

AN75779

Interfacing an Image Sensor to EZ-USB® FX3™ in a USB video class (UVC) Framework

Author: Karnik Shah
Associated Project: Yes
Software Version: FX3 SDK1.1
Related Application Notes: None

If you have a question, or need help with this application note, contact the author at SHAH@cypress.com.

Cypress's EZ-USB® FX3™ is a USB 3.0 peripheral controller with a general programmable interface, called GPIF II, which can connect easily to a range of devices. This application note describes a video streaming application compatible with USB video class (UVC) where FX3 streams images from an image sensor to a USB host.

Table of Contents

Introduction	1
Designing the GPIF II interface	2
Image Sensor Interface	2
Requirements for GPIF II Descriptor	3
Pin Mapping of Image Sensor Interface	3
GPIF II DMA Capabilities.....	4
GPIF II State Machine Design	7
Correlation between Image Sensor Waveforms, Data Path and State Machine	8
Integration of the GPIF II Descriptor	9
USB Video Class Requirements	9
Creating a DMA channel to stream data from GPIF II to USB.....	10
Execution path of the firmware application.....	10
Application Threads.....	10
Enumeration.....	11
Starting the streaming	11
Handling the buffers during streaming.....	11
Clean Up	11
Aborting the streaming	12
Firmware Example Project Details	12
Summary.....	13
About the Author	13
Appendix A.....	14
Designing with the GPIF II Designer	14
Creating the Project.....	14
Choosing the Interface Definition	14
Drawing the State Machine on the Canvas	17

Basic Drawing Actions on the Canvas	17
Drawing Image Sensor Interface State Machine	20
Editing the GPIF II interface details.....	25
Document History.....	26

Introduction

EZ-USB® FX3™ is the USB 3.0 peripheral controller that enables developers to add USB 3.0 device functionality to any system. FX3 has a fully configurable General Programmable Interface (GPIF™ II), which can interface with virtually any processor, ASIC, image sensor or FPGA. UVC is a USB standard class that allows a video streaming device to be connected to a USB host to stream video like a webcam using standard UVC driver. This application note discusses how to design an application, which is compatible with UVC, by interfacing FX3 and an image sensor with an interface that has the following signals: frame valid, line valid, pixel clock, and 8bit to 32bit parallel data bus.

This application has been designed for an image sensor with an 8-bit parallel data interface, 16bits per pixel, YUY2 color space, 1280 x 720 pixels resolution, 30 frames per second, active high frame/line valid signals and positive clock edge polarity. With some minor customizations, a variety of sensors with a similar interface can also be interfaced. Editing the GPIF II interface details section in Appendix A describes how to change the data bus width as an example of customization.

This application discusses the flow of data from the image sensor to the internal buffers and then to the USB3 host,

while explaining the parts of design required to implement this data flow. These parts include designing an interface waveform for the GPIF II, creating a USB descriptor to enumerate FX3 as a UVC device, and connecting the GPIF II to the USB via a DMA channel for streaming. An overview of the whole system is shown in the system block diagram [Figure 1](#). A host application like AMCap talks through the UVC driver to configure the image sensor over the video control interface and receive video data over the video streaming interface. FX3 firmware translates the video control settings to the Image Sensor over GPIO or I2C interface. FX3 DMA channel streams data from the Image Sensor to internal buffers where UVC header is added to the image data. This video data is then sent to the video streaming endpoint.

Designing the GPIF II interface

To design an interface waveform, an understanding of the interface requirements and DMA capabilities of FX3 is required. Additionally, the class specific requirements of

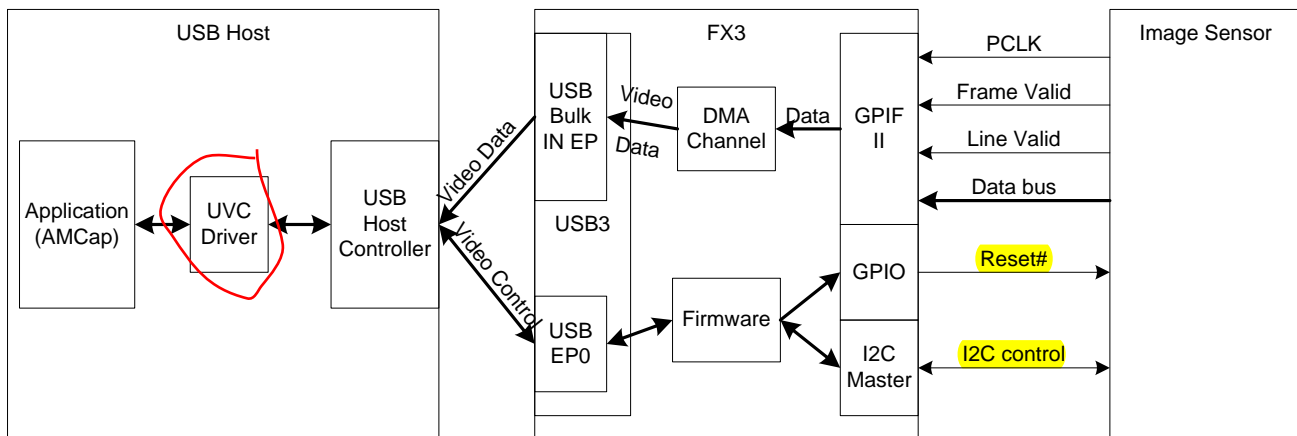
USB video class (UVC) also need to be addressed to make the application compatible with UVC standards.

Image Sensor Interface

A standard image sensor interface looks like the interface between the image sensor and FX3 shown in [Figure 1](#). Usually, the image sensor requires a reset signal from the FX3 controller. This can be handled by using an FX3 GPIO. Image sensors typically use an I²C connection to allow a controller to configure the image sensor parameters. The I²C block of FX3 can act as an I²C master to configure the image sensor with the correct parameters. The various signals that are associated with transferring an image are as follows: (these unidirectional signals are from image sensor to FX3)

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 – 8 to 32 bit data lines for image data

Figure 1. System block diagram



[Figure 2](#) shows the [timing diagram](#) of the FV, LV, PCLK and Data signals. The FV signal is asserted 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 when the data is driven by the image sensor. This data can be 8bit to 32bit simultaneous transfer from the image sensor.

Note This parallel data bus width is defined by the data width as defined in the image sensor datasheet. Do not confuse the data bus width with the bit depth/resolution of the image sensor.

FX3 GPIF II bus can be configured only in 8, 16 or 32bit data interfaces. Hence, the data bus width from FX3 should be set such that it is larger than or equal to the image sensor data width. This size will be the interface

width over GPIF II. If the interface width is larger than the image sensor data width, the additional padded bits transferred should be discarded by the end application – usually the software running on host PC. For example, if the image sensor sends out 12bits in parallel, the interface width should be set to 16bits and the unconnected 4bits will behave as padding. The time between each line is called the horizontal blanking. During horizontal blanking no data is transferred and the LV signal is de-asserted. Once all the lines are transferred from the image sensor, the FV signal is de-asserted, and it remains de-asserted for the time equivalent to the vertical blanking. A slave state machine for this interface can be implemented using the GPIF II to receive the image data from the image sensor.

Requirements for GPIF II Descriptor

The previous section described a typical image sensor interface. This section describes the design requirements, which can be derived from the image sensor interface, for the GPIF II state machine. Looking at the image sensor interface timing diagram we can infer that:

- For image resolution C x R, where C is the number of columns and R is the number of rows of pixels, and P is the bytes per pixel (including the padding), an image sensor continuously sends CxP bytes of image data; R times with short pauses. So once a line starts, GPIF II design should be able to move CxP bytes without a break as there is no flow control on the image sensor interface.

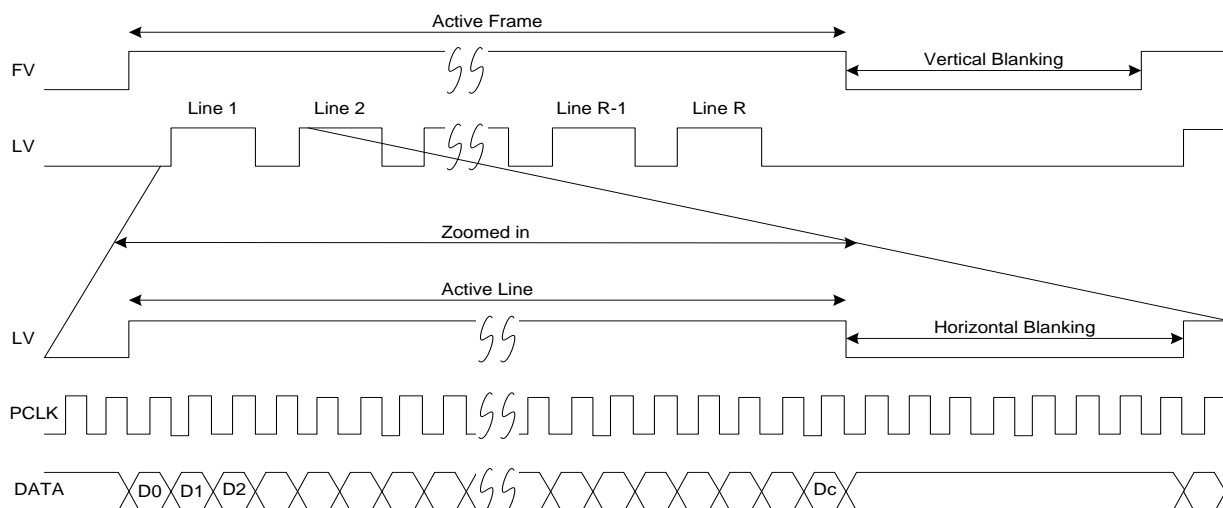
- At the end of the frame, a signal must be generated from GPIF II to the FX3 CPU so that the CPU indicates to the host that the current frame data transfer is complete.

- FX3 can only operate in 8bit, 16bit or 32bit modes. Hence, the data bus width for the GPIF II must be set such that it is greater or equal to the parallel data coming out of the image sensor.

An example GPIFII design for an image sensor has been created using the GPIFII Designer tool and is attached with this application note for reference. Steps to build it are explained in detail in the “Designing with the GPIF II designer” section (Appendix A).

The following section shows the pin mapping used in the GPIFII implementation of the image sensor interface in this application note.

Figure 2. Image Sensor Interface Timing Diagram



Pin Mapping of Image Sensor Interface

The pin mapping used in the GPIFII implementation of the parallel image sensor interface in this application note is shown in Table 1. Also shown are the GPIO pins and other serial interfaces (UART/SPI/I2S) available when you use the GPIF II design available with this application note.

Table 1. Pin Mapping for Parallel Image Sensor Interface Descriptor

EZ-USB FX3 Pin	Synchronous Parallel Image Sensor Interface with 16-bit Data Bus	Synchronous Parallel Image Sensor Interface with 8-bit Data Bus
GPIO[28]	LV	LV
GPIO[29]	FV	FV
GPIO[0:7]	DQ[0:7]	DQ[0:7]
GPIO[8:15]	DQ[8:15]	Unused
GPIO[16]	PCLK	PCLK
GPIO[17:27]	Available as GPIOs	Available as GPIOs

GPIO[33:45]	Available as GPIOs	Available as GPIOs
GPIO[46]	GPIO/UART_RTS	GPIO/UART_RTS
GPIO[47]	GPIO/UART_CTS	GPIO/UART_CTS
GPIO[48]	GPIO/UART_TX	GPIO/UART_TX
GPIO[49]	GPIO/UART_RX	GPIO/UART_RX
GPIO[50]	GPIO/I2S_CLK	GPIO/I2S_CLK
GPIO[51]	GPIO/I2S_SD	GPIO/I2S_SD
GPIO[52]	GPIO/I2S_WS	GPIO/I2S_WS
GPIO[53]	GPIO/SPI_SCK /UART_RTS	GPIO/SPI_SCK /UART_RTS
GPIO[54]	GPIO/SPI_SSN/UART_CTS	GPIO/SPI_SSN/UART_CTS
GPIO[55]	GPIO/SPI_MISO/UART_TX	GPIO/SPI_MISO/UART_TX
GPIO[56]	GPIO/SPI_MOSI/UART_RX	GPIO/SPI_MOSI/UART_RX
GPIO[57]	GPIO/I2S_MCLK	GPIO/I2S_MCLK

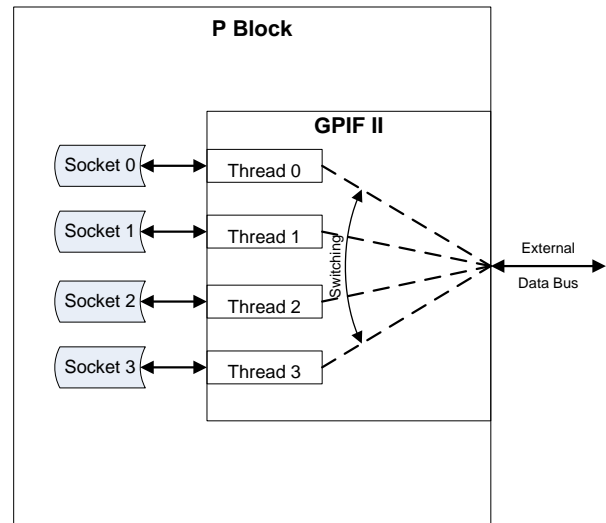
Note For the complete pin mapping of EZ-USB FX3, please refer to the datasheet “EZ-USB FX3 SuperSpeed USB Controller.”

The following section explains the DMA configuration used to achieve video streaming from FX3’s GPIF II to the USB interface.

GPIF II DMA Capabilities

The GPIF II block, as a part of P (processor port) block, is capable of running up to 100MHz with 32 bits of data (400MBps). To transfer the data to internal buffers, GPIF II uses threads connected to DMA producer sockets of the P block. The sockets point to a DMA descriptor that sets DMA buffer size, count and chaining sequence. The GPIF II has four threads, and these four threads can be associated with any four sockets of the 32 P block sockets. Default settings are used for this application. The default threads are associated with the sockets of the same number. As shown in Figure 3, GPIF II has 4 threads through which it can transfer data. The switching between these threads is accomplished by in-state thread switching as explained in the “GPIF II State Machine Design” section.

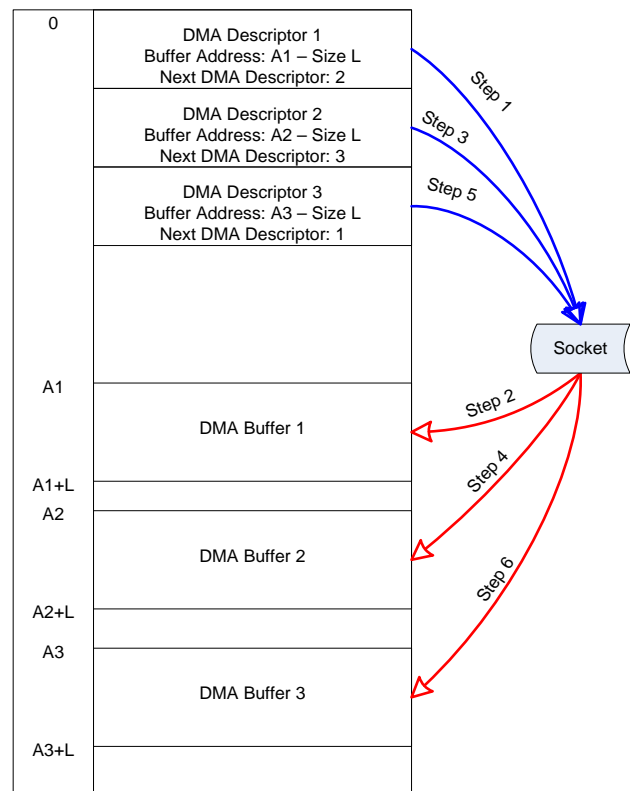
Figure 3. Default settings for GPIF II threads and P block sockets



The DMA data transfer mechanism is shown in Figure 4. The example is of a DMA channel where there is a single producer socket (consumer side not shown) and three buffers, of length L, chained in a linear circular loop. The figure shows the internal memory of FX3. The left column in the diagram shows the memory offset and the right column shows what is stored in that memory location. The red arrows (data path) indicate how the buffers will be accessed by the socket. The blue arrows (connecting the DMA descriptor chain to the socket) show how the socket loads the descriptors from memory. The following

execution steps show the process or mechanism of data moving from a socket to the internal buffers. The steps are also marked in [Figure 4](#) for reference.

Figure 4. Example to Explain DMA Data Transfer Operation



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

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

Step 3: Load DMA descriptor 2 as pointed to by the current DMA descriptor 1. Get the buffer location (A2),

buffer size (L) and next descriptor (DMA descriptor 3) information. Go to step 4.

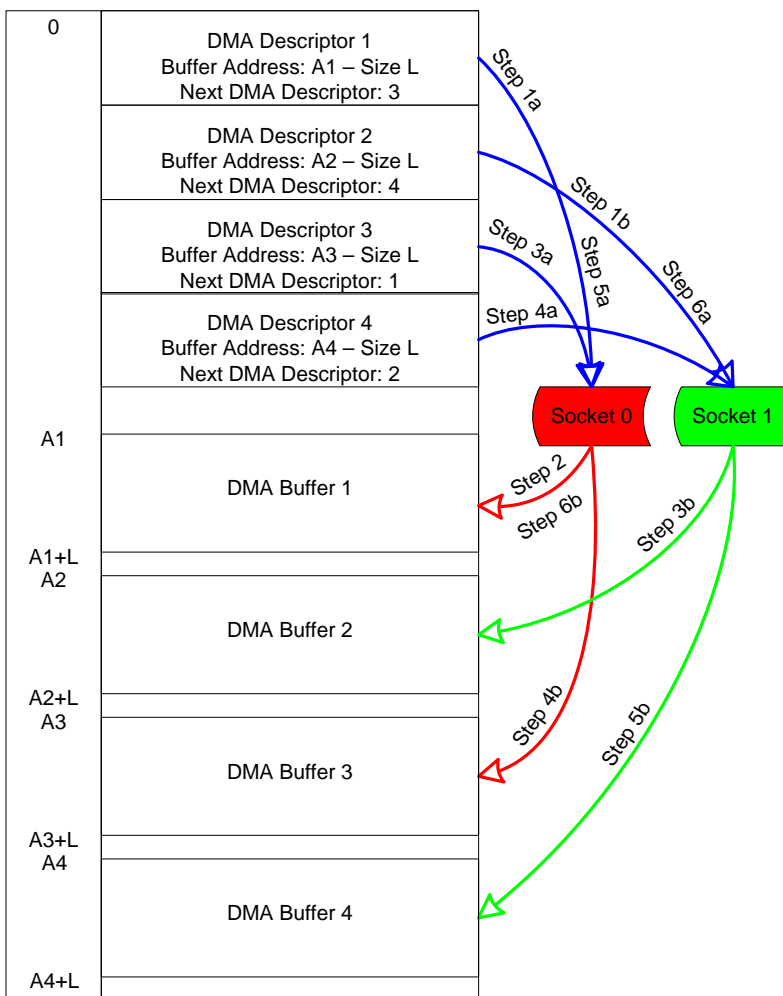
Step 4: Transfer data to the buffer location starting at A2. After transferring buffer size L amount of data, go to step 5.

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

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

Notice that in this execution path, the socket is not continuously transferring data. The socket pauses to get configuration in between two buffers. There is a non zero (typically 1us) delay when the socket switches buffers. For only one socket implementation, the first requirement from the [requirement section](#) can fail at buffer boundaries if the buffers are not a multiple of line length.

One obvious solution is to use buffer sizes equal to a multiple of the line size. Under such implementation, if the resolution of the image changes, the buffer size needs to be changed. Setting buffer size equal to line size will not give maximum throughput either. USB3 allows a maximum of 16 bursts of 1024 bytes over bulk endpoint. This feature should be utilized for maximum throughput. To enable USB burst of 16KB, the USB DMA buffer size (consumer side) should be set to 16KB and so the P block DMA buffer size (producer side) will be similar – explained in [USB video class requirements](#) section. An alternate method can be implemented where two sockets are used from GPIF II side to write data in an interleaved fashion. Since switching of sockets for GPIF II has no time latency, the sockets can be switched when the buffer associated with the active socket is full i.e. socket switching at the exact buffer boundary as compared to line boundary. The data transfer using dual sockets is described in [Figure 5](#) with the execution steps labeled. Socket0 and Socket1 access to DMA buffers is differentiated with red and green colored arrows (data paths for individual sockets/threads) respectively. The ‘a’ and ‘b’ parts of each step occur simultaneously. This parallel operation of the hardware helps mask the buffer switching time and allows GPIF II to stream data continuously into internal memory.

Figure 5. Dual Socket Data Transfer Architecture


Step 1: At initialization of the sockets, both Socket0 and Socket1 will load the DMA Descriptor 1 and DMA Descriptor 2 respectively.

Step2: As soon as the data is available, Socket0 transfers data to DMA buffer 1. The transfer length is L and at the end of this transfer goes to step 3.

Step 3: GPIF II switches the thread and, hence, the socket for data transfer. So, Socket1 now starts transferring data to DMA buffer 2 and at the same time Socket0 will load the DMA descriptor 3. By the time Socket1 finishes transferring L amount of data, Socket0 will be ready to transfer data into DMA buffer 3.

Step 4: GPIF II now switches back to the original thread. So, Socket0 will now transfer the data of length L into DMA buffer 3. At the same time, Socket1 will load the DMA Descriptor 4 and be ready to transfer data to DMA buffer 4. After Socket0 finishes transferring the data of length L, go to step 5.

Step 5: GPIF II switches thread and Socket1 transfers data of length L into DMA buffer 4. Socket0 loads DMA descriptor 1 and gets ready to transfer data into DMA buffer 1 at the same time. Notice that step 5a is same as step 1a, except that Socket1 is not initializing but transferring data simultaneously.

Step 6: GPIF II switches back the thread and Socket0 starts transferring data of length L into DMA buffer 1. It is assumed that by now, the buffer is empty (consumed by a consumer socket – UIB socket as USB is the usual consumer). At the same time, Socket1 will load the DMA descriptor 2 and be ready to transfer data into DMA buffer 2. The cycle now goes to Step 3 in the execution path.

It is assumed that the buffers are consumed by the consumer (USB), so that when the execution path comes to fill the buffers, there is no data loss. GPIF II state machine can implement the in-state thread switching at buffer boundary via counters. The counter value needs to be set according to the buffer size on the producer side.

If the consumer is not fast enough, the Sockets would drop data as the buffer, which the producer tries to write to, is inaccessible. If there is data drop, the counters continue to increment while the data is not transferred to the buffer, resulting in premature thread switching. This implies that the whole data transfer streaming breaks. Thus, it is necessary to put a cleanup mechanism at the end of every frame so that such misalignment does not carry forward into the next frame transfer. This is described in the [Clean Up](#) section.

With an execution path described in [Figure 5](#), the data transfer ends when a frame ends with four possible states:

Socket0 has transferred a full buffer

Socket1 has transferred a full buffer

Socket0 has transferred a partial buffer

Socket1 has transferred a partial buffer

In case either Socket0 or Socket1 has filled a buffer partially, the CPU needs to commit this partial buffer so that the consumer socket can consume this data. The next step is creating a state machine based on all the understanding from previous sections. This is described in the following sections.

GPIF II State Machine Design

The GPIF™ II block is a versatile state machine with 256 states. In each state, a multitude of actions including and not limited to driving multiple control lines, sending or receiving data and/or address, and sending interrupts to internal CPU can be performed. State transitions can have internal or external signals like DMA ready and Frame/Line Valid as deciding factors.

To begin designing, a point should be chosen on the image sensor waveform where the state machine is to start. The start of a frame is the most convenient position and is indicated by a positive transition on the frame valid signal. GPIF II can only detect the state of an input and not the edge. Positive edge detection of frame valid signal is necessary to make sure that FX3 does not start transferring frame data from the middle of a frame. So, the start of the state machine (see [Figure 6](#) for reference) should wait for FV to be de-asserted (State 1), and then it should wait for the FV signal to be asserted (State 2). Also, the counter should be initialized to keep track of the

data transferred, so that state machine can switch threads (State 2).

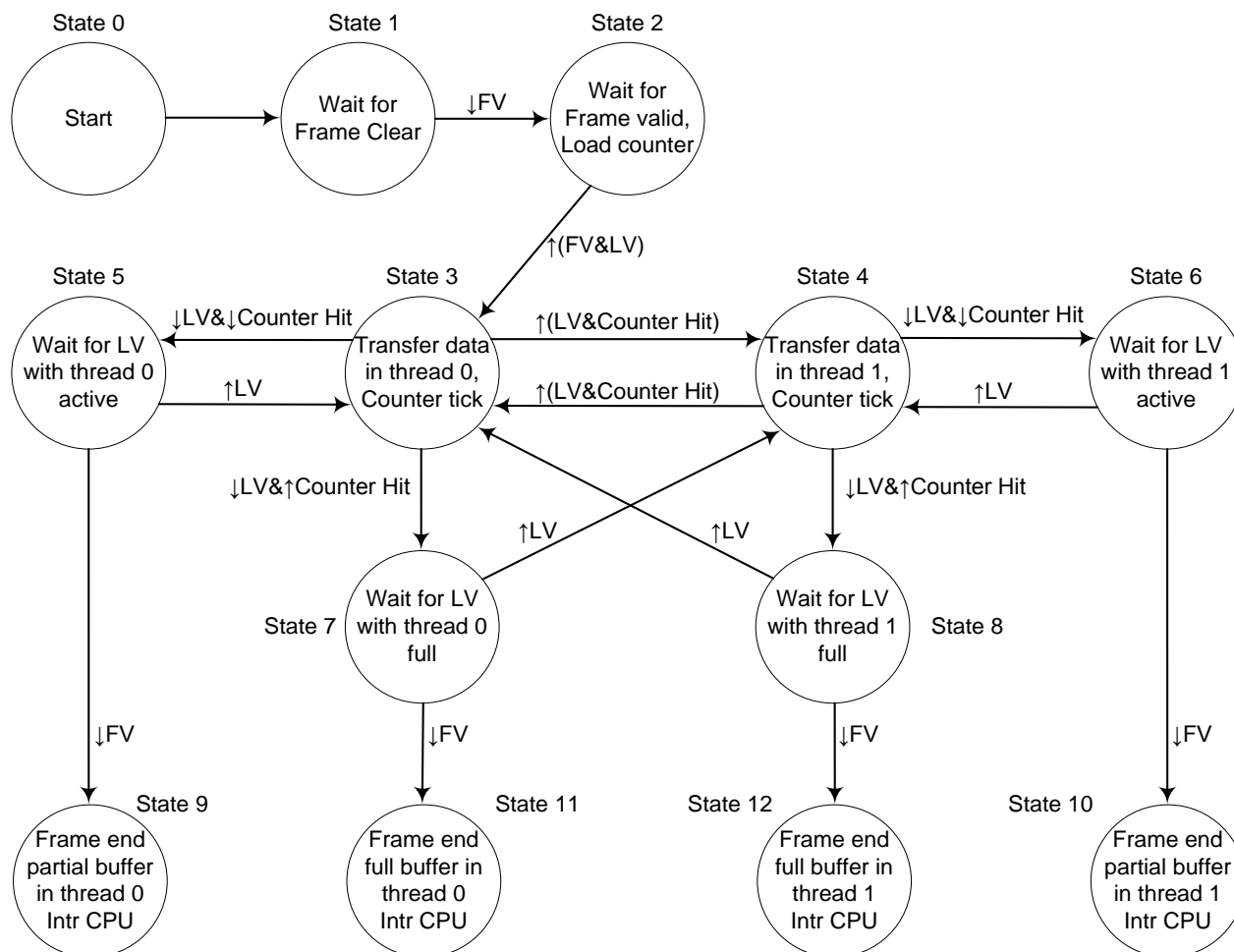
Then, the state machine should transfer data into the threads (State 3 and State 4). While the GPIF II state machine is transferring the data into a thread, two conditions can happen:

1. The current line that is being transferred is completed and so the LV signal is de-asserted. The data transfer can have two different states for the threads.
 - a. The data transfer ended at the buffer boundary. In this case, the next data transfer needs to happen in the next thread. (State 7 and State 8)
 - b. The data transfer did not end at the buffer boundary. In this case, the state machine waits for LV signal to get asserted back, to start transferring data into same thread. (State 5 and State 6)
2. When the data for a line is still being transferred, the buffer associated with the current thread gets full. So the state machine needs to switch to transferring the data through the next thread. (transitions between State 3 and State 4)

Under any circumstances, whenever the GPIF II state machine has to switch the threads, the counter needs to be re-initiated. The GPIF II state machine can automatically reload the counter value; however, it takes one clock cycle for the reload to occur. Hence, the actual limit of the counter should be 1 less. Equation 1 helps calculate the value of the counter.

$$\text{Equation 1} \quad \text{count} = \left(\frac{\text{producer_buffer_size}(L)}{\text{Interface_width}} \right) - 1$$

For example: The UVC example as described has a producer buffer size of 16368 bytes. Its interface width is 8bits. Hence, the counter value to be set is 16367 (0x3FEF). If the interface width is 16bits, this value should be 8183 (0x1FF7). If the interface width is 32bits, this value should be 4091 (0xFFB). It is important to note here that the buffer size is independent of the interface width; however, the value of the count that goes in GPIF II descriptor design is dependent on both. [Editing the GPIF II interface details](#) section in [Appendix A](#) shows how to change the data bus width using the GPIF II Designer tool.

Figure 6. GPIF II State Machine Diagram


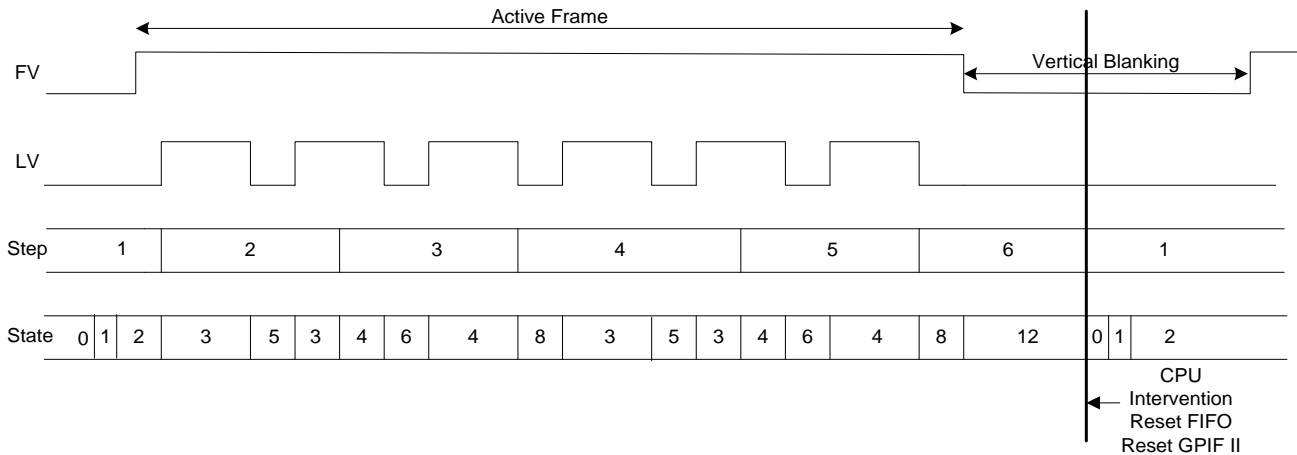
Note With such a state machine design, the CPU will get an interrupt at the end of every frame. This can be used to call a call back function that will enable CPU to handle several different tasks including but not limited to the following:

1. Commit last buffer if there is a partial buffer at the end of a frame. (State 9 and State 10)
2. Let the consumer socket drain all data and then reset the channel and reset the GPIF II state machine. (Resetting the state machine to State 0 so that it can start streaming data from start of a frame)
3. Handle any special application specific tasks to indicate the consumer of the change of frame. For example: UVC implementation requires the change of frame indication as a toggling bit in the 12 byte header.

Correlation between Image Sensor Waveforms, Data Path and State Machine

The last section showed the GPIFII state machine design for an image sensor interface. This section explains the correlation between the image sensor interface, the GPIFII state machine and the DMA data path diagrams. Figure 7 shows how the following three components: the image sensor waveform described in Figure 2, the FX3 data path described in Figure 5 and the state machine described in Figure 6 are correlated. To explain the correlation, a dummy image sensor waveform is taken as an example. In this example, the buffer size L is equal to 1.5 times the line size C. R = 6. FV and LV waveforms show the image sensor timing. “Step” shows how the data is being routed from Figure 5. “State” shows how the state machine changes states in response to FV and LV from Figure 6. With CPU intervention the data pipe is cleared and GPIF II state machine is reset.

Figure 7. Image Sensor Interface, Data Path Execution and State Machine Correlation



Refer to Appendix A to see how to create the GPIFII state machine described in the previous sections using the GPIF II designer tool. The sample project created in the “Designing with the GPIF II designer” section (Appendix A) is attached with this application note for reference. The project name is **ImageSensorInterface.cyfx** and is stored under the folder **ImageSensorInterface.cydsn**.

Integration of the GPIF II Descriptor

Once a GPIFII Designer project is created as explained in Appendix A, building the project will generate a header file “cyfxgpiif2config.h”. To integrate the GPIF II descriptor into the project that was created, the cyfxgpiif2config.h file is copied into the eclipse project directory. This file has a structure called “CyFxGpiifConfig”. In the firmware application that is included with this application note, this structure needs to be passed in the function “`uvc.c/UVCAppThread_Entry`→`gpiif_interface.c/gpiif_init`→`gpiif_interface.c/CyU3PGpiifLoad`” to load the GPIF II state machine descriptor into the memory. Once loaded, the function “`gpiif_interface.c/gpiif_init`→`gpiif_interface.c/CyU3PGpiifSMStart`” starts the GPIF II state machine. “CyU3PGpiifSMStart” requires the start state as the parameter. The start state declaration can be found from the cyfxgpiif2config.h file. Please refer to the FX3 SDK API GUIDE for more information on how to use the GPIF II related functions including “CyU3PGpiifLoad” and “CyU3PGpiifSMStart”.

Note Convention used here for indicating the location of the function in a file is filename/function. → is used to indicate the function call hierarchy. Example: If a function F1 in file1.c is calling a function F2 in file2.c, it is indicated as file1.c/F1→file2.c/F2.

So far this application note explained the details of the image sensor interface and the GPIFII state machine design. The next sections explain the details of the USB Video Class (UVC) requirements, DMA channel to stream data, and the details of the firmware that supports UVC.

USB Video Class Requirements

There is an **associated example firmware project** available with this application note that supports the UVC class. This section explains the details of the example firmware that supports the UVC class.

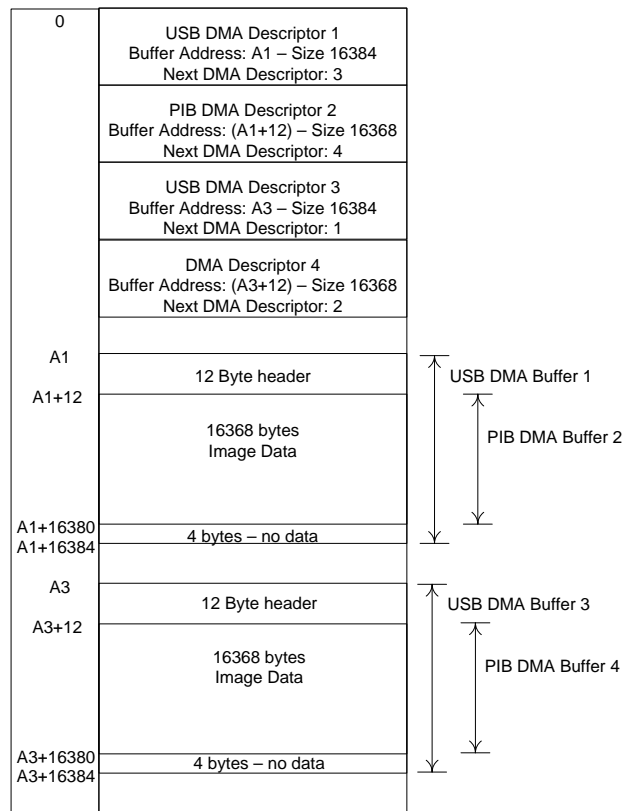
The UVC class requires a device to enumerate with the UVC class specific descriptors. The documentation for UVC standard can be found at http://www.usb.org/developers/devclass_docs. The UVC class requires a **12 byte header** that describes some properties of the image data being transferred. For example - it has a new frame bit which needs to be toggled every frame. There is an error bit that can be set to indicate a problem in the streaming of the current frame. **This header is required for every USB transfer.** So, for a burst of 16KB, one header needs to be included in the 16KB USB payload. To accommodate this header, FX3 CPU needs to intervene to commit every packet. This action is accomplished in the firmware in function `uvc.c/CyFxDMACallback`, whenever there is a CY_U3P_DMA_CB_PROD_EVENT indicating GPIF II filled a buffer.

There is a special case when at the end of a frame; GPIF II may not have filled the last buffer. In this case, firmware must **wrap up** (`uvc.c/CyFxDMACallback`→`gpiif_interface.c/CyFxDMACommitEOF`→`gpiif_interface.c/CyU3PDmaMultiChannelSetWrapUp`) the partial buffer that was being filled. Then CPU should commit this packet with the number of bytes present in the buffer.

The 12 byte header required by the UVC class also impacts the size of the DMA buffer that is associated with the P block socket. The architecture of FX3 is designed such that any buffer associated with a descriptor needs to be a multiple of 16 bytes. To accommodate the 12 byte header offset, the buffer pointed by the DMA descriptor for the P block (PIB DMA descriptor) must start from 12 bytes after the byte address where the buffer pointed by the USB DMA descriptor starts. Since we are using 16KB

(16,384) as the size of the USB buffer to maximize the throughput, the PIB buffer should be at max 16K bytes – 12 bytes (16,372 bytes). However, due to the restriction, that the buffer size must be a multiple of 16 bytes, PIB buffer needs to be set as $(16,372 - (16,372 \text{ modulo } 16)) = 16,368$. This means that USB will transfer $16,368 + 12 = 16,380$ bytes in one transaction except when the buffer is partially filled at the end of a frame. Figure 8 explains this setup. In the firmware, this is done in the `uvc.c/CyFxCVAppInit` function. Details are explained in the next section.

Figure 8. USB and PIB DMA descriptors for UVC



Creating a DMA channel to stream data from GPIF II to USB

The 12 byte offset and 16 byte boundary condition in the PIB buffer can be configured via the header and footer settings of a DMA channel configuration. The `uvc.c/CyFxCVAppInit` function initializes GPIO for sensor reset, PIB block to enable GPIF II function, I2C block to configure the image sensor, the sensor to start streaming image data, USB block to enumerate as UVC device, USB endpoints to stream data to USB host, and a DMA channel to connect the data pipe from GPIF II to USB endpoint.

The requirements for the DMA channel that we need to implement are:

1. USB buffer size should be 16,384 to maximize throughput
2. PIB buffer should start at 12 bytes after USB buffer to accommodate the 12 byte UVC header
3. PIB buffer size should be 16,368 to make sure buffer size is less than USB buffer size, is a multiple of 16 bytes and allows 12 bytes UVC header.
4. There should be two interleaved producer sockets (PIB is the producer here) to make sure buffer switching time does not cause data drop
5. One consumer socket (USB is consumer) that can read all data off the memory in the order it was stored by the producer.
6. The data needs to be committed by the FX3 CPU.

This is implemented by setting the `dmaMultiConfig` structure in the `uvc.c/CyFxCVAppInit` function and creating a `MANUAL_MANY_TO_ONE` channel. The USB endpoint that will stream data to the USB3.0 host also needs to be configured to enable a burst of 16 over the 1024 byte bulk endpoint. This is set via `endPointConfig` structure passed in the `uvc.c/CyFxCVAppInit` → `uvc.c/CyU3PSetEpConfig` function.

Execution path of the firmware application

It is now time to understand the UVC application from functional standpoint after understanding the interface, interface design, UVC requirements, and DMA channel details for streaming. This section describes the flow of the firmware to explain how the application works. The FX3 firmware runs on top of a ThreadX Real Time Operating System (RTOS). In the `uvc.c/main` function the firmware defines the configuration of FX3 I/Os. Here FX3 is configured to enable use of particular peripherals and set the data width of the GPIF II interface to either 32bits or less. These settings can be changed on the fly later but care must be taken to avoid any I/O conflicts. Then the firmware creates two threads. The main application thread, which runs in `uvc.c/UVCAppThread_Entry` function and control request handler thread, which runs in `uvc.c/UVCAppEP0Thread_Entry` function. These are software application threads that run on the ThreadX operating system; they should not be confused with the GPIF II hardware thread implementation.

Application Threads

The reason to have these as separate threads is to enable concurrent functionality. The main thread is responsible to wait for a stream event to occur before the streaming starts, commit buffers once the streaming has started and then clean up the FIFO after a frame is transmitted or if the streaming stops. However, when this thread is waiting,

the firmware also needs to handle some UVC class specific control requests on EP0 like probe/commit control, set_cur, get_cur, get_min, and get_max. Any class specific control request is handled in `uvc.c/CyFxCVAppInUSBSetupCB` function. This function is implemented in the main application thread. Whenever one of these control requests are received by FX3, this function will set corresponding events and immediately free the main application thread to do its other concurrent tasks. EP0Thread will trigger on these events to serve the control request.

Enumeration

The `uvc.c/UVCAppThread_Entry` calls `uvc.c/CyFxCVAppInDebugInit` to initialize the UART debugging capability, `uvc.c/CyFxCVAppInI2CInit` to initialize the I2C block of FX3, and `uvc.c/CyFxCVAppInInit` to initialize the rest of the required blocks, DMA channels and USB endpoints. In the `uvc.c/CyFxCVAppInInit` function, `CyU3PUsbSetDesc` function calls 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 16bits per pixel on average, uncompressed YUY2 image format (4:2:2 down sampled), 1280 x 720 pixels at 30 frames per second. Please refer to the UVC specification if you need to change these settings.

Starting the streaming

USB host application (like AMCAP), which sits on top of the UVC driver to stream the images, sets the USB interface and USB alternate setting to the one that streams the video (usually Interface 0 Alternate setting 1), and sends a probe/commit control. This is an indication of the stream event. On stream event, the USB host application will start requesting image data from FX3 and FX3 is supposed to start sending the image data from the Image Sensor to the USB3.0 host. In the firmware, `uvc.c/UVCAppThread_Entry` function has in infinite for loop. When there is no streaming, the application thread will wait in this for loop till there is a stream event.

Note If there is no stream event, FX3 does not need to transfer any data. So, GPIF II state machine need not be initialized for transferring data. Otherwise, all the buffers will be full before the host application starts pulling data out of the buffers and FX3 will end up transmitting a bad frame. Hence, GPIF II state machine should be initialized only if there is stream event.

When FX3 receives a stream event, the main application thread will start the GPIF II state machine. There are three functions to do this. On power up, the GPIF II waveform descriptor is not loaded into the memory. So, the firmware loads the GPIF II waveform in the memory by using `uvc.c/UVCAppThread_Entry`→`gpiif_interface.c/gpiif_init`→`gpiif_interface.c/CyU3PGpifLoad` function. Then, firmware starts the GPIF II state machine using `uvc.c/UVCAppThread_Entry`→`gpiif_interface.c/gpiif_init`→`g`

`pif_interface.c/CyU3PGpifSMStart` function. In case, the USB host utility is shutdown, or if the streaming is stopped while the power to FX3 is not cycled, GPIF II will still be running. In such cases, `uvc.c/CyU3PGpifSMSwitch` function is used to reset the GPIF II state machine so that it can start transferring data on the next start of frame.

Handling the buffers during streaming

When designing the DMA channel in the `uvc.c/CyFxCVAppInInit` function, the firmware created a manual channel with call back notifications for both the produce and the consume events. This is because as per the requirements mentioned in "Creating a DMA channel to transfer data from GPIF to USB" section, FX3 CPU needs to intervene every buffer.

In the DMA call back function, `uvc.c/CyFxDMACallback`, the firmware handles the buffers when streaming the data. The produce event occurs when the producer buffer (PIB buffer in our case) is committed from the GPIF II side or if it is forcefully committed by the FX3 CPU. During a frame valid period, the image sensor is streaming data and GPIF II will produce full PIB buffers. At this time, the FX3 CPU has to commit 16,380 bytes of data to the USB. However, at the end of a frame, usually the last buffer is partially filled. In this case, the firmware must forcefully wrap up the buffer on the producer side to trigger a produce event and then commit the buffer to the USB with appropriate byte count. The forceful wrap up of the PIB buffer is executed in the GPIF II call back function `uvc.c/CyFxCVAppInInit`→`gpiif_interface.c/CyFxCVAppInInit`→`gpiif_interface.c/CyU3PDmaMultiChannelSetWrapUp`. The `uvc.c/CyFxCVAppInInit` function is triggered when GPIF II sets a CPU interrupt. As shown in Figure 6 from the GPIF II State Machine Design section, this interrupt is caused when a frame valid signal is de-asserted (basically end of frame). It should be noted that `hitFV` variable is set to indicate that the frame has ended and it is used to distinguish between the two scenarios of produce event.

Note UVC header carries information regarding the frame identifier and end of frame marker. At the end of a frame the last significant bit of the second byte of the UVC header needs to be set to '1' (refer `uvc.c/CyFxCVAppInInit`→`gpiif_interface.c/CyFxCVAppInInit`→`gpiif_interface.c/CyU3PDmaMultiChannelSetWrapUp`) and at every start of a new frame, the second last significant bit of the second byte of the UVC header needs to be toggled (refer to clean up).

Every produce and consume event is also tracked by the firmware via `dmaDone` variable. This variable helps keep track of the buffers that were produced and consumed, to make sure that all the data has been drained from the FX3 FIFO. This is useful to know for the firmware so that it can reset the channel at the end of a frame.

Clean Up

At the end of a frame, the GPIF II state machine generates a CPU interrupt resulting in a call back function. This call back function helps facilitate committing the last buffer to the USB. Now, the firmware waits for the USB to drain all

the data in the for loop of `uvc.c/UVCAppThread_Entry` function. Here it checks for `hitFV` (which is set by the GPIF call back function) and `dmaDone` to reach '0' (`dmaDone` is decremented on each consume event). As soon as both the conditions are met, firmware cleans up the FIFO, resets the channel, toggles the UVC header bit for frame index and calls `uvc.c/CyU3PGpifSMSwitch` function. Looking at the state machine diagram in Figure 6, once the GPIF II state machine issues a CPU interrupt, it stops transferring any data. `uvc.c/CyU3PGpifSMSwitch` function switches the state of the GPIF II state machine to start and it can hence start streaming data on the next frame start event.

Aborting the streaming

The FX3 can be disconnected from the host, the USB host utility may be turned off or USB host may issue a reset to FX3. In any of these events, the streaming of the image data stops. All of these actions trigger the "CY_FX_UVC_STREAM_ABORT_EVENT" event (refer `uvc.c/CyFxUVCAppInAbortHdlr` function). This action does not always happen when there is no data in the FX3 FIFO. This means `dmaDone` can have a non Zero value. Firmware resets `dmaDone` to '0', cleans up the FIFO, and resets the channel in the `uvc.c/UVCAppThread_Entry` for loop. It will not call the `CyU3PGpifSMSwitch` function as there is no streaming required and then wait for the next streaming event to occur.

Firmware Example Project Details

This section explains the details of the different files and placeholders contained in the example firmware project associated with this application note.

The example **firmware project** incorporates the GPIFII descriptor for an image sensor interface and also supports UVC. To use this project, first install the FX3 SDK and then import the project into the Eclipse IDE workspace. Please note this is not a complete project and may not function as it is. Certain sections of code specific to the image sensor being used may need to be filled out if the application is such that FX3 needs to configure the image sensor. In case the configuration is handled by a different controller and FX3 only needs to stream the image data, no modifications may be required. The sections that need to be completed are pointed out below. The following are the main components of the firmware project:

cyfxgpif2config.h

This is the **header file generated by the GPIF II Designer tool**. It contains the GPIF II descriptor for the image sensor interface. The GPIF II Designer project is available with this application note. Use this GPIF II Designer project for any required customization to the descriptor.

gpif_interface.c

The `gpif_init()` function in this file loads and starts the GPIF II descriptor.

uvc.c file

This file illustrates the UVC application example. The video streaming is accomplished by configuring **a many-to-one manual DMA channel (two producer sockets at PIB side and one consumer socket at UIB side)**. The two producer sockets alternately transfer data from the sensor to the DMA buffer for consumption by USB. This allows continuous streaming of video. The UVC specific header is added to the video payload data in the callback function `CyFxDMAcallback()`. The callback function is called when a DMA produce event occurs.

cyfxuvcdscr.c file

This file contains the USB enumeration descriptors for the UVC application example. Using these descriptors, the FX3 device enumerates as a UVC device on the USB host. The class specific descriptors indicate the supported configurations of the image sensor to the host. These configurations include image resolution, frame rate, and video control support like brightness or contrast.

sensor.c file

This part of the firmware **is specific to the image sensor being used**. Certain sections of the code specific to the **sensor being used must be filled in if FX3 needs to control or configure the image sensor**. A placeholder is added for the `sensor_init()` function in the `sensor.c` file. Example implementations of I2C Write and Read commands to the sensor are provided. **The user must define the value of `SensorSlaveAddress` in the `sensor.h` header file.**

fx3.ld file

The FX3 SDK installation provides a **standard `fx3.ld` script** file that specifies the code memory, data memory, and heap memory locations in the internal RAM of FX3. These memory locations are different for the UVC firmware project, **so a separate `fx3.ld` file is included with the project**. Use this file instead of the `fx3.ld` provided with the SDK installation (in the `/firmware/common` directory).

Because the following components in the firmware example are specific to the GPIF II descriptor implementation, incorporate them as is:

DMA Channel Configuration

The P-to-U DMA channel must be set up as a many-to-one channel as shown in the `uvc.c` file in the firmware project. The firmware uses PIB sockets 0 and 1 as the two producer sockets. Note that these socket numbers are configured by the GPIF II descriptor. The descriptor writes data into thread 0 and thread 1, which are mapped to socket 0 and socket 1. Do not change the PIB socket numbers in the DMA channel configuration. You can change the UIB consumer socket number to whichever IN socket is used for video streaming.

Interrupt handling

The GPIF II descriptor generates several CPU interrupts that the firmware must handle. The CyFxFpifCB() in the uvc.c file is the callback function that handles the GPIF interrupts.

Table 2. Summary of the main files included in the project associated with this application note

File	Sections to be added/changed
sensor.c	Sensor_init() function needs to be completed I2C commands to initialize sensor need to be added (examples of I2C writes and reads are provided)
fx3.ld	The default fx3.ld file found in the SDK <install path>/firmware/common directory must be replaced with the fx3.ld file included with the project associated with this application note.
cyfctx.c	No changes needed. Please use this file as provided with the project associated with this application note. Note, the default cyfctx.c file provided with other example projects in the SDK may be different.
cyfxgpif2config.h	Header file generated by the GPIF II Designer tool. No direct changes are required. If the interface needs to be changed, a new header file should be generated from the GPIF II Designer tool.
gpif_interface.c	Loads and starts the GPIF II descriptor. No changes needed
uvc.c	Main source file for UVC application. No changes needed
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 needs to be changed. The UVC specification has all the required details.

Summary

This application note described how to implement a UVC application using the FX3 and an image sensor.

About the Author

Name: Karnik Shah
Title: Applications Engineer
Contact: shah@cypress.com

Appendix A

Designing with the GPIF II Designer

Once a state machine is fixed as shown in [Figure 6](#), it is very easy to create the configuration file using the GPIF II designer. The example project shown here is one for creating a GPIF II descriptor for image sensor interface as described in the [introduction](#) of this application note while also adhering to the [UVC requirements](#). This process involves three steps.

1. Creating a project using the GPIF II designer.
2. Choosing the Interface Definition
3. Drawing the state machine on the canvas

Creating the Project

Start GPIF II designer. The GUI appears as [Figure 9](#). Follow the step-by-step instructions as indicated in the figures. Each figure has sub steps indicated. For advance information regarding any steps refer to the GPIF II user guide.

Figure 9. Start GPIF II Designer

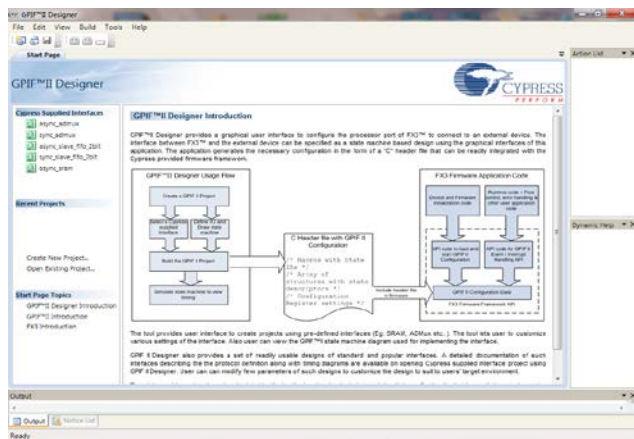


Figure 10. Open File Menu and Select New Project

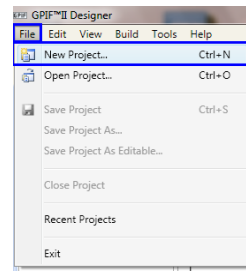
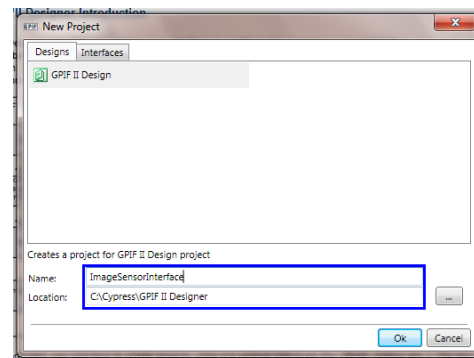


Figure 11. Enter Project Name and Location



Project creation is complete at this point. Once the project is created, the GPIF II designer opens up access to the interface definition and state machine tabs. The next step is to set the interface.

Choosing the Interface Definition

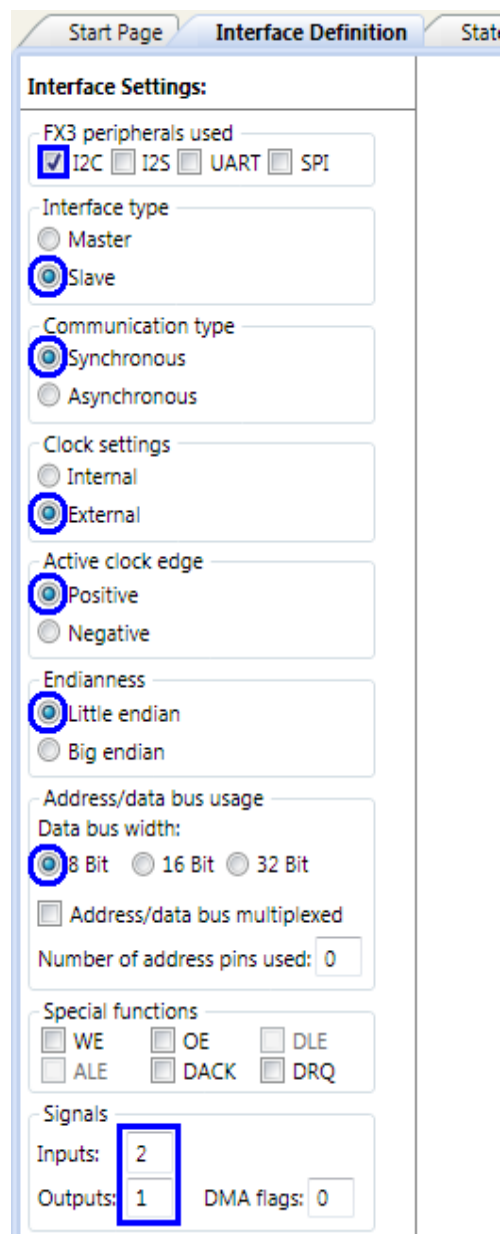
In this project, the image sensor connected to the FX3 device has 8bit data bus interface width. It uses GPIO 28 for Line valid signal, GPIO 29 for Frame valid signal, GPIO 22 for reset as an active low input. This image sensor also uses the I²C connection for FX3 to load the register values for configuring the sensor in 720p mode. By selecting the choices on the interface definition page, the direction and polarity of the GPIO signals are set. Also, the indicated input signals become available in the next phase for creating transition equations. [Figure 12](#) shows which interface settings (marked in blue boxes or circles) to choose from the "Interface Definition" tab. These are listed as below in order:

1. Choose 2 inputs (for LV and FV)
2. Choose 1 output (for nSensor_Reset)
3. Select I²C in the "FX3 peripherals used"
4. Select "Interface type" as "Slave"

5. Select "Communication type" as "Synchronous"
6. Select "Clock settings" as "External"
7. Select "Active clock edge" as "Positive"
8. Select "Endianness" as "Little endian" (unless the bytes need to be flipped)
9. Select "Address/data bus usage" as "8 bit"

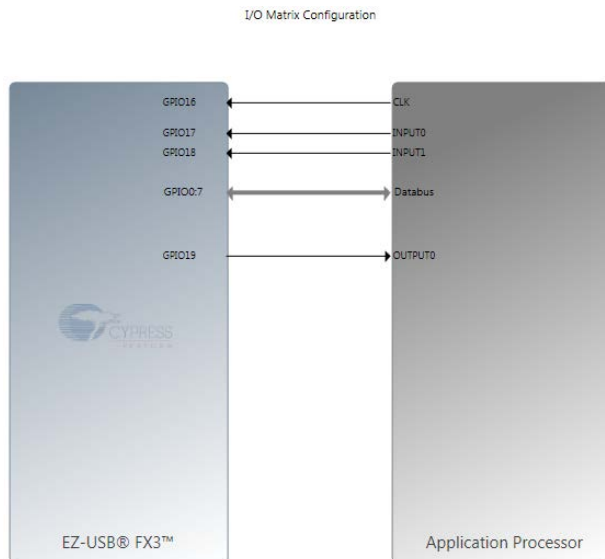
Once all these settings are complete the "I/O Matrix configuration" should look like that shown in Figure 13. Now, the properties of the input and the output signals can be modified. The properties include but not limited to name of the signals, the pin mapping (i.e. which GPIO acts as the input or the output), the polarity of the signal, and the initial value of the output signal.

Figure 12. Selecting the Interface Settings



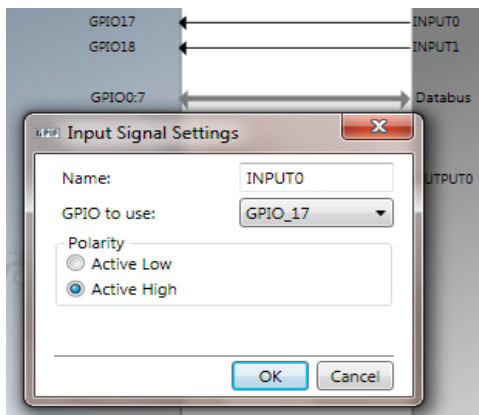
The screenshot shows the 'Interface Definition' tab of the Cypress EZ-USB FX3 configuration tool. The 'Interface Settings' section is expanded, showing various configuration options. The 'FX3 peripherals used' section has 'I2C' checked. The 'Interface type' section has 'Slave' selected. The 'Communication type' section has 'Synchronous' selected. The 'Clock settings' section has 'External' selected. The 'Active clock edge' section has 'Positive' selected. The 'Endianness' section has 'Little endian' selected. The 'Address/data bus usage' section has '8 Bit' selected. The 'Special functions' section has 'WE', 'OE', 'DLE', 'ALE', 'DACK', and 'DRQ' all checked. The 'Signals' section shows 'Inputs: 2' and 'Outputs: 1'.

Figure 13. I/O Matrix Configuration without Modification



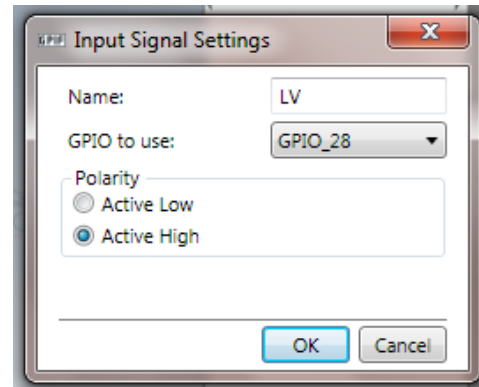
Double clicking “INPUT0” text in the Application Processor area opens the properties for that input signal as shown in [Figure 14](#).

Figure 14. “INPUT0” Default Properties



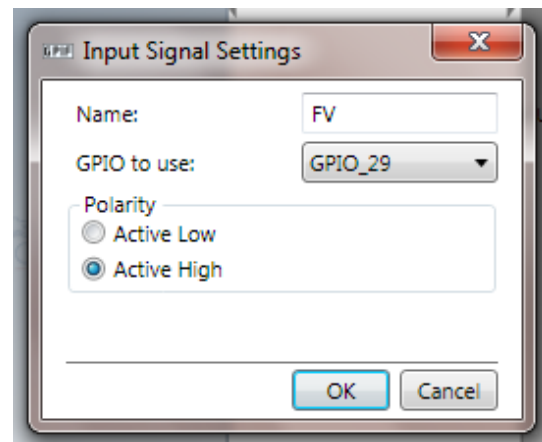
Change the name of the signal to “LV” for line valid. Change the “GPIO to use” to the line used for LV. This value is GPIO_28. Keep the polarity to “Active High”. The properties appear as [Figure 15](#). Click OK.

Figure 15. Changing Properties of Input0 Signal



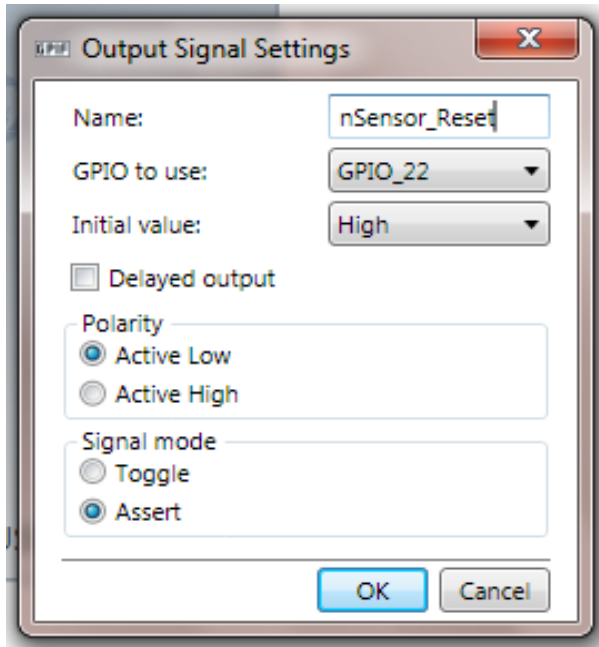
Next, open the properties box of “INPUT1” signal and change the properties as follows: “Name” -> “FV” for frame valid, “GPIO to use” -> GPIO_29, “Polarity” -> “Active High”. The properties box appears as [Figure 16](#).

Figure 16. Changing Properties of Input1 Signal



Change the properties of the “OUTPUT0” signal to the following: “Name” -> “nSensor_Reset”, “GPIO to use” -> “GPIO_22”, “Initial value” -> “High”, “Polarity” -> “Active Low”, and “Signal mode” -> Assert. The properties for the output signal will look as [Figure 17](#).

Figure 17. Changing Properties of Output0 Signal



This sets all the required Interface Settings. With these settings in place, state machine canvas should give options for LV and FV input signals in transition equations and nSensor_Reset output signal in DR_GPIO action.

Drawing the State Machine on the Canvas

Click on the State Machine tab to open the state machine canvas. Drawing on the canvas is easy. It involves creating new states, changing state properties, creating transitions between states, creating transition equations for each transition, adding actions for the state machine to do in each state. Figure 18 through Figure 39 show how to design the complete waveform, which satisfies the requirements described in the Image Sensor Interface section and keeps in mind the DMA capabilities of FX3. Usually, the interface width, the GPIO line numbers for FV/LV/Reset signals and the count limit for the counter are the only values that are specific for a given interface. The other values remain the same from design to design. The final design should appear similar to the design in Figure 6.

Note If the interface width is 32bits, the iomatrix configuration in the SDK should reflect "isDQ32bit" setting as "CyTrue". Refer to the FX3 SDK API guide for setting this parameter. In all other cases "isDQ32bit" can be set to false to utilize the pins for other purposes.

Basic Drawing Actions on the Canvas

The following steps are basic actions that are used to draw on the canvas.

1. Adding a new state

To add a new state, right click on the empty space in the canvas and choose Add State from the menu as shown in Figure 18. Figure 19 shows the newly added state.

Figure 18. Right Click to Open Menu for Adding New State

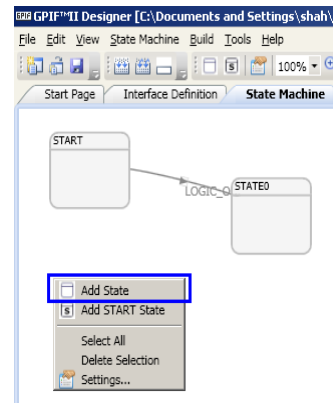
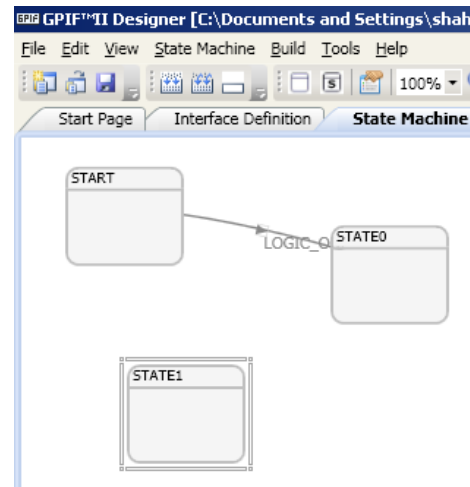


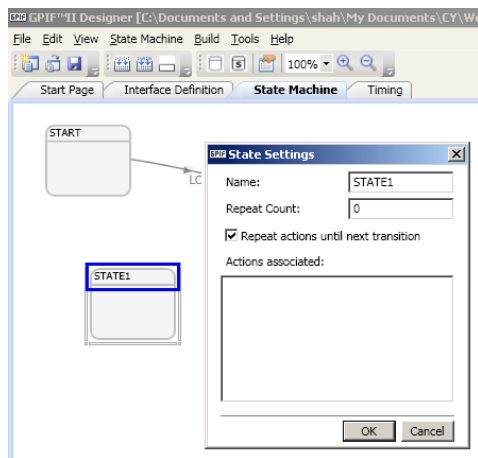
Figure 19. New State Added to the Canvas



2. Change the properties of the state

Double click on the highlighted part (blue rectangle) of the state in Figure 20 to open the properties dialog box for that state. After making the required changes, click OK to apply the changes.

Figure 20. Change the State Properties



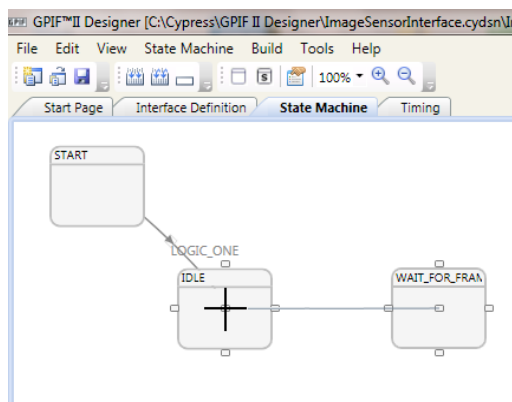
3. Move the state location on the canvas

Dragging and dropping the state using the same area as indicated by the blue rectangle in Figure 20 moves the location of the state on the canvas. However, this does not change any properties or connections associated with the state.

4. Connecting two states – drawing transition

To connect two states draw a transition line from the center of one state to the center of another. There is a direction associated with this transition line. The direction of this transition line is from the state where the transition line is started to the state where the transition line is dropped. See Figure 21; the transition from IDLE state to WAIT_FOR_FRAME state has been drawn. Step 1: Point the mouse exactly between the square in the center of the IDLE state. When the mouse reaches the center, it turns into the black plus sign. Step 2: Drag and drop a line from the black plus sign to the center of the WAIT_FOR_FRAME state.

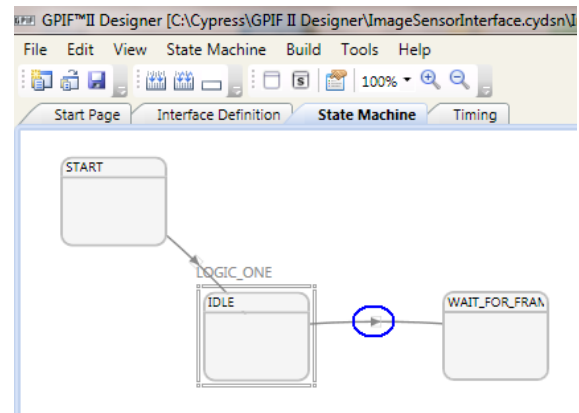
Figure 21. Drawing Transition between Two States



Note While drawing the transition, the center squares (or any other small squares around the states) of the states must be connected. If the mouse click is dropped somewhere other than these small squares of the states, the transition line does not appear.

The transition between the states appears as Figure 22. Notice that the transition has an arrow (circled blue), and it is in the direction from IDLE to WAIT_FOR_FRAME. This indicates that this transition carries unidirectional properties. To have a transition in the other direction (i.e. WAIT_FOR_FRAME to IDLE) another transition must be drawn but in the reverse direction.

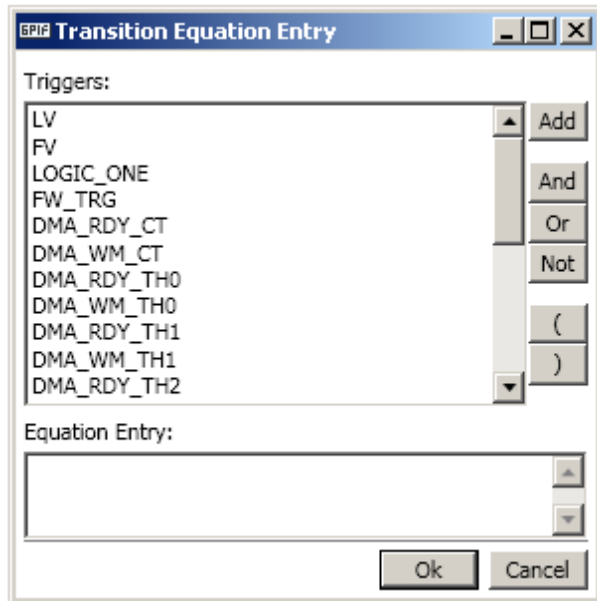
Figure 22. Transition between Two States



5. Changing properties of the transition – adding the transition equation

Double clicking on the transition arrow, highlighted by the blue circle in Figure 22, opens the Properties dialog box for the transition called the Transition Equation Entry dialog box. This is used to add the transition equation which the GPIF II will use to switch from one state to another as linked by the transition line. This dialog box allows access to all trigger options and logic operations. The triggers include the inputs defined in the Interface Definition tab. Figure 23 shows a sample Transition Equation Entry dialog box.

Figure 23. Transition Equation Entry Dialog Box



To add an entry in the Equation Entry field, type the equation in the box or click to add. Triggers are added by double clicks and logic operations are added by single clicks. For example, to add an entry of “(FV) and (not LV)” use the following steps:

Step 1: Double click on **FV** in the Triggers.

Step 2: Single Click on the **And** button.

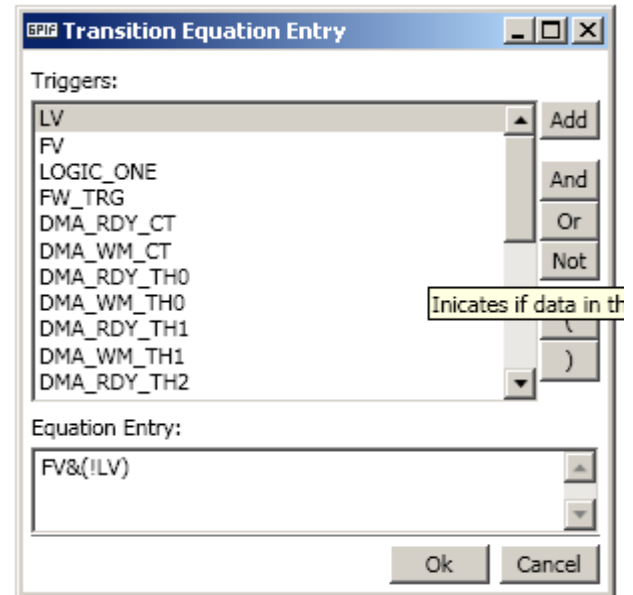
Step 3: Single Click on the **(** button on the right.

Step 4: Single click on the **Not** button..

Step 5: Double click on **LV** in the Triggers.

Step 6: Single Click on the **)** button. The result is shown in Figure 24.

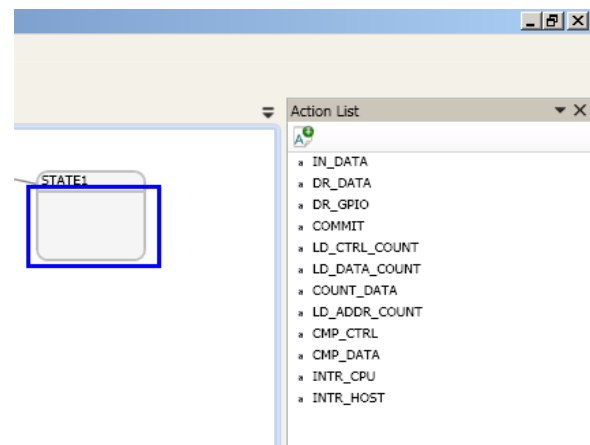
Figure 24. Transition Equation “(FV) and (not LV)”



6. Adding action to the states and modifying

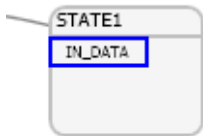
There are various actions available on the right side of the GPIF II designer in the State Machine tab under the Action List pane. These are actions that can be performed by the GPIF II state machine while in a particular state and include reading data into buffers from the data bus or writing data on the data bus from the buffers, loading or counting the counters, driving outputs and interrupting CPU. To add these actions to a state, drag and drop the action from the actions list into the marked area (blue rectangle in Figure 25) inside the state.

Figure 25. Adding Action to State



After adding the action, the action appears inside the state. For example, if IN_DATA action is added to the state in Figure 25, the result is Figure 26.

Figure 26. Action IN_DATA Added to the State



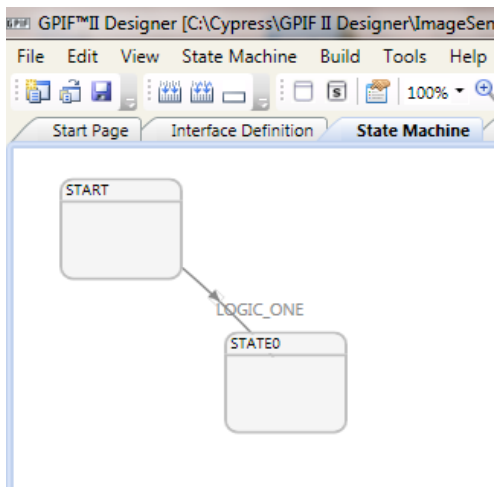
Some actions have properties associated with them. These properties can be changed by double clicking the action inside the state (marked by blue rectangle in Figure 26)

All these basic actions are already described in detail in the GPIF II designer's user guide. With these basic actions, the state machine diagram as described in Figure 6 can be created.

Drawing Image Sensor Interface State Machine for GPIF II

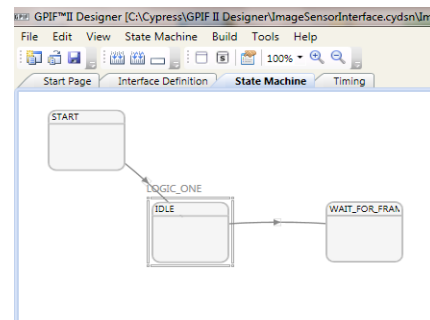
Click on the State Machine tab to open the canvas. An unedited canvas has only two states. A START state, which has a transition to STATE0 with a LOGIC_ONE transition equation as shown in Figure 27 is the blank canvas.

Figure 27. Blank Canvas at Start



1. Edit the name of STATE0 to IDLE
2. Add a new state and change the name to WAIT_FOR_FRAME_START.
3. Create transition from IDLE to WAIT_FOR_FRAME_START.

Figure 28. Result of Steps 1, 2, and 3



4. Edit the equation for the transition from IDLE to WAIT_FOR_FRAME_START to "not FV" as shown in Figure 29. The transition appears as Figure 30.

Figure 29. Transition Equation Entry from IDLE State to WAIT_FOR_FRAME_START State

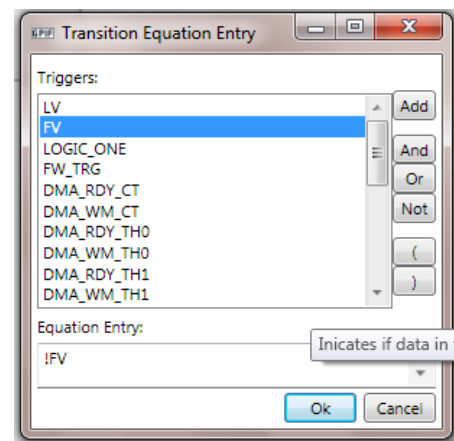
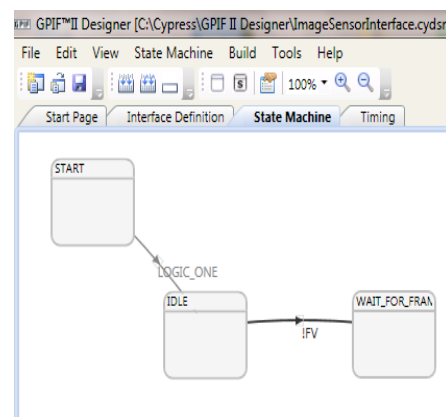


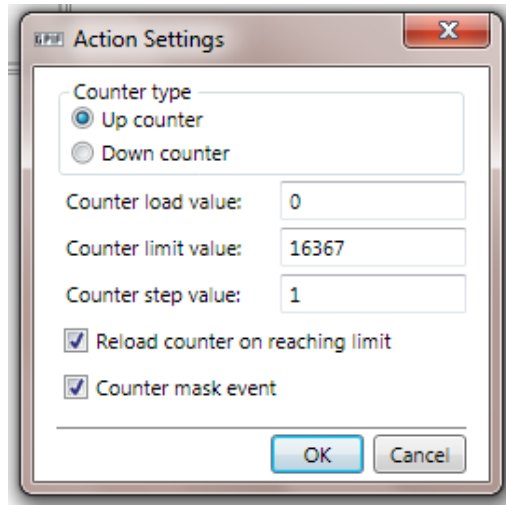
Figure 30. Transition from IDLE to WAIT_FOR_FRAME_START



5. Add action "LD_DATA_COUNT" to WAIT_FOR_FRAME_START state. Edit the properties of the action "LD_DATA_COUNT" as shown in Figure 31.

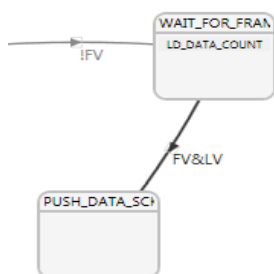
The limit value is calculated for the example project as per equation 1. This same counter is used for counting data to switch threads/sockets at buffer boundaries so; it must be automatically reloaded on reaching the limit.

Figure 31. LD_DATA_COUNT Action Settings



6. Add a new state with the name PUSH_DATA_SCK0.
7. Create a transition from WAIT_FOR_FRAME_START state to PUSH_DATA_SCK0 state.
8. Edit this transition equation entry to "FV and LV". The resulting state machine is shown in Figure 32.

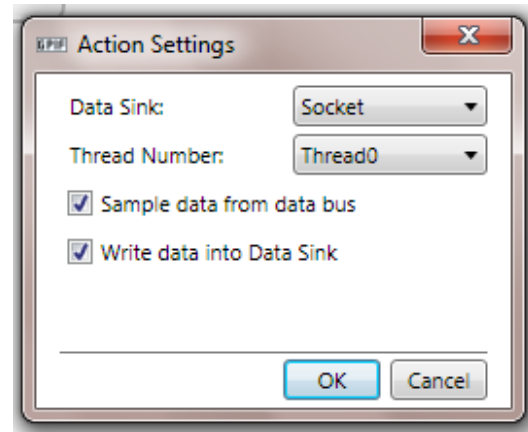
Figure 32. PUSH_DATA_SCK0 State Created



9. Add action COUNT_DATA to PUSH_DATA_SCK0 state. This increments the counter value.
10. Add action IN_DATA to PUSH_DATA_SCK0 state. This action helps read the data from the data bus into the DMA buffers.

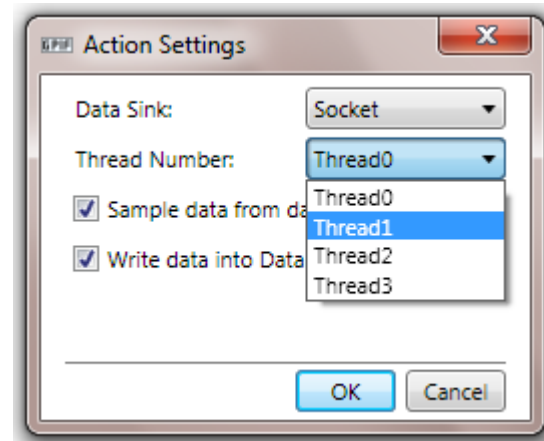
11. Edit the properties of the action IN_DATA in the PUSH_DATA_SCK0 state as shown in Figure 33.

Figure 33. IN_DATA Action for PUSH_DATA_SCK0



12. Add a new state with the name PUSH_DATA_SCK1.
13. Add action COUNT_DATA to PUSH_DATA_SCK1 state.
14. Add action IN_DATA to PUSH_DATA_SCK1 state.
15. Edit the properties of the IN_DATA action of the PUSH_DATA_SCK1 state as shown in Figure 34.

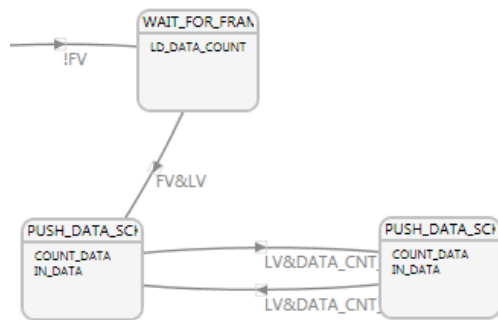
Figure 34. IN_DATA Action for PUSH_DATA_SCK1



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

These transitions occur when the state machine has to switch between sockets/threads during an active line at buffer boundaries to prevent data loss. Figure 35 shows the resulting state machine diagram.

Figure 35. PUSH_DATA_SCK0 and PUSH_DATA_SCK1



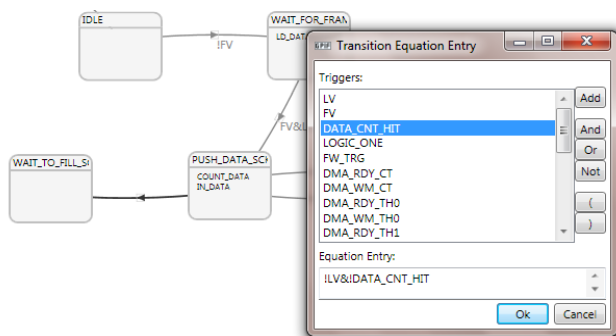
20. Add a new state “WAIT_TO_FILL_SCK0” to the left of the PUSH_DATA_SCK0 state.

21. Add a new state “WAIT_TO_FILL_SCK1” to the right of the PUSH_DATA_SCK1 state.

These two states are entered when the buffers are not full, but the line valid is de-asserted (i.e. image sensor switching) to the next line. Since the same operation is being done but in different sockets/threads alternatively, the states and the transition will be mirrored.

22. Create a transition from the PUSH_DATA_SCK0 state to the WAIT_TO_FILL_SCK0 state with the transition equation “(not LV) and (not DATA_CNT_HIT)” as shown in Figure 36.

Figure 36. PUSH_DATA_SCK0 to WAIT_TO_FILL_SCK0



23. Create 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.

24. Create transition from the “PUSH_DATA_SCK1” state to the “WAIT_TO_FILL_SCK1” state with the transition equation “(not LV) and (not DATA_CNT_HIT)”.

25. Create transition back from the “WAIT_TO_FILL_SCK1” state to the “PUSH_DATA_SCK1” state with the equation “LV”.

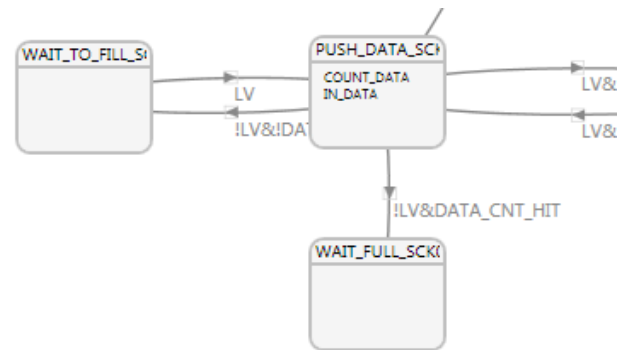
26. Add a new state “WAIT_FULL_SCK0_NEXT_SCK1” below the “PUSH_DATA_SCK0” state.

27. Add a new state “WAIT_FULL_SCK1_NEXT_SCK0” below the “PUSH_DATA_SCK1” state.

28. During these two states (WAIT_FULL_x), the image sensor is switching lines at buffer boundaries. Hence, the next data transfer has to happen through the alternate socket/thread.

29. Create transition from the “PUSH_DATA_SCK0” state to the “WAIT_FULL_SCK0_NEXT_SCK1” state with the equation “(not LV) and DATA_CNT_HIT”. The state machine appears as Figure 37.

Figure 37. WAIT_FULL_SCK0_NEXT_SCK1 state added

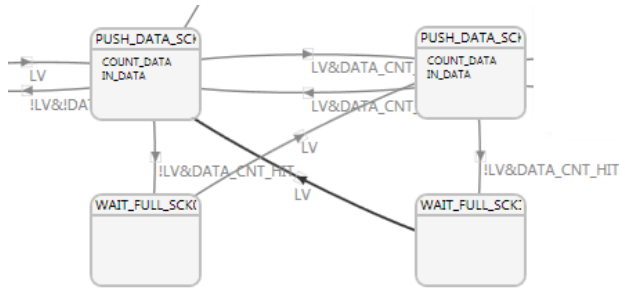


30. Create transition from the “PUSH_DATA_SCK1” state to the “WAIT_FULL_SCK1_NEXT_SCK0” state with the equation “(not LV) and DATA_CNT_HIT”.

31. Create transition from the “WAIT_FULL_SCK0_NEXT_SCK1” state to the “PUSH_DATA_SCK1” state with the equation “LV”. Notice that this is going to create a cross link in the diagram.

32. Create transition from the “WAIT_FULL_SCK1_NEXT_SCK0” state to the “PUSH_DATA_SCK0” state with the equation “LV”. The resultant diagram appears as Figure 38.

Figure 38. Wait When Buffers Are Full



33. Add a new state “PARTIAL_BUF_IN_SCK0” below the “WAIT_TO_FILL_SCK0” state.

34. Add a new state “PARTIAL_BUF_IN_SCK1” below the “WAIT_TO_FILL_SCK1” state.

The PARTIAL_BUF_x states are an indication of the end of the frame where the last buffer being written to has some data which is not equal to full length L. CPU must commit this partial buffer manually.

35. Add a new state “FULL_BUF_IN_SCK0” below the “WAIT_FULL_SCK0_NEXT_SCK1” state.

36. Add a new state “FULL_BUF_IN_SCK1” below the “WAIT_FULL_SCK1_NEXT_SCK0” state.

The FULL_BUF_x states are an indication that the end of the frame data is the last byte in the buffer associated with the corresponding socket/thread. There may not be any special handling required depending on the application used.

37. Create transition from the “WAIT_TO_FILL_SCK0” state to the “PARTIAL_BUF_IN_SCK0” state with the equation “not FV”.

38. Create transition from the “WAIT_TO_FILL_SCK1” state to the “PARTIAL_BUF_IN_SCK1” state with the equation “not FV”.

39. Create transition from the “WAIT_FULL_SCK0_NEXT_SCK1” state to the “FULL_BUF_IN_SCK0” state with the equation “not FV”.

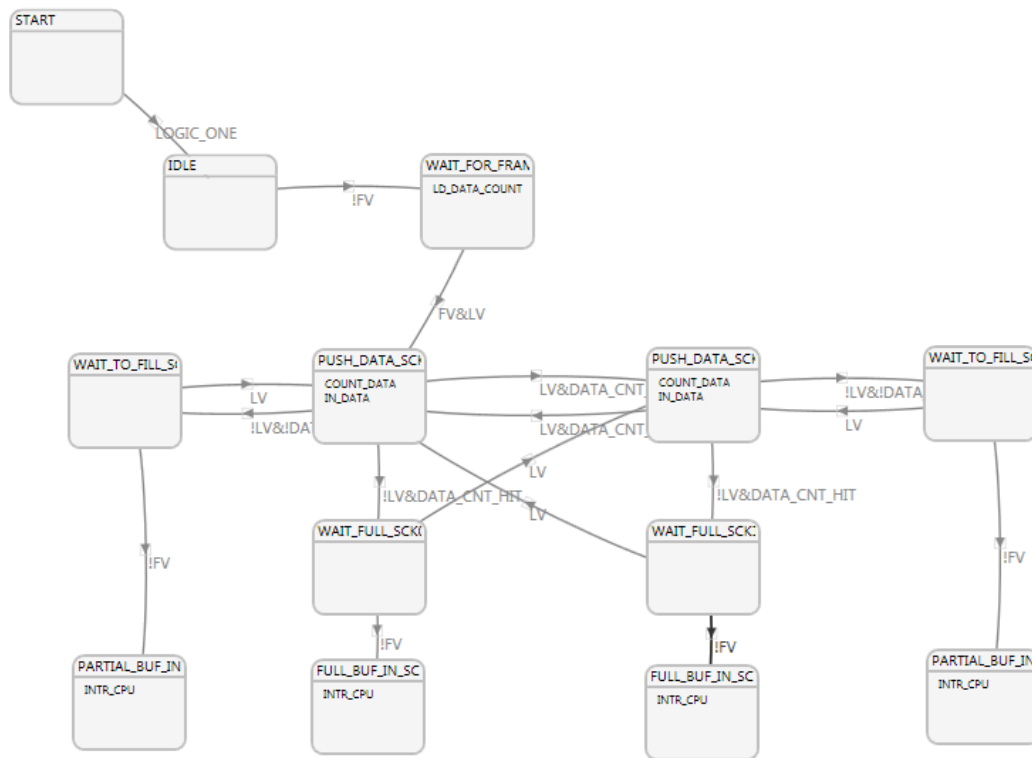
40. Create transition from the “WAIT_FULL_SCK1_NEXT_SCK0” state to the “FULL_BUF_IN_SCK1” state with the equation “not FV”.

41. 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 39](#). This is the final image that needs to be created. This can be compared directly with state machine created in [Figure 6](#).

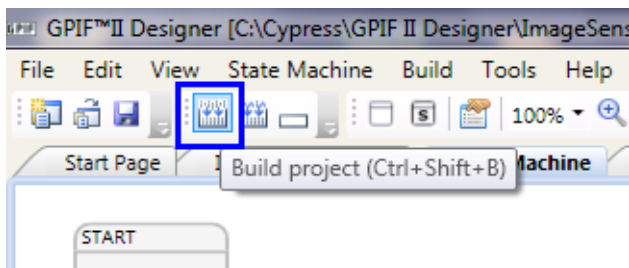
42. Save the project.

Figure 39. Final State Machine Diagram for Image Sensor Interface



43. Build the project using the Build icon highlighted in Figure 40.

Figure 40. Building the Project

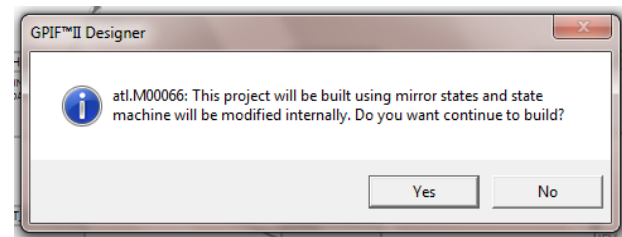


When building, the GPIF II designer tool recognizes that the state machine has states which have three transitions going out of them (PUSH_DATA_X states). GPIF II hardware architecture can only support binary transitions from a single state. To overcome this, there is a special feature called 'mirroring.' This method is described in detail in the GPIF II designer user guide. The GPIF II

designer tool generates a warning message as shown in Figure 41.

44. Click Yes to proceed.

Figure 41. Warning to Use Mirrors



45. Check the output of the project. This appears as a header file named **cyfxgpiif2config.h** generated under the project directory as shown in Figure 42. The project output window should indicate that the project was build successfully.

Figure 42. Project Output in the GPIF II Designer Output Window (Located Below)

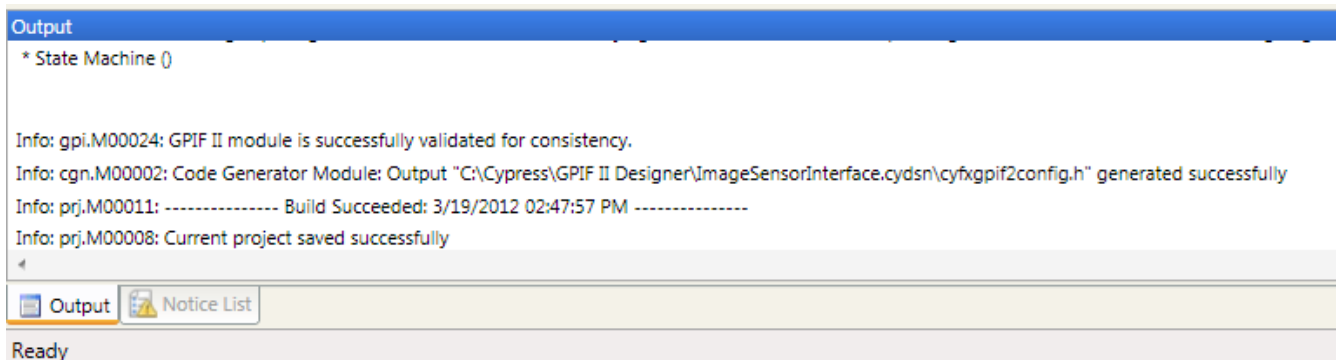
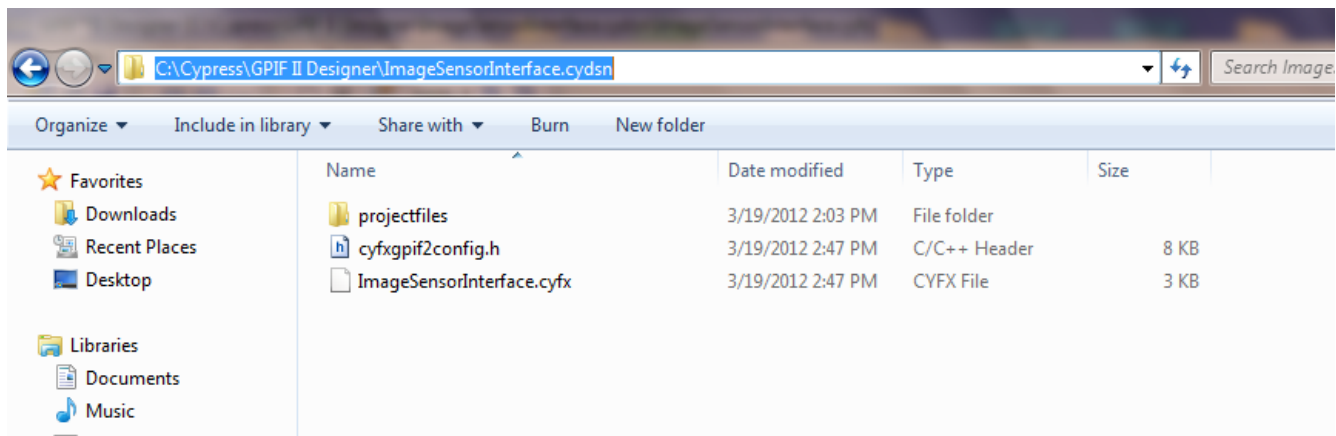


Figure 43. Project Output on the Hard Drive



Editing the GPIF II interface details

This section describes how to change the interface settings if required. As an example, the interface needs to be changed to accommodate a 16bit wide data bus from the image sensor.

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 following dialog box.
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 **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 as stated in the [GPIF II State Machine Design](#) section based on [Equation 1](#).

9. Save the Project.
10. Build the Project.
11. Copy the newly generated cyfxgpi2config.h header file from the location selected in step 3 to the firmware project directory. (Overwriting the existing cyfxgpi2config.h file might be necessary)

Document History

Document Title: Interfacing an Image Sensor to EZ-USB® FX3™ in a USB video class (UVC) Framework

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	<p>Changed AN title to match the scope of the new version of AN</p> <p>Added firmware project</p> <p>Added explanation of the firmware project</p> <p>Added the UVC application related details,</p> <p>Revised the functional block diagram</p> <p>Moved the steps to generate the GPIF II descriptor using the GPIF II Designer tool to Appendix A</p> <p>Added section in Appendix to show users how to modify a the given GPIF II -Designer project</p> <p>Clarified certain topics with explicit information</p> <p>Updated the all the links in the document to point to the correct locations within and outside the document</p> <p>Removed references to AN75310</p> <p>Removed references to the slave fifo application note</p>

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 5](#)

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 Phone : 408-943-2600
198 Champion Court Fax : 408-943-4730
San Jose, CA 95134-1709 Website : www.cypress.com

© Cypress Semiconductor Corporation, 2012. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, 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 products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

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.