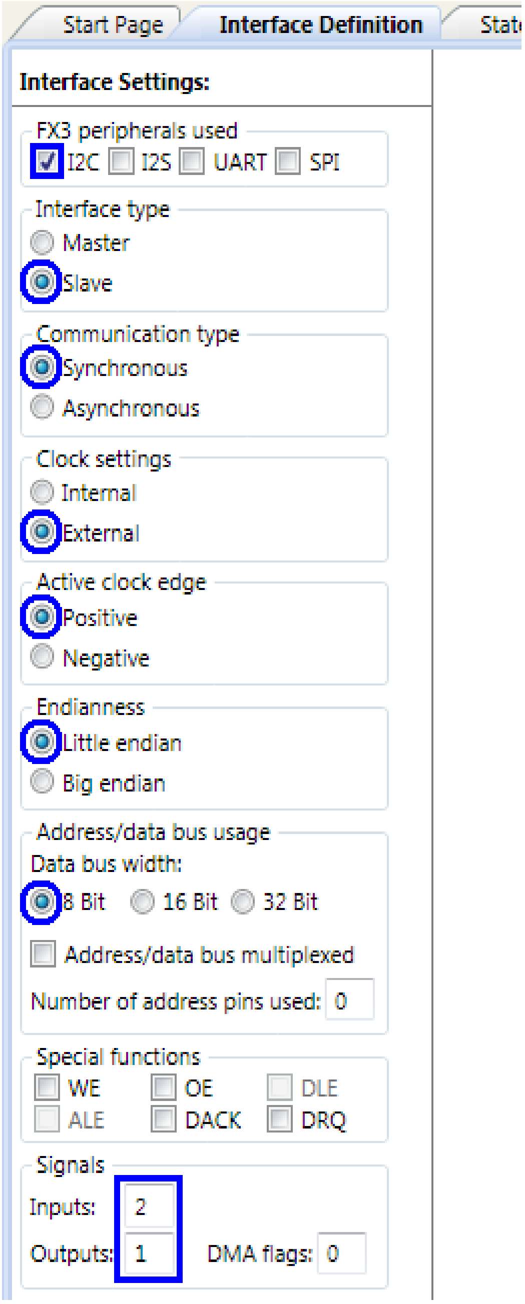
3.6.2 Define the Interface

In this project, the image sensor connected to the FX3 device has an 8-bit data bus. It uses GPIO 28 for the LV signal, GPIO 29 for the FV signal, and GPIO 22 for sensor reset as an active low input (Table 6). This image sensor also uses an I^{2}C connection to FX3 to access image sensor registers, for example, to configure the sensor for 720p mode. The “Interface Settings” column provides the necessary choices.

In addition, the indicated input signals become available in the next phase to create transition equations. Figure 17 shows the interface settings to choose.

1. In “Signals”, choose two inputs for LV and FV.
2. In “Signals”, choose one output for nSensor_Reset.
3. In “FX3 peripherals used”, select I^{2}C. This activates the FX3 I^{2}C master.
4. In “Interface type”, select “Slave”. Because the image sensor supplies the clock and drives the data bus, the GPIF II acts as a slave device.
5. In “Communication type”, select “Synchronous”. This reflects the presence of a synchronizing clock from the image sensor.
6. In “Clock settings”, select “External”. The image sensor supplies its PCLK signal to the GPIF II.
7. In “Active clock edge”, select “Positive”. The image sensor references data transitions to its positive edge.
8. In “Endianness”, select “Little endian”. Endianness refers to the byte ordering in data buses wider than one byte. For our 8-bit interface, this setting doesn’t matter.
9. In “Address/data bus usage”, select “8 bit”. The image sensor supplies an 8-bit data bus.

Figure 17. Interface Setting Selections



Start Page | Interface Definition | Stat

Interface Settings:

FX3 peripherals used
☑ I2C ☐ I2S ☐ UART ☐ SPI

Interface type
○ Master
● Slave

Communication type
● Synchronous
○ Asynchronous

Clock settings
○ Internal
● External

Active clock edge
● Positive
○ Negative

Endianness
● Little endian
○ Big endian

Address/data bus usage
Data bus width:
● 8 Bit ○ 16 Bit ○ 32 Bit

☐ Address/data bus multiplexed

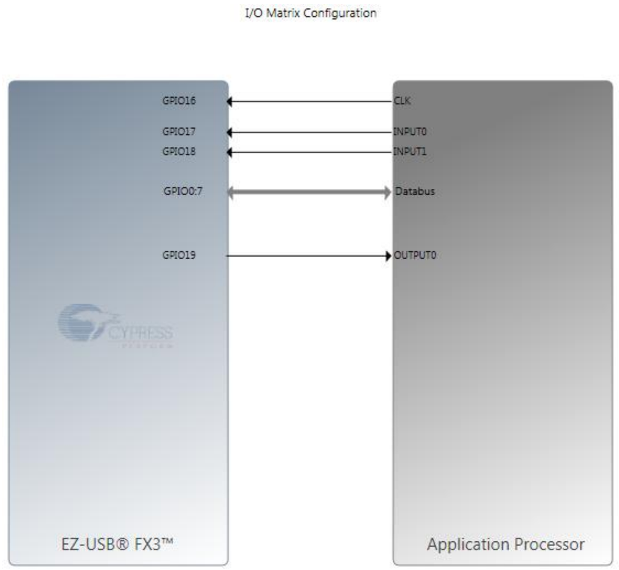
Number of address pins used: 0

Special functions
☐ WE ☐ OE ☐ DLE
☐ ALE ☐ DACK ☐ DRQ

Signals
Inputs: 2
Outputs: 1 DMA flags: 0

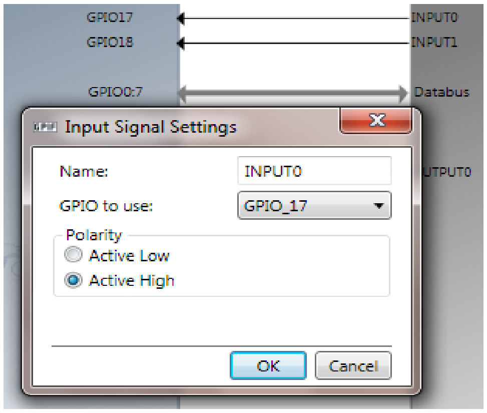
The “I/O Matrix configuration” should now look like Figure 18. The next step is to modify the properties of the input and the output signals. The properties include the name of the signals, the pin mapping (i.e., which GPIO acts as the input or the output), the polarity of the signal, and the initial value of the output signal.

Figure 18. The Block Diagram So Far



Double-click the INPUT0 label in the Application Processor area to open the properties for that input signal, as Figure 19 shows.

Figure 19. INPUT0 Default Properties



Input Signal Settings

Name: INPUT0

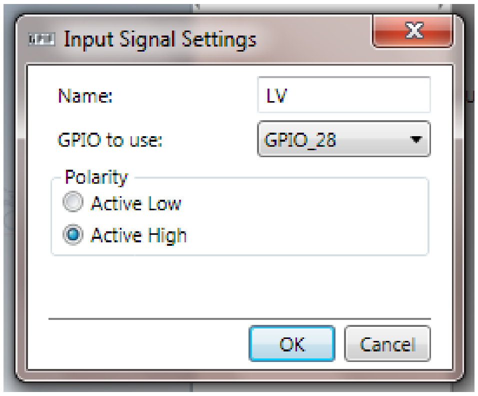
GPIO to use: GPIO_17

Polarity
○ Active Low
● Active High

OK Cancel

Change the name of the signal to "LV", and change "GPIO to use:" to GPIO_28 (Table 6). Keep the polarity as "Active High". The properties now appear as shown in Figure 20. Click OK.

Figure 20. Edited INPUT0 Properties



Input Signal Settings

Name: LV

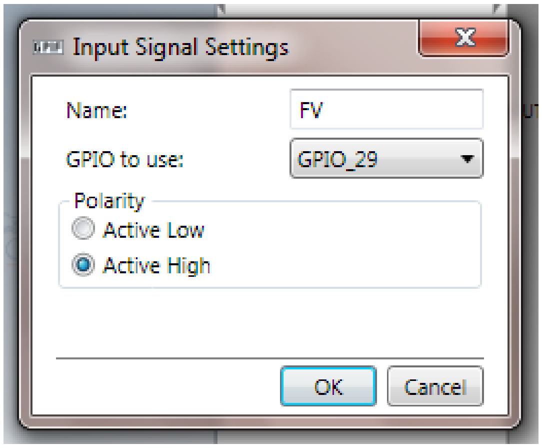
GPIO to use: GPIO_28

Polarity
○ Active Low
● Active High

OK Cancel

Next, double-click on the INPUT1 label, change the name to FV and the GPIO assignment to GPIO_29, and keep the polarity as Active High (Figure 21).

Figure 21. Edited INPUT1 Properties



Input Signal Settings

Name: FV

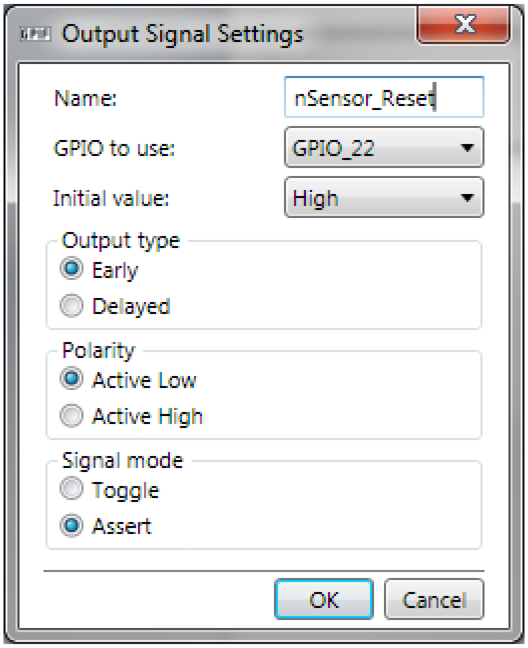
GPIO to use: GPIO_29

Polarity
○ Active Low
● Active High

OK Cancel

Double-click the OUTPUT0 label and change the settings according to Figure 22.

Figure 22. Edited OUTPUT0 Properties



Output Signal Settings

Name: nSensor_Reset

GPIO to use: GPIO_22

Initial value: High

Output type
● Early
○ Delayed

Polarity
● Active Low
○ Active High

Signal mode
○ Toggle
● Assert

This completes the Interface Settings. As you set properties, such as signal names, all other parts of the GPIF II designer will update to reflect the changes. For example, the block diagram now has signal names instead of generic input and output names. Also, as you use the state machine designer, the available signal options, for example the LV and FV signals, automatically appear as choices in drop-down lists.

3.6.3 Draw the State Machine

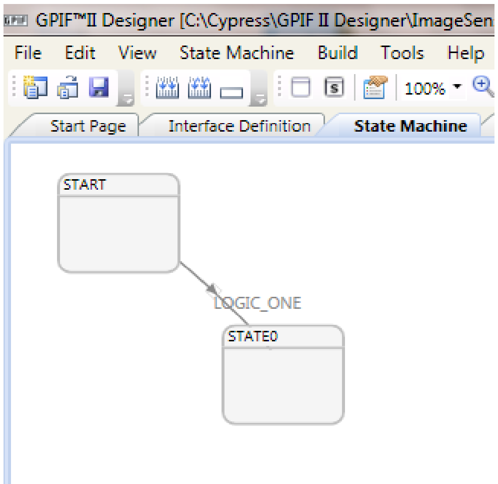
Click on the State Machine tab to open the state machine canvas. State machine design involves three basic operations:

1. Create states
2. Add actions within the states
3. Create transitions between states using conditions required to make the transitions

Draw the GPIF II State Machine

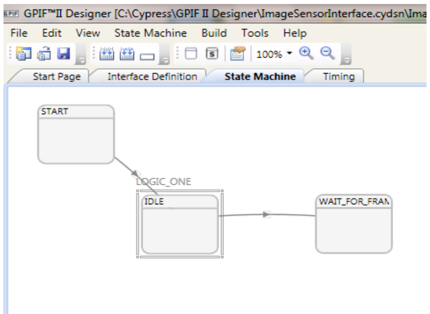
Click on the State Machine tab to open the canvas. An unedited canvas has two states: START and STATE0. An unconditional transition (LOGIC_ONE) connects the states (Figure 23).

Figure 23. Initial State Machine Canvas



1. Edit the name of STATE0 to IDLE by double-clicking inside the STATE0 box and editing the name text box. The "Repeat actions until next transition" determines if an action occurs once when the state is entered or if it occurs on every clock while inside the state. For states with no actions, such as this one, the checkbox state is irrelevant.
2. Add a new state by right-clicking on an empty spot in the canvas and selecting "Add State". Double-click inside this new state and change its name to WAIT_FOR_FRAME_START.
3. Create a transition from the IDLE state to the WAIT_FOR_FRAME_START state. Position the cursor in the center of the IDLE state box. The cursor changes to a + sign to indicate transition entry. Drag the mouse to the center of the WAIT_FOR_FRAME_START state. A small selection square appears inside the state box to help locate its center. If the mouse is released anywhere but in the center of the state box, the transition is not created. Try again. Notice that the state transition does not yet have any conditions.

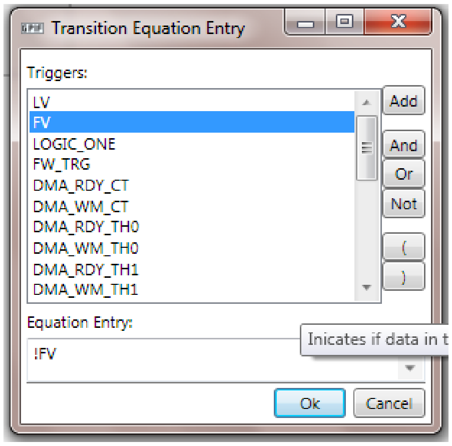
Figure 24. Results of Steps 1-3



4. Edit the equation for the transition from IDLE to WAIT_FOR_FRAME_START by double-clicking on the transition line (Figure 25). Notice that LV and FV

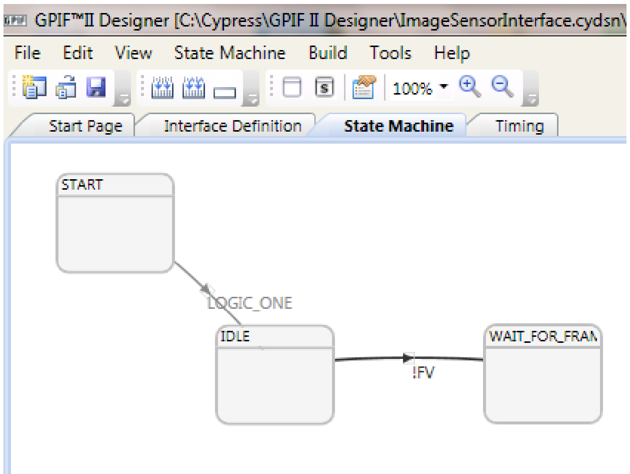
appear as equation terms to correspond to the renamed block diagram signals. To select FV low, build the equation using buttons and signal selections. Click the Not button, and the "!" symbol appears in the Equation Entry window. Then select the FV signal and click the "Add" button (or double-click the FV entry) to create the final "!FV" equation. You could also directly enter the equation by typing "!FV" in the Equation Entry window. The transition now appears as shown in Figure 26.

Figure 25. Double-Clicking a Transition Line Brings Up This Window



!FV

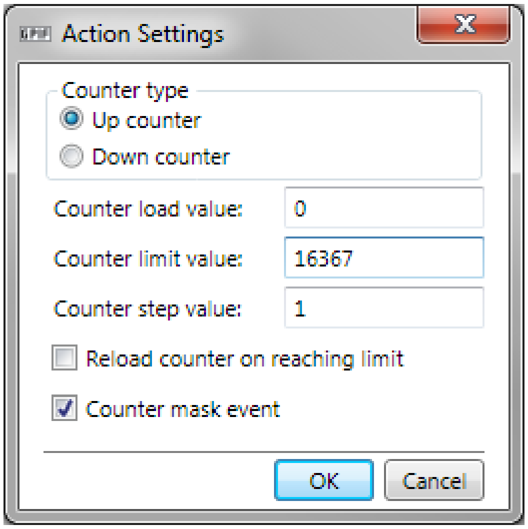
Figure 26. The Edited Transition



5. In the WAIT_FOR_FRAME_START state, we want to initialize our two byte counters because counters must be initialized in a state that precedes the state that increments them. Recall that this design uses the DATA counter for the Socket 0 buffer and the ADDR counter for the Socket 1 buffer. The "Action List" window in GPIF II Designer's top-right window displays the action choices. To add an action to a state, drag its name in the Action List into the state box. Drag the LD_DATA_COUNT and LD_ADDR_COUNT actions into the WAIT_FOR_FRAME_START state box.

6. To edit action properties, double-click the action name inside the state box. Edit the properties of both actions as shown in Figure 27. The "Counter mask event" checkbox disables an interrupt request when the counter reaches its limit value.

Figure 27. LD_DATA/ADDR_COUNT Action Settings



Counter type
● Up counter
○ Down counter

Counter load value: 0

Counter limit value: 16367

Counter step value: 1

☐ Reload counter on reaching limit

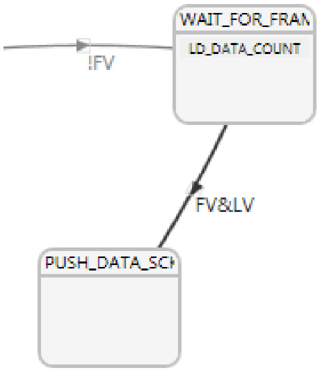
☑ Counter mask event

7. Add a new state with the name PUSH_DATA_SCK0 (Push image sensor data into Socket 0). This state transfers one image sensor byte per clock into the GPIF II interface, which routes it to Socket 0.

8. Create a transition from the WAIT_FOR_FRAME_START state to the PUSH_DATA_SCK0 state.

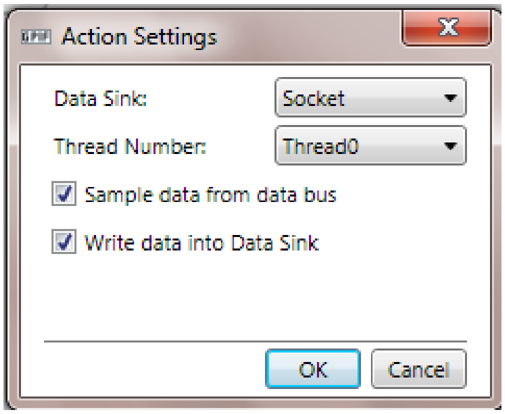
9. Edit this transition equation entry to occur when FV and LV are both asserted by creating the equation FV&LV. The resulting state transition is shown in Figure 28.

Figure 28. PUSH_DATA_SCK0 and Its Transition Condition FV&LV



10. Add the COUNT_DATA action to the PUSH_DATA_SCK0 state. This increments the DATA (Socket 0) counter value on every PCLK rising edge.
11. Add the IN_DATA action to the PUSH_DATA_SCK0 state. This action reads data from the data bus into the DMA buffers.
12. Add the LD_ADDR_COUNT action to the PUSH_DATA_SCK0 state to reload the ADDR counter. As before, the counter load is done in the state that actually increments the counter. The ADDR counter counts the bytes transferred into SCK1.
13. Edit the properties of the action IN_DATA in the PUSH_DATA_SCK0 state, as Figure 29 shows.

Figure 29. PUSH_DATA_SCK00 Action Settings



Action Settings

Data Sink: Socket

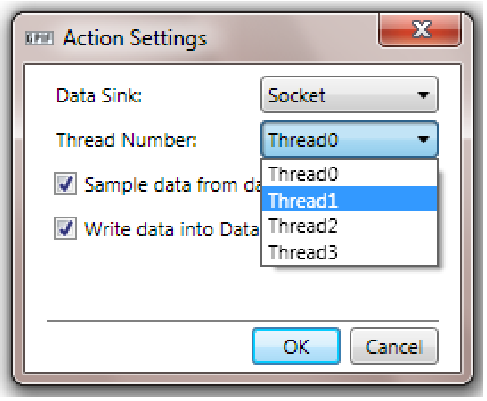
Thread Number: Thread0

☑ Sample data from data bus

☑ Write data into Data Sink

14. Add a new state with the name PUSH_DATA_SCK1. Add the COUNT_ADDR, IN_DATA and LD_DATA_COUNT actions to this state.
15. Edit the properties of the IN_DATA action in the PUSH_DATA_SCK1 state to use Thread1, as shown in Figure 30.

Figure 30. PUSH_DATA_SCK1 IN_DATA Action



Action Settings

Data Sink: Socket

Thread Number: Thread0

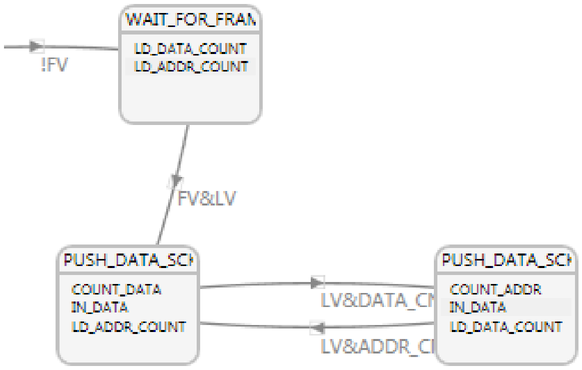
☑ Sample data from da

☑ Write data into Data

16. Create a transition from the PUSH_DATA_SCK0 state to the PUSH_DATA_SCK1 state. Edit this transition's equation entry using the equation "LV and DATA_CNT_HIT".
17. Create a transition from the PUSH_DATA_SCK1 state to the PUSH_DATA_SCK0 state. Reverse direction. Edit this transition's equation entry with the equation "LV and ADDR_CNT_HIT".

These transitions define state transitions that switch between GPIF threads during an active line at the DMA buffer boundaries. Figure 31 shows this portion of the state machine.

Figure 31. The Ping-Pong Action From Figure 10

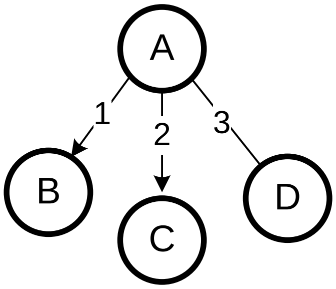


18. Add a new state "LINE_END_SCK0" to the left of the PUSH_DATA_SCK0 state.
19. Add a new state "LINE_END_SCK1" to the right of the PUSH_DATA_SCK1 state.

These two states are entered when the LV signal is deasserted while the image sensor switches to the next video line. Because the same operation is alternately executed using different GPIF threads, the states are required on both socket0 and socket1 sides.

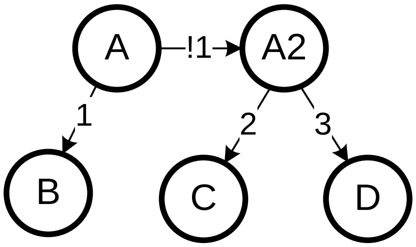
The PUSH_DATA state requires three possible exit transitions, but the GPIF II hardware implements a maximum of two per state. The three-transition requirement is handled by creating a dummy state that does nothing but distribute three exit conditions between two states. To demonstrate this, Figure 32 shows a state A transitioning to states B, C, or D based on conditions 1, 2, or 3.

Figure 32. State A Requires Exit Conditions 1, 2, and 3



To make this GPIF II compatible, add the dummy state A2, as shown in Figure 33.

Figure 33. Add a Dummy State A2 for GPIF II Compatibility



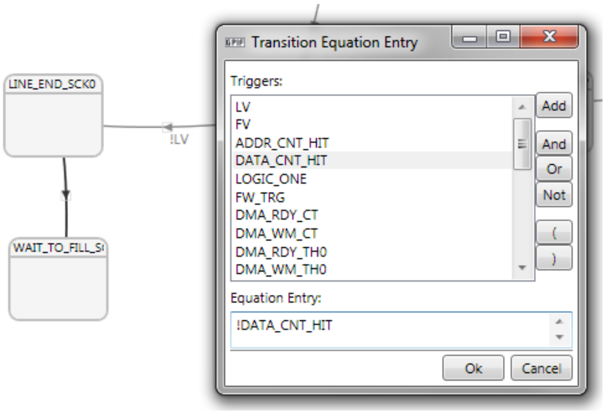
Create the LINE_END state to serve as this dummy state.

20. Create a transition from the PUSH_DATA_SCK0 state to the LINE_END_SCK0 state with the transition equation "(not LV)".
21. Create a transition from the PUSH_DATA_SCK1 state to the LINE_END_SCK1 state with the transition equation "(not LV)".
22. Add a new state "WAIT_TO_FILL_SCK0" below the LINE_END_SCK0 state.
23. Add a new state "WAIT_TO_FILL_SCK1" below the LINE_END_SCK1 state.

These two states are entered when the DMA buffers are not full but the line valid is deasserted.

24. Create a transition from the LINE_END_SCK0 state to the WAIT_TO_FILL_SCK0 state with the transition equation "(not DATA_CNT_HIT)", as Figure 34 shows.

Figure 34. DATA_CNT_HIT Indicates the Counter Has Reached Its Programmed Limit



Triggers:

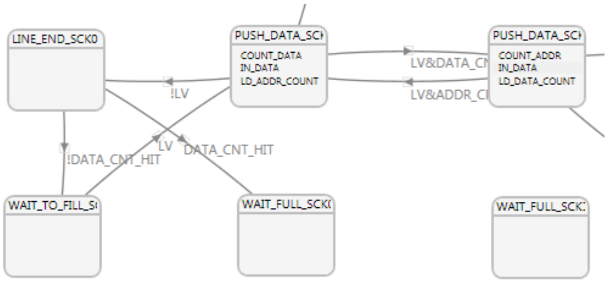
LV
FV
ADDR_CNT_HIT
DATA_CNT_HIT
LOGIC_ONE
FW_TRG
DMA_RDY_CT
DMA_WM_CT
DMA_RDY_TH0
DMA_WM_TH0

Equation Entry:

!DATA_CNT_HIT

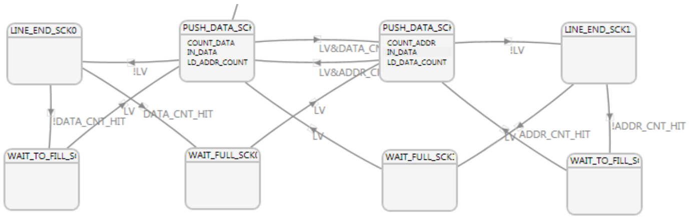
25. Create a transition back from the "WAIT_TO_FILL_SCK0" state to the "PUSH_DATA_SCK0" state with the equation "LV". The data transfer resumes in the same socket as soon as the line is active.
26. Create a transition from the "LINE_END_SCK1" state to the "WAIT_TO_FILL_SCK1" state with the transition equation "(not ADDR_CNT_HIT)".
27. Create a transition back from the "WAIT_TO_FILL_SCK1" state to the "PUSH_DATA_SCK1" state with the equation "LV".
28. Add a new state "WAIT_FULL_SCK0_NEXT_SCK1" below the "PUSH_DATA_SCK0" state.
29. Add a new state "WAIT_FULL_SCK1_NEXT_SCK0" below the "PUSH_DATA_SCK1" state.
30. During these two states (WAIT_FULL_), the image sensor is switching lines at the DMA buffer boundaries. Therefore, the next byte transfer must use the currently inactive GPIF thread.
31. Create a transition from the "LINE_END_SCK0" state to the "WAIT_FULL_SCK0_NEXT_SCK1" state with the equation "DATA_CNT_HIT". This portion of the state machine appears as in Figure 35.

Figure 35. WAIT_FULL States Added



32. Create a transition from the “LINE_END_SCK1” state to the “WAIT_FULL_SCK1_NEXT_SCK0” state with the equation “ADDR_CNT_HIT”.
33. Create a transition from the “WAIT_FULL_SCK0_NEXT_SCK1” state to the “PUSH_DATA_SCK1” state with the equation “LV”. Notice that doing so creates a crossed link in the diagram.
34. Create a transition from the “WAIT_FULL_SCK1_NEXT_SCK0” state to the “PUSH_DATA_SCK0” state with the equation “LV”. The resultant state machine portion appears as in Figure 36.

Figure 36. Wait When DMA Buffers Are Full



35. Add a new state “PARTIAL_BUF_IN_SCK0” below the “WAIT_TO_FILL_SCK0” state.
36. Add a new state “PARTIAL_BUF_IN_SCK1” below the “WAIT_TO_FILL_SCK1” state.

The PARTIAL_BUF_ states are an indication of the end of the frame, where the last DMA buffer has not completely filled. The CPU must commit this partial DMA buffer manually when it responds to a GPIF II-generated interrupt request.

37. Add a new state “FULL_BUF_IN_SCK0” below the “WAIT_FULL_SCK0_NEXT_SCK1” state.
38. Add a new state “FULL_BUF_IN_SCK1” below the “WAIT_FULL_SCK1_NEXT_SCK0” state.

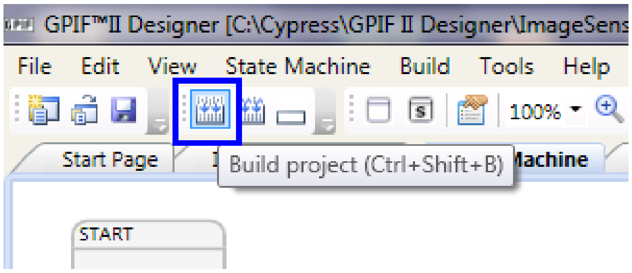
The FULL_BUF_ states are an indication that the end of the frame data is the last byte in the DMA buffer associated with the corresponding GPIF thread. The GPIF II hardware has taken care of committing this full DMA buffer, so any further action here would be application-specific.

39. Create a transition from the “WAIT_TO_FILL_SCK0” state to the “PARTIAL_BUF_IN_SCK0” state with the equation “not FV”.
40. Create a transition from the “WAIT_TO_FILL_SCK1” state to the “PARTIAL_BUF_IN_SCK1” state with the equation “not FV”.
41. Create a transition from the “WAIT_FULL_SCK0_NEXT_SCK1” state to the “FULL_BUF_IN_SCK0” state with the equation “not FV”.
42. Create a transition from the “WAIT_FULL_SCK1_NEXT_SCK0” state to the “FULL_BUF_IN_SCK1” state with the equation “not FV”.
43. Add action “Intr_CPU” in each of the states “PARTIAL_BUF_IN_SCK0”, “PARTIAL_BUF_IN_SCK1”, “FULL_BUF_IN_SCK0”, and “FULL_BUF_IN_SCK1”.

The final state machine appears as Figure 38. This can be compared with Figure 12, with the only difference being the addition of the PUSH_DATA states to accommodate the two-transition maximum for any state.

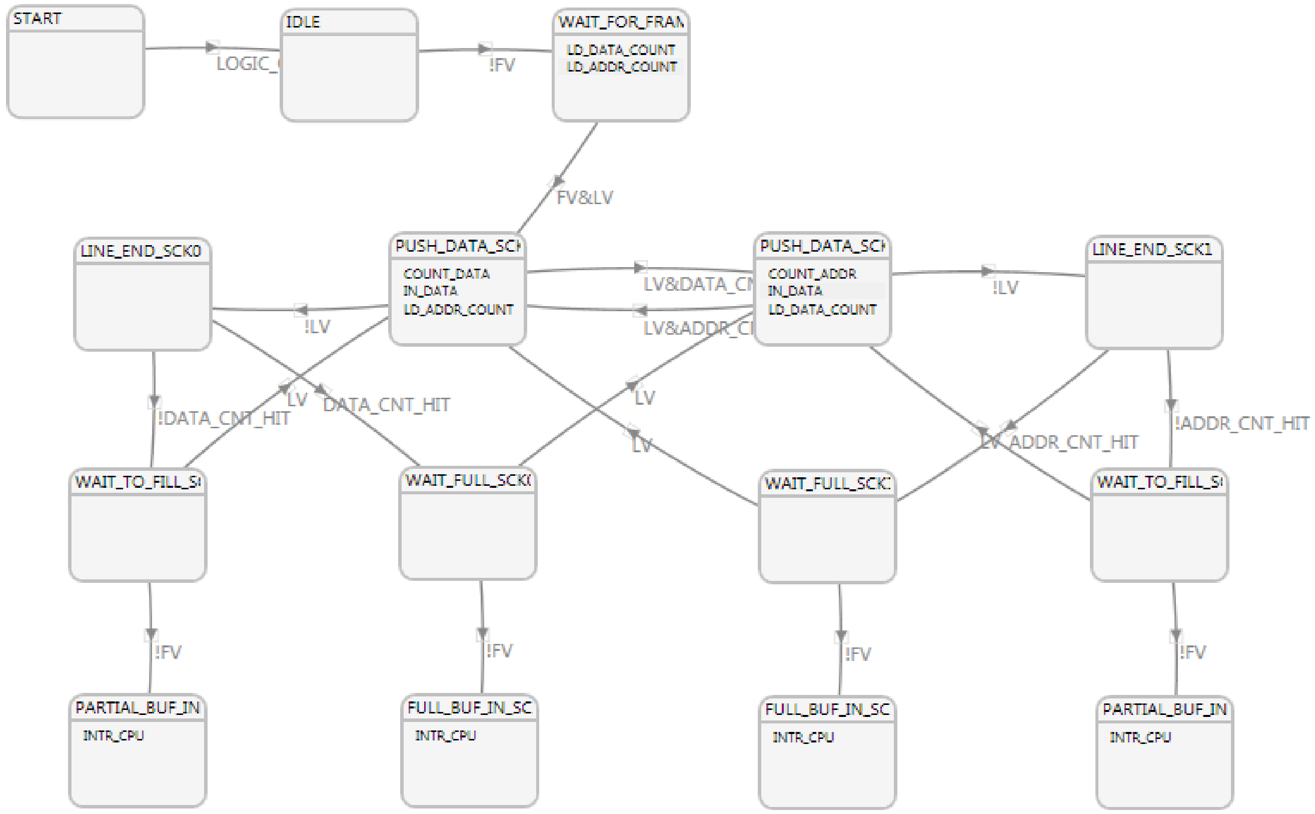
44. Save the project by selecting “File-Save Project As”.
45. Build the project using the Build icon highlighted in Figure 37. The project output window indicates the build status.

Figure 37. The BUILD Button


GPIF™II Designer [C:\Cypress\GPIF II Designer\ImageSens
File
Edit
View
State Machine
Build
Tools
Help
100%
Start Page
Build project (Ctrl+Shift+B)
lachine
START

46. Check the output of the project. This appears as a header file called **cyfxgpif2config.h** in the project directory. If you examine this header file, you’ll see that GPIF Designer II has created arrays of GPIF II internal settings that will be used by the FX3 firmware to configure the GPIF II and to define its state machine. Never directly edit this file; let GPIF II Designer do the work for you.

Figure 38. Final State Machine Diagram



3.6.4 Editing GPIF II Interface Details

This section describes how to change the interface settings, if required. As an example, if the image sensor/ASIC has a 16-bit-wide data bus, you would need to change the GPIF II interface to accommodate the data bus. To accomplish this, take the following steps:

1. Open the **ImageSensorInterface.cyfx** project in the GPIF II Designer. (This project may not be directly compiled.)
2. Go to File->Save Project As.
3. Save the project in a convenient location with a convenient name in the dialog box that will appear.
4. Close the currently open project (File->Close Project).
5. Open the project that was saved in step 3.
6. Go to the **Interface Definition** tab and choose the **16 Bit** option for **Address/Data Bus Usage** setting.
7. Go to the **State Machine** tab.
8. In the state machine canvas, double-click the LD_DATA_COUNT action inside the WAIT_FOR_FRAME_START state. Change the counter limit value to 8183.
9. Do the same for LD_ADDR_COUNT action.
10. Save the Project.
11. Build the Project.
12. Copy the newly generated **cyfxgpif2config.h** header file from the location selected in Step 3 to the firmware project directory. You will overwrite the existing **cyfxgpif2config.h** file if there is one. Find the firmware project directory, **USBVideoClass**, inside the zip file of the attached source.

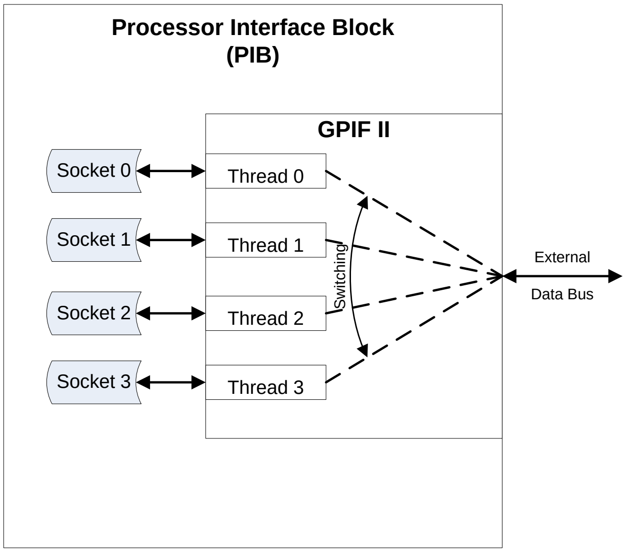
Note: If you are changing the GPIF II bus width to 32 bits, make sure the iomatrix configuration in the firmware has the isDQ32Bit parameter set to CyTrue.

The next sections explain the details of the DMA channel to stream data and the firmware that supports UVC.

4. Setting up the DMA System

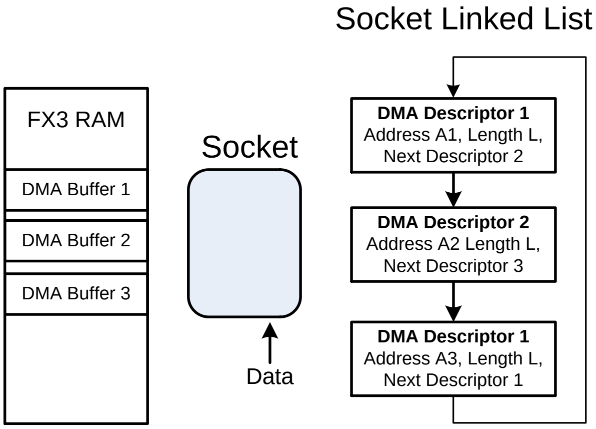
The GPIF II block, a part of the processor interface block (PIB), can run up to 100 MHz with 32 bits of data (400 MBps). To transfer the data into internal DMA buffers, GPIF II uses multiple GPIF threads connected to DMA producer sockets (explained in Section 3.3). Two of the four GPIF threads are used in this application. Default mapping (Figure 39) of the sockets and GPIF threads is used for this application—Socket 0 is connected to GPIF thread 0, and Socket 1 is connected to GPIF thread 1. The GPIF thread switching is accomplished in the GPIF II State machine designed in the previous section.

Figure 39. Default GPIF II Socket/Thread Mapping



To understand DMA transfers, the concept of a socket is further explored in the following four figures. Figure 40 shows the two main socket attributes, a linked list, and a data router.

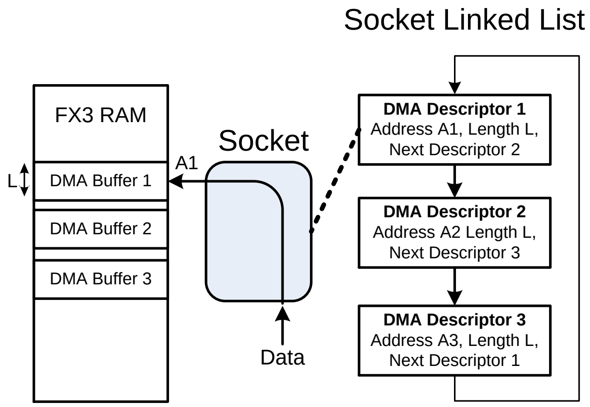
Figure 40. A Socket Routes Data According to a List of DMA Descriptors



The Socket linked list is a set of data structures in main memory called DMA Descriptors. Each Descriptor specifies a DMA buffer address and length as well as a pointer to the next DMA Descriptor. As the socket operates, it retrieves the DMA Descriptors one at a time, routing the data to the DMA buffer specified by the Descriptor address and length. When L bytes have transferred, the socket retrieves the next Descriptor and continues transferring bytes to a different DMA buffer.

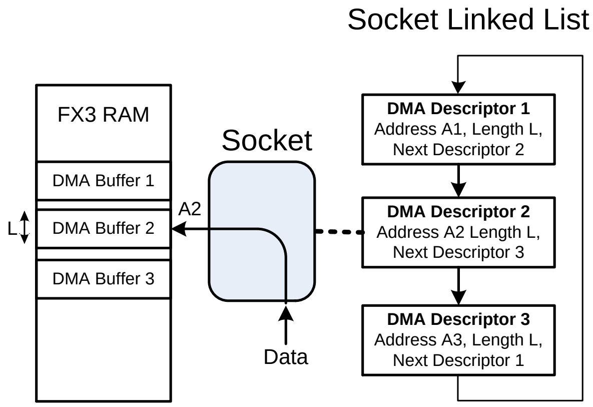
This structure makes a socket extremely versatile because any number of DMA buffers can be created anywhere in memory and be automatically chained together. For example, the socket in Figure 40 retrieves DMA Descriptors in a repeating loop.

Figure 41. A Socket Operating with DMA Descriptor 1



In Figure 41, the socket has loaded DMA Descriptor 1, which tells it to transfer bytes starting at A1 until it has transferred L bytes, at which time it retrieves DMA Descriptor 2 and continues with its address and length settings A2 and L (Figure 42).

Figure 42. A Socket Operating with DMA Descriptor 2



In Figure 43 the Socket retrieves the third DMA Descriptor and transfers data starting at A3. When it has transferred L bytes, the sequence repeats with DMA Descriptor 1.

Figure 43. A Socket Operating with DMA Descriptor 3

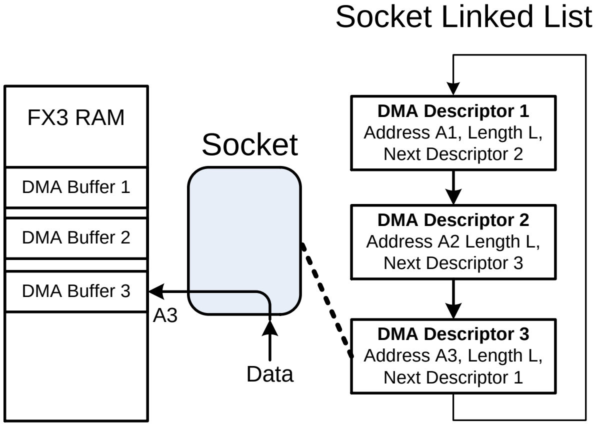
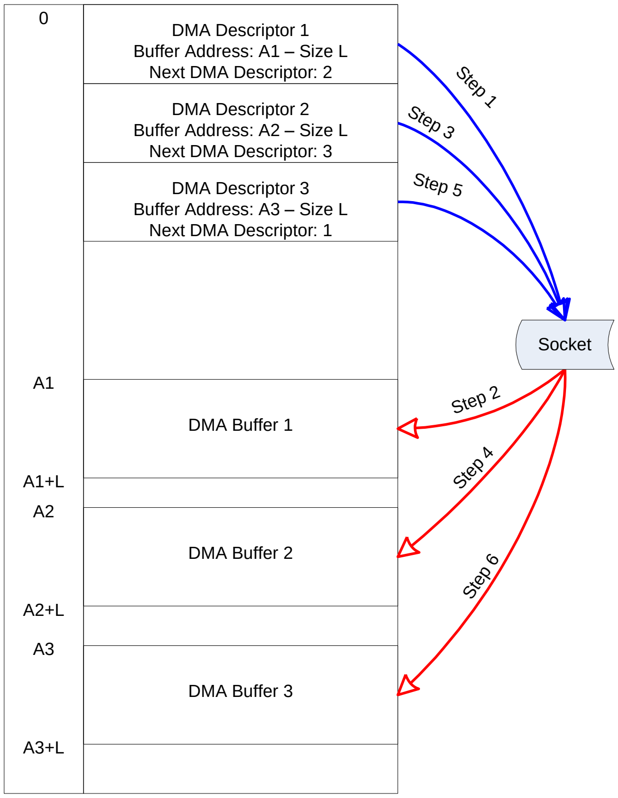


Figure 44 shows a DMA data transfer in more detail. This example uses three DMA buffers of length L chained in a circular loop. FX3 memory addresses are on the left. The blue arrows show the socket loading the socket linked list Descriptors from memory. The red arrows show the resulting data paths. The following steps show the socket sequence as data is moved to the internal DMA buffers.

Step 1: Load DMA Descriptor 1 from the memory into the socket. Get the DMA buffer location (A1), DMA buffer size (L), and next Descriptor (DMA Descriptor 2) information. Go to step 2.

Step 2: Transfer data to the DMA buffer location starting at A1. After transferring DMA buffer size L amount of data, go to step 3.

Figure 44. DMA Transfer Example



Step 3: Load DMA Descriptor 2 as pointed to by the current DMA Descriptor 1. Get the DMA buffer location (A2), DMA buffer size (L), and next Descriptor (DMA Descriptor 3) information. Go to step 4.

Step 4: Transfer data to the DMA buffer location starting at A2. After transferring DMA buffer size L amount of data, go to step 5.

Step 5: Load DMA Descriptor 3 as pointed to by the current DMA Descriptor 2. Get the DMA buffer location (A3), DMA buffer size (L), and next Descriptor (DMA Descriptor 1) information. Go to step 6.

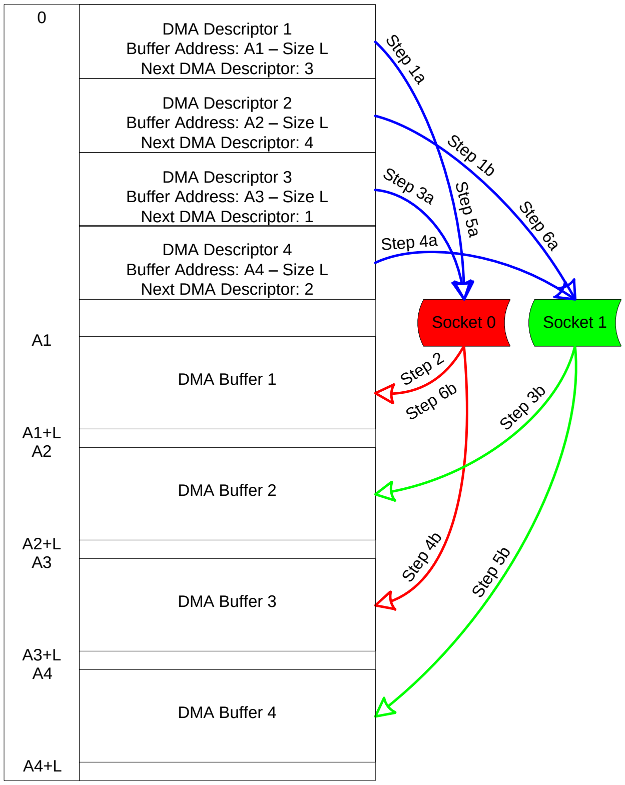
Step 6: Transfer data to the DMA buffer location starting at A3. After transferring DMA buffer size L amount of data, go to step 1.

This simple scheme has an issue in the camera application. A socket takes time to retrieve the next DMA Descriptor from memory, typically 1 microsecond. If this transfer pause occurs in the middle of a video line, the video data is lost. To prevent this loss, the DMA buffer size can be set as a multiple of the video line length. This would make the DMA buffer switching pause coincide with the time that the video line is inactive (LV=0). However, this approach lacks flexibility if, for example, the video resolution is changed.

Setting DMA buffer size exactly equal to line size is also not a good solution because it does not take advantage of the USB 3.0 maximum burst rate for BULK transfers. USB 3.0 allows a maximum of 16 bursts of 1024 bytes over BULK Endpoints. This is why the DMA buffer size is set to 16 KB.

A better solution is to take advantage of the fact that sockets can be switched without latency—in one clock cycle. Therefore, it makes sense to use *two* Sockets to store data into four interleaved DMA buffers. Data transfer using dual sockets is described in Figure 45, again with numbered execution steps. Socket0 and Socket1 access to DMA buffers is differentiated by red and green arrows (data paths for individual sockets), respectively. The ‘a’ and ‘b’ parts of each step occur simultaneously. This parallel operation of the hardware eliminates DMA Descriptor retrieval dead time and allows the GPIF II to stream data continuously into internal memory. These steps correspond to the “Step” line in Figure 13.

Figure 45. Dual Sockets Yield Seamless Transfers



Step 1: At initialization of the sockets, Socket 0 and Socket 1 load the DMA Descriptor 1 and DMA Descriptor 2, respectively.

Step 2: As soon as the data is available, Socket 0 transfers the data to DMA buffer 1. The transfer length is L. At the end of this transfer, go to step 3.

Step 3: GPIF II switches the GPIF thread and, therefore, the socket for data transfer. Socket 1 starts to transfer data to DMA buffer 2, and, at the same time, Socket 0 loads the DMA Descriptor 3. By the time Socket 1 finishes transferring L amount of data, Socket 0 is ready to transfer data into DMA buffer 3.

Step 4: GPIF II now switches back to the original GPIF thread. Socket 0 now transfers the data of length L into DMA buffer 3. At the same time, Socket 1 loads the DMA Descriptor 4, making it ready to transfer data to DMA buffer 4. After Socket 0 finishes transferring the data of length L, go to step 5.

Step 5: GPIF II routes Socket 1 data into DMA buffer 4. At the same time, Socket 0 loads DMA Descriptor 1 to prepare to transfer data into DMA buffer 1. Notice that Step 5a is the same as Step 1a except that Socket 1 is not initializing but, rather, transferring data simultaneously.

Step 6: GPIF II switches sockets again, and Socket 0 starts to transfer data of length L into DMA buffer 1. It is assumed that by now, the DMA buffer is empty, having been depleted by the UIB consumer socket. At the same time, Socket 1 loads the DMA Descriptor 2 and is ready to transfer data into DMA buffer 2. The cycle now goes to Step 3 in the execution path.

GPIF II Sockets can transfer video data only if the consuming side (USB) empties and releases the DMA buffers in time to receive the next chunk of video data from GPIF II. If the consumer is not fast enough, the sockets drop data because their DMA buffer writes are ignored. As a result, the byte counters lose sync with the actual transfers, which can propagate to the next frame. Therefore, a cleanup mechanism is required at the end of every frame. This mechanism is described in the Clean Up section.

According to the flowchart in Figure 12, a frame transfer ends with four possible states:

- Socket 0 has transferred a full DMA buffer
- Socket 1 has transferred a full DMA buffer
- Socket 0 has transferred a partial DMA buffer
- Socket 1 has transferred a partial DMA buffer

In the partial DMA buffer cases, the CPU needs to commit the partial DMA buffer to the USB consumer.

The DMA channel is initialized using a function in the **uvc.c** file called **CyFxUVCApplnInit**. The DMA channel configuration details are customized in the “dmaMultiConfig” structure in the **CyFxUVCApplnInit** function. The DMA channel type is set to MANUAL_MANY_TO_ONE.

In addition, the USB Endpoint that streams data to the USB 3.0 Host is configured to enable a burst of 16 over the 1024-byte BULK Endpoint. This is set using the “endPointConfig” structure passed in the “CyU3PSetEpConfig” function, with the Endpoint constant set to “CY_FX_EP_BULK_VIDEO”.

4.1 About DMA Buffers

This section summarizes how FX3 DMA buffers are created and used in this application.

- The integral parts of a DMA channel are described in Section 3.3.
- In this application, the GPIF II unit is the producer, and the FX3 USB unit is the consumer.
- This application uses the GPIF thread switching feature in the GPIF II block to avoid data drops.
- When a producer socket loads a DMA Descriptor, it checks the associated DMA buffer to see if it is ready for a write operation. The producer socket changes its state to “active” for writing data into FX3 RAM if it finds that the DMA buffer is empty. The producer socket locks the DMA buffer for write operations.
- When a producer socket is finished writing to a DMA buffer, it releases the lock so that the consumer socket can access the DMA buffer. The action is called “buffer wrap-up“ or simply “wrap-up”. The DMA unit is then said to commit the DMA buffer to the FX3 RAM. The producer socket is said to have produced a DMA buffer. A DMA buffer should be wrapped up only while the producer is not actively filling it. This is the reason for the FV=0 test in the state diagrams.
- If a DMA buffer fills completely, as it does repeatedly during a frame, the wrap-up operation is automatic. The producer socket releases the lock on the DMA buffer, commits it to the FX3 RAM, switches to an empty DMA buffer, and continues to write the video data stream.
- When a consumer socket loads a DMA Descriptor, it checks the associated DMA buffer to see if it is ready for a read operation. The consumer socket changes its state to “active” for reading data from FX3 RAM if it finds that the DMA buffer is committed. The consumer socket locks the DMA buffer for read operations.
- After the consumer socket has read all the data from the DMA buffer, it releases the lock so that the producer socket can access the DMA buffer. The consumer socket is said to have consumed the DMA buffer.
- If the same DMA descriptors are used by the producer and consumer sockets, the DMA buffer full/empty status is communicated automatically between the producer and consumer sockets via the DMA Descriptors and inter-socket events.
- In this application, because the CPU needs to add a 12-byte UVC header, the producer socket and the consumer socket need to load different sets of DMA Descriptors. The DMA Descriptors loaded by the producer socket will point to DMA buffers that are at a 12-byte offset from the corresponding DMA buffers that the DMA Descriptors loaded by the consumer socket point to.
- Due to different DMA Descriptors for producer and consumer sockets, the CPU must manage the communication of the DMA buffer status between the producer and the consumer sockets. This is why the DMA channel implementation is called a “Manual DMA” channel.
- After a DMA buffer is produced by the GPIF II block, the CPU is notified via an interrupt. The CPU then adds the header information and commits the DMA buffer to the consumer (USB Interface Block).
- On the GPIF II side, the video data in DMA buffers are automatically wrapped up and committed to the FX3 RAM for all but the last DMA buffer in the frame. No CPU intervention is required.
- At the end of a frame, the final DMA buffer is likely not filled completely. In this case, the CPU intervenes and manually wraps up the DMA buffer and commits it to the FX3 RAM. This is called a “forced wrap-up”.

5. FX3 Firmware

The example firmware supplied with this application note is located under the **USBVideoClass** folder of the zip file of the source. Please refer to AN75705, “Getting Started with EZ-USB FX3”, to import the firmware project in the Eclipse workspace. This example firmware will compile and enumerate, but the user must tailor it to a particular image sensor using the **sensor.c** and **sensor.h** files to be able to stream video from the image sensor. These files are provided in the example project as placeholders where you should implement code to configure the image sensor.

If the Aptina sensor board is used as described in Section 6 of this application note, and if an Aptina NDA is signed, Cypress will provide **sensor.c** and **sensor.h** files to correspond with this particular sensor. A pre-compiled load image is provided in the project to try out the FX3-Aptina configuration without the NDA (Section 7).

Table 8 summarizes the code modules and the functions implemented in each module.

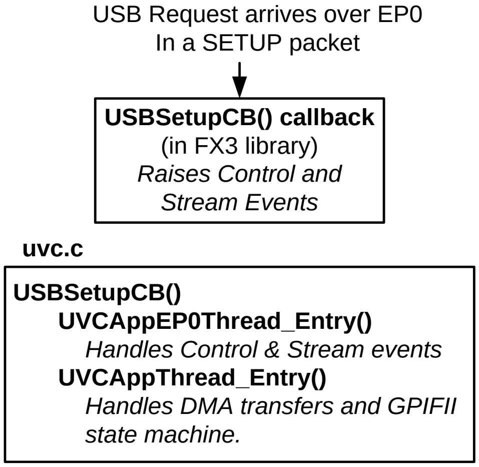
Table 8. Example Project Files

File	Description
sensor.c	• Defines **SensorWrite2B**, **SensorWrite**, **SensorRead2B**, and **SensorRead** functions to write and read the image sensor configuration over I^{2}C. These functions assume that the register addresses on the I^{2}C bus of the image sensor are 16-bit wide. • Defines **SensorReset** function to control the reset line of the image sensor, and **SensorInit** function to test the I^{2}C connection and configure the image sensor in default streaming mode (using **SensorScaling_HD720p_30fps** function) • Defines **SensorScaling_VGA** and **SensorScaling_HD720p_30fps** functions to configure the image sensor in desired streaming modes. These functions are placeholder functions. They need to be populated with image sensor-specific configuration commands. • Defines **SensorGetBrightness** and **SensorSetBrightness** functions to read and to write the brightness value in the image sensor brightness control register. These functions are placeholder functions. They need to be populated with image sensor-specific configuration commands.
sensor.h	• Contains the constants used for the image sensor (its I^{2}C slave address and reset GPIO number). You have to define the I^{2}C slave address of the image sensor here. • Contains the declarations of all the functions defined in **sensor.c**
camera_ ptzcontrol.c	• Defines the functions to read and to write the PTZ values of the camera. These are placeholder functions. They need to be populated with image sensor-specific configuration commands. • Uncomment the line “#define UVC_PTZ_SUPPORT” in **uvc.h** to enable PTZ control code placeholders.
camera_ ptzcontrol.h	• Contains the constants used for the PTZ control implementation • Contains the declarations of the functions defined in **camera_ptzcontrol.c** file
cyfxtx.c	• No changes needed. Use this file as provided with the project associated with this application note. • It contains the variables that RTOS and the FX3 API library use for memory mapping, and functions that the FX3 API library uses for memory management.
cyfxgpif2config.h	• Header file generated by the GPIF II Designer tool. No changes are required. If the interface needs to be changed, a new header file should be generated from the GPIF II Designer tool. The new file should replace this one. • Contains the structure and constants that are passed to API calls in the **uvc.c** file to start and to run the GPIF II state machine.
uvc.c	• Main source file for UVC application. Changes are needed when modifying the code to support controls other than brightness and PTZ, and when modifying to add support for different video streaming modes. • Contains the following functions: • **Main**: Initializes the FX3 device, sets up caches, configures the FX3 I/Os, and starts RTOS kernel • **CyFxApplicationDefine**: Defines the two application threads that are executed by the RTOS • **UVCAppThread_Entry**: First application thread function, calls initialization functions for internal blocks of FX3, enumerates the device, and then handles video data transfers • **UVCAppEP0Thread_Entry**: Second application thread function, waits for events that indicate UVC-specific requests are received and calls corresponding functions to handle these requests • **UVCHandleProcessingUnitRqts**: Handles VC requests targeted to Processing Unit capabilities • **UVCHandleCameraTerminalRqts**: Handles VC requests targeted to Input Terminal capabilities • **UVCHandleExtensionUnitRqts**: Handles VC requests targeted to Extension Unit capabilities

File	Description
	• **UVCHandleInterfaceCtrlRqts**: Handles generic VC requests not targeted to any terminal or unit • **UVCHandleVideoStreamingRqts**: Handles VS requests to change streaming mode • **CyFxUVCAppInDebugInit**: Initializes FX3's UART block for printing debug messages • **CyFxUVCAppInI2CInit**: Initializes FX3's I^{2}C block for image sensor configuration • **CyFxUVCAppInInit**: Initializes FX3's GPIO block, Processor Block (GPIF II is a part of the processor block), sensor (sets configuration to 720p 30fps), USB block for enumeration, Endpoint configuration memory for USB transfers and DMA channel configuration for data transfers from GPIF II to USB • **CyFxUvcAppGpifInit**: initializes the GPIF II state machine and starts it • **CyFxGpifCB**: Handles CPU interrupts generated from the GPIF II state machine • **CyFxUvcAppCommitEOF**: Commits the partial DMA buffers produced by the GPIF II state machine to FX3 RAM • **CyFxUvcAppInDmaCallback**: Keeps track of outgoing video data from FX3 to Host • **CyFxUVCAppInUSBSetupCB**: Handles all control requests sent by Host, sets events indicating UVC-specific requests have been received from Host, detects when streaming is stopped • **CyFxUVCAppInUSBEventCB**: Handles USB events such as suspend, cable disconnect, reset, and resume • **CyFxUVCAppInAbortHandler**: Aborts streaming video data when any error or streaming stop request is detected • **CyFxAppErrorHandler**: Error handler function. This is a placeholder function for you to implement error handling if necessary • **CyFxUVCAddHeader**: Adds UVC header to the video data during active streaming
cyfxuvcdscr.c	• Contains the USB enumeration Descriptors for the UVC 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.
uvc.h	• Contains switches to modify the application behavior to turn ON/OFF the PTZ support, debug interface, and debug prints for frame counts. • Contains constants that are used in common by the **uvc.c, cyfxuvcdscr.c, camera_ptzcontrol.c**, and **sensor.c** files.
cyfx_gcc_startup.s	• This assembly source file contains the FX3 CPU startup code. It has functions that set up the stacks and interrupt vectors. • No changes needed.

The firmware first initializes the FX3 CPU and configures its I/O. Then, it makes a function call (**CyU3PKernelEntry**) to start the ThreadX Real Time Operating System (RTOS). It creates two application threads: **uvcAppThread** and **uvcAppEP0Thread**. The RTOS will allocate resources to run these application threads and schedule the execution of the application thread functions: **UVCAppThread_Entry** and **UVCAppEP0Thread_Entry** respectively. Figure 46 shows the basic program structure.

Figure 46. Camera Project Structure



5.1 Application Thread

The two application threads enable concurrent functionality. The **UVCAppThread** (application) thread manages video data streaming. It waits for stream events before streaming starts, commits DMA buffers after the streaming starts, and cleans up FIFOs after each frame or when streaming stops.

The firmware handles UVC-specific control requests (SET_CUR, GET_CUR, GET_MIN, and GET_MAX) over EP0, such as brightness, PTZ, and PROBE/COMMIT control. Class-specific control requests are handled by the **CyFxUVCApplnUSBSetupCB** (CB=Callback) function in the application thread. Whenever one of these control requests is received by FX3, this function raises corresponding events and immediately frees the main application thread to perform its other concurrent tasks. EP0Thread then triggers on these events to serve the class-specific requests.

5.2 Initialization

UVCAppThread_Entry calls **CyFxUVCApplnDebugInit** to initialize the UART debugging capability, **CyFxUVCApplnI2CInit** to initialize the I^{2}C block of FX3, and **CyFxUVCApplnInit** to initialize the rest of the required blocks, DMA channels, and USB Endpoints.

5.3 Enumeration

In the **CyFxUVCApplnInit** function, **CyU3PUsbSetDesc** function calls to ensure that FX3 enumerates as a UVC device. The UVC Descriptors are defined in **cyfxuvcdscr.c** file. These Descriptors are defined for an image sensor sending 16 bits per pixel using the uncompressed YUY2 format, with a resolution of 1280 x 720 pixels at 30 FPS. Refer to Section 2.3.1 if you need to change these settings.

5.4 Configuring the Image Sensor Through the I^{2}C Interface

The image sensor is configured using the I^{2}C master block of FX3. **SensorWrite2B**, **SensorWrite**, **SensorRead2B**, and **SensorRead** functions (defined in **sensor.c**) are used to write and read image sensor configuration over I^{2}C. The functions **SensorWrite2B** and **SensorWrite** call the standard API **CyU3PI2cTransmitBytes** to write data to the image sensor. The functions **SensorRead2B** and **SensorRead** call the standard API **CyU3PI2cReceiveBytes** to read data from the image sensor. For more details on these APIs, refer to the FX3 SDK API Guide.

5.5 Starting the Video Streaming

The USB Host application, such as the VLC Player or AMCap, or VirtualDub, sits on top of the UVC driver to display the video, to set the USB interface and the USB alternate setting combination to one that streams video (usually Interface 0 Alternate setting 1), and to send a PROBE/COMMIT control. This 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 FX3; FX3 is supposed to start sending the image data from the image sensor to the USB 3.0 Host. In the firmware, the **UVCAppThread_Entry** 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, FX3 does not need to transfer any data. Therefore, the GPIF II state machine need not be initialized to transfer data. Otherwise, all of the DMA buffers would fill before the Host application starts to pull data out of the DMA buffers, and FX3 would transmit a bad frame. Hence, the GPIF II state machine should be initialized only if there is a stream event.

When FX3 receives a stream event, the main application thread calls **CyFxUvcAppGpifInit** function to start the GPIF II state machine. Inside this function, the firmware loads the GPIF II parameters (register settings and waveform data) into FX3 memory using the **CyU3PGpifLoad** function. Then, firmware starts the GPIF II state machine using the **CyU3PGpifSMStart** function. Pass the **CyFxGpifConfig** structure defined in the **cyfxgpif2config.h** file as the parameter to the **CyU3PGpifLoad** function. Pass the start state name and start condition as arguments to the **CyU3PGpifSMStart** function. The start state (**START**) and the start condition (**ALPHA_START**) are defined in **cyfxgpif2config.h** file. The **cyfxgpif2config.h** file was generated from the GPIF II Designer tool as described in Section 3.6.

Note: The FX3 SDK API Guide contains detailed information about GPIF II-related functions, such as **CyU3PGpifLoad** and **CyU3PGpifSMStart**.

5.6 Setting Up DMA Buffers

The UVC spec requires adding a 12-byte header to each USB transfer (meaning each 16-KB DMA buffer in this application). However, the FX3 architecture requires that any DMA buffer associated with a DMA Descriptor must have a size that is a multiple of 16 bytes.

Reserving 12 bytes in a DMA buffer for the FX3 CPU to fill in would place the DMA buffer boundary at a non-multiple of 16 bytes. Therefore, the DMA buffer size should be 16,384 minus 16, not 12. This makes the DMA buffer size, not including the 12-byte header that the FX3 firmware adds, 16,384-16=16,368 bytes. The DMA buffer is ordered as the 12-byte header, the 16,368 video bytes, and finally 4 unused bytes at the end of the DMA buffer. This creates a DMA buffer that can utilize the maximum USB BULK burst size of 16 x 1024 byte packets, or 16,384 bytes.

5.7 Handling the DMA Buffers During Video Streaming

The **CyFxUVCApplnInit** function creates a manual DMA channel with callback notification for the consumer events. This notification is used to track the amount of data read by the Host. At the end of video frame transmission, the DMA channel is reset as a part of a cleanup process. It is safe to reset the DMA channel only after all the data has been drained out of the DMA buffers. If the DMA channel is reset with valid image frame data in the pipe, the data is lost. Therefore, this notification plays an important role in keeping the firmware operational.

The firmware handles the DMA buffers when streaming the data. In the main application thread, the FX3 firmware checks for a produced DMA buffer via **CyU3PDmaMultiChannelGetBuffer**. A DMA buffer becomes available to the FX3 CPU when the GPIF II producer DMA buffer is committed or if it is forcefully wrapped up by the FX3 CPU. During an active frame period, the image sensor is streaming data and GPIF II produces full DMA buffers. At this time, the FX3 CPU has to commit 16,380 bytes of data to USB.

At the end of a frame, usually the last DMA buffer is partially filled. In this case, the firmware must forcefully wrap up the DMA buffer on the producer side to trigger a produce event and then commit the DMA buffer to the USB with the appropriate byte count. The forceful wrap-up of the DMA buffer (produced by GPIF II) is executed in the GPIF II callback function **CyU3PDmaMultiChannelSetWrapUp**. The **CyFxGpifCB** callback function is triggered when GPIF II sets a CPU interrupt. As shown in the GPIF II state diagram, this interrupt is raised when the frame valid signal is deasserted. The hitFV variable is set at this time to indicate that the frame capturing from the image sensor has ended.

The UVC header carries information about the frame identifier and an end-of-frame marker. At the end of a frame, the FX3 firmware sets bit 1 and toggles bit 0 of the second UVC header byte (see “CyFxUVCAddHeader”).

The variables prodCount and consCount track every DMA buffer produce and consume event, respectively. These variables help keep track of the DMA buffers to ensure that all the data has been drained from the FX3 FIFO. The firmware uses these variables to reset the channel at the end of a frame transmission over USB.

5.8 Cleaning Up at the End of a Frame

At the end of a frame, the GPIF II state machine generates a CPU interrupt, which starts the chain of events described above. The firmware waits for the USB side to drain the DMA buffer data using a loop in the **UVCAppThread_Entry** function. As soon as hitFV (which is set by the GPIF callback function) is set, the last DMA buffer is committed to USB, and the prodCount equals the consCount, the firmware will do the following:

- Reset the DMA channel (which resets its FIFO)
- Toggle the UVC header FID bit
- Call the CyU3PGpifSMSwitch function in order to restart the GPIF II state machine at its START state
- Wait for FV to go HIGH again

5.9 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, or the USB Host may issue a reset or suspend request to FX3. All of these actions trigger the CY_FX_UVC_STREAM_ABORT_EVENT event (refer to the **CyFxUVCApplnAbortHdlr** function). This action does not always happen when there is no data in the FX3 FIFO. This means that proper cleanup is required. The firmware resets streaming-related variables, and resets the DMA channel in the **UVCAppThread_Entry** loop. It does not call the **CyU3PGpifSMSwitch** function because no streaming is required. The firmware then 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. This request is handled in the **CyFxUVCApplnUSBSetupCB** function under switch case CY_U3P_USB_TARGET_ENDPT and request CY_U3P_USB_SC_CLEAR_FEATURE.

5.10 Adding a “Debug” Interface

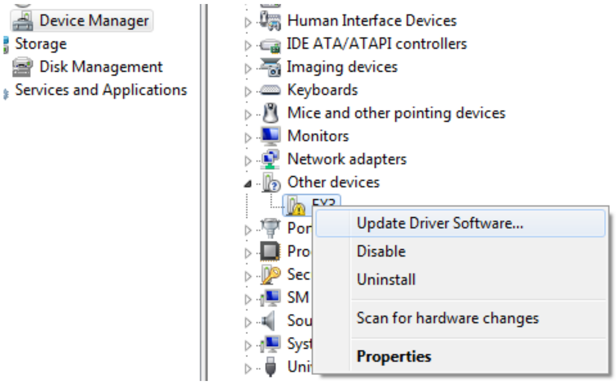
This section shows how to add a third USB interface to the camera application to use for debug or other custom purposes.

The #define USB_DEBUG_INTERFACE switch in the **uvc.h** file enables code in **cyfxuvcdscr.c, uvc.h** and **uvc.c** files. The example provided reads and writes image sensor registers over I^{2}C.

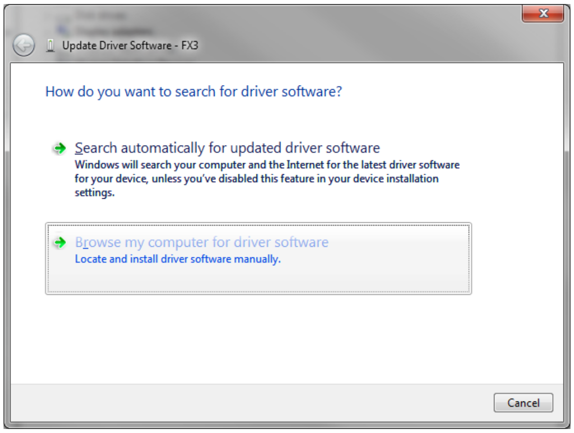
5.10.1 Debug Interface Details

Two bulk Endpoints are defined for this interface. EP 4 OUT is configured as the debug command BULK Endpoint, and EP 4 IN is configured as the debug response BULK Endpoint. When the firmware with the enabled debug interface runs, FX3 reports three interfaces during enumeration. The first two are the UVC control and streaming interfaces, and the third is the debug interface. The third interface needs to be bound to the CyUSB3.sys driver, which is provided as a part of the FX3 SDK. You can install the driver according to the following instructions. (A 64-bit Windows 7 system is used as an example. XP users will have to modify the .inf file to include the VID/PID and then use the modified .inf file to bind the cyusb3.sys driver to this interface.)

1. Open Device Manager, right-click on FX3 (or equivalent) under “Other devices”, and choose “Update Driver Software…”



2. Choose “Browse my computer for driver software” on the next screen



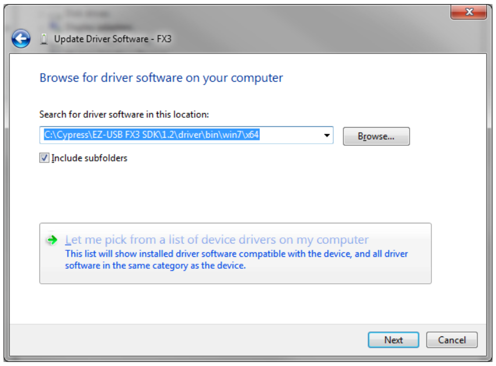
Update Driver Software - FX3

How do you want to search for driver software?

Search automatically for updated driver software
Windows will search your computer and the Internet for the latest driver software for your device, unless you've disabled this feature in your device installation settings.

Browse my computer for driver software
Locate and install driver software manually.

3. Choose “Let me pick from a list of device drivers on my computer” on the following screen



Update Driver Software - FX3

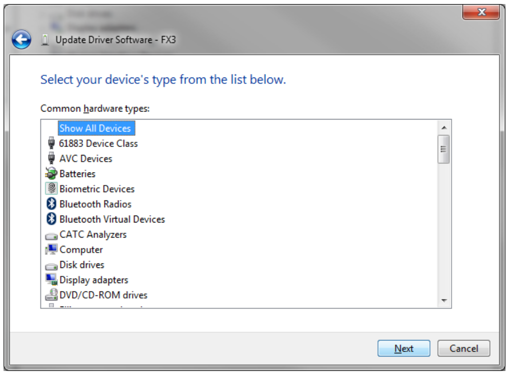
Browse for driver software on your computer

Search for driver software in this location:
C:\Cypress\EZ-USB FX3 SDK\1.2\driver\bin\win7\x64

☑ Include subfolders

Let me pick from a list of device drivers on my computer
This list will show installed driver software compatible with the device, and all driver software in the same category as the device.

4. Click “Next” on the following screen

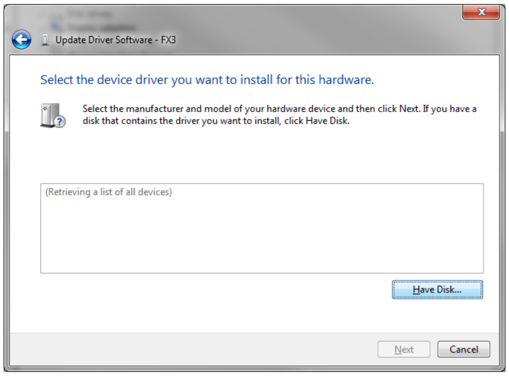


Update Driver Software - FX3

Select your device's type from the list below.

Common hardware types:

5. Click “Have Disk…” to choose the driver



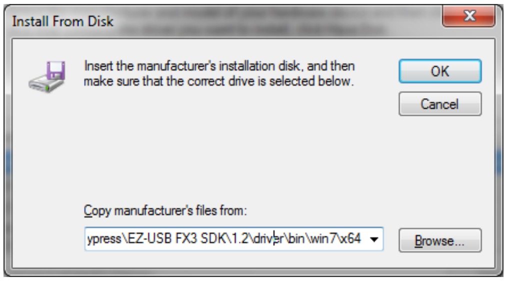
Update Driver Software - FX3

Select the device driver you want to install for this hardware.

Select the manufacturer and model of your hardware device and then click Next. If you have a disk that contains the driver you want to install, click Have Disk.

(Retrieving a list of all devices)

6. Browse for the cyusb3.inf file in the **<SDK installation>\<version>\driver\bin\<OS>\x86** or **\x64** folder, and click "OK"

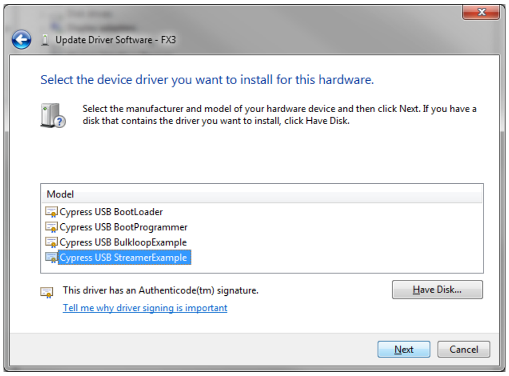


Insert the manufacturer's installation disk, and then make sure that the correct drive is selected below.

Copy manufacturer's files from:

ypress\EZ-USB FX3 SDK\1.2\driver\bin\win7\x64

7. Choose an OS version and click "Next" to install



Select the device driver you want to install for this hardware.

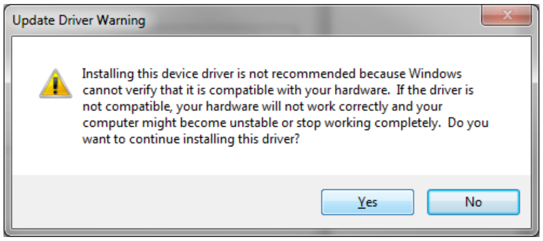
Select the manufacturer and model of your hardware device and then click Next. If you have a disk that contains the driver you want to install, click Have Disk.

Cypress USB BootLoader
Cypress USB BootProgrammer
Cypress USB BulkloopExample
Cypress USB StreamerExample

This driver has an Authenticode(tm) signature.

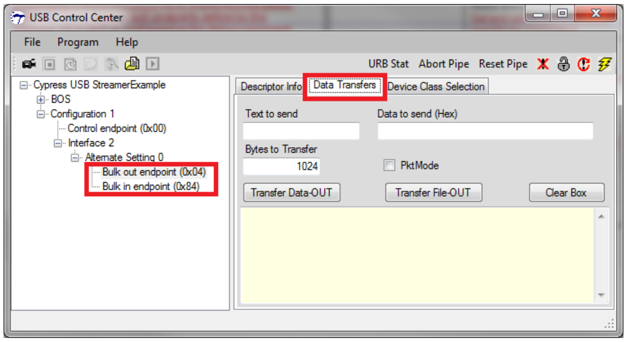
Tell me why driver signing is important

8. Click "Yes" on the warning dialog box if it appears



Installing this device driver is not recommended because Windows cannot verify that it is compatible with your hardware. If the driver is not compatible, your hardware will not work correctly and your computer might become unstable or stop working completely. Do you want to continue installing this driver?

After the driver is bound to the third interface, the device will show up in the Cypress USB Control Center application. You can access the FX3 firmware from here as a vendor-specific device. The current implementation of the debug interface allows reading and writing of the image sensor registers by using the EP4-OUT command and the EP4-IN response Endpoints. These Endpoints are accessed in the Control Center under <FX3 device name> → Configuration 1 → Interface 2 → Alternate Setting 0, as shown in the following figure. Use the Data Transfer tab to send a command or to retrieve a response after sending a command.



Cypress USB StreamerExample
BOS
Configuration 1
Control endpoint (0x00)
Interface 2
Alternate Setting 0
Bulk out endpoint (0x04)
Bulk in endpoint (0x84)

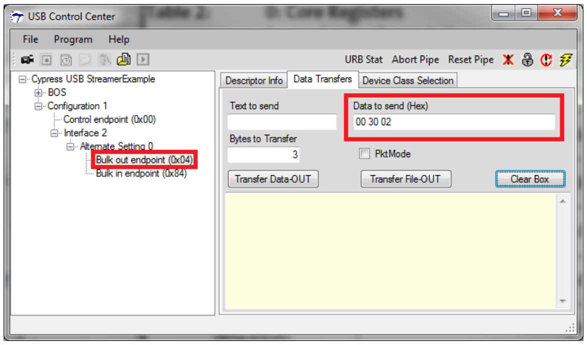
Bytes to Transfer: 1024

5.10.2 Using the Debug Interface

Four commands are implemented in the debug interface: single read, sequential read, single write, and sequential write. There is no error checking, so enter commands carefully. You can implement error checks to ensure proper functionality. The I^{2}C registers are 16 bits wide and are addressed using 16 bits.

Single Read:

1. Choose the command Endpoint and type the command in hex under the Data transfers. The format of a single read is 0x00 <register address high byte> <register address low byte>. The figure shows the read command for register address 0x3002. Do not use a space while typing in the hex data box. For example, click on the hex data field and type "003002".

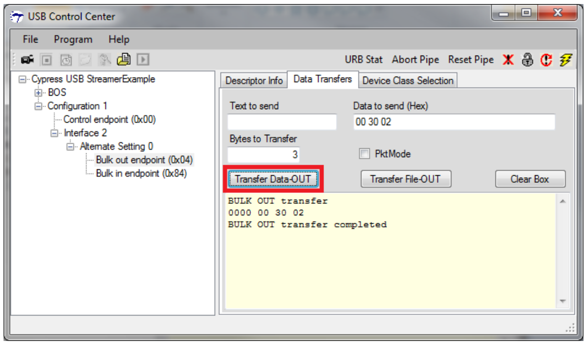


Cypress USB StreamerExample
BOS
Configuration 1
Control endpoint (0x00)
Interface 2
Alternate Setting 0
Bulk out endpoint (0x04)
Bulk in endpoint (0x84)

Data to send (Hex): 00 30 02

Bytes to Transfer: 3

2. Click "Transfer Data-Out" to send the command.



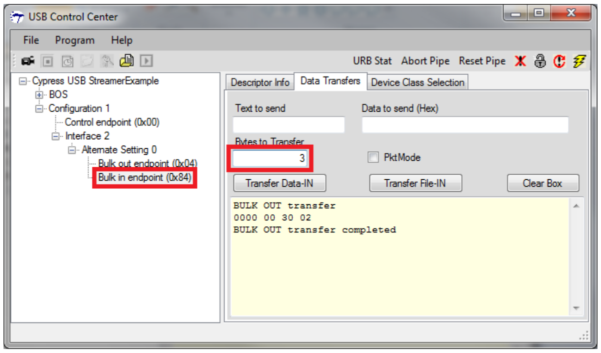
Cypress USB StreamerExample
BOS
Configuration 1
Control endpoint (0x00)
Interface 2
Alternate Setting 0
Bulk out endpoint (0x04)
Bulk in endpoint (0x84)

Data to send (Hex): 00 30 02

Bytes to Transfer: 3

```
BULK OUT transfer
0000 00 30 02
BULK OUT transfer completed
```

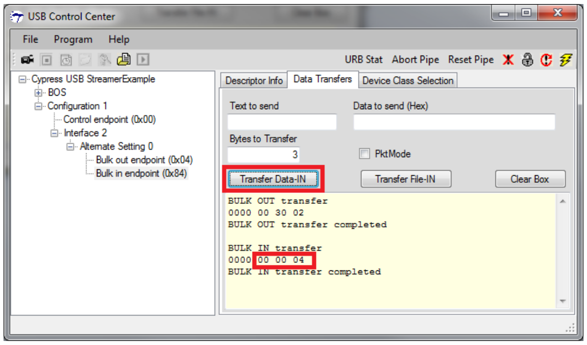
3. Choose the response Endpoint and set the Bytes to Transfer field to “3” to read out the response of the single read command.



Bytes to Transfer: 3

```
BULK OUT transfer
0000 00 30 02
BULK OUT transfer completed
```

4. Click “Transfer Data-IN” to receive the response

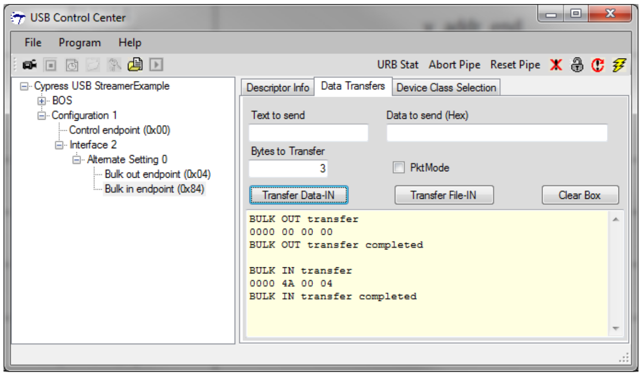


Bytes to Transfer: 3

```
BULK OUT transfer
0000 00 30 02
BULK OUT transfer completed

BULK IN transfer
0000 00 00 04
BULK IN transfer completed
```

5. The first byte of response is the status. Status=0 means the command passed; any other value means that the command failed. The following bytes indicate that the value of the register read back is 0x0004. This example shows a failed transfer in which the status is non-zero and the rest of the bytes are stale values from a previous transfer.



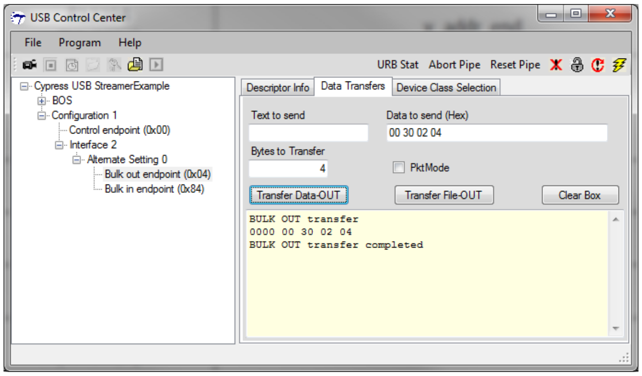
Bytes to Transfer: 3

```
BULK OUT transfer
0000 00 00 00
BULK OUT transfer completed

BULK IN transfer
0000 4A 00 04
BULK IN transfer completed
```

Sequential Read:

1. Choose the command Endpoint and type the command in hex under Data Transfers. The format of a sequential read is 0x00 <register address high byte> <register address low byte> <N>. The figure shows a read command for four (N=4) registers starting at register address 0x3002.

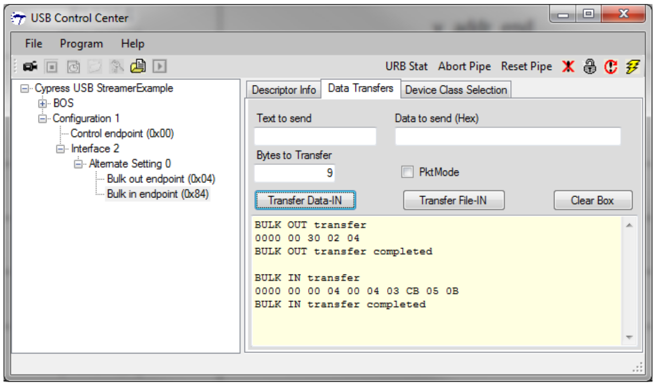


Data to send (Hex): 00 30 02 04

Bytes to Transfer: 4

```
BULK OUT transfer
0000 00 30 02 04
BULK OUT transfer completed
```

2. The Bytes to Transfer for the response is (N*2+1) = 9 for this case. The figure shows the values read by FX3.



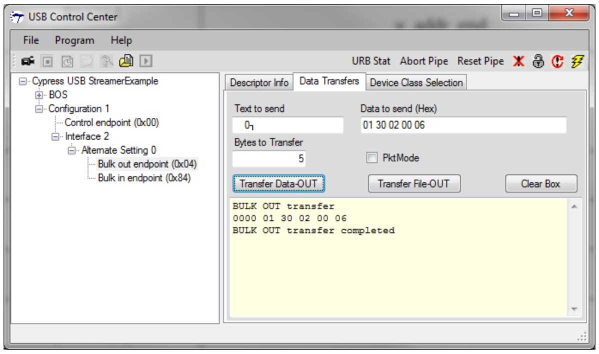
Bytes to Transfer: 9

```
BULK OUT transfer
0000 00 30 02 04
BULK OUT transfer completed

BULK IN transfer
0000 00 00 04 00 04 03 CB 05 0B
BULK IN transfer completed
```

Singe Write:

1. Choose the command Endpoint and type the command in hex under Data Transfers. The format of a single write is 0x01 <register address high byte> <register address low byte> <register value high byte> <register value low byte>. The figure shows the write command to write a value of 0x0006 at register address 0x3002.



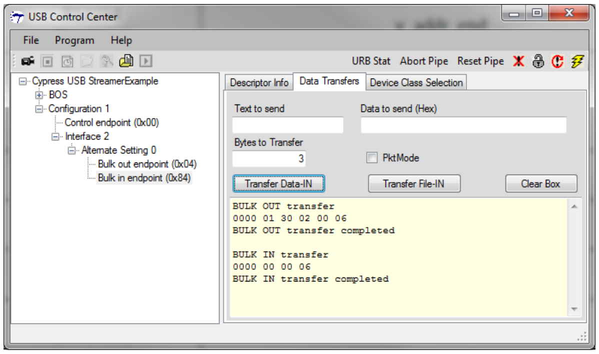
Text to send: 0

Data to send (Hex): 01 30 02 00 06

Bytes to Transfer: 5

```
BULK OUT transfer
0000 01 30 02 00 06
BULK OUT transfer completed
```

2. The response for a single write contains three bytes: <Status> <register value high byte> <register value low byte>. These register values are read back after writing, which means you will see the same values sent in the command.



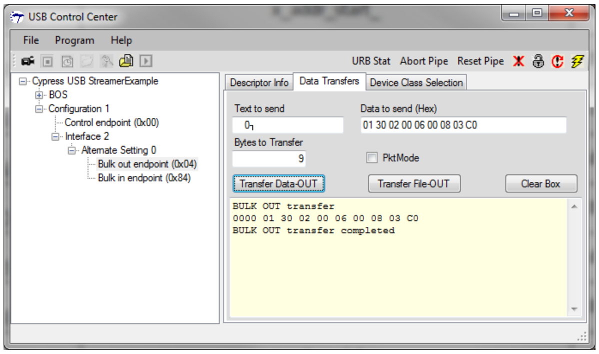
Bytes to Transfer: 3

```
BULK OUT transfer
0000 01 30 02 00 06
BULK OUT transfer completed

BULK IN transfer
0000 00 00 06
BULK IN transfer completed
```

Sequential Write:

1. Choose the command Endpoint and type the command in hex under Data Transfers. The format of a sequential write is 0x01 <register address high byte> <register address low byte> ((<register value high byte> <register value low byte>) * N times) to write N number of registers. The figure shows writing values 0x0006, 0x0008, and 0x03C0 to registers 0x3002, 0x3004, and 0x3006 sequentially (N=3).



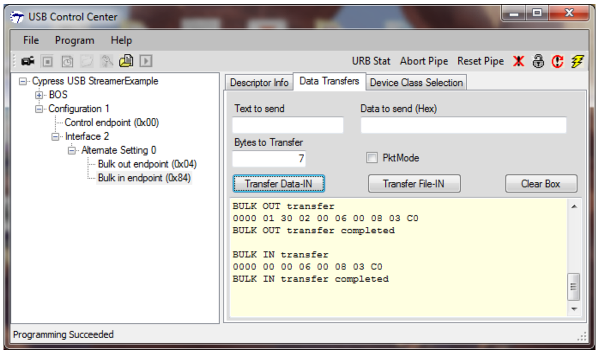
Text to send: 0

Data to send (Hex): 01 30 02 00 06 00 08 03 C0

Bytes to Transfer: 9

```
BULK OUT transfer
0000 01 30 02 00 06 00 08 03 C0
BULK OUT transfer completed
```

2. The response is <Status> (<register value high byte> <register value low byte>) * N values. For this example, the total bytes to transfer is (2*N+1) = 7.



Bytes to Transfer: 7

```
BULK OUT transfer
0000 01 30 02 00 06 00 08 03 C0
BULK OUT transfer completed

BULK IN transfer
0000 00 00 06 00 08 03 C0
BULK IN transfer completed
```

Programming Succeeded

6. Hardware Setup

The current project has been tested on a setup that includes the FX3 DVK and an Aptina image sensor MT9M114 along with an interconnect board. Details about how to obtain these components are available on the Cypress website at EZ-USB® FX3™ HD 720p Camera Kit and are summarized below:

6.1 Hardware Procurement

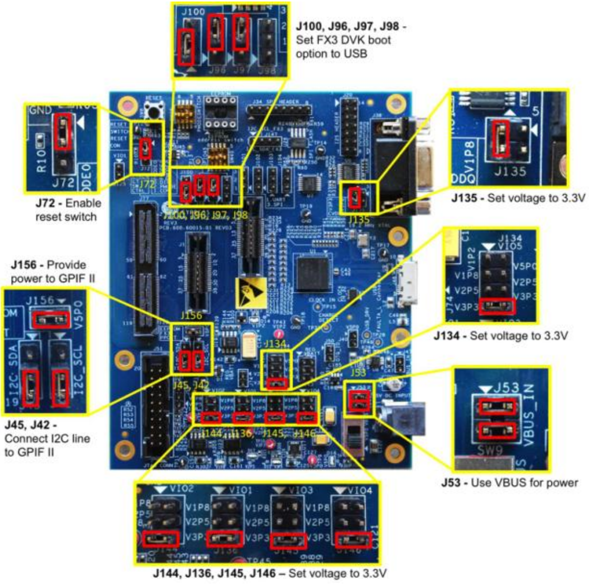
1. Sign an NDA with Aptina (email your request to fx3@cypress.com for an expedited response). If you have the NDA, send it to fx3@cypress.com. Cypress will provide the Aptina-specific **sensor.c** and **sensor.h** source files for the project after NDA verification.
2. Buy the Aptina MT9M114 Image Sensor headboard from Aptina Distributors.
3. Buy the EZ-USB FX3 Development Kit (CYUSB3KIT-001).
4. Contact Cypress at fx3@cypress.com to obtain an interconnection board.
5. Use a USB 3.0 Host-enabled computer to evaluate SuperSpeed performance.

6.2 FX3 DVK Board Setup

Prepare the board for testing the video application using these steps:

1. Configure the jumpers on the FX3 DVK board as shown in Figure 47. Do not load the jumpers that are not highlighted in the figure.

Figure 47. FX3 DVK Jumpers

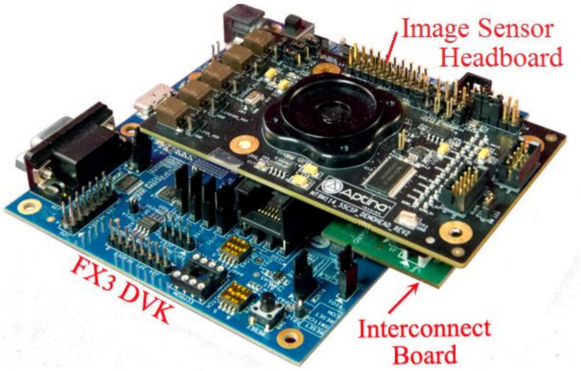

J100, J96, J97, J98 - Set FX3 DVK boot option to USB
J72 - Enable reset switch
J135 - Set voltage to 3.3V
J156 - Provide power to GPIF II
J134 - Set voltage to 3.3V
J45, J42 - Connect I2C line to GPIF II
J53 - Use VBUS for power
J144, J136, J145, J146 – Set voltage to 3.3V

2. Plug the interconnect board into FX3 DVK J77. The connector types on the interconnect board are

unique, and the sockets are keyed so that they will fit only in the correct orientation.

3. Connect the image sensor module to the interconnect board. Figure 48 shows the three-board assembly.

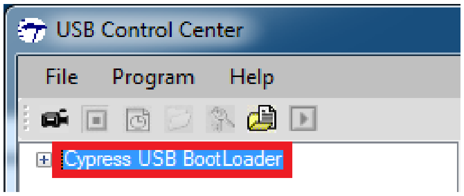
Figure 48. The Three-Board 720P Camera Assembly



4. Plug the FX3 DVK board into the USB computer using the USB 3.0 cable provided with the DVK.
5. Load the firmware into the board using the USB Control Center application provided as part of the FX3 SDK. For detailed instructions, see AN75705, “Getting Started with EZ-USB FX3”. Here are brief instructions:

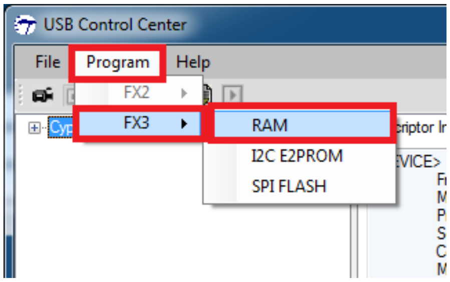
   a. Start the USB Control Center. When you plug in the FX3 EV board, it is recognized as a USB Bootloader (Figure 49).

Figure 49. FX3 Enumerates as a Bootloader



   b. Select Program>FX3>RAM and navigate to the **cyfxuvc.img** file provided with the attachment to this application note (Figure 50).

Figure 50. Loading Code into FX3 RAM



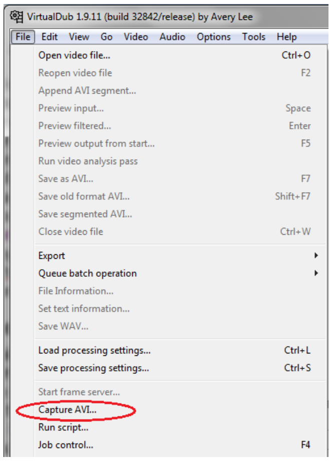
7. UVC-Based Host Applications

Various Host applications allow you to display and capture video from a UVC device. The VLC Media Player is a popular choice. Another widely used Windows application is AMCap. We recommend AMCap Version 8.0 because it has demonstrated stability when streaming, whereas later versions of AMCap can slow down stream rendering. Two additional Windows apps are VirtualDub (an open-source application) and Debut Video Capture software. Linux systems can use the V4L2 driver and VLC Media Player to stream video. The VLC Media Player is available on the web. Mac platforms can use FaceTime, iChat, Photo Booth, and Debut Video Capture software to create an interface with the UVC device to stream video.

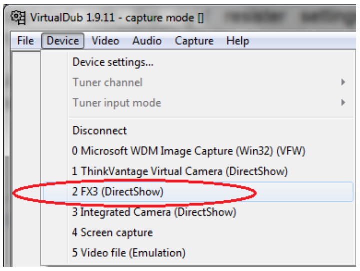
7.1 Running the Demo

A precompiled code image file for the 720p kit in Figure 48 is available at http://www.cypress.com/?docID=41913.
Run this code using the following steps:

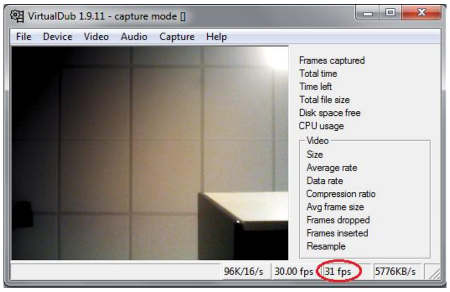
1. Load the precompiled firmware image into the FX3 UVC setup as described in Section 6.1.2.
2. At this point, the setup will re-enumerate as a UVC device. The operating system installs UVC drivers; no additional drivers are required.
3. Open the Host application (for example, VirtualDub).
4. Choose File → Capture AVI



5. Choose Device → FX3 (Direct Show), and this will start streaming images

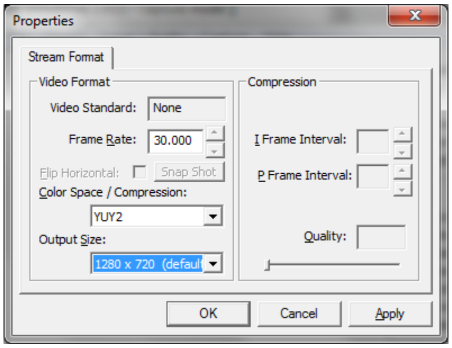

VirtualDub 1.9.11 - capture mode []
File
Device
Video
Audio
Capture
Help
Device settings...
Tuner channel
Tuner input mode
Disconnect
0 Microsoft WDM Image Capture (Win32) (VFW)
1 ThinkVantage Virtual Camera (DirectShow)
2 FX3 (DirectShow)
3 Integrated Camera (DirectShow)
4 Screen capture
5 Video file (Emulation)

6. The bottom right shows the actual frame rate



96K/16/s | 30.00 fps | 31 fps | 5776KB/s

7. Video → Capture Pin can be used to select among different supported resolutions and to see which resolution is currently active.



Video Standard: None

Frame Rate: 30.000

Flip Horizontal: □

Color Space / Compression: YUY2

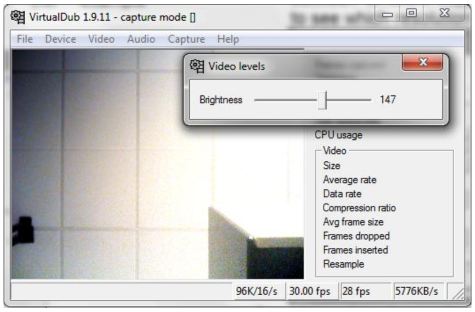
Output Size: 1280 x 720 (default)

I Frame Interval:

P Frame Interval:

Quality:

8. Video → Levels can be used to change brightness (change slider position to change value) or other supported control commands. Find additional control commands under Video->Capture Filter.



Brightness 147

96K/16/s | 30.00 fps | 28 fps | 5776KB/s

8. Troubleshooting

If you have a black screen, try these debugging steps:

1. There is a "DEBUG_PRINT_FRAME_COUNT" switch in the **uvc.h** file. Enable it to find out whether FX3 is streaming images. This switch will enable UART prints for frame counts. Short pins 1 and 2 on jumpers J101, J102, J103, and J104 on the FX3 DVK. This will connect the UART pins to the RS232 port on the FX3 DVK. Connect the UART port on the FX3 DVK to a PC via a UART cable or a USB-to-UART bridge. Open Hyperterminal, Tera Term, or another utility that gives access to the COM port on PC. Set the UART configuration as follows before starting transfers: 115,200 baud, no parity, 1 stop bit, no flow control, and 8-bit data. This should be sufficient to capture debug prints. If you do not see the incremental frame counter in the PC terminal program, there is probably a problem with the interface between FX3 and the image sensor (GPIF or sensor control/initialization).
2. If you see the prints of the incremental frame counter, the image data that is being sent out needs to be verified. A USB trace can show the amount of data being transferred per frame.
3. To check the total amount of data being sent out per frame, find the data packets that have the end-of-frame bit set in the header. (The second byte of the header is either 0x8E or 0x8F for the end-of-frame transfer). The total image data transferred in a frame (not including the UVC header) should be: width * height * 2.
4. If this is not the amount from the USB trace, there is an issue with the image sensor setting or with the GPIF interface.
5. If the total amount of image data is correct and a Host application still does not show any images, change the Host computer.
6. If the problems still persist, create a Cypress technical support case.

9. Connecting Two Image Sensors

Host applications, such as 3-D imaging or motion tracking, require FX3 to stream simultaneous video data from two image sensors. If the sensors are different, an FPGA must be inserted between the image sensor modules and FX3 to reconcile formats and synthesize a single video data channel. Such a setup with two different image sensors is beyond the scope of this application note.

A more practical approach uses identical image sensors. This approach is detailed in this section.

Figure 51 shows the connection details. The green blocks are inside the GPIF II and the red block is a part of an FX3 low-bandwidth peripheral (I^2C/GPIO). The idea is to synchronize the two sensors to achieve identical frame timing and have the GPIF II simultaneously input each of their 8-bit video streams on a 16-bit data bus.

Figure 51. Two Identical Image Sensors Connected to FX3

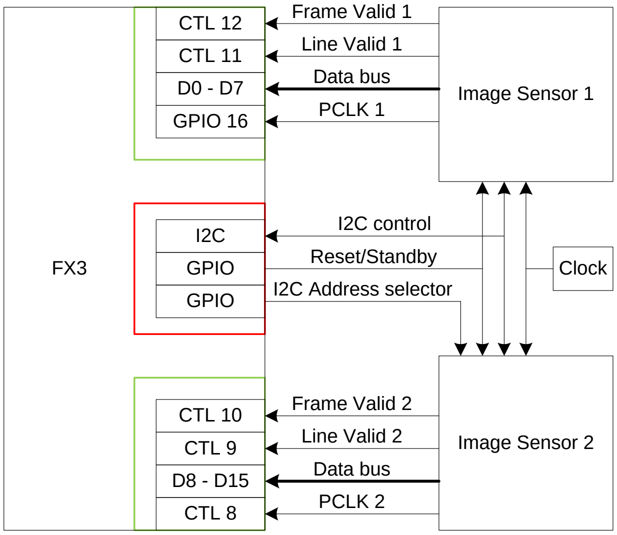


Figure 51 makes assumptions about the two image sensors:

1. The bus width of each image sensor is 8 bits, making the GPIF II bus width 16 bits.
2. The two image sensors are synchronized. This means that both image sensors use the same clock, LV and FV transitions, and pixel timing. In other words, frames from both image sensors could be exactly overlaid. Some image sensors have an external trigger input that can be used to synchronize two video streams. Other image sensors may use a different synchronization method. The applicable sensor datasheet gives details.
3. Both image sensors use I^{2}C for configuration. The image sensor used in this application note requires I^{2}C control registers to be written at exactly the same time to achieve synchronization between the image sensor modules. This is accomplished by controlling the I^{2}C address of one of the sensors using FX3 GPIO pins. FX3 configures both image sensors simultaneously using I^{2}C writes. FX3 reads from individual sensors by switching to a different I^{2}C address on the image sensor with the configurable address pins.
4. Each sensor's Reset signals should be driven by the same FX3 GPIO output pin. Similarly, each sensor's Standby pin, if present, should share another FX3 GPIO output pin.
5. Automatic configurations are disabled on the image sensors. For example, features such as auto exposure, auto gain, and auto white balance are turned OFF on both sensors. Turning them OFF ensures that the integration plus any image processing on the image sensors will take the same time. The result is that frames from both sensors are output from the image sensor simultaneously.

As shown in the connection diagram, Frame Valid 2, Line Valid 2, and PCLK 2 signals are connected to FX3, but they are not utilized by the GPIF II block because the image sensors are assumed to be synchronized. These signals are connected to FX3 so it can monitor the signals to check the accuracy of the synchronization between the image sensors during debug and development.

9.1 Transferring the Interleaved Image over UVC

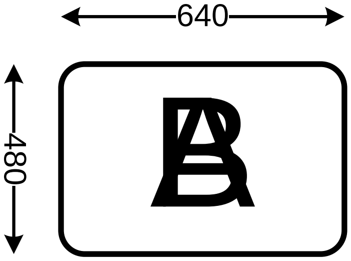
The UVC specification cannot discern how many image sensors are used in a transmitted image. It deals only with one set of image parameters: frame width and height and total number of bytes per frame. To accommodate multiple image sensors, the UVC driver must read Descriptors that have been modified so that the driver can perform and pass internal consistency checks. If these consistency checks fail, the UVC driver cannot pass image data to the Host application and the application fails. The Descriptors need to be modified so that the extra frame is accounted for in a manner that looks like a single image sensor to the UVC driver.

Two examples illustrate how this is done.

9.1.1 Example 1: Two 640 x 80 monochrome sensors.

Two image sensors provide 640 x 480 monochrome (1 byte per pixel) data. FX3 simultaneously receives two complete images per frame, as shown in Figure 52.

Figure 52. What FX3 Receives From Two Image Sensors



The Descriptors still report a 640 x 480 image size. The doubled data size is accommodated by placing the A image into the Y data and the B image into the U and V data.
The bytes per frame can be calculated as:

Bytes per frame = Bytes per pixel x Number of image sensors x Resolution

Where:

Resolution = Width in pixels x Height in pixels

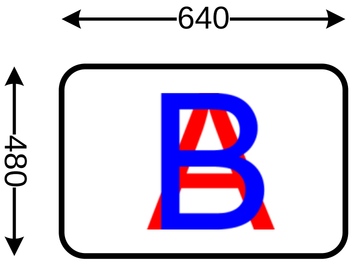
For this example:

Bytes per frame = 1 x 2 x 640 x 480 = 614,400 bytes

9.1.2 Example 2: Two 640 x 480 color sensors

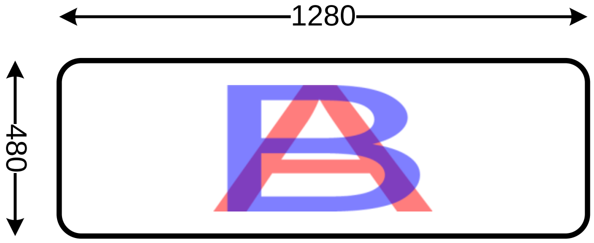
Consider two color sensors that send YUY2 data with 640 x 480 resolution. FX3 receives two complete color images per frame (Figure 53).

Figure 53. FX3 Receives Two Color Images



The only difference from Example 1 is that each pixel in the YUY2 format now uses two bytes instead of one. To accommodate the pixel doubling, the reported image size is doubled (Figure 54).

Figure 54. Reported Image Size



Because each pixel now occupies two bytes, the data per frame is:

Bytes per frame = 2 x 2 x 640 x 480 = 1,228,800 bytes

Note that any arbitrary frame width and height could be reported as long as they reflect the pixel doubling. Doubling only one dimension simplifies the downstream math. Doubling the width, as opposed to the height, has the advantage of overlaying the vertical lines, which simplifies application image processing.

These Descriptor modifications allow double images to be transported from the image sensors to the Host application in an interleaved fashion: Any two consecutive bytes are from different image sensors.

The above modifications pass the UVC driver consistency checks, allowing the driver to pass the video data to Host applications. The video streamed in this manner is not meant to be comprehended when viewed directly. You can use a standard UVC Host application to perform a sanity check of the implementation. However, the images are streamed in a non-standard manner, so the applications will not display them correctly. A custom application needs to be built to separate these images and to view and calculate useful information from the interleaved video.

9.2 Firmware Modification Checklist

As a summary, when modifying the given example firmware, take the following steps to ensure that all of the required changes have been made:

- Image sensor control: The example has reference I^{2}C code as the control interface. This may need modifications or may require a different kind of interface.
- Additional code to control the GPIO that acts as an image sensor selector for the control interface. When FX3 writes to the sensors used in this application note, it is a multicast message to both image sensors, but FX3 requires exclusive access when it reads the I^{2}C registers from the image sensors.
- Additional code to control the GPIO that controls the standby or low-power mode of the image sensors.
- Changes to the PROBE/COMMIT control structure to reflect the modified frame rate and frame resolution.
- Changes to the UVC-specific high-speed and SuperSpeed USB configuration Descriptors for frame and format. This is to report the modified frame resolution, frame rate, and width-to-height ratio.
- Changes to the GPIF II Descriptor to increase the bus width and to update the counter limits according to the bus width. These changes are made in the GPIF II Designer to generate the header file. Remember to copy this newly generated file in your project to ensure that the changes take effect.

10. Summary

This application note describes how an image sensor conforming to the USB Video Class can be implemented using Cypress's EZ-USB FX3. Specifically, it shows:

- How the Host application and driver interact with a UVC device
- How the UVC device manages UVC-specific requests
- How to program an FX3 interface using GPIF II Designer to receive data from typical image sensors
- How to display video streams and change camera properties in a Host application
- How to add a USB interface to the UVC device for debugging purposes
- How to find Host applications available on different platforms, including an open-source Host application project
- How to connect multiple image sensors, synchronize them externally, and stream from them simultaneously over the UVC
- How to troubleshoot and debug the FX3 firmware, if required.

11. About the Author

Name:	Karnik Shah
Title:	Applications Engineer

12. Document History

Document Title: How to Implement an Image Sensor Interface Using EZ-USB® FX3™ in a USB Video Class (UVC) Framework - AN75779

Document Number: 001-75779

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3591590	SHAH	04/19/2012	New Application Note
*A	3646722	SHAH	06/14/2012	Changed application note title to match the scope of the new version of application note Added firmware project Added explanation of the firmware project Added the UVC application related details, Revised the functional block diagram Moved the steps to generate the GPIF II Descriptor using the GPIF II Designer tool to Appendix A Added section in Appendix to show users how to modify a the given GPIF II -Designer project Clarified certain topics with explicit information Updated the all the links in the document to point to the correct locations within and outside the document Removed references to AN75310 Removed references to the Slave FIFO application note
*B	3938382	SHAH	03/20/2013	Updated application note title Updated the Software Version required Updated the Abstract with information on newly added features Updated TOC Added more description on how UVC application works Added general block diagram of UVC class requests Modified the description of the file structure based on the new structure in the associated project for ease of use Added a section on USB Descriptors for UVC application Added details section on UVC class requests Added description of sample control requests (brightness and PTZ) included as new features in the updated associated project Added a section on the UVC streaming requests Added a section on the UVC video format and UVC header insertion Updated the firmware application description section with appropriate content to reflect the changes in the associated firmware project Added a section describing an optional debug interface implemented as a new feature and documentation on how to use this new interface Added a section on the hardware setup instructions Added a section on Host applications available in market for viewing video over UVC Added a section on basic troubleshooting Updated the GPIF II state machine design steps in the Appendix A to accommodate the updated state machine used in the associated project
*C	4041078	SHAH	06/27/2013	Updated the abstract to include reference to two image sensor interface Added section to explain how to connect two image sensors Added section on how to transfer data with two image sensors connected

Revision	ECN	Orig. of Change	Submission Date	Description of Change
				Added section called Firmware Modification Checklist To improve quality of graphics, replaced Hardware Setup screen shot on page 20
*D	4125334	SHAH	09/27/2013	Rewrite and overhaul of application note

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
Optical Navigation Sensors	cypress.com/go/ons
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

EZ-USB® and FX3™ are registered trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.
Use may be limited by and subject to the applicable Cypress software license agreement.