AN65974

Designing with the EZ-USB® FX3™ Slave FIFO Interface

Author: Sonia Gandhi
Associated Project: Yes
Software Version: EZ-USB FX3 SDK1.2
Related Application Notes: AN75705 - Getting Started with EZ-USB FX3

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

AN65974 describes the Synchronous Slave FIFO interface of EZ-USB® FX3™. The hardware interface and configuration settings for the FLAGs are described in detail and examples are provided. References to the GPIF II Designer are included in order to make the Slave FIFO interface easy to design with. Finally, a complete design example is included to demonstrate how an FPGA can be interfaced to FX3 using synchronous Slave FIFO.

Contents

Introduction

Cypress's EZ-USB FX3, which is the next-generation USB 3.0 peripheral controller, lets developers add USB 3.0 functionality to any system. The controller works well with such applications as imaging and video devices, printers, and scanners.

The EZ-USB FX3 has a fully configurable, parallel, general programmable interface, called GPIF II, that can connect to an external processor, ASIC, or FPGA. The GPIF II is an enhanced version of the GPIF in FX2LP, Cypress's flagship USB 2.0 product. The GPIF II provides glue-less connectivity to popular devices such as FPGAs, image sensors and processors with interfaces such synchronous address data multiplexed interface.

One popular implementation of GPIF II is the synchronous Slave FIFO interface. This interface is used for applications in which the external device connected to EZ-USB FX3 accesses the EZ-USB FX3 FIFOs, reading from or writing data to them. Direct register accesses are not performed over the Slave FIFO interface.

This application note begins with a brief introduction to GPIF II and then describes the details of the synchronous Slave FIFO interface.

After a detailed description of the synchronous Slave FIFO interface, a complete design example is provided. This example shows how a master interface compatible with synchronous Slave FIFO can be implemented on an FPGA. The Verilog files for a Xilinx Spartan 6 FPGA are provided. The corresponding FX3 firmware project for synchronous Slave FIFO is also included as part of the example. The example has been developed using a Xilinx SP601 Evaluation kit for the Spartan 6 FPGA, an FX3 DVK and the FX3 SDK.

Figure 1 shows an example application diagram where the synchronous Slave FIFO interface is used.

GPIF II

The GPIF II is a programmable state machine that enables the flexibility of implementing an industry standard or proprietary interface. The GPIFII can function either as master or slave.

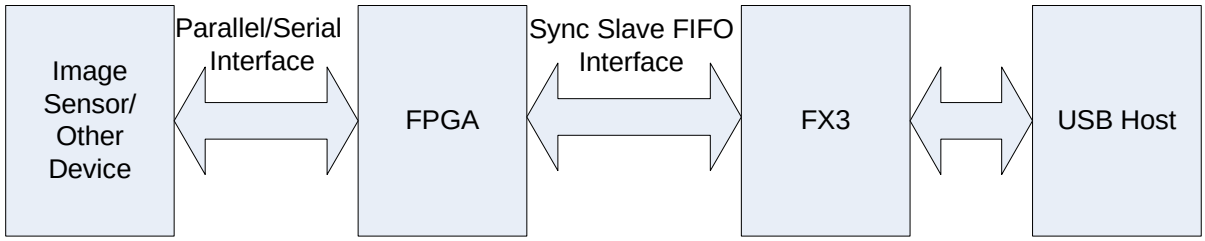
The GPIF II has the following features:

- Functions as master or slave
- Offers 256 firmware programmable states
- Supports 8-bit, 16-bit, and 32-bit parallel data bus
- Enables interface frequencies up to 100 MHz
- Supports 14 configurable control pins when 32-bit data bus is used; all control pins can be either input/output or bidirectional
- Supports 16 configurable control pins when 16/8 data bus is used; all control pins can be either input/output or bidirectional

GPIF II state transitions occur based on control input signals. Control output signals are driven by GPIF II state transitions. The behavior of the state machine is defined by a descriptor, which is designed to meet the required interface specifications. The GPIF II descriptor is essentially a set of programmable register configurations. In the EZ-USB FX3 register space. 8 kB is dedicated as GPIF II waveform memory, where the GPIF II descriptor is stored.

A popular implementation of GPIF II is the synchronous Slave FIFO interface, which is described in detail in the following sections.

Figure 1. Example Application Diagram



Synchronous Slave FIFO Interface

This section shows the interconnect diagram for the synchronous Slave FIFO interface, followed by the pin mapping of the signals.

The synchronous Slave FIFO interface is suitable for applications in which an external processor or device needs to perform data read/write accesses to EZ-USB FX3's internal FIFO buffers. Register accesses are not done over the Slave FIFO interface. The synchronous Slave FIFO interface is generally the interface of choice for USB applications, in order to support high throughput requirements.

Figure 2 shows the interface diagram for the synchronous Slave FIFO interface. The signals shown in Figure 2 are described in Table 1.

Figure 2. Synchronous Slave FIFO Interface Diagram

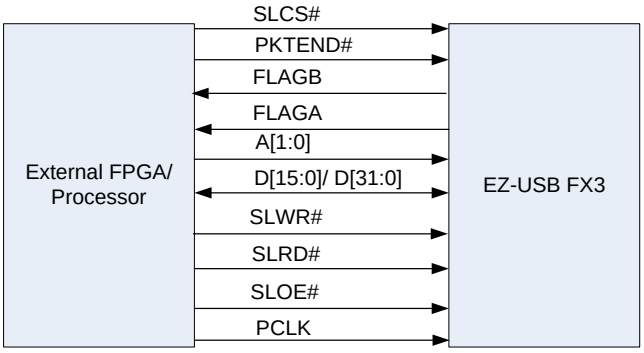


Table 1. Synchronous Slave FIIFO Interface Signals

Signal Name	Signal Description
SLCS#	This is the chip select signal for the Slave FIFO interface, which needs to be asserted to perform any access to the Slave FIFO interface.
SLWR#	This is the write strobe for the Slave FIFO interface. It must be asserted for performing write transfers to Slave FIFO.
SLRD#	This is the read strobe for the Slave FIFO interface. It must be asserted for performing read transfers from Slave FIFO.
SLOE#	This is the output enable signal. It causes the data bus of the Slave FIFO interface to be driven by FX3. It must be asserted for performing read transfers from Slave FIFO.
FLAGA/ FLAGB	These are the FLAG outputs from FX3. The FLAGs indicate the availability of an FX3 socket[1]
A[1:0]	This is the 2-bit address bus of the Slave FIFO interface. D[15:0]/D[31:0] : This is the 16-bit or 32-bit data bus of the Slave FIFO interface.
PKTEND#	The PKTEND# signal is asserted in order to write a short packet or a zero length packet to Slave FIFO
PCLK	This is the Slave FIFO interface clock.

Note The Slave FIFO interface described in this application note has only two address lines; hence only up to four sockets may be accessed.

In order to access more than four sockets, the Slave FIFO interface with five address lines should be used. Please refer to the application note AN68829 - Slave FIFO Interface for EZ-USB FX3: 5-Bit Address Mode.

Difference between Slave FIFO with Two and Five Address Lines

The synchronous Slave FIFO interface with two address lines supports access to up to four sockets.

If access to more than four sockets is required, then the synchronous Slave FIFO interface with five address lines should be used. In addition to the extra address lines, this interface also has a signal called EPSWITCH#. Due to the increased pins, fewer pins are available for use as FLAGs, necessitating that the FLAG be configured as a current_thread FLAG.

Extra latencies are incurred when using the synchronous Slave FIFO interface with five address lines:

- a two cycle latency from address to FLAG valid is incurred at the beginning of every transfer.
- whenever a socket address is switched, multiple cycles of latency are incurred in order to complete the socket switching.

Due to the increased latencies and additional interface protocol requirements, it is recommended that the synchronous Slave FIFO interface with five address lines be used only if the application requires access to more than four GPIFII sockets. For details on this interface, refer to the application note AN68829 - Slave FIFO Interface for EZ-USB FX3: 5-Bit Address Mode.

The remainder of this application note describes the details of the synchronous Slave FIFO interface with two address lines only. The following section describes the pin mapping of the signals shown in the Slave FIFO interface diagrams.

[1] The section Treads and Sockets explains the concept of sockets for data transfers. The FLAGs are described in detail in the section on Configuration of Flags.

Pin Mapping of Slave FIFO Interface

The default pin mapping of the Slave FIFO interface is shown in Table 2. The table also shows the GPIO pins and other serial interfaces (UART/SPI/I2S) available when GPIF II is configured for the Slave FIFO interface.

Note The pin mapping may be changed if needed and FLAGs may be added or reconfigured by using the GPIFII Designer tool. Further details on this are provided in the section on Configuration of Flags.

Table 2. Pin Mapping for Slave FIFO Interface

EZ-USB FX3 Pin	Synchronous Slave FIFO Interface with 16-bit Data bus	Synchronous Slave FIFO Interface with 32-bit Data bus
GPIO[17]	SLCS#	SLCS#
GPIO[18]	SLWR#	SLWR#
GPIO[19]	SLOE#	SLOE#
GPIO[20]	SLRD#	SLRD#
GPIO[21]	FLAGA	FLAGA
GPIO[22]	FLAGB	FLAGB
GPIO[24]	PKTEND#	PKTEND#
GPIO[28]	A1	A1
GPIO[29]	A0	A0
GPIO[0:15]	DQ[0:15]	DQ[0:15]
GPIO[16]	PCLK	PCLK
GPIO[33:44]	Available as GPIOs	DQ[16:27]
GPIO[45]	GPIO	GPIO
GPIO[46]	GPIO/UART_RTS	DQ28
GPIO[47]	GPIO/UART_CTS	DQ29
GPIO[48]	GPIO/UART_TX	DQ30
GPIO[49]	GPIO/UART_RX	DQ31
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/UART_RTS
GPIO[54]	GPIO/SPI_SSN/UART_CTS	GPIO/UART_CTS
GPIO[55]	GPIO/SPI_MISO/UART_TX	GPIO/UART_TX
GPIO[56]	GPIO/SPI_MOSI/UART_RX	GPIO/UART_RX
GPIO[57]	GPIO/I2S_MCLK	GPIO/I2S_MCLK

Note

For complete pin mapping of EZ-USB FX3, refer to the datasheet "EZ-USB FX3 SuperSpeed USB Controller."

This section described the interconnect and the pin mapping of the synchronous Slave FIFO interface. The following section describes the interface access sequence and timing parameters for the synchronous Slave FIFO interface.

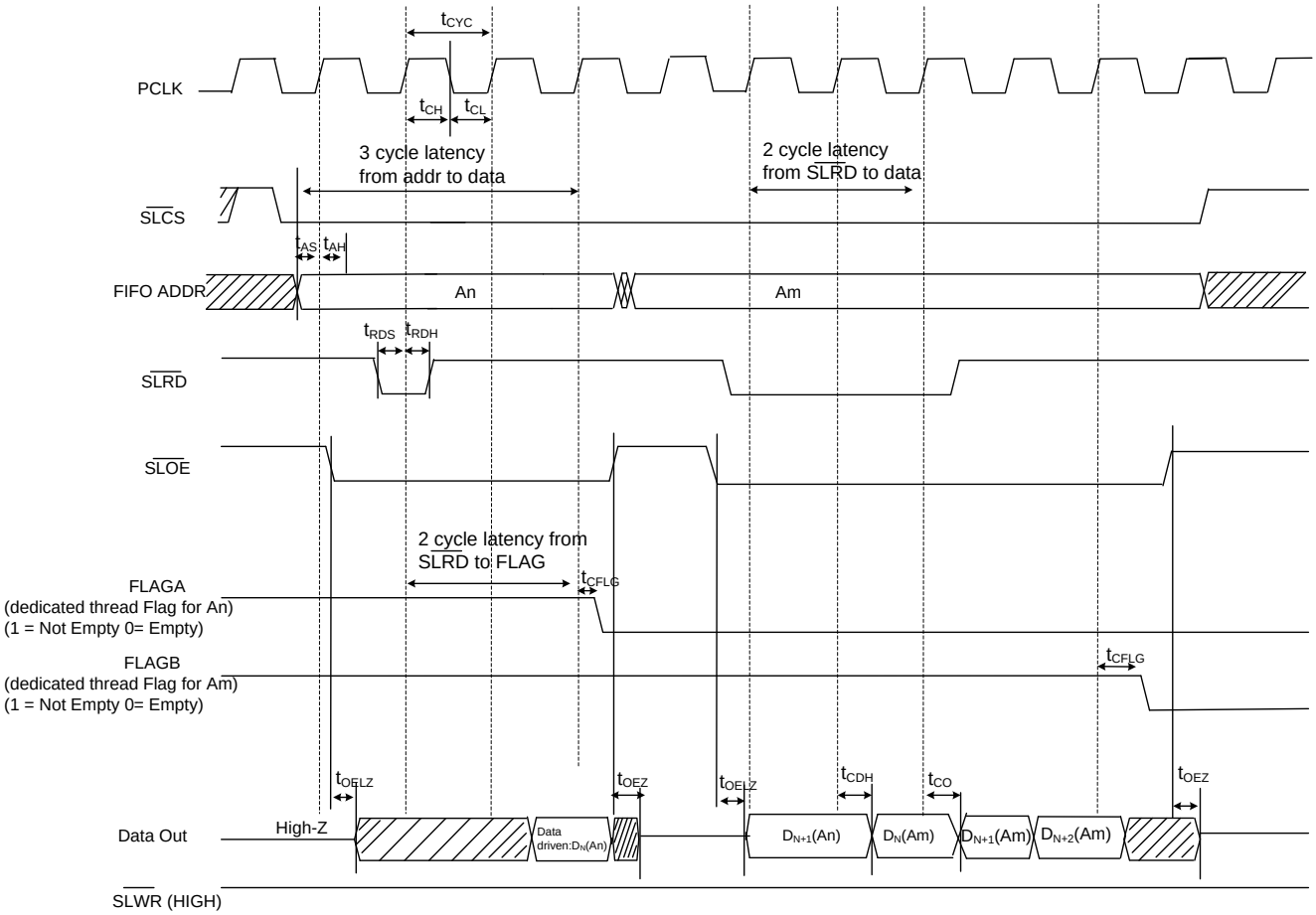
Synchronous Slave FIFO Access Sequence and Interface Timing

This section describes the access sequence and timing of the Synchronous Slave FIFO interface.

An external processor or device (functioning as master of the interface) may perform single-cycle or burst data accesses to EZ-USB FX3's internal FIFO buffers. The external master drives the 2-bit address on the ADDR lines and asserts the read or write strobes. EZ-USB FX3 asserts the FLAG signals to indicate empty or full conditions of the buffer.

Synchronous Slave FIFO Interface Timing

Figure 3. Synchronous Slave FIFO Read Sequence



Synchronous Slave FIFO Read Sequence Description

The sequence for performing reads from the synchronous Slave FIFO interface is as follows:

1. FIFO address is stable and SLCS# is asserted.
2. SLOE# is asserted. SLOE# is an output enable only whose sole function is to drive the data bus.
3. SLRD# is asserted.

The FIFO pointer is updated on the rising edge of the PCLK, while SLRD# is asserted. This action starts the propagation of data from the newly addressed FIFO to the data bus. After propagation delay of tCO (measured from the rising edge of PCLK), the new data value is present. N is the first data value read from the FIFO. To drive the data bus, SLOE# must also be asserted.

The same sequence of events is shown for a burst read.

Note For burst mode, the SLRD# and SLOE# are left asserted during the entire duration of the read. When SLOE# is asserted, the data bus is driven (with data from the previously addressed FIFO). For each subsequent

rising edge of PCLK, while the SLRD# is asserted, the FIFO pointer is incremented and the next data value is placed on the data bus.

FLAG Usage: The FLAG signals are monitored by the external processor for flow control. FLAG signals are outputs from EZ-USB FX3 that may be configured to show empty/full/partial status for a dedicated thread or the current thread being addressed.

Figure 4. Synchronous Slave FIFO Write Sequence

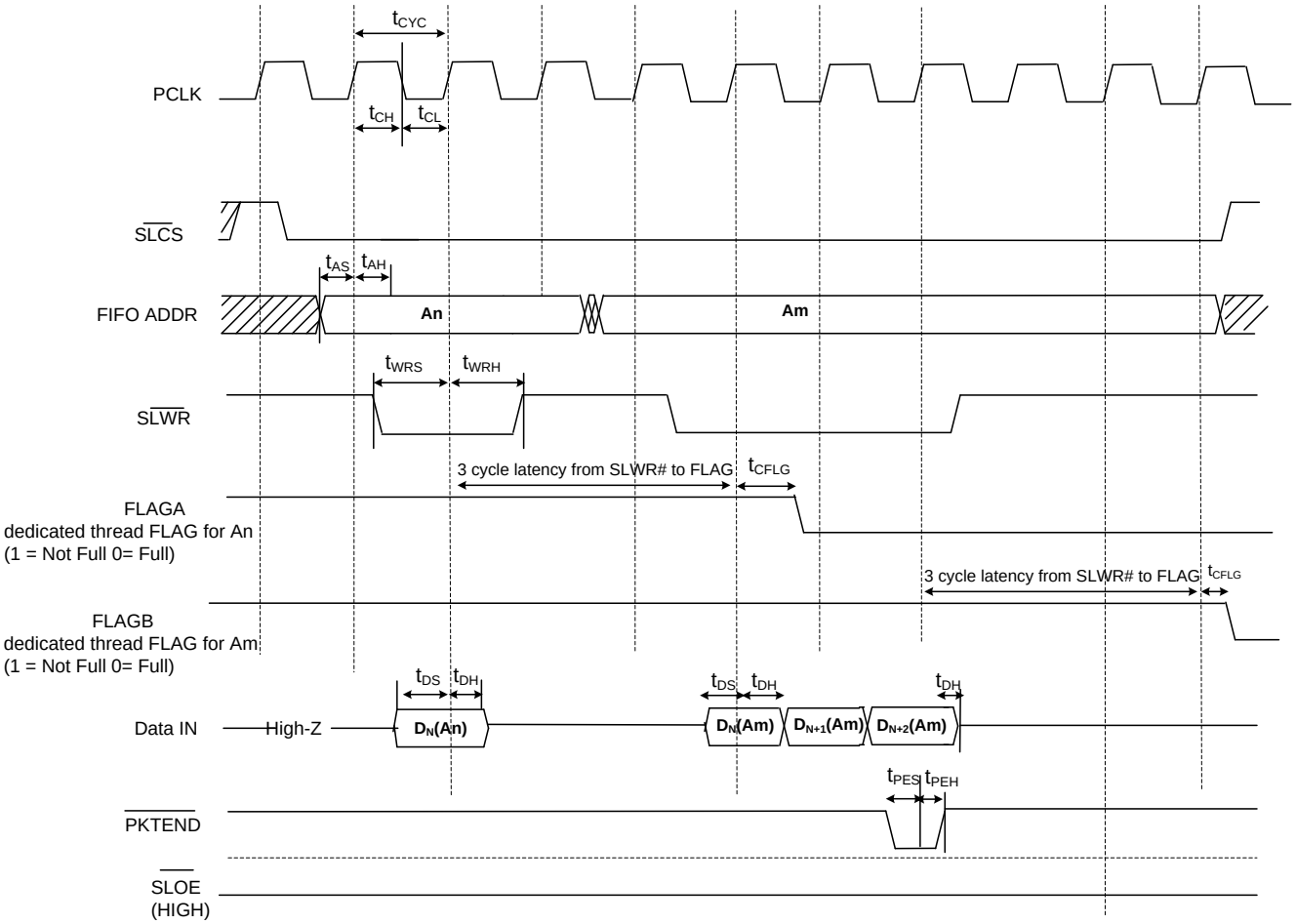
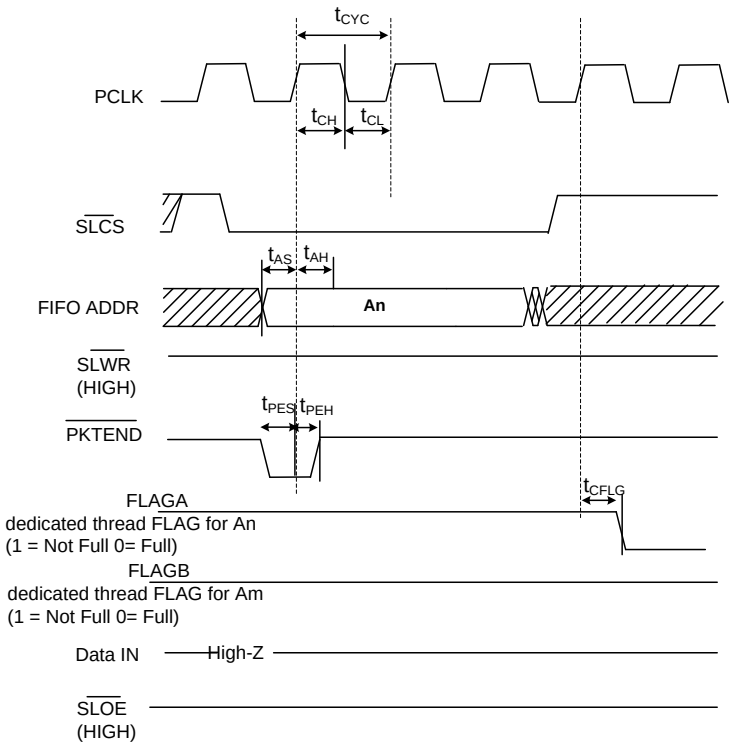


Figure 5. Synchronous ZLP Write Cycle Timing



Synchronous Slave FIFO Write Sequence Description

The sequence for performing writes to the synchronous Slave FIFO interface is as follows

1. FIFO address is stable and the signal SLCS# is asserted.
2. External master/peripheral outputs the data onto the data bus
3. SLWR# is asserted.
4. While the SLWR# is asserted, data is written to the FIFO and on the rising edge of the PCLK, the FIFO pointer is incremented.
5. The FIFO flag is updated after a delay of tCFLG from the rising edge of the clock.

The same sequence of events also is shown for a burst write.

Note For the burst mode, SLWR# and SLCS# are left asserted for the entire duration of the burst write. In the burst write mode, after the SLWR# is asserted, the value on the data bus is written into the FIFO on every rising edge of PCLK. The FIFO pointer is updated on each rising edge of PCLK.

Short Packet: A short packet can be committed to the USB host by using the PKTEND# signal. The external device/processor should be designed to assert the PKTEND# along with the last word of data and SLWR# pulse corresponding to the last word. The FIFOADDR lines must be held constant during the PKTEND# assertion. On assertion of PKTEND# with SLWR#, the GPIF II state machine interprets the packet to be a short packet and commits it to the USB interface. If the protocol does not require any short packets to be transferred, the PKTEND# signal may be pulled high.

Note that in the read direction, there is no specific signal to indicate that a short packet has been sourced from USB. The empty FLAG must be monitored by the external master in order to determine when all the data has been read.

Zero Length Packet: The external device/processor can signal a Zero Length Packet (ZLP) by asserting PKTEND#, without asserting SLWR#. SLCS# and address must be driven as shown in Figure 5.

FLAG Usage: The FLAG signals are monitored by the external processor for flow control. FLAG signals are outputs from the EZ-USB FX3 device that may be configured to show empty/full/partial status for a dedicated thread or the current thread being addressed.

Table 3. Synchronous Slave FIFO Timing Parameters

Parameter	Description	Min	Max	Unit
FREQ	Interface clock frequency	–	100	MHz
tCYC	Clock period	10	–	ns
tCH	Clock high time	4	–	ns
tCL	Clock low time	4	–	ns
tRDS	SLRD# to CLK Setup time	2	–	ns
tRDH	SLRD# to CLK Hold time	0.5	–	ns
tWRS	SLWR# to CLK Setup time	2	–	ns
tWRH	SLWR# to CLK Hold time	0.5	–	ns
tCO	Clock to valid data	–	8	ns
tDS	Data input Setup time	2	–	ns
tDH	CLK to Data Input Hold	0	–	ns
tAS	Address to CLK Setup time	2	–	ns
tAH	CLK to Address Hold time	0.5	–	ns
tOELZ	SLOE# to Data low-Z	0	–	ns
tCFLG	CLK to Flag Output propagation delay	–	8	ns
tOEZ	SLOE# de-assert to Data Hi Z	–	8	ns
tPES	PKTEND# to CLK setup	2	–	ns
tPEH	CLK to PKTEND# Hold	0.5	–	ns
tCDH	CLK to Data output hold	2	–	ns
Note Three-cycle latency from ADDR to DATA				

So far this application note has described the hardware details of the Slave FIFO interface, including the pin mapping and interface timing. The following sections described the configuration of the FLAG signals that may be done using the GPIFII Designer and the EZ-USB FX3 SDK.

Before describing the various FLAG configurations, it is important to introduce the concept of threads, sockets and DMA channel.

Threads and Sockets

This section briefly explains the concept of threads and sockets, which is important for understanding how to configure the FLAGs.

The EZ-USB FX3 FIFOs are associated with sockets. Sockets on the GPIF II side are similar to endpoints on the USB interface.

EZ-USB FX3 provides four physical hardware threads for data transfer over the GPIF II. At a time, any one socket is mapped to a physical thread. By default, PIB socket 0 is mapped to thread 0, PIB socket 1 is mapped to thread 1, PIB socket 2 is mapped to thread 2 and PIB socket 3 is mapped to thread 3.

Note that the address signals A1:A0 on the interface indicate the thread to be accessed. FX3's DMA fabric then routes the data to the socket mapped to that thread. Therefore when A1:A0 = 0, thread 0 is accessed, and any data that is transferred over thread 0 is routed to socket 0. Similarly, when A1:A0 = 1, data is transferred in and out of socket 1.

Note The Slave FIFO interface described in this application note has only two address lines; hence only up to four sockets may be accessed.
In order to access more than four sockets, the Slave FIFO interface with five address lines should be used. Refer to the application note, AN68829 - Slave FIFO Interface for EZ-USB® FX3™: 5-Bit Address Mode.

The sockets to be accessed must be specified by configuring a DMA channel.

DMA Channel Configuration

The firmware must configure a DMA channel with the required producer and consumer sockets.

Note that if data is to be transferred from the Slave FIFO interface to the USB interface, then P-port is the producer and USB is the consumer, and vice-versa.

So if data is to be transferred in both directions over the Slave FIFO interface, two DMA channels should be configured, one with P-port as the producer and another with P-port as the consumer.

The P-port producer socket is the socket that the external device will write to over the Slave FIFO interface and the P-port consumer socket is the one that the external device will read from over the Slave FIFO interface.

Note that the P-port socket number in the DMA channel should be the socket number that will be addressed on A1:A0.

Multiple buffers can be allocated to a particular DMA channel, when configuring the channel. Note, that the FLAGs will indicate full/empty on a per buffer basis.

(The maximum buffer size for any one buffer is 64 kB-16.)

For example, if two buffers of 1024 bytes have been allocated to a DMA channel, the full FLAG will indicate full when 1024 bytes have been written into the first buffer. It will continue to indicate full, until the DMA channel has switched to the second buffer. The time taken for the DMA channel to switch to the next buffer is not deterministic, although it is typically of the order of a few microseconds. The external master must monitor the FLAG to determine when the switching is complete and the next buffer has become available for data access.

The following section describes how FLAGs may be configured to indicate the status of different threads.

Configuration of Flags

Flags may be configured as empty, full, partially empty, or partially full signals. These are not controlled by the GPIF II state machine, but, rather, by the DMA hardware engine internal to EZ-USB FX3. Flags are associated with specific threads or the currently addressed thread and, therefore, indicate the status of the socket mapped to that thread.

Flags indicate empty or full, based on the direction of the socket (configured during socket initialization). Therefore, a flag indicates empty/not empty status if data is being read out of the socket and indicates full/not full status if data is being written into the socket.

The different types of FLAGs that can be used are:

- Dedicated thread flag (empty/full or partially empty/full)
- Current thread flag (empty/full or partially empty/full)

These different types of FLAGs are described below. Different FLAG configurations result in different latencies, which are summarized in Table 4.

Dedicated Thread Flag

A flag can be configured to indicate the status of a particular thread. In this case, that flag is dedicated only to that thread and always indicates the status of the socket mapped to that particular thread only, irrespective of which thread is being addressed on the address bus.

In this case, the external processor/device must keep track of which flag is dedicated to which thread and monitor the correct flag every time a different thread is addressed.

For example, if FLAGA is dedicated to thread 0, and FLAGB is dedicated to thread 1, then when the external processor performs accesses to thread 0, it must monitor FLAGA. When the external processor accesses thread 1, it must monitor FLAGB.

A flag may be dedicated for every thread that is going to be accessed. If the application requires four threads to be accessed, then there may be four corresponding flags.

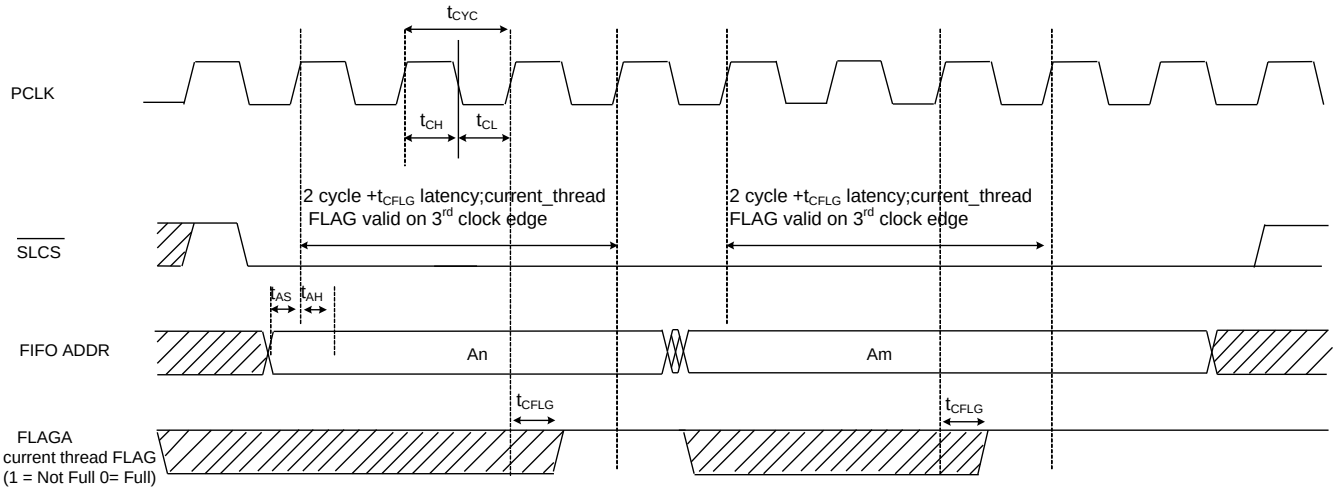
Note, when performing write transfers a 3-cycle latency for the flag is always incurred at the end of the transfer. The 3-cycle latency is from the write cycle that causes the buffer to become full to the time that the flag gets asserted low. At the fourth clock edge the external master can sample the flag low. This is shown in Figure 4.

When performing read transfers a two-cycle latency for the flag is always incurred at the end of the transfer. The two-cycle latency is from the read (last SLRD# assertion) cycle that causes the buffer to become empty to the time that the flag gets asserted low. At the third clock edge the external master can sample the flag low. This is shown in Figure 3.

Current Thread Flag

A flag can be configured to indicate the status of the currently addressed thread. In this case, the GPIF II state machine samples the address on the address bus and then updates the flags to indicate the status of that thread. This configuration requires fewer pins, because a single “current_thread” flag can be used to indicate the status of all four threads. However, two-cycle latency is incurred when the current_thread flag is used for a synchronous Slave FIFO interface, because the GPIF II first must sample the address and then update the flag. The two-cycle latency starts when a valid address is presented on the interface. On the third clock edge after this, the valid state of the FLAG of the newly addressed thread can be sampled. (Note that the Slave FIFO descriptors included in the SDK use the “current_thread” flag configuration.)

Figure 6. Additional Latency Incurred at Start of Transfer when using a Current Thread FLAG



Note When performing write transfers a 3-cycle latency for the flag is always incurred at the end of the transfer. The 3-cycle latency is from the write cycle that causes the buffer to become full to the time that the flag gets asserted low. At the fourth clock edge the external master can sample the flag low. This is shown in Figure 4.

When performing read transfers, a 2-cycle latency for the flag is always incurred at the end of the transfer. The 2-cycle latency is from the read (last SLRD# assertion) cycle that causes the buffer to become empty to the time that the flag gets asserted low. At the third clock edge the external master can sample the flag low. This is shown in Figure 3.

Partial FLAG

A flag can be configured to indicate the partially empty/full status of a socket. A watermark value must be selected such that the flag gets asserted when the number of 32-bit words that may be read or written is less than or equal to the watermark value.

Note The latency for a partial FLAG is dependent on the watermark value specified for the partial FLAG.

The following table summarizes the latencies incurred when using different FLAG configurations. Table 4 also shows the setting that must be selected in GPIFII Designer for a particular FLAG setting. Examples and screenshots of the GPIFII Designer settings for FLAGs may be found in the section on Configuration of Flags.

Table 4. Latencies Associated with Different FLAG Configurations

FLAG Configuration	GPIFII Designer FLAG setting selection	Address to FLAG latency at start of transfer	FLAG latency at end of transfer		Additional API call required
			For Write transfers to Slave FIFO (latency from last SLWR# assertion to full FLAG assertion)	For Read transfers from Slave FIFO (latency from last SLRD# assertion to empty FLAG assertion)	
Full/Empty FLAG dedicated to a specific thread "n"	Thread_n_DMA_Ready	0 cycles	3 cycles + tCFLG (external device can sample valid FLAG on 4th clock edge)	2 cycles + tCFLG (external device can sample valid FLAG on 3rd clock edge)	N/A
Full/Empty FLAG for currently addressed thread	Current_thread_DMA_Ready	2 cycles + tCFLG (external device can sample valid FLAG on 3rd clock edge)	3 cycles + tCFLG (external device can sample valid FLAG on 4th clock edge)	2 cycles + tCFLG (external device can sample valid FLAG on 3rd clock edge)	N/A
Partially full/empty FLAG dedicated to a specific thread "n"	Thread_n_DMA_Watermark	0 cycles	Dependent on watermark level	Dependent on watermark level	Set watermark level by calling the CyU3SocketConfigure() API. Note: Watermark is in terms of a 32-bit data word. Examples: CyU3PGpifSocketConfigure (0,PIB_SOCKET_0,4,CyFalse,1) sets the watermark for thread 0 to 4 CyU3PGpifSocketConfigure (3,PIB_SOCKET_3,4,CyFalse,1) sets the watermark for thread 3 to 4
Partially full/empty FLAG for currently addressed thread	Current_thread_DMA_Watermark	2 cycles + tCFLG (external device can sample valid FLAG on 3rd clock edge)	Dependent on watermark level	Dependent on watermark level	Set watermark level by calling the CyU3SocketConfigure() API. Note: Watermark is in terms of a 32-bit data word. Examples: CyU3PGpifSocketConfigure (0,PIB_SOCKET_0,4,CyFalse,1) sets the watermark for thread 0 to 4 CyU3PGpifSocketConfigure (3,PIB_SOCKET_3,4,CyFalse,1) sets the watermark for thread 3 to 4

This section described the different FLAG configurations that are available and the associated latencies.

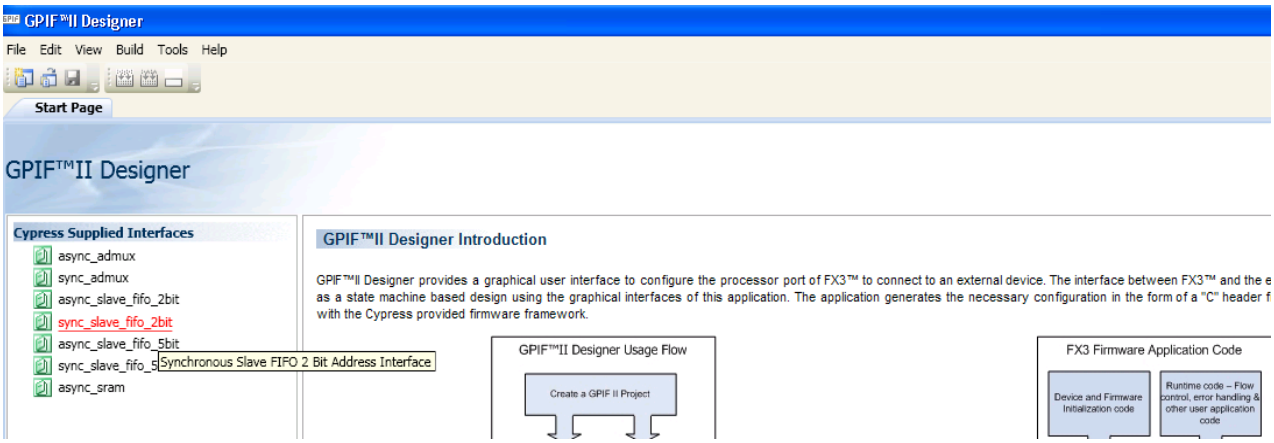
The following sections describe how to configure the FLAGs using the GPIFII Designer tool and the EZ-USB FX3 SDK.

GPIFII Designer

Implementation of Synchronous Slave FIFO Interface

The GPIFII implementation of the Slave FIFO interface can be found by installing the GPIFII Designer tool. The GPIFII Designer tool can be installed from Cypress's website. On launching the GPIFII Designer, on the Start Page you will find the Cypress Supplied Interfaces.

Figure 7. Slave FIFO Projects in GPIFII Designer – Cypress Supplied Interfaces



The sync_slave_fifo_2bit project is the GPIFII implementation of the synchronous Slave FIFO interface with 2-bit address. The next section explains how a partial FLAG may be configured using GPIFII Designer.

Configuring a Partial FLAG

A partial FLAG is configured by the following two steps:

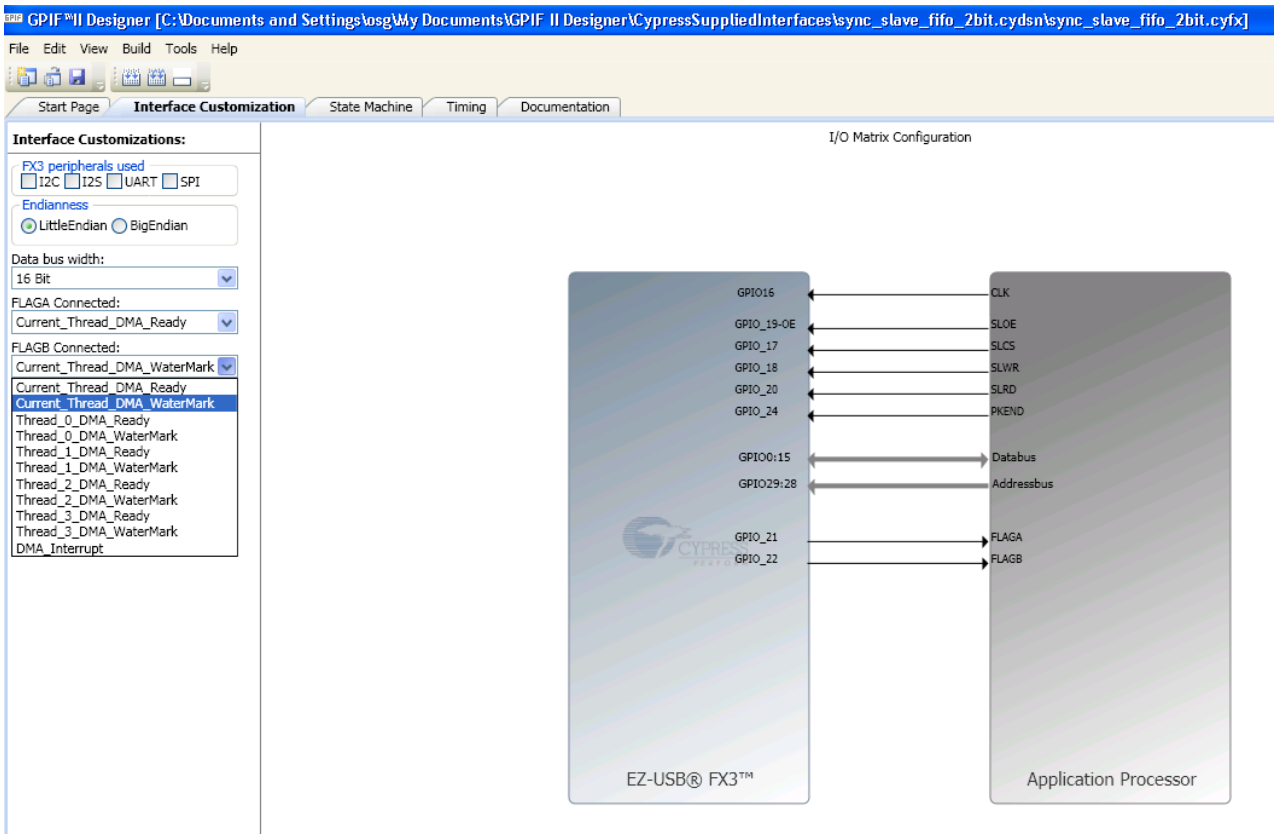
- The partial FLAG setting must be selected in the GPIFII Designer tool
- The watermark level for the partial FLAG must be set using the the CyU3PGpifSocketConfigure() API in firmware

In the GPIFII Designer tool, on opening the sync_slave_fifo_2bit project from the Cypress Supplied Interfaces, under "FLAGA Connected" or "FLAGB Connected" select "Current_Thread_DMA_Watermark" to configure the FLAG as a partial FLAG for current thread.

Or select "Thread_n_DMA_Watermark" to configure the FLAG as a partial FLAG dedicated for thread "n".

A screenshot of how this is done is on the next page:

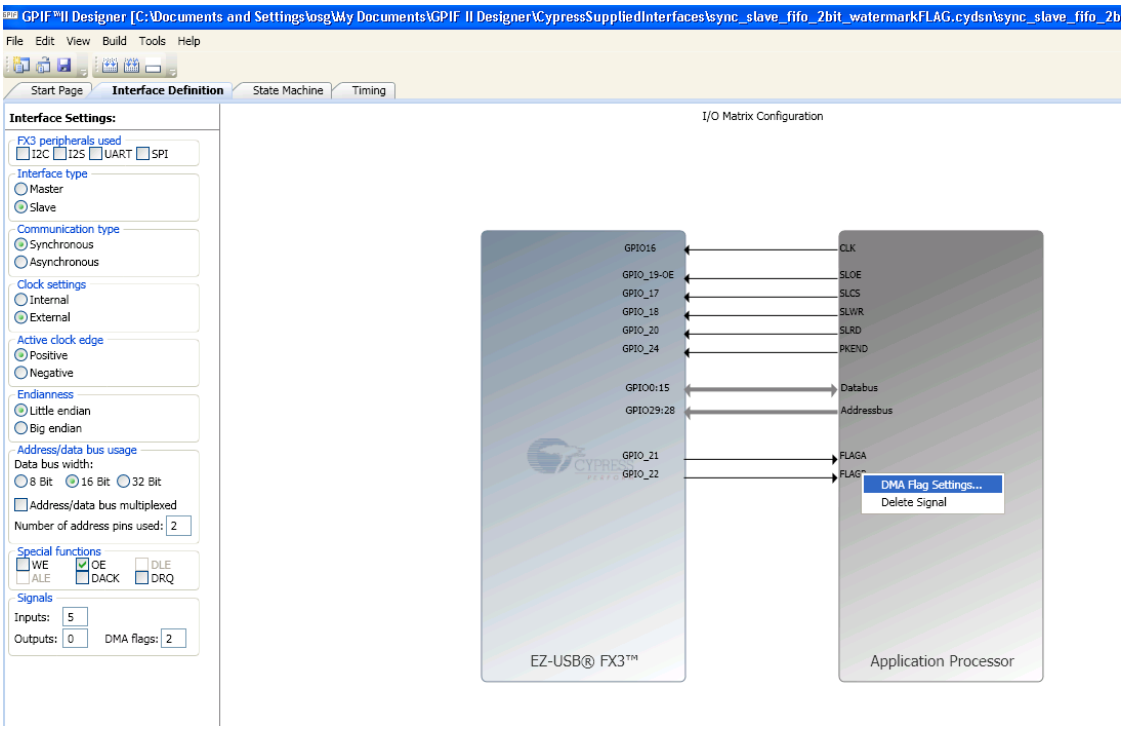
Figure 8. FLAG Settings in GPIFII Designer – Cypress Supplied Interfaces sync_slave_fifo_2bit



Note that if you would like to add more FLAGs or make changes other than what is allowed in the *sync_slave_fifo_2bit.cyfx* project, you can click on **File > Save Project as Editable**. This will allow you to save the project under a different name, after which changes can be made to the newly saved project.

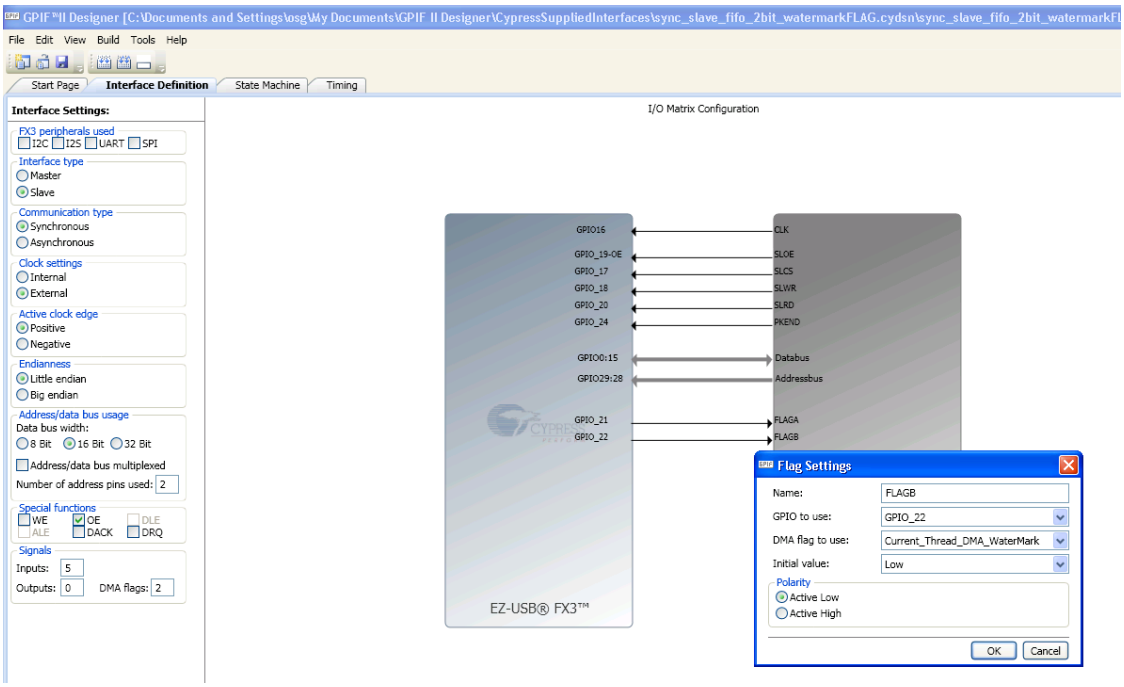
In this case to configure a FLAG as a partial FLAG, right click on the FLAG in the I/O Matrix Configuration diagram. Then click on DMA Flag Settings. This is shown in the following screenshot.

Figure 9. FLAG Settings after Creating a New Project by "Save Project as Editable" on the Default sync_slave_fifo_2bit



This will allow you to select the FLAG configuration as follows:

Figure 10. Selecting Specific FLAG Settings



The second step to configure a partial FLAG is to specify a watermark value for the partial FLAG. This watermark value must be specified in the firmware project. In the cyfxslfifo.c file add a call to the CyU3PGpifSocketConfigure() API to specify the watermark value. This call may be added just after the call to the CyU3PGPIFLoad() API. Refer to the EZ-USB FX3 SDK API Guide for a complete description of the CyU3PGpifSocketConfigure() API. One of the parameters input to this API is the watermark value.

The watermark value determines when a partial FLAG will be asserted. The following section describes the formula to calculate the number of data words that may be read or written after the partial FLAG is asserted.

General Formulas for using Partial FLAGs

The previous sections described the possible configurations for FLAGs and the steps to configure a partial FLAG. This section explains how a watermark value should be determined for a partial FLAG.

The following formulas should be used to precisely calculate the number of data words that may be read or written after the partial FLAG is asserted.

Note The watermark number specified in the CyU3PGpifSocketConfigure() API is in terms of a 32-bit data word.

1. When writing from external master to Sync Slave FIFO:

The number of data words that may be written AFTER the clock edge at which the partial FLAG is sampled low
= watermark x (32/bus width) - 4

2. When reading into external master from Sync Slave FIFO:

(a) The number of data words available for reading (while keeping SLOE# asserted) AFTER the clock edge at which the partial FLAG is sampled asserted = watermark x (32/bus width) - 1

(b) **Note** There is already two cycle latency from SLRD# to data. Hence, the number of cycles for which SLRD# may be kept asserted AFTER the clock edge at which the partial FLAG is sampled asserted = watermark x (32/bus width) -3

The following section describes some examples of how to set a watermark value and the corresponding effect on the FLAG. Screenshots are shown in order to clearly show the behavior of a partial FLAG for different watermark values. Note, these examples are with the FLAG polarities set to active low. Therefore the FLAGs go low to indicate full/empty or partially full/empty

Examples of CyU3PGpifSocketConfigure() API Usage

In the previous sections it was stated that the behavior of the partial FLAGs is dependent on the watermark value specified and the formulas for precise calculation of the watermark value were provided.

This section provide some examples of the effect of the watermark value specified by using the CyU3PGpifSocketConfigure() API. Screenshots are also shown in order to clearly show the behavior of a partial FLAG for different watermark values.

Note In these examples the FLAG polarities set to active low. Therefore the FLAGs go low to indicate full/empty or partially full/empty

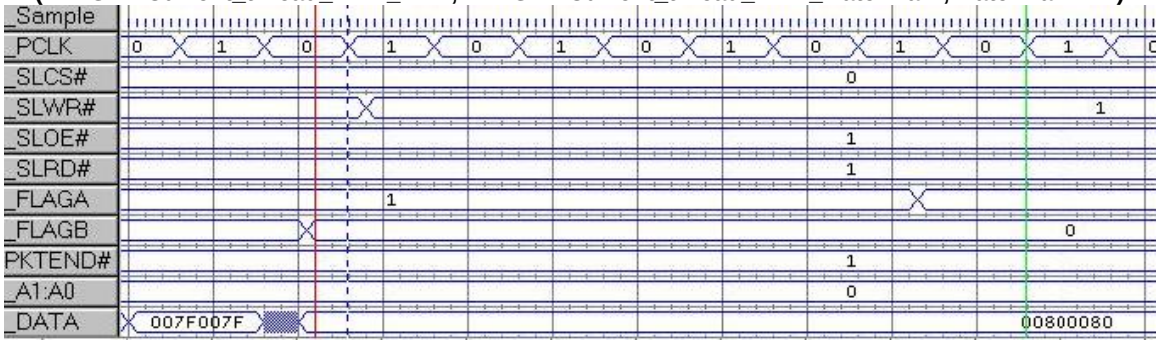
Example 1

- Sync Slave FIFO with 32-bit data bus:
- In GPIFII Designer, FLAGA is configured as Current_thread_DMA_RDY and FLAGB is configured as Current_thread_DMA_watermark.
- CyU3PGpifSocketConfigure (0,PIB_SOCKET_0,4,CyFalse,1)
- Burst write is performed from external FPGA to EZ-USB FX3 over Sync Slave FIFO. (last data to be written is 0x00800080)

Following is a logic analyzer screenshot of how the FLAGs go to 0 at the end of the transfer. It can be seen that FLAGB (partial FLAG) goes low in the same cycle in which the last word of data is written.

Figure 11. Burst Write Transfer with 32-bit Data Bus Width

(FLAGA= Current_thread_DMA_RDY, FLAGB= Current_thread_DMA_watermark, watermark = 4)



Example 2

- Sync Slave FIFO with 32-bit data bus:
- In GPIFII Designer, FLAGA is configured as Current_thread_DMA_RDY and FLAGB is configured as Current_thread_DMA_watermark.
- CyU3PGpifSocketConfigure (3, PIB_SOCKET_3, 4, CyFalse, 1)
- Burst read is performed by external FPGA from EZ-USB FX3 over Sync Slave FIFO. (last data to be read is 0x00000080)

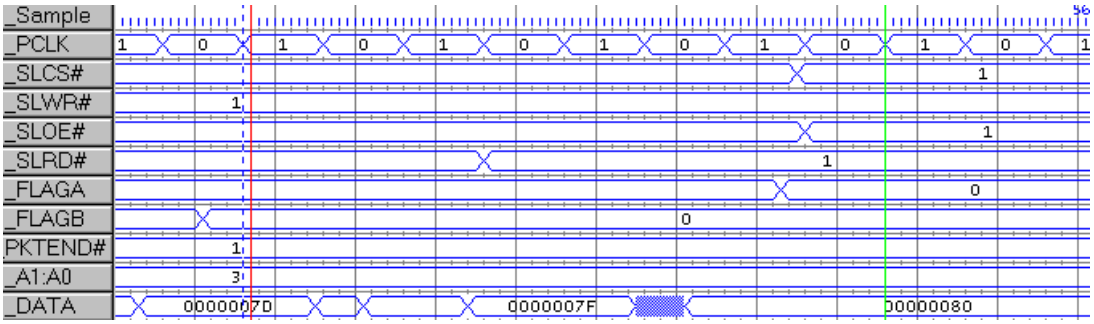
Following is a logic analyzer screenshot of how the FLAGs go to 0 at the end of the transfer. It can be seen that FLAGB (partial FLAG) goes low four cycles before the last data is read. That is, three words of data are available to be read out after the cycle in which FLAGB goes low.

Formula 2(a) from the section on General Formulas for using Partial FLAGs can be applied to this example as follows:

Watermark value =4, bus width =32

Therefore, number of 32-bit data words available for reading AFTER the clock edge at which the partial FLAG is sampled asserted = 4 x (32/32) – 1 = 3

Figure 12. Burst Read Transfer with 32-bit Data Bus Width (FLAGA= Current_thread_DMA_RDY, FLAGB= Current_thread_DMA_watermark, watermark = 4)



Example 3

- Sync Slave FIFO with 16-bit data bus:
- In GPIFII Designer, FLAGA is configured as Current_thread_DMA_RDY and FLAGB is configured as Current_thread_DMA_watermark.
- CyU3PGpifSocketConfigure (0, PIB_SOCKET_3, 3, CyFalse, 1)
- Burst write is performed from external FPGA to EZ-USB FX3 over Sync Slave FIFO. (last data to be written is 0x0200)

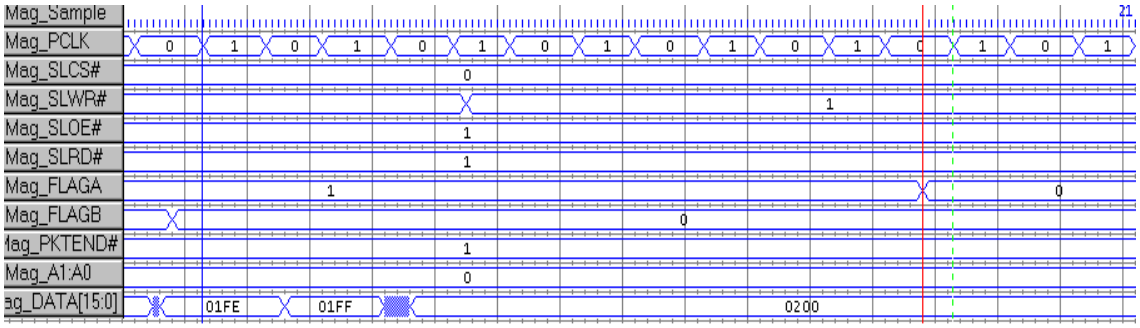
The following figure is a logic analyzer screenshot of how the FLAGs go to 0 at the end of the transfer. It can be seen that FLAGB (partial FLAG) goes low 3 cycles before the last data. i.e. 2 words of data may be written after the cycle in which FLAGB goes low.

Formula 1 from the section on General Formulas for using Partial FLAGs can be applied to this example as follows:

Watermark value =3, bus width =16

Therefore, number of 16-bit data words that may be written AFTER the clock edge at which the partial FLAG is sampled asserted = 3 x (32/16) – 4 = 2

Figure 13. Burst Write Transfer with 16-bit Data Bus Width (FLAGA= Current_thread_DMA_RDY, FLAGB= Current_thread_DMA_watermark, watermark = 3)

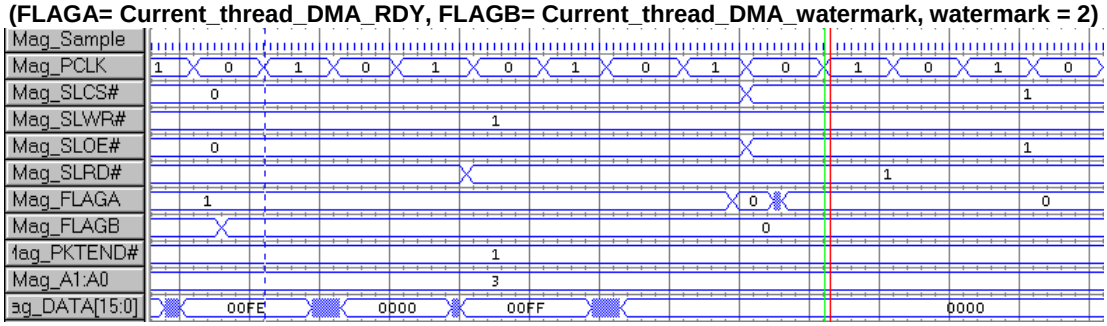


Example 4

- Sync Slave FIFO with 16-bit data bus:
- In GPIFII Designer, FLAGA is configured as Current_thread_DMA_RDY and FLAGB is configured as Current_thread_DMA_watermark.
- CyU3PGpifSocketConfigure (3, PIB_SOCKET_3, 2, CyFalse, 1)
- Burst read is performed by external FPGA from EZ-USB FX3 over Sync Slave FIFO. (last data to be read is 0x0000)

Following is a logic analyzer screenshot of how the FLAGs go to 0 at the end of the transfer. It can be seen that FLAGB (partial FLAG) goes low four cycles before the last data. That is, three words of data may be read after the cycle in which FLAGB goes low.

Figure 14. Burst Read Transfer with 16-bit Data Bus Width (FLAGA= Current_thread_DMA_RDY, FLAGB= Current_thread_DMA_watermark, watermark = 2)



Other Important Considerations when using Partial FLAG:

- A partial FLAG may only be used to decide when to end a transfer. The full/empty FLAG must be monitored at the start of transfer to ensure the availability of the socket. This means that a partial FLAG cannot be used by itself; it must be used in conjunction with a full/empty FLAG.
- The use of a partial FLAG may be completely avoided if the external master can implement a counting mechanism and always write an amount of data that equals the size of EZ-USB FX3's DMA buffer. The external master should count the data being written or read and ensure that the count does not exceed the buffer size set up when creating the DMA channel. In this case, a full/empty FLAG should be monitored to decide when to begin a transfer.
- If a counting mechanism is not implemented as described in (2) above and a partial FLAG is used one of the following will need to be done:
  - If the external master always bursts a fixed amount of data, this burst size must be considered when selecting a watermark value. For example, if the external master always writes in bursts of 8 words, then the watermark value may be set such that the partial FLAG goes low when there is space for a burst of 8 in EZ-USB FX3's DMA buffer. Then, having seen the partial FLAG as low, the external master can write one complete burst of 8. To achieve this for the write (to Slave FIFO) direction in 16-bit mode, the watermark value should be set to 6.
  - An alternative to (a) is that once the partial FLAG goes to 0, instead of performing a burst access, the external master can switch to single cycle access mode. Then at each cycle, before doing a write, the external master can check the full/empty FLAG to ensure that the buffer still has space.

Error Conditions Due To Flag Violations

A data read or write access to the Slave FIFO interface must not be done when the partial FLAG or full/empty FLAG indicates that a buffer is not available.

If a read access is performed on an empty buffer, a buffer under-run error will occur. If a write access is performed on a full buffer, a buffer over-run error will occur. These errors can lead to data corruption at buffer boundaries. The Slave FIFO interface is in the PIB block domain of FX3, hence any errors related to the Slave FIFO interface are indicated by a PIB error.

If a PIB error has occurred, a PIB interrupt will be triggered. The FX3 SDK allows a callback function to be registered for PIB interrupts. The callback function for the PIB interrupt can check the PIB error code. Following is an example of how a callback function can be registered and the actual callback function that checks for under-run/overrun errors and prints out a debug message.

The error code indicates which thread the error occurred on, as shown in Table 5. If one of these errors does occur, it is recommended to analyze the interface timing very carefully, using a logic analyzer, paying special attention to the number of read or write cycles being executed after the partial FLAG goes to 0. (For examples, refer to the screenshots shown in Figure 11 to Figure 14 in the previous section.

```
/* Register a callback for notification of PIB interrupts*/
CyU3PPibRegisterCallback(gpif_error_cb,intMask);

/* Callback function to check for PIB ERROR*/
void gpif_error_cb(CyU3PPibIntrType cbType, uint16_t cbArg)
{
       if(cbType==CYU3P_PIB_INTR_ERROR)
       {
              switch(CYU3P_GET_PIB_ERROR_TYPE(cbArg))
               {
                      case CYU3P_PIB_ERR_THR0_WR_OVERRUN:
                             CyU3PDebugPrint (4, "CYU3P_PIB_ERR_THR0_WR_OVERRUN");
                      break;
                      case CYU3P_PIB_ERR_THR1_WR_OVERRUN:
                             CyU3PDebugPrint (4, "CYU3P_PIB_ERR_THR1_WR_OVERRUN");
                      break;
                       case CYU3P_PIB_ERR_THR2_WR_OVERRUN:
                             CyU3PDebugPrint (4, "CYU3P_PIB_ERR_THR2_WR_OVERRUN");
                      break;
                      case CYU3P_PIB_ERR_THR3_WR_OVERRUN:
                             CyU3PDebugPrint (4, "CYU3P_PIB_ERR_THR3_WR_OVERRUN");
                      break;

                      case CYU3P_PIB_ERR_THR0_RD_UNDERRUN:
                             CyU3PDebugPrint (4, "CYU3P_PIB_ERR_THR0_RD_UNDERRUN");
                      break;
                      case CYU3P_PIB_ERR_THR1_RD_UNDERRUN:
                             CyU3PDebugPrint (4, "CYU3P_PIB_ERR_THR1_RD_UNDERRUN");
                      break;
                      case CYU3P_PIB_ERR_THR2_RD_UNDERRUN:
                             CyU3PDebugPrint (4, "CYU3P_PIB_ERR_THR2_RD_UNDERRUN");
                      break;
                      case CYU3P_PIB_ERR_THR3_RD_UNDERRUN:
                             CyU3PDebugPrint (4, "CYU3P_PIB_ERR_THR3_RD_UNDERRUN");
                      break;

                      default:
                             CyU3PDebugPrint (4, "No Underrun/Overrun Error");
                      break;
              }
        }
}
```

Table 5. PIB Error Codes for Overrun/Under-run Conditions

PIB Error Code	Description
CYU3P_PIB_ERR_THR0_WR_OVERRUN	Write overrun on thread 0 buffer
CYU3P_PIB_ERR_THR1_WR_OVERRUN	Write overrun on thread 1 buffer
CYU3P_PIB_ERR_THR2_WR_OVERRUN	Write overrun on thread 2 buffer
CYU3P_PIB_ERR_THR3_WR_OVERRUN	Write overrun on thread 3 buffer
CYU3P_PIB_ERR_THR0_RD_UNDERRUN	Read under-run on thread 0 buffer
CYU3P_PIB_ERR_THR1_RD_UNDERRUN	Read under-run on thread 1 buffer
CYU3P_PIB_ERR_THR2_RD_UNDERRUN	Read under-run on thread 2 buffer
CYU3P_PIB_ERR_THR3_RD_UNDERRUN	Read under-run on thread 3 buffer

Slave FIFO Firmware Examples in Software Development Kit

So far this application note described the details of the synchronous Slave FIFO interface and how the FLAGs can be configured. After having made the required configurations in the GPIFII Designer tool, the updated configuration needs to be integrated into the firmware. On building the project in GPIFII Designer, a header file *cyfxgpifconfig.h* is generated. This header file needs to be included in the firmware project. The EZ-USB FX3 SDK includes a firmware example that integrates the Slave FIFO interface.

After you install the EZ-USB FX3 software development kit (SDK), you can find the firmware example that integrates the synchronous Slave FIFO interface in the following directory:

```
[FX3 SDK Install Path] \EZ-USB FX3 SDK\1.2\firmware\slavefifo_examples\slfifosync
```

This firmware example supports both 16- and 32-bit data bus width. The constant CY_FX_SLFIFO_GPIF_16_32BIT_CONF_SELECT is defined in the header file *cyfxslfifosync.h*. To select the 32-bit data bus width, set this constant to 1; to select the 16-bit data bus width, set this constant to 0.

Note If the Slave FIFO should function at 100 MHz with 32 b data, you must configure the PLL frequency to 400 MHz. To do this, set the setSysClk400 parameter, as an input to the CyU3PDeviceInit() function. For more information, refer to the API Guide available with the FX3 SDK.

Design Example: Interfacing an FPGA to FX3's Synchronous Slave FIFO Interface

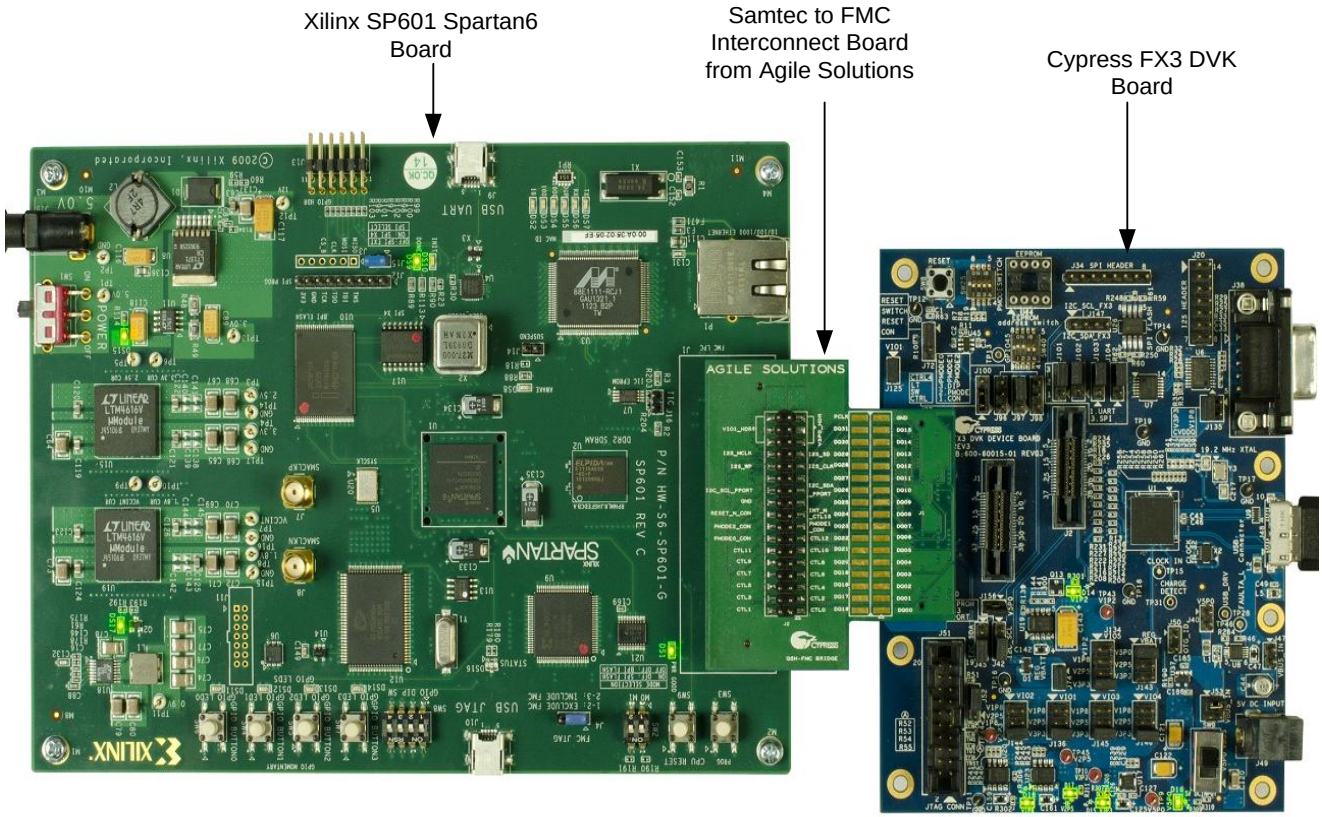
So far this application note described the details of the synchronous Slave FIFO interface, including the configuration and usage of different FLAGs. This section provides a complete design example in which, a Xilinx Spartan 6 FPGA is connected to FX3 over the synchronous Slave FIFO interface. Following are the hardware, firmware and software components that were used to implement this design:

Hardware Setup

The project provided in this example can be executed on a hardware setup consisting of a Cypress FX3 DVK board interconnected with a Xilinx Spartan 6 SP601 evaluation kit. The FX3 board and the Xilinx board are connected using a Samtec to FMC interconnect board. The interconnect board mates with the Samtec connector on the Cypress FX3 DVK board and the FMC connector on the Xilinx board. The interconnect board is available for ordering from Agile SDR Solutions.

Figure 15 shows a picture of this hardware setup.

Figure 15. Cypress FX3 DVK Board Connected to Xilinx SP601 Board using FMC Interconnect Board

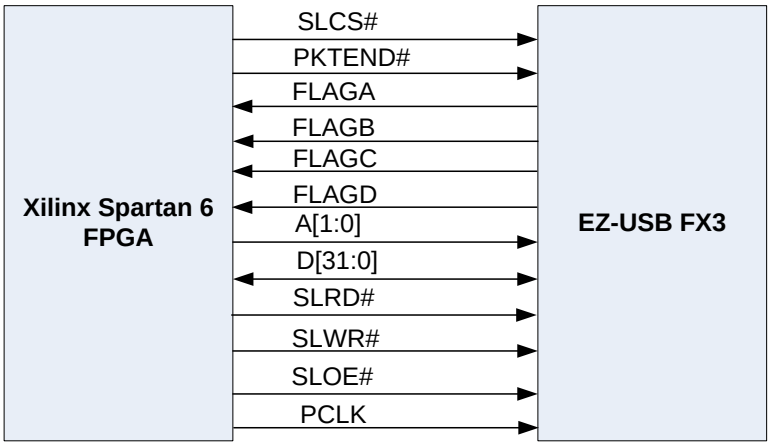


Firmware and Software Component of the Example

- FX3 synchronous Slave FIFO firmware project available with the FX3 SDK
- Control Center and Streamer software utilities available with the FX3 SDK

The following figure shows the interconnect diagram between the FPGA and FX3.

Figure 16. Interconnect Diagram between FPGA and FX3



The following are the components of this example:

1. Loopback transfer: In this component of the design, the FPGA first reads a complete buffer from FX3 and then writes it back to FX3. The USB host should issue OUT/IN tokens to transmit and then receive this data. The Control Center utility provided with the EZ-USB FX3 SDK can be used for this purpose.
2. Short packet: In this component of the design, the FPGA transfers a full packet followed by a short packet to FX3. The USB host should issue IN tokens to receive this data.
3. Zero length packet (ZLP) transfer: In this component of the design, the FPGA transfers a full packet followed by a zero length packet to FX3. The USB host should issue IN tokens to receive this data.
4. Streaming (IN) data transfer: In this component of the design, the FPGA does one directional transfers, that is, continuously writes data to FX3 over synchronous Slave FIFO. The USB host should issue IN tokens to receive this data. The Control Center or Streamer utility provided with the EZ-USB FX3 SDK can be used for this purpose.
5. Streaming (OUT) data transfer: In this component of the design, the FPGA does one directional transfers, i.e. continuously reads data from FX3 over synchronous Slave FIFO. The USB host should issue OUT tokens to provide this data. The Control Center or Streamer utility provided with the EZ-USB FX3 SDK can be used for this purpose.

Details of Project Files

1. Loopback project

In this component of the design, the FPGA first reads a complete buffer from FX3 and then writes it back to FX3. The USB host should issue OUT/IN tokens to transmit and then receive this data. The Control Center utility provided with the EZ-USB FX3 SDK can be used for this purpose.

The purpose of the loopback example is to demonstrate read and write operations over Slave FIFO.

This project is contained in the file *Loopback.zip*. On extracting, this folder will contain the following:

- *Fpga_master.bit* – This is the FPGA Verilog code that implements the loopback transfer. The source for this is contained in the fpga_write folder.
- *SlaveFifoSync_loopback.img* – This is the FX3 firmware image that contains the Slave FIFO implementation and sets up the DMA channels required for loopback transfer. The source for this is found in the FX3_fw_slfifosync directory.
- *TEST.txt* – This is the file that contains the data pattern that is sent using the Transfer-FILE OUT button in the Control Center utility.

2. Stream IN/OUT + Short packet + ZLP project

This project demonstrates the following types of transfers:

- **Streaming (IN) data transfer:** In this component of the design, the FPGA does one directional transfers, i.e. continuously writes data to FX3 over synchronous Slave FIFO. The USB host should issue IN tokens to receive this data. The Control Center or Streamer utility provided with the EZ-USB FX3 SDK can be used for this purpose.
- **Streaming (OUT) data transfer:** In this component of the design, the FPGA does one directional transfers, i.e. continuously reads data from FX3 over synchronous Slave FIFO. The USB host should issue OUT tokens to provide this data. The Control Center or Streamer utility provided with the EZ-USB FX3 SDK can be used for this purpose.

  The streaming IN and OUT transfers demonstrate the performance that can be achieved over the synchronous Slave FIFO interface.
- **Short packet:** In this component of the design, the FPGA transfers a full packet followed by a short packet to FX3. The USB host should issue IN tokens to receive this data.
- **Zero length packet (ZLP) transfer:** In this component of the design, the FPGA transfers a full packet followed by a zero length packet to FX3. The USB host should issue IN tokens to receive this data.

  The purpose of the short packet and ZLP examples is to demonstrate how short packets and ZLPs can be transferred over the synchronous Slave FIFO interface.

This project is contained in the file *Stream_Shrt_ZLP.zip*. On extracting, this folder will contain the following:

- *fpga_master.bit* - This is the FPGA Verilog code that implements the streaming (IN and OUT) transfers, as well as short packet and ZLP transfers. The source for this is found in the fpga_write directory.
- *SF_streamIN.img, SF_streamOUT.img* – These are the FX3 firmware images that contains the Slave FIFO implementation and set up the DMA channels required for optimized performance when streaming data.
- *SF_shrt_ZLP.img* – This is the FX3 firmware image that contains the Slave FIFO implementation and sets up the DMA channels required for demonstrating short packet and ZLP transfers.

The source for the FX3 firmware is found in the *FX3_fw_slfifosync* directory.

Note The only differences between the different FX3 firmware images provided is in the way the DMA channels are set up, for the ease of demonstration of the different types of transfers. But any one firmware image can be used for demonstrating any type of transfer.

Details of FX3 Firmware

The FX3 firmware is based on the example project contained in the FX3 SDK.

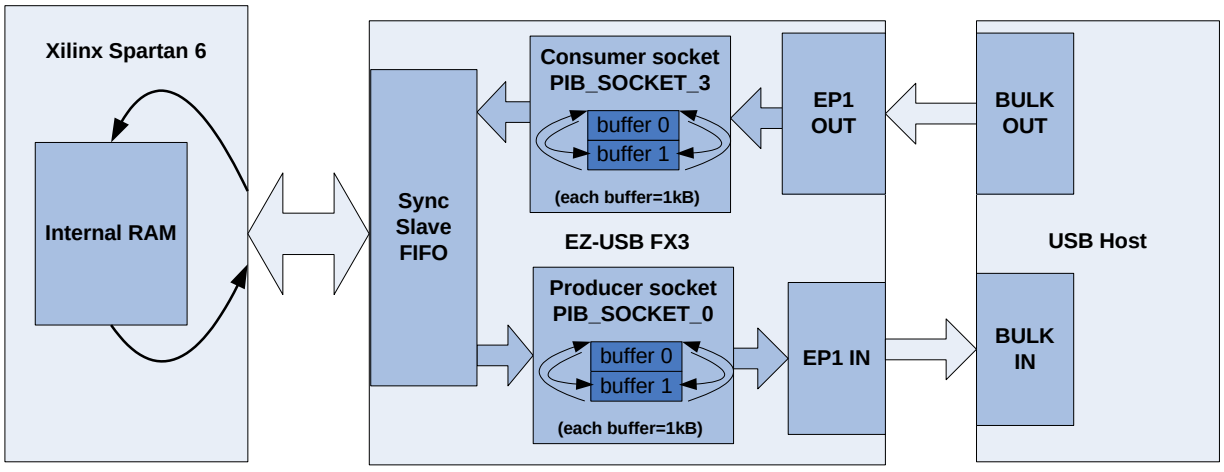
Following are the main features of this firmware:

1. Enables both USB3.0 and USB2.0
2. Enumerates with the Cypress VID/PID = 0x04B4/0x00F1. This enables the use of the Cypress utilities Control Center and Streamer for initiating USB transfers.
3. Integrates the synchronous Slave FIFO descriptor which:
   a. Supports access to up to 4 sockets.
   b. Configures data bus width to 32 b
   c. Works on the 100 MHz PCLK input clock
   d. Configures 4 FLAGS as follows
      i. FLAGA: Full flag dedicated to thread0
      ii. FLAGB: Partial flag with watermark value 6, dedicated to thread0
      iii. FLAGC: Empty flag dedicated to thread3
      iv. FLAGD: Partial flag with watermark value 6, dedicated to thread3

   Note The GPIFII Designer project provided with this application note shows how the above settings are done. Also, the firmware project shows the usage of the CyU3PGpifSocketConfigure() API to configure the watermark value.
4. Configures the PLL frequency to 400 MHz. This is done by setting the setSysClk400 parameter, as an input to the CyU3PDeviceInit() function.

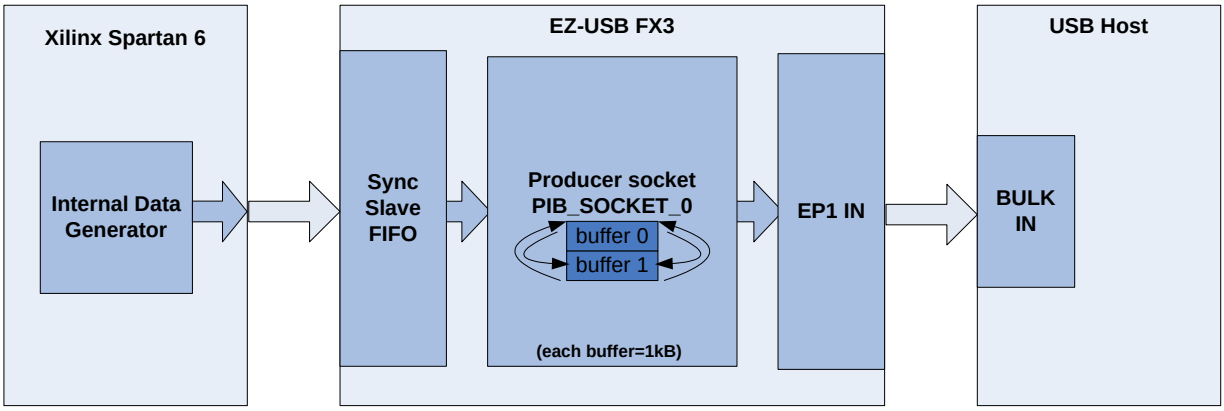
   Note This setting is essential for the functioning of Slave FIFO at 100 MHz with 32 b data.
5. Sets up the DMA channels as follows:
   a. For loopback transfers, short packet and ZLP transfer, two DMA channels are created:
      i. A P2U channel with PIB_SOCKET_0 as the producer and UIB_SOCKET_1 as the consumer. The DMA buffer size is 512 or 1024 depending on whether the USB connection is USB2.0 or USB3.0. The DMA buffer count is 2.
      ii. A U2P channel with PIB_SOCKET_3 as the consumer and UIB_SOCKET_1 as the producer. The DMA buffer size is 512 or 1024 depending on whether the USB connection is USB2.0 or USB3.0. The DMA buffer count is 2.

Figure 17. Setup for Loopback Transfer



Note Only the P2U channel is used for short packets and ZLPs.

Figure 18. Setup for Short Packet and ZLP Transfers



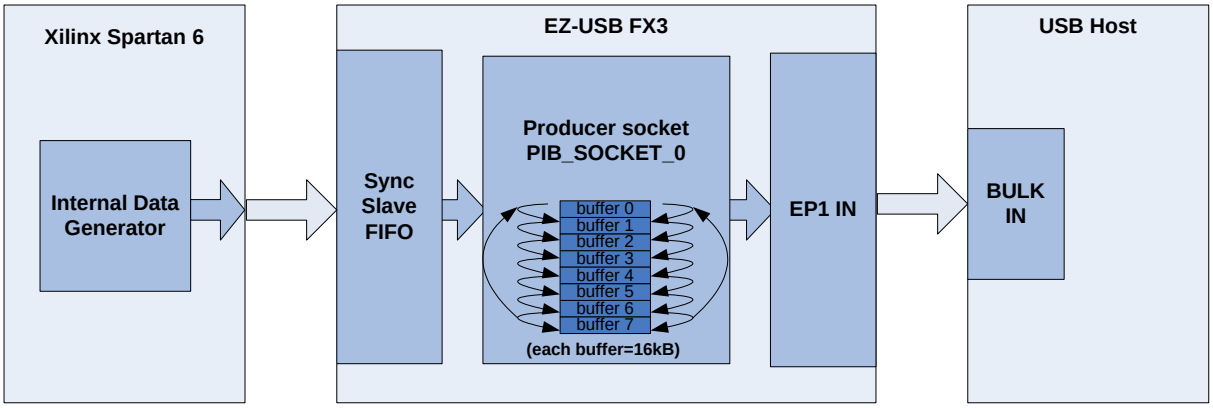
The DMA channels as described above are set up if the following define is enabled in the *cyfxslfifosync.h* file in the FX3 firmware project provided with this application note.

```
/* set up DMA channel for loopback/short packet/ZLP transfers */
#define LOOPBACK_SHRT_ZLP
```

b. For streaming, two DMA channels are created:

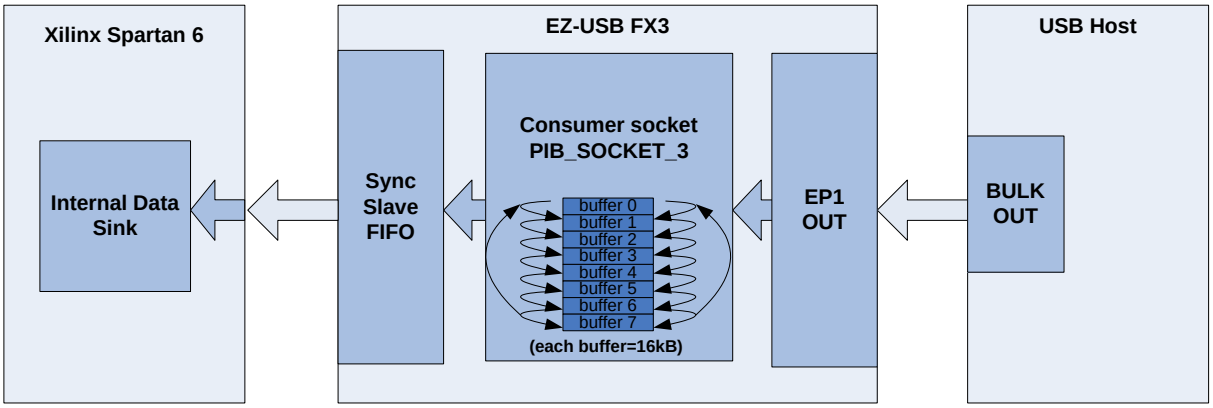
i. A P2U channel with PIB_SOCKET_0 as the producer and UIB_SOCKET_1 as the consumer. The DMA buffer size is 16*1024 (for a USB3.0 connection) or 16*512 (for a USB2.0 connection) depending on whether the USB connection is USB2.0 or USB3.0. The DMA buffer count is 8. This buffer size and count is chosen to provide high throughput performance.

Figure 19. Setup for Stream IN Transfers – Buffer Count and Size Optimized for Performance



ii. A U2P channel with PIB_SOCKET_3 as the consumer and UIB_SOCKET_1 as the producer. The DMA buffer size is 16*1024 (for a USB3.0 connection) or 16*512 (for a USB2.0 connection). The DMA buffer count is 4. Note, the buffer count can be increased further to enhance performance, but then the buffer count of the P2U channel should be reduced. This is because the FX3 SDK does not provide enough buffer memory such that both channels can have a buffer size of 16*1024 and buffer count of 8.

Figure 20. Setup for Stream OUT Transfers – Buffer Count and Size Optimized for Performance



The DMA channels as above are set up if the following define is enabled in the *cyfxslfifosync.h* file in the FX3 firmware project provided with this application note.

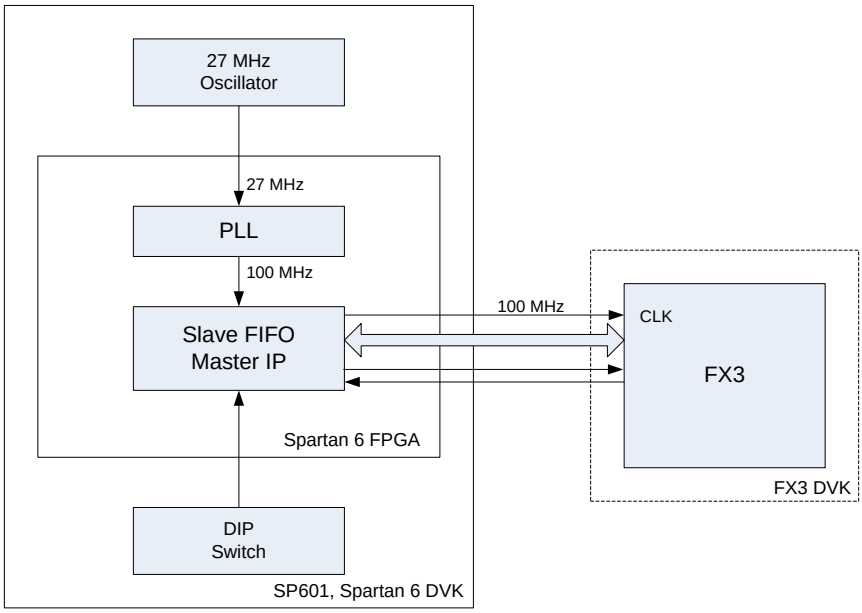
```
/* set up DMA channel for stream IN/OUT transfers */
#define STREAM_IN_OUT
```

The buffer count allocated to the P2U and U2P DMA channels can be controlled by using the following defines, also in the *cyfxslfifosync.h* file :

```
/* Slave FIFO P_2_U channel buffer count */
#define CY_FX_SLFIFO_DMA_BUF_COUNT_P_2_U      (4)
/* Slave FIFO U_2_P channel buffer count */
#define CY_FX_SLFIFO_DMA_BUF_COUNT_U_2_P      (8)
```

Details of FPGA Implementation

Figure 21. Xilinx Spartan 6 (XC6SLX16) FPGA Implementation using SP601 Evaluation Kit



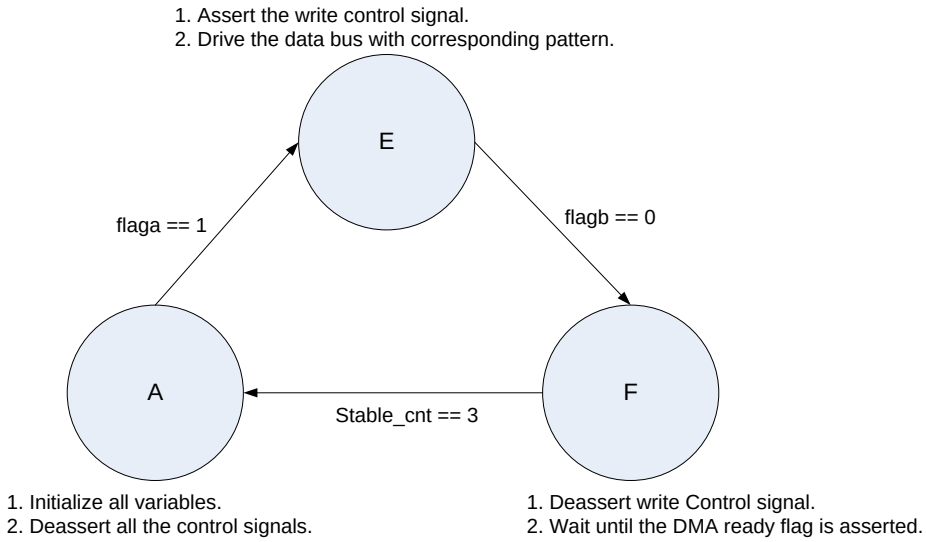
To demonstrate the maximum performance of FX3 the GPIF interface is running at 100 MHz. The SP601 has an on - board 27 MHz single ended oscillator. The FPGA uses a PLL to generate a 100 MHz clock from the 27 MHz clock.

Following are the state implementations of the different types of transfers.

Stream IN example [FPGA writing to Slave FIFO]

The state machine implemented in Verilog RTL for the stream IN example is shown below.

Figure 22. FPGA State Machine for Stream IN



State A:

This state initializes all the registers and signals used in the state machine. The status of the Slave FIFO control lines is as follows:

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 0

State E:

Whenever FLAGA =1 the state machine will enter this state. Here the state machine will assert the write control signal as follows,

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 0; A[1:0] = 0

State F:

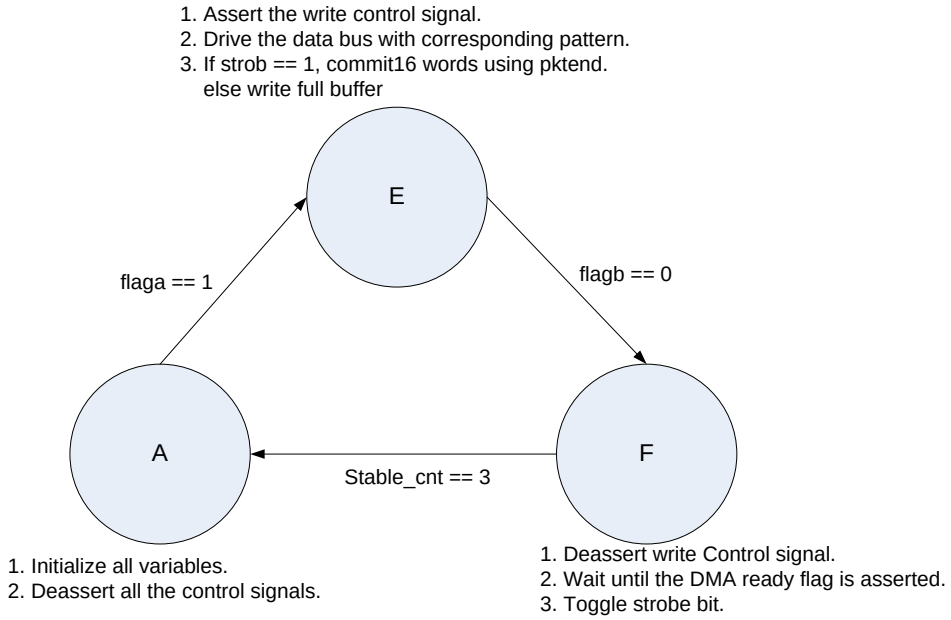
This state is entered after 1 clock cycle of FLAGB assertion. This state de-asserts all the write control lines. As per the formula (1) in the section "General Formulas for using Partial FLags", FX3 should sample SLWR# asserted for 2 cycles after the partial FLAG goes to 0. Considering a one cycle propagation delay through the FPGA and to the interface, the FPGA asserts SLWR# for a count of 1 cycle after sampling the FLAGB as 0. As the watermark value is 6 we expect FLAGA to go to 0, only 6 clock cycles after the FLAGB. This state holds the execution for 4 clock cycles to ensure the availability of a valid status on FLAGA.

Short packet example [FPGA writing full packet followed by short packet to Slave FIFO]

This example demonstrates the short packet commit procedure using PKTEND#.

The state machine implemented in verilog RTL for short packet example is shown on the next page.

Figure 23. State Machine for Short Packet Transfer



State A:

This state initializse all the registers and signals used in the state machine. This state initializes all the registers and signals used in the state machine. The status of the Slave FIFO control lines is as follows:

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 0

State E:

Whenever FLAGA=1 the state machine will enter this state. If strob == 1 then commit a short packet of 16 words, else commit a full packet.

State F:

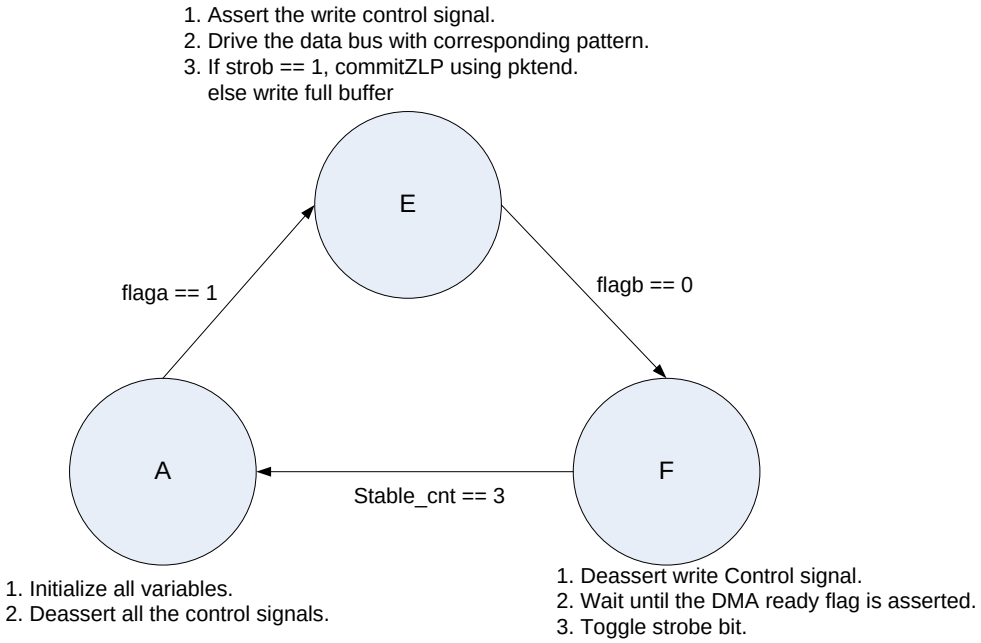
This state is entered after 1 clock cycle of FLAGB assertion. This state de-asserts all the write control lines. As per the formula (1) in the section "General Formulas for using Partial FLags", FX3 should sample SLWR# asserted for 2 cycles after the partial FLAG goes to 0. Considering a one cycle propagation delay through the FPGA and to the interface, the FPGA asserts SLWR# for a count of 1 cycle after sampling the partial FLAG as 0. As the watermark value is 6 we expect FLAGA to go to 0, only 6 clock cycles after the partial flag. This state holds the execution for 4 clock cycles to ensure the availability of a valid status on FLAGA.

Zero Length packet example [FPGA writing full packet followed by ZLP to Slave FIFO]

This example demonstrates the ZLP commit procedure using PKTEND#.

The state machine implemented in Verilog RTL for ZLP example is shown on the next page.

Figure 24. State Machine for ZLP Transfer



State A:

This state initializes all the registers and signals used in the state machine. This state initializes all the registers and signals used in the state machine. The status of the Slave FIFO control lines is as follows:

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 0

State E:

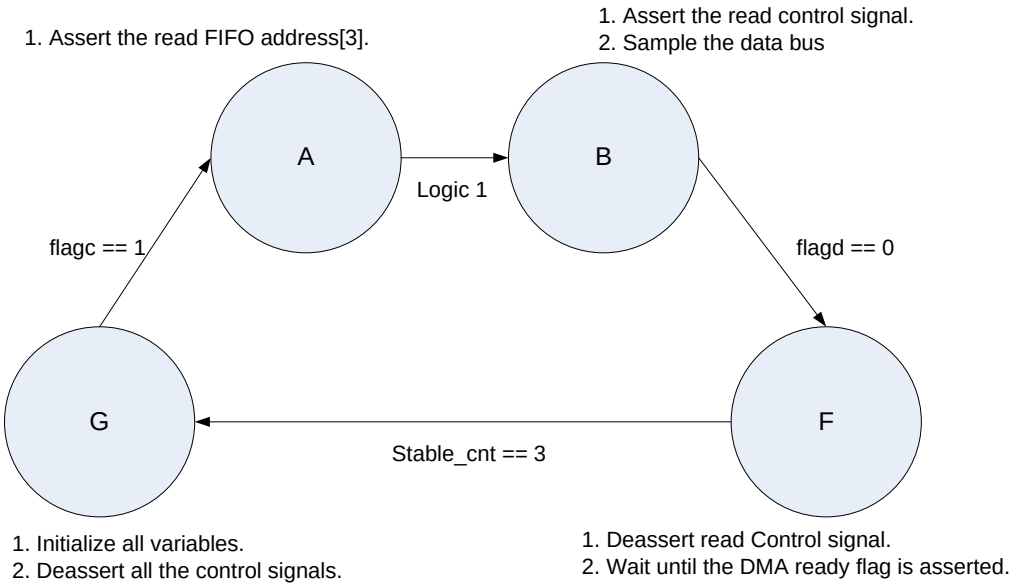
Whenever FLAGA=1 the state machine will enter this state. If strob == 1 then commit a ZLP else commit a full packet.

State F:

This state is entered after 1 clock cycle of FLAGB assertion. This state de-asserts all the write control lines. As per the formula (1) in the section General Formulas for using Partial FLAGs, FX3 should sample SLWR# asserted for 2 cycles after the partial FLAG goes to 0. Considering a one cycle propagation delay through the FPGA and to the interface, the FPGA asserts SLWR# for a count of 1 cycle after sampling the partial FLAG as 0. As the watermark value is 6 we expect FLAGA to go to 0, only 6 clock cycles after the partial flag. This state holds the execution for 4 clock cycles to ensure the availability of a valid status on FLAGA.Stream OUT example [FPGA reading from Slave FIFO]:

State Machine Implemented in Verilog RTL for Stream OUT Example

Figure 25. State Machine for Stream OUT Transfer



State G:

This state initializes all the registers and signals used in the state machine. The status of the Slave FIFO control lines is as follows:

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] =3

State A:

Whenever FLAGC=1, the state machine will enter in to this state.

State B:

Here the state machine will assert read control signals as follows,

PKTEND# = 1; SLOE# = 0; SLRD# = 0; SLCS# = 0; SLWR# = 1; A[1:0] =3

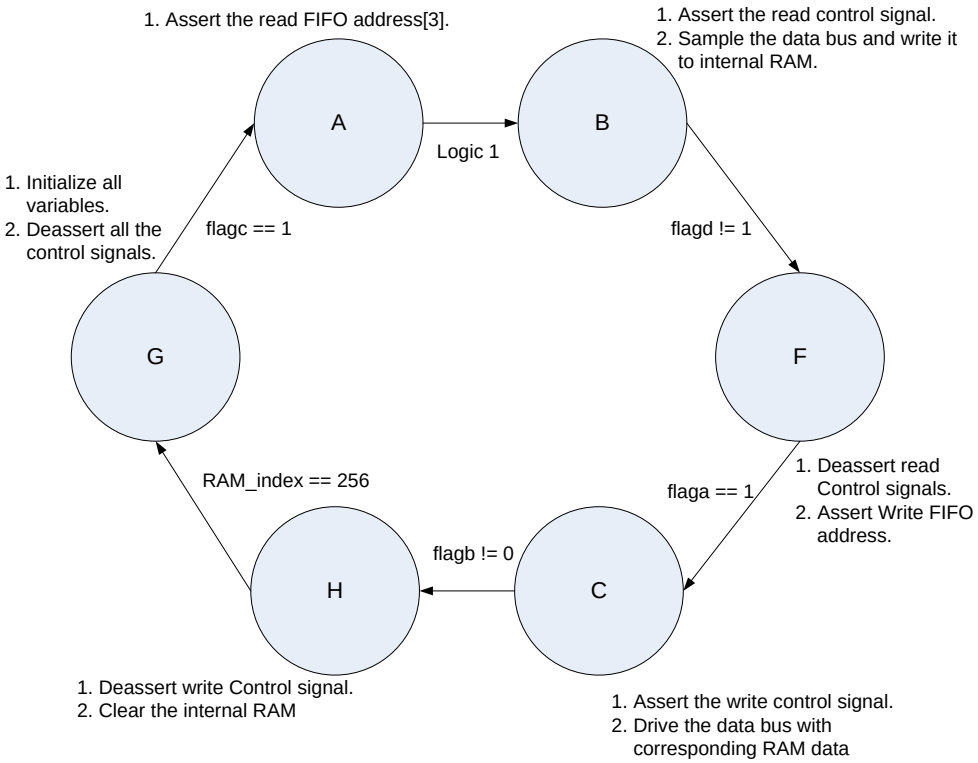
State F:

This state is entered after 4 clock cycles of FLAGD going to 0. As per the formula (2b) in the section General Formulas for using Partial FLAGs FX3 should sample SLRD# asserted for 3 cycles after the partial FLAG goes to 0. Considering a one cycle propagation delay through the FPGA and to the interface, the FPGA asserts SLRD# for a count of 2 cycles after sampling FLAGD as 0. As the watermark value is 6 we expect FLAGC to go to 0, only 6 clock cycles after the partial flag. This state holds the execution for 4 clock cycles to ensure the availability of a valid status on FLAGC.

Loopback example [FPGA reading from Slave FIFO and writing the same data back to Slave FIFO]:

There are six states through which the state machine moves before the completion of one loopback cycle. The state machine along with the corresponding actions is shown below.

Figure 26. State Machine for Loopback Transfer



State G:

This state initializes all the registers and signals used in the state machine.

The status of the Slave FIFO control lines is as follows:

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] =3

State A:

Whenever the FLAGC=1 the state machine will enter this state. This state ensures that the address is setup on the interface one cycle before SLRD# is asserted.

State B:

The control signals for Slave FIFO read operation are asserted in this state. The data bus is sampled and data is written to internal RAM in this state. The status of the Slave FIFO control lines is as follows:

PKTEND# = 1; SLOE# = 0; SLRD# = 0; SLCS# = 0; SLWR# = 1; A[1:0] =3

Note, there is a 2 cycle latency from SLRD# to data. This latency is accounted for when reading the data. This state keeps polling the partial flag and exits when it is not 1.

State F:

This state deasserts all the read control lines, and it drives the FIFO address bus with FIFO address to be written (A[1:0] = 0).

State C:

Whenever the FLAGA=1, the state machine will enter this state. Here the state machine will assert write control signals as follows

PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 0; A[1:0] =0

The data bus will be driven with data in this state.

State H:

This state deasserts all the write control lines. The internal RAM buffer will be cleared in this state. The state of the control signals is as follows:

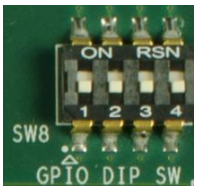
PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] =0

Project Operation

Steps to Test Loopback Transfer

1. Connect the FX3 DVK board with the Xilinx SP601 DVK board via the FMC connector and power on both the FX3 DVK board and the Xilinx SP601 DVK board
2. The FPGA and FX3 both must be configured before the FPGA may start any transaction. The FPGA code uses a GPIO input to determine when it may start. GPIO switch SW8 on the SP601 DVK board is used for this purpose. Before configuring the FPGA, ensure that position 4 of SW8 is in the OFF position as shown in the following figure.

Figure 27. SW8 on Xilinx SP601 Board – all OFF before FPGA and FX3 Configuration



3. Program the Xilinx Spartan 6 FPGA with the file fpga_loopback.bit. The FPGA can be programmed with any standard programmer such as the iMPACT application available with the Xilinx ISE Design Suite.
4. Program the FX3 device with the firmware image file SF_loopback.img. The FX3 can be programmed from the USB host using the Control Center utility available with the FX3 SDK.

Figure 28. Programming FX3 Firmware using Control Center

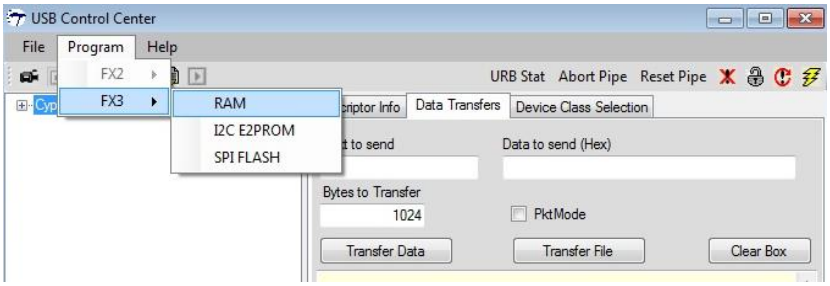
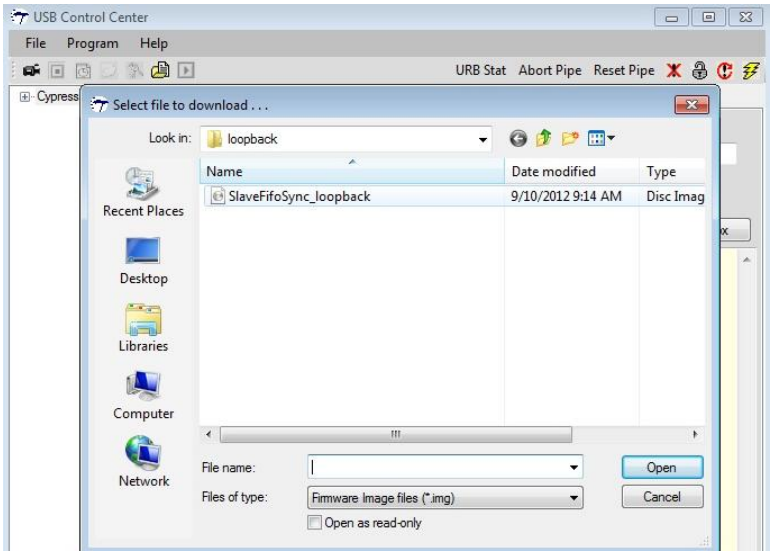
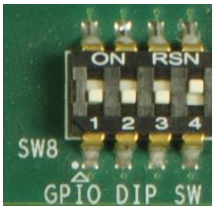


Figure 29. Programming FX3 Firmware for Loopback Testing using Control Center



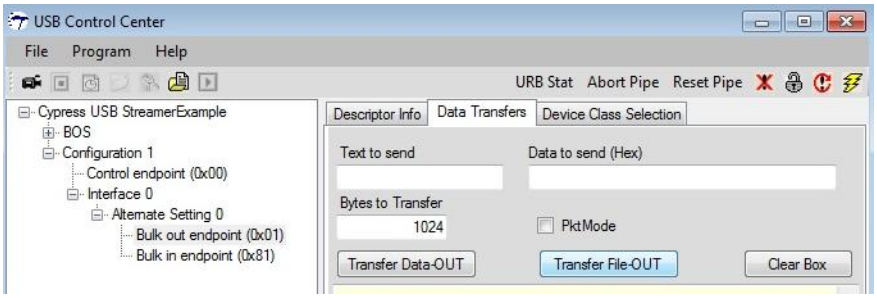
5. On firmware download, the FX3 device will enumerate as a SuperSpeed device (if connected to a USB3.0 port). Now before data transfers can be initiated, the position 4 on SW8 on the Xilinx SP601 board must be turned to the ON position as shown in the following figure.

Figure 30. SW8 on Xilinx SP601 Board after FPGA and FX3 Configuration is Complete



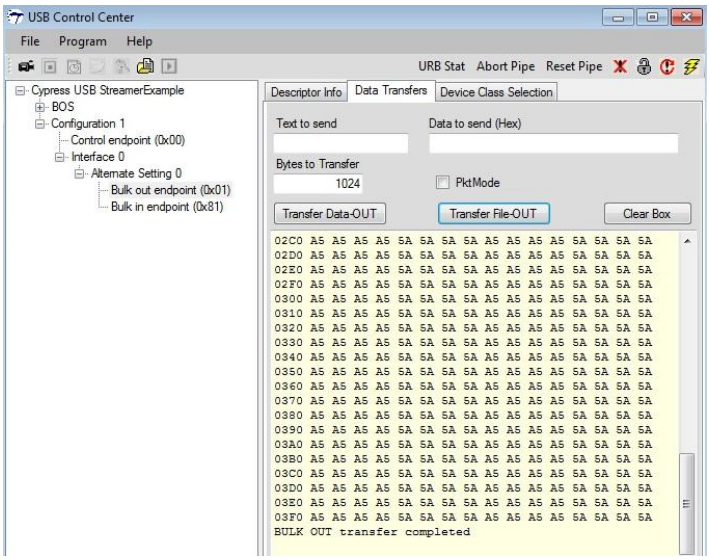
6. Now transfers can be initiated from the Control Center utility. First initiate a Bulk OUT transfer from the USB host. Select the Bulk OUT endpoint in Control Center and then click on the Transfer File OUT button.

Figure 31. Initiate Bulk OUT Transfer using the Transfer File OUT option



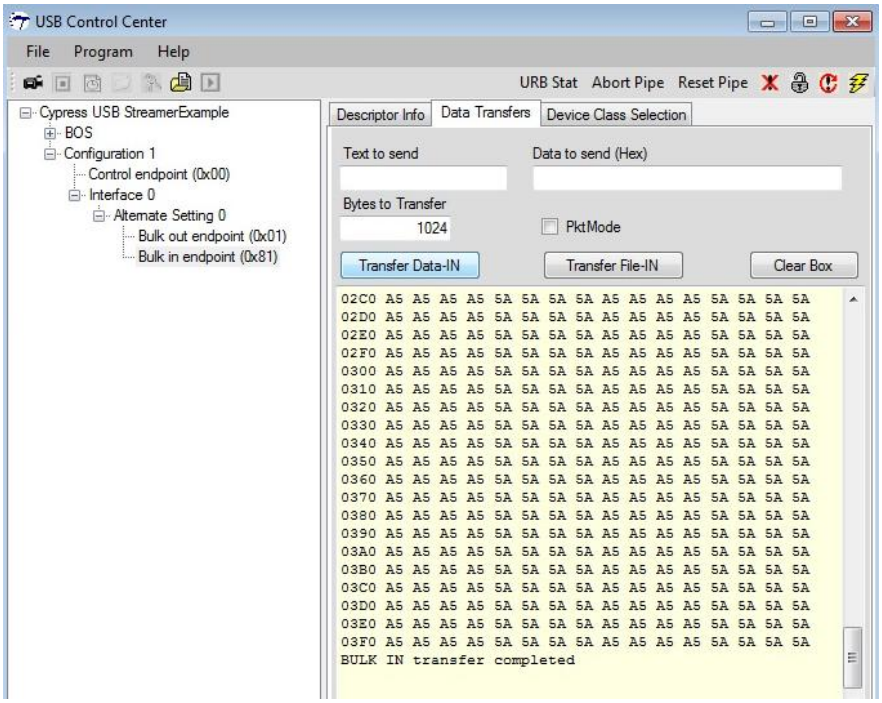
7. This will allow you to browse and select a file containing the data to transfer. In the attachment to this appnote, in the Loopback folder, you will find a TEST.txt file. This contains a data pattern which will send out “0xA5A5A5A5 0x5A5A5A5A” in an alternating manner. Double click to select the file and send out the data.

Figure 32. Data Pattern Transferred by Selecting TEST.txt File for Transfer FILE-OUT



8. Now the FPGA is already in a state where it is waiting for FLAGA to equal 1. As soon as the data is available in the buffer of PIB_SOCKET_0, the FPGA will read it. The FPGA will then loopback the same data and write it to FX3’s PIB_SOCKET_3.
9. Now from the USB host you can issue a Bulk IN transfer. Select the BULK IN endpoint in Control Center and click on Transfer Data IN. The same data that was previously written is now read back.

Figure 33. Complete Loopback Test by Initiating a BULK IN Transfer using Transfer Data-IN



Steps to Test Streaming Transfers

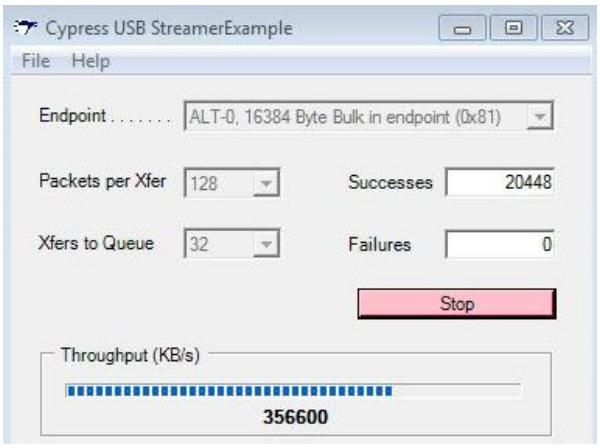
1. Connect the FX3 DVK board with the Xilinx SP601 evaluation board via the FMC connector and power on both the FX3 DVK board and the Xilinx SP601 board.
2. Program the Xilinx Spartan 6 FPGA with the file fpga_stream_short_zlp.bit. The FPGA can be programmed with any standard programmer such as the iMPACT application available with the Xilinx ISE Design Suite. Note, the Verilog code for the streaming, short packet and ZLP is contained in the same file. The FPGA uses GPIO inputs to determine which of these transfers to execute. Switch SW8 on the SP601 DVK board is used for this purpose. The switch setting required for the different transfers is shown in Table 6.
3. When the FPGA is first programmed, SW8 should be configured with SW8[4] = OFF. This is so that the FX3 firmware can be programmed before the FPGA starts any transfers. Once the FX3 firmware has been programmed, SW8 should be configured for the required transfer.

Table 6. Configuration of FPGA Transfer Modes in fpga_stream_short_zlp.bit

SW8[4]	SW8[3]	SW8[2]	SW8[1]	FPGA Transfer mode
ON	OFF	OFF	OFF	FPGA continuously writes a full packet followed by short packet
ON	OFF	OFF	ON	FPGA continuously writes full packet followed by ZLP
ON	OFF	ON	OFF	FPGA continuously writes full packets (Stream Bulk IN packets from the host)
ON	OFF	ON	ON	FPGA continuously reads full packets (Stream Bulk IN packets from the host)
OFF	X	X	X	Invalid

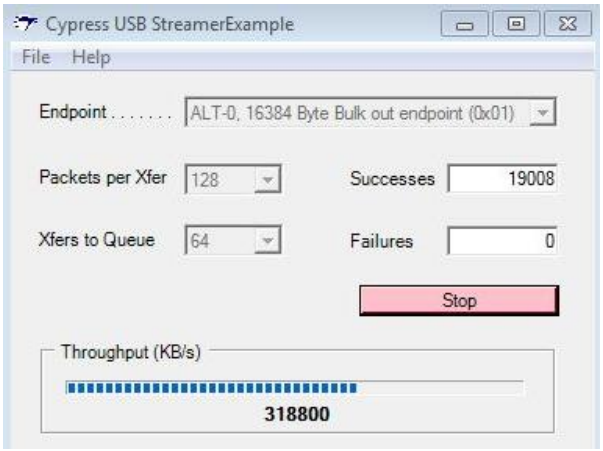
4. For Stream IN or OUT, program the FX3 device with the firmware image file SF_streamIN.img. The FX3 can be programmed from the USB host using the Control Center utility available with the FX3 SDK.
5. On firmware download, the FX3 device will enumerate as a SuperSpeed device (if connected to a USB3.0 port). Now data transfers can be initiated from the Control Center utility and also the Streamer utility provided with the FX3 SDK.
6. Set the switch SW8 on the SP601 board appropriately, for the required transfer, as shown in Table 6.
7. In the stream IN case, now the FPGA is already in a state where it is waiting for FLAGA to equal 1. As soon as the buffer is available, the FPGA will start writing continuously to FX3's PIB_SOCKET_0. Now from the USB host you can issue continuous Bulk IN transfers. Select the BULK IN endpoint in the Cypress Streamer utility and click on Start. The performance number is displayed. The performance shown in Figure 34 is observed on a Win7 64-bit PC with an Intel Z77 Express Chipset.

Figure 34. Streaming IN Performance Displayed in the Cypress Streamer Utility



8. In the stream OUT case, now the FPGA is already in a state where it is waiting for FLAGC to equal 1. As soon as the data is available, the FPGA will start reading continuously from FX3's PIB_SOCKET_3. Now from the USB host you can issue continuous Bulk OUT transfers. Select the BULK OUT endpoint in the Cypress Streamer utility and click on Start. The performance number is displayed. The performance shown in Figure 35 is observed on a Win7 64-bit PC with an Intel Z77 Express Chipset.

Figure 35. Streaming OUT Performance displayed in the Cypress Streamer Utility



Note the same FX3 firmware image SF_streamIN.img can be used for both streaming IN and OUT. In the SF_streamIN.img, 8 buffers have been allocated to the P2U DMA channel and 4 buffers have been allocated to the U2P channel. Whereas in the SF_streamOUT.img, 8 buffers have been allocated to the U2P DMA channel and 4 buffers have been allocated to the P2U channel. Hence higher P2U performance will be demonstrated by the SF_streamIN.img firmware and a higher U2P performance will be demonstrated by the SF_streamOUT.img firmware file.

Steps to test Short packet and ZLP transfers:

1. The FPGA can be kept programmed with the same bit file as above for the streaming transfers. You only need to configure the switch settings as shown in Table 6.
2. Program the FX3 device with the firmware image file SF_shrt_ZLP.img. The FX3 can be programmed from the USB host using the Control Center utility available with the FX3 SDK.
3. As soon as the FX3 firmware is programmed, a buffer allocated to PIB_SOCKET_0 becomes available. The FPGA is already in a state where is it waiting for this condition, by monitoring FLAGA. As soon as FLAG equals 1, the FPGA starts writing to FX3.
4. If the switch is configured for short packet transfers, the FPGA writes a full packet (1024 bytes) followed by a short packet. If the switch is configured for ZLP transfers, the FPGA writes a full packet (1024 bytes) followed by a ZLP.
5. Now the USB host can issue Bulk IN tokens. In the Control Center utility, select the Bulk IN endpoint and then click on Transfer DATA IN. First the full packet will be received. Click on Transfer DATA IN again. Now the short packet or ZLP will be received.

Figure 36. Full Packet Followed by Short Packet Received by Consecutive Transfer Data-IN Operations

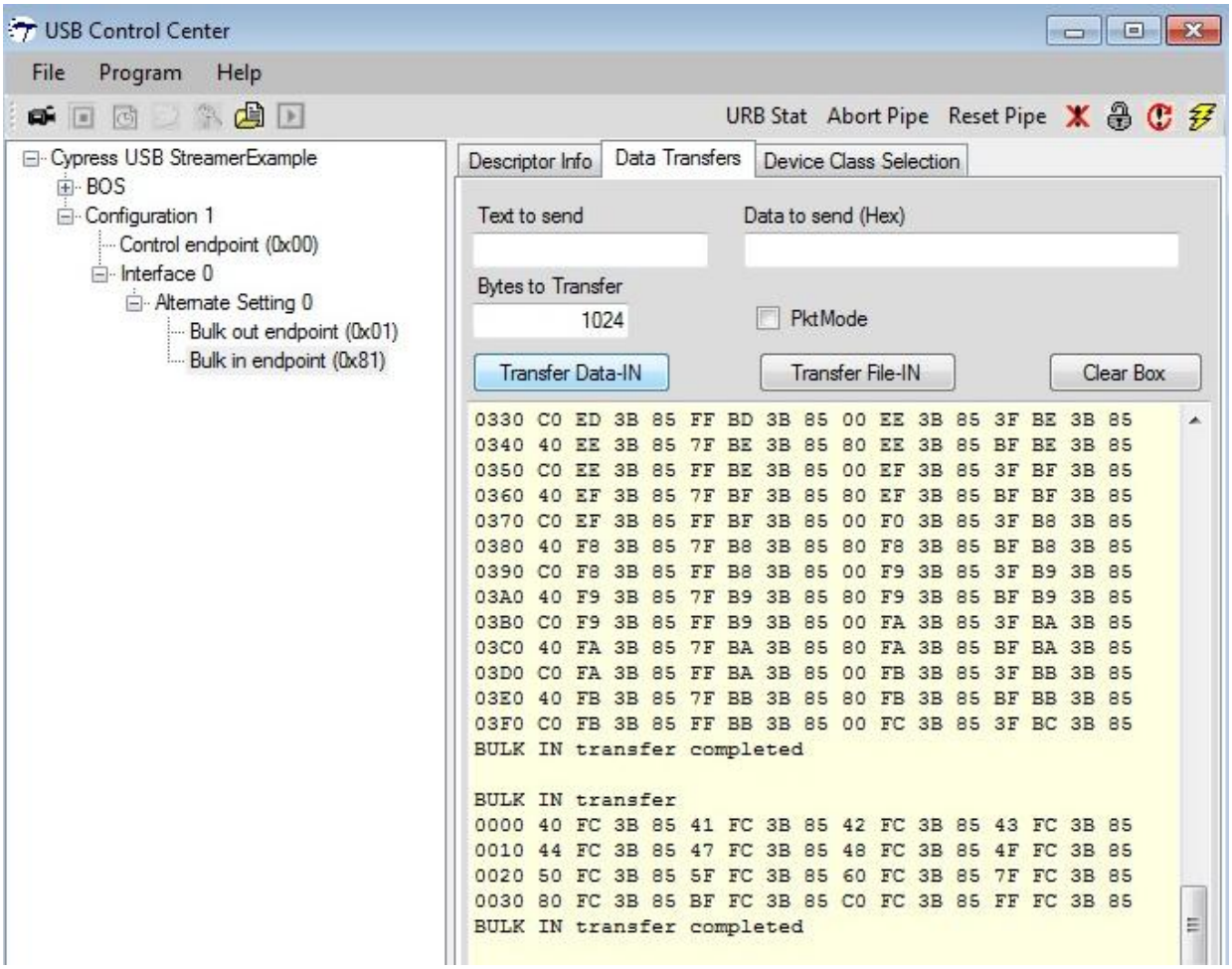
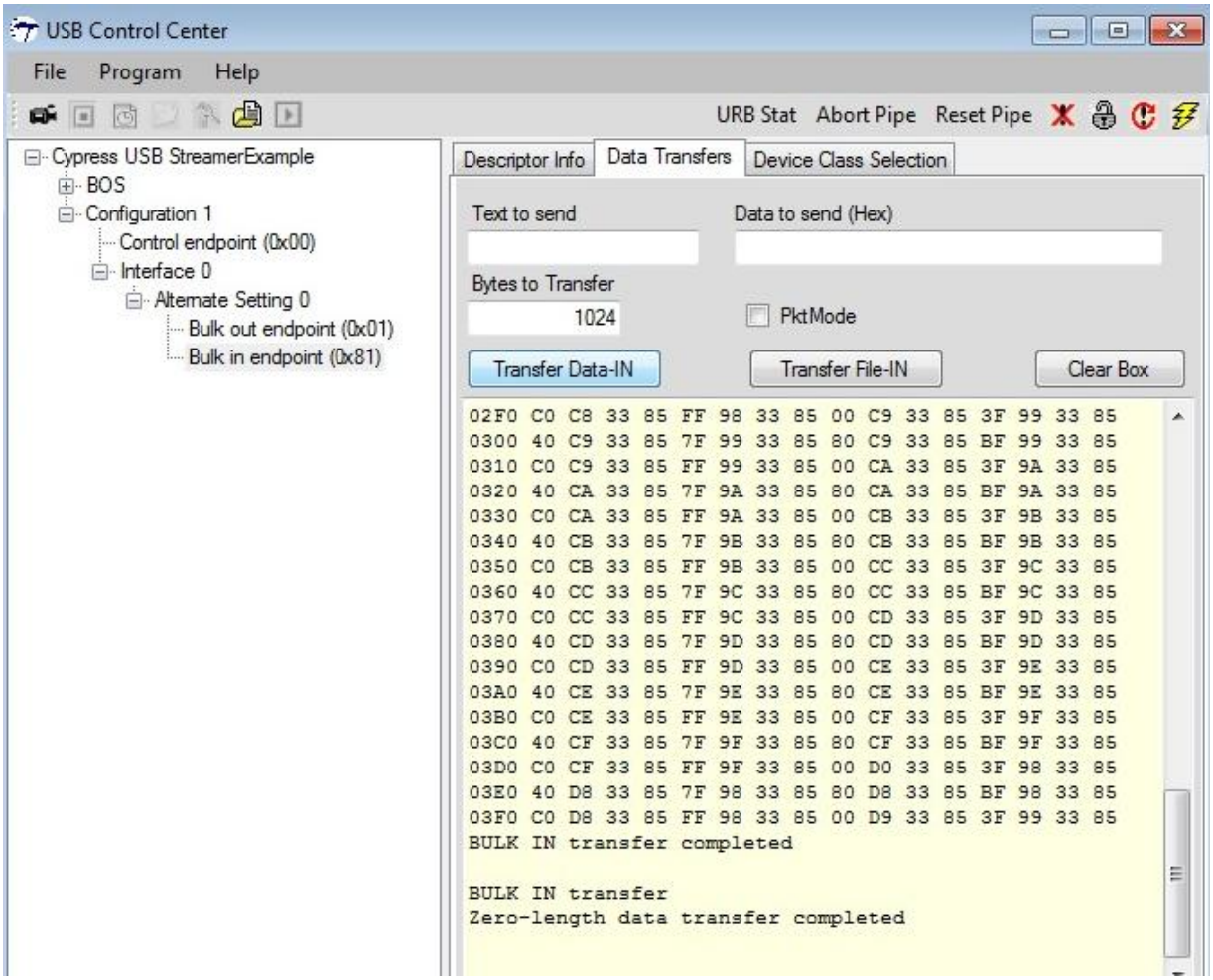


Figure 37. Full Packet Followed by Zero Length Packet Received by Consecutive Transfer Data-IN Operations



6. Note, the FPGA is continuously writing data, so multiple BULK IN transfers can be done.

Summary

The Slave FIFO interface is suitable for applications in which an external FPGA, processor or device needs to perform data read/write accesses to the EZ-USB FX3's internal FIFO buffers.

This application note described the synchronous Slave FIFO interface in detail. The different FLAG configurations available were also described along with an explanation of how the GPIFII Designer tool may be used to configure the FLAGs. A complete design example was also presented.

About the Author

Name: Sonia Gandhi

Title: Applications Engineer Sr. Staff

Contact: osg@cypress.com

Document History

Document Title: Designing with the EZ-USB® FX3™ Slave FIFO Interface - AN65974

Document Number: 001-65974

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3117206	OSG	12/21/2010	New application note.
*A	3261257	OSG	04/15/2011	Updated abstract; added information on synchronous Slave FIFO interfaces; expanded flag configuration section.
*B	3319015	OSG	07/18/2011	Updated description and pin mapping with information on 32-bit data bus support. Also added GPIO and serial interface availability to pin mapping. Clarified PKTEND# usage for short packets.
*C	3473293	OSG	12/22/2011	Updated Async Slave FIFO tPEL parameter Updated Sync Slave FIFO Read and Write Timing diagrams
*D	3656500	OSG	06/25/2012	Updated to new application note template Complete rewrite of application note
*E	3711053	OSG	08/13/2012	Corrected broken links in the document.
*F	3751229	OSG	09/21/2012	Added a design example describing how a Xilinx FPGA can be interconnected with FX3 over Slave FIFO Clarified the difference between Slave FIFO interface with 2 address lines and Slave FIFO interface with 5 address lines Added a timing diagram to show the latency when using a current thread FLAG Added an example application diagram
*G	3843462	OSG	12/17/2012	Template Update Added to the hardware setup options available for executing the design example provided Added pictures of the hardware setup Added a section on error conditions that may occur if FLAGs are violated

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
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 5LP

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

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

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