

Designing A GPIF™ II Master Interface

Author: Sai Krishna Vakkantula

Associated Project: Yes

Associated Part Family: **CYUSB3014**

For latest FX3 SDK: [click here](#)

Related Application Notes: **AN65974**

More code examples? We heard you.

For a consolidated list of USB SuperSpeed Code Examples, visit <http://www.cypress.com/101781>

AN87216 shows how to design an EZ-USB® FX3™ master interface to communicate with an external synchronous Slave FIFO. The design uses the FX3 GPIF™ II Designer tool to develop the interface using a graphical state machine entry. To test this design, we connected two FX3 development kits back to back over the GPIF II interface, one acting as the master (the subject of this note) and the other as a test slave. Firmware source code and GPIF II state machines for both master and slave FX3 kits are attached to this application note.

Contents

1	Introduction.....	1	5.3	GPIF II Actions.....	14
2	More Information.....	2	5.4	GPIF Events.....	19
2.1	EZ-USB FX3 Software Development Kit.....	3	6	GPIF II Master State Machine Implementation.....	19
2.2	GPIF™ II Designer.....	3	6.1	FX3 GPIF II Master Complete State Machine.....	20
3	Introduction to GPIF II.....	3	6.2	FX3 Master Write to FX3 Slave.....	21
4	Synchronous Slave FIFO Interface.....	5	6.3	FX3 Master Read from FX3 Slave.....	22
4.1	Synchronous Slave FIFO Access Sequence and Interface Timing.....	6	7	FX3 Master Firmware Implementation.....	23
4.2	Synchronous Slave FIFO Read Sequence.....	7	8	Hardware Connections.....	23
4.3	Synchronous Slave FIFO Write Sequence.....	8	9	Steps to Run the Demo.....	24
4.4	Basics of the FX3 DMA Architecture.....	9	10	Associated Project Files.....	28
4.5	DMA Channel Configuration in Slave FX3 Firmware.....	9	11	Summary.....	28
4.6	Configuration of Flags.....	10	12	Related Application Notes.....	28
5	GPIF II Designer Tool.....	11		Appendix: Hardware Setup Using FX3 Development Kit (CYUSB3KIT-001).....	29
5.1	Interface Definition.....	12		Document History.....	30
5.2	State Machine Implementation.....	13		Worldwide Sales and Design Support.....	31

1 Introduction

Cypress's FX3 is a USB 3.0 peripheral controller, which provides highly integrated and flexible features to enable developers to add USB 3.0 functionality to any system.

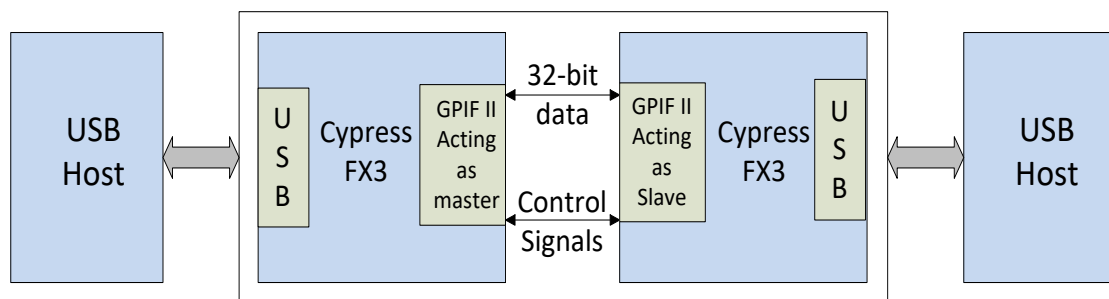
FX3 has a configurable, parallel, general programmable interface, called GPIF II, which is Cypress's General Programmable Interface, generation II. GPIF II can connect to an external processor, ASIC, or FPGA. GPIF II provides glueless connectivity to widely used interfaces, such as asynchronous SRAMs as well as asynchronous and synchronous address data multiplexed interfaces.

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

This application note focuses on the design of a synchronous FIFO master interface. A master initiates transfers, drives an address bus (if present), and usually supplies a clock to the slave. The slave device used in this design is another FX3 device whose GPIF II unit is programmed to act as a Slave FIFO. To test the design, we connected two FX3 Development Kits (DVKs) using each one's GPIF II interface, and one FX3 DVK is programmed to act as the FIFO slave unit. The behavior for this Slave FIFO interface is detailed in this note before delving into the FIFO master implementation. For more details about a Slave FIFO interface, see [AN65974, Designing with the EZ-USB® FX3™ Slave FIFO Interface](#).

Although this example uses a single PC to transfer test data over two of its USB 3.0 ports, a second PC could be employed to move data between the two PCs (as shown in [Figure 1](#)).

Figure 1. Host-to-Host Interconnection Cable



The following sections give an overview of the GPIF II, Slave FIFO interface and its state machine implementation using the GPIF II Designer tool.

2 More Information

Cypress provides a wealth of data at www.cypress.com to help you to select the right device for your design, and to help you to integrate the device into your design quickly and effectively. For a comprehensive list of resources, see the knowledge base article [KBA87889, How to design with FX3/FX3S](#).

- Overview: [USB Portfolio](#), [USB Roadmap](#)
- USB 3.0 Product Selectors: [FX3](#), [FX3S](#), [CX3](#), [HX3](#), Benicia Application notes: Cypress offers a large number of USB application notes covering a broad range of topics, from basic to advanced level. Recommended application notes for getting started with FX3 are:
 - [AN75705](#) – Getting Started with EZ-USB FX3
 - [AN76405](#) – EZ-USB FX3 Boot Options
 - [AN70707](#) – EZ-USB FX3/FX3S Hardware Design Guidelines and Schematic Checklist
 - [AN65974](#) – Designing with the EZ-USB FX3 Slave FIFO Interface
 - [AN75779](#) – How to Implement an Image Sensor Interface with EZ-USB FX3 in a USB Video Class (UVC) Framework
 - [AN86947](#) – Optimizing USB 3.0 Throughput with EZ-USB FX3
 - [AN84868](#) – Configuring an FPGA over USB Using Cypress EZ-USB FX3
 - [AN68829](#) – Slave FIFO Interface for EZ-USB FX3: 5-Bit Address Mode
 - [AN73609](#) – EZ-USB FX2LP/ FX3 Developing Bulk-Loop Example on Linux
 - [AN77960](#) – Introduction to EZ-USB FX3 High-Speed USB Host Controller
 - [AN76348](#) – Differences in Implementation of EZ-USB FX2LP and EZ-USB FX3 Applications
 - [AN89661](#) – USB RAID 1 Disk Design Using EZ-USB FX3S
- Code Examples:
 - [USB Hi-Speed](#)

- [USB Full-Speed](#)
- [USB SuperSpeed](#)
- Technical Reference Manual (TRM):
 - [EZ-USB FX3 Technical Reference Manual](#)
- Development Kits:
 - [CYUSB3KIT-003, EZ-USB FX3 SuperSpeed Explorer Kit](#)
 - [CYUSB3KIT-001, EZ-USB FX3 Development Kit](#)
- Models: [IBIS](#)

2.1 EZ-USB FX3 Software Development Kit

Cypress delivers the complete software and firmware stack for FX3 to easily integrate SuperSpeed USB into any embedded application. The [Software Development Kit](#) (SDK) comes with tools, drivers, and application examples, which help accelerate application development.

2.2 GPIF™ II Designer

The [GPIF II Designer](#) is a graphical software that allows designers to configure the GPIF II interface of the EZ-USB FX3 USB 3.0 Device Controller.

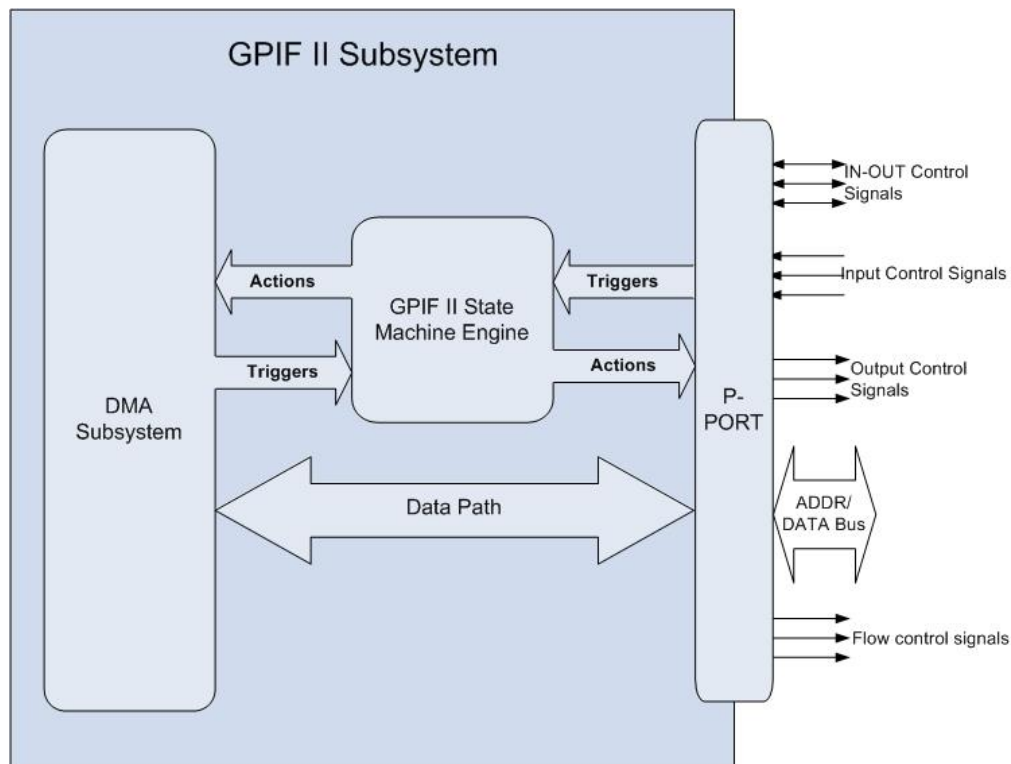
The tool allows users the ability to select from one of five Cypress-supplied interfaces or choose to create their own GPIF II interface from scratch. Cypress has supplied industry-standard interfaces such as asynchronous and synchronous Slave FIFO, and asynchronous and synchronous SRAM. Designers who already have one of these pre-defined interfaces in their system can simply select the interface of choice, choose from a set of standard parameters such as bus width (x8, 16, x32) endianness, clock settings, and then compile the interface. The tool has a streamlined three-step GPIF interface development process for users who need a customized interface. Users can first select their pin configuration and standard parameters. Secondly, they can design a virtual state machine using configurable actions. Finally, users can view the output timing to verify that it matches the expected timing. After this three-step process is complete, the interface can be compiled and integrated with FX3.

3 Introduction to GPIF II

The GPIF II is a programmable state machine that gives you the flexibility to implement an industry standard or a proprietary interface. The GPIF II can function either as a master or a slave to an outside device.

The GPIF II port, also known as the P-Port of FX3, provides a parallel interface with a maximum of 32 bi-directional data lines. The data lines can be split into or time-multiplexed into address lines. There are 13 control lines that are configurable as IN or OUT. A programmable state machine engine controls the GPIF II port operations and control signals. The state machine operations can be controlled by the external processor using control signals configured as input to FX3.

Figure 2. GPIF II Subsystem of FX3



The GPIF hardware also supports the generation of a set of DMA status flags indicating buffer readiness or based on a user-programmed threshold, which can be used by the external processor for flow control. Use these flags to ensure transfers without data loss.

A set of counter and comparator blocks is associated with the GPIF hardware and can be used by the state machine. The comparators can check whether the current state of the address, data or control signals match a specific pattern and generate a match signal that can be used as a trigger by the state machine. The counters can be reset or updated through state machine actions and also generate a limit match signal that can be used as a trigger.

The GPIF II Designer tool generates the configuration data corresponding to the design entered by the user. This is used as an input by the APIs running in the FX3 firmware to initialize the GPIF II hardware. The FX3 SDK includes a set of APIs that can be used to configure, control, and monitor the GPIF interface.

The GPIF hardware can trigger interrupts to the on-board ARM core on the FX3, as well as to an external processor via user-specified actions selected on the state machine canvas. The GPIF II state machine can also make use of a firmware-generated input signal to control the operation of the GPIF interface.

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 a 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 input signals, and control output signals are driven by GPIF II states. 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 values. In the FX3 register space, eight kilobytes are dedicated as GPIF II waveform memory, where the GPIF II descriptor is stored. For more information on the GPIF-II Designer tool, please refer to the [GPIF-II Designer User Guide](#).

A popular implementation of GPIF II is the synchronous Slave FIFO interface, which is described in the following sections. Later sections explain how to design a master interface compatible with synchronous Slave FIFO using the FX3 GPIF II Designer tool.

4 Synchronous Slave FIFO Interface

To make the most of later sections of this application note, you should have a basic understanding of a synchronous FIFO interface. Synchronous Slave FIFO interface details, which are required to understand the GPIF II master state machine, are described here.

This section shows the interconnect diagram for the synchronous Slave FIFO interface, followed by the pin mapping of the signals. Timing diagrams to perform read and write operations on the Slave FIFO interface are shown in the [Synchronous Slave FIFO Access Sequence and Interface Timing](#) section. Details of P-port sockets, flags, and DMA channel configuration in the slave FX3 firmware are described in the [DMA Channel Configuration in Slave FX3 Firmware](#) section.

The synchronous Slave FIFO interface is suitable for applications in which an external processor or device needs to perform sequential data read/write accesses to 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, to support high-throughput requirements.

Figure 3 shows the interface diagram for the synchronous Slave FIFO interface.

Figure 3. Synchronous Slave FIFO Interface

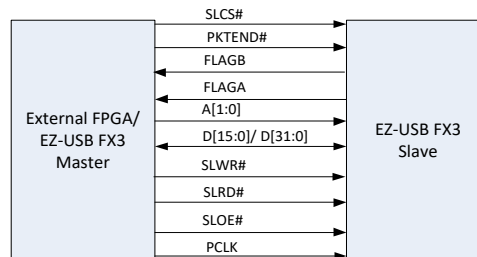


Table 1. Synchronous Slave FIFO Interface Signals

Signal Name	Signal Description
SLCS#	Slave Chip Select, active-low. A master asserts this signal to access the slave interface.
SLWR#	Slave Write Strobe, active-low. A master asserts this signal to perform a write operation (master to slave).
SLRD#	Slave Read Strobe, active-low. A master asserts this signal to perform a read operation (slave to master).
SLOE#	Slave Output Enable, active-low. A master asserts this signal to cause the slave to drive its data bus. If unasserted, the slave data bus floats.
FLAGA/ FLAGB	FX3 flag outputs. These can be programmed in FX3 to indicate various slave states. In this application FLAGA is configured to indicate the availability of data on the slave side and FLAGB is configured to indicate the availability of a free buffer on the slave side.
A[1:0]	Address Bus. In this FX3 GPIF II example two address lines are required to select thread on the slave FX3.
D[31:0]	This is the 16-bit or 32-bit data bus of the Slave FIFO interface.
PKTEND#	Packet End, negative-true. A master asserts this signal to instruct the FX3 slave to send a zero-length or short packet over USB.
PCLK	The master provides this clock to the slave. All data and timing operations are referenced to this clock.

4.1 Synchronous Slave FIFO Access Sequence and Interface Timing

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

In the design attached to this application note, the master FX3 performs burst data accesses to the slave FX3's internal FIFO buffers. The master FX3 drives the 2-bit address on the ADDR lines and asserts the read or write strobes. The Slave FX3 carries out USB transfers and asserts FIFO FLAG signals to indicate empty (on FIFO read) or full (on FIFO write) conditions.

Figure 4 shows a logical diagram of the Slave FIFO interface as seen by the external FPGA/processor. Table 2 shows the FX3 GPIF-II Master timing parameters.

Table 2. FX3 GPIF-II Master Timing Parameters

Parameter	Description	Min	Max	Unit
Freq	Clock Frequency	100	–	MHz
tDS_addr	Input Address Setup Time	3	–	ns
tDH_addr	Input Data Hold Time	1.5	–	ns
tDO_addr	Clock to Output Address Delay	–	8	ns
tDOH_addr	Output Address Hold Time	2	–	ns
tS	Input control Setup Time	3	–	ns
tH	Input control Hold Time	1.5	–	ns
tDS	Input Data Setup Time	3	–	ns
tDH	Input Data Hold Time	1.5	–	ns
tDO	Clock to Output Data Delay	–	8	ns
tDOH	Output Data Hold Time	2	–	ns
tCTLO	Clock to Output Control Delay	–	8	ns
tCOH	Output Control Hold Time	2	–	ns
tHZ	Clock to High-Z	–	8	ns

Figure 4. Logical Diagram of the Slave FIFO Interface

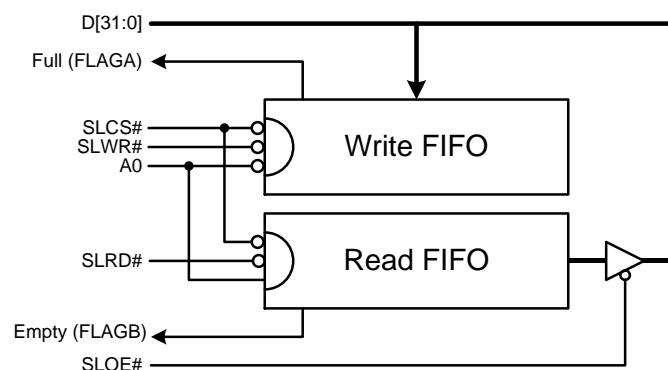
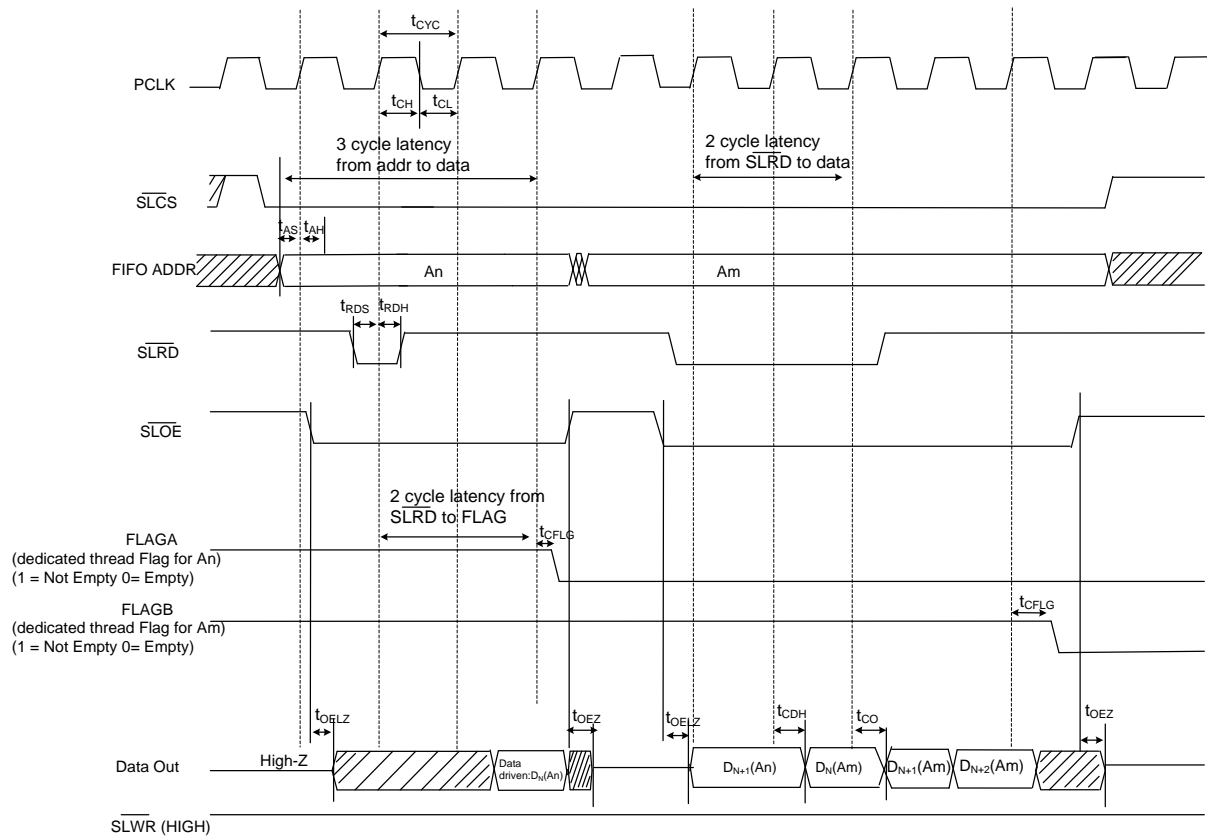


Figure 5. Synchronous Slave FIFO Read Sequence



4.2 Synchronous Slave FIFO Read Sequence

Figure 5 shows how the master reads a single 32-bit word from the slave, which is a pre-programmed FX3 DVK. The figure shows a single-word read from FIFO address “An” followed by a burst read from FIFO address “Am”.

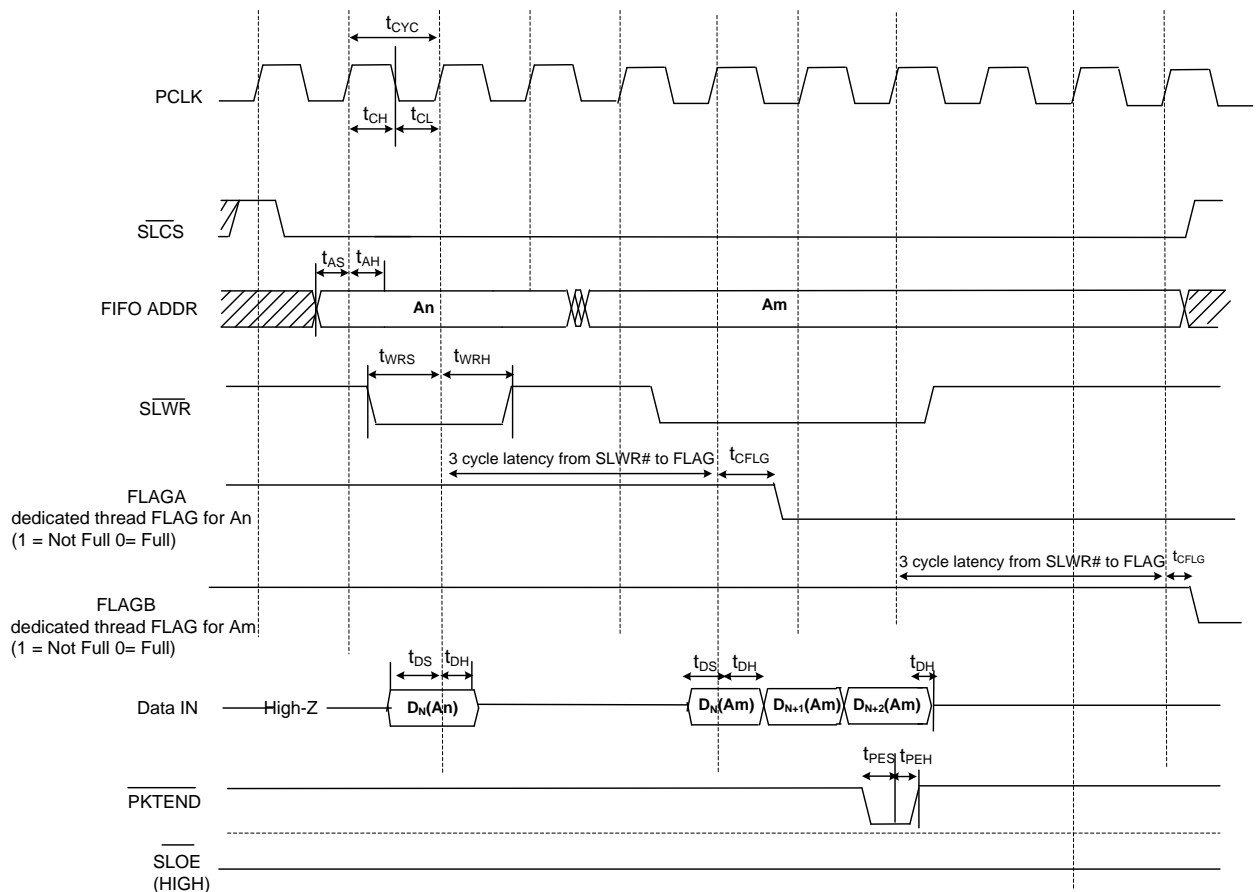
A single 32-bit word read transfer follows these steps:

1. The master drives FIFO address A_n , then asserts $SLCS\#$. Note that the master must ensure that the address meets the setup time t_{AS} requirement for the PCLK rising edge that samples $SLCS\#$.
2. The master asserts $SLOE\#$ to cause the slave to start driving the data bus.
3. The master asserts $SLRD\#$ and holds its LOW for one PCLK. This starts the data propagation from the addressed FIFO to the data bus.

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. Two clocks later (measured from the rising edge of PCLK), the new data value is present t_{CO} after the clock edge. N is the first data value read from the FIFO. To drive the data bus, $SLOE\#$ must also be asserted.

A read burst transfer is shown in the second half of Figure 5. For burst mode, the $SLRD\#$ and the $SLOE\#$ are left asserted during the duration of the read. When the $SLOE\#$ is first 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. In the Figure 5 example, the retrieved FIFO word is the last word in the FIFO, indicated by the FLAGB signal indicating “empty”.

Figure 6. Synchronous Slave FIFO Write Sequence



4.3 Synchronous Slave FIFO Write Sequence

Figure 6 illustrates a single-word write FIFO to address “An” followed by a burst write to FIFO address “Am”.

A single 32-bit word write transfer follows these steps:

1. The master drives FIFO address An, then asserts SLCS#. Note that the master must ensure that the address meets the setup time requirement t_{AS} for the PCLK rising edge that samples SLCS#.
2. The master drives its data onto the data bus.
3. The master asserts SLWR# as early as the next clock after SLCS# assertion.
4. While the SLWR# is asserted, the master writes data to the FIFO and on the rising edge of the PCLK, the FIFO pointer is incremented.
5. The FIFO flag is updated after three clocks plus a delay of t_{CFLG} from the rising edge of the clock.

The same sequence of events shows a burst write.

For the burst mode, the master keeps SLWR# and SLCS# asserted for the entire duration of the burst write. In the burst write mode, after the master asserts SLWR#, 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 as long as SLWR# is asserted.

Short Packet: The master can commit a short packet to the USB Host by using the PKTEND# signal. The master FX3 should assert the PKTEND# along with the last word of data and the SLWR# pulse corresponding to the last word. Otherwise, asserting PKTEND# without the SLWR# pulse results in a ZLP (Zero Length Packet). The master must hold the FIFOADDR lines constant during the PKTEND# assertion. On assertion of PKTEND# with SLWR#, the GPIF II state machine of the slave FX3 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 tied 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 master FX3 to determine when all the data has been read.

4.4 Basics of the FX3 DMA Architecture

The FX3 device has an internal DMA fabric that connects the GPIF II interface to internal system memory and other serial peripherals. All data transfer through the GPIF II interface is done from or into an internal memory buffer. The firmware application running on the FX3 is responsible for connecting this data path (using the DMA fabric) to the appropriate source or sink, such as the USB Host or a serial peripheral.

4.4.1 Sockets

Each port on the USB 3.0 device supports sockets that correspond to one end of a data flow and can be independently addressed. The FX3 P-port (Processor port) or GPIF II ports support a maximum of 32 sockets, which means that a total of 32 independent data paths can be configured across this interface.

The firmware application is responsible for allocating memory buffers corresponding to all utilized sockets and for connecting sockets to appropriate data sources or sinks. Refer to the DMA Channel APIs in the [FX3 SDK API guide](#) for the functions that execute this configuration.

4.4.2 Threads

Although the GPIF port on the FX3 device makes 32 addressable sockets available for connections between data providers and consumers, there are only four data highways, or “threads,” that can simultaneously transfer data. This means that the application can select a maximum of four sockets, which are bound to these threads, and then switch between them with no added latency.

A typical FX3 application initializes one or more of these threads by associating sockets at each end, one socket providing data, and the other consuming data. The thread to be used for each word of data transferred can be specified directly by providing an input address, or by specifying the target thread in the IN_DATA or DR_DATA action settings (described in [GPIF II Actions](#) section) in the GPIF state machine.

Note: You also can dynamically (while the app is running) change socket-to-thread assignments, though this technique is not required for this application. Changing the active socket that is bound to a thread requires additional latency. This switching can be done automatically by specifying the socket address (in cases where the address bus is between 3 and 5 bits wide) or it can be done with firmware intervention within the FX3 device.

The GPIF hardware provides one set of DMA status flags per thread. The DMA flags that are specific to a thread reflect the status of the active socket on that thread, all the time. When switching the active socket on a thread, allow enough time to ensure that a flag reflects the state of the new socket before using it to control data transfers.

Note that the socket to thread mapping is not completely flexible. Each socket N can be bound only to the thread numbered (N MOD 4). For example, socket 7 can only use thread 3 (remainder of 7 divided by 4). Socket 11 can also use thread 3. This means that it is not possible to bind sockets 0, 4, 8, and so on to different threads in hopes of using them at the same time. This does not prove a limitation. Because there are many more sockets than threads (32 versus 4), many choices are available. Remember this constant when you select the sockets for GPIF data transfers.

4.5 DMA Channel Configuration in Slave FX3 Firmware

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

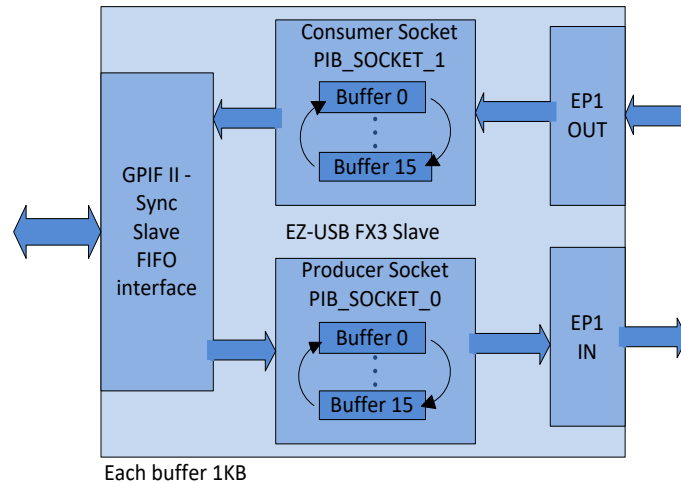
Note that if data is transferred from the Slave FIFO interface to the USB interface, then P-port or GPIF II port is the producer and USB is the consumer, and vice-versa.

Therefore, if data is to be transferred in both directions over the Slave FIFO interface, configure two DMA channels. Use the CyU3PDmaChannelCreate API in firmware (refer to [FX3 API guide](#) for more details of APIs), one with P-port as the producer and the other 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. The P-port consumer socket is the one that the external device will read from over the Slave FIFO interface.

In the example attached to this application note, socket 0 is configured as the producer socket and socket 1 as the consumer socket on the P-port side (see [Figure 7](#)).

Figure 7. PIB Sockets Used in Synchronous Slave FIFO Firmware



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

When you configure the channel, you can allocate multiple buffers to a particular DMA channel. Note that the flags will indicate full/empty on a per-buffer basis. (The maximum buffer size 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.

4.6 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 that makes FIFOs out of FX3 RAM. Flags can be associated with specific threads or dynamically associated with the currently addressed thread. In either case, the flags 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 an empty status if data is being read out of the socket and indicates full if data is being written into the socket.

These are the types of FLAGS that can be used:

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

4.6.1 Dedicated Thread Flag

A flag can be configured to indicate the status of a thread 0-3. 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 it must monitor the correct flag every time a different thread is addressed.

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

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

In this example application, FLAGA is dedicated to thread 1, and FLAGB is dedicated to thread 0. Therefore, if the master FX3 needs to perform a write operation on the Slave FIFO interface, then it needs to wait for HIGH on FLAGB. HIGH on FLAGB indicates that the DMA buffer allocated to this data path is not FULL. FLAGB goes LOW when the DMA buffer is FULL. Similarly, when you perform a read operation on slave FX3 device, the master FX3 needs to wait for HIGH on FLAGA. HIGH on FLAGA indicates that the DMA buffer allocated to this data path is not EMPTY. FLAGA stays LOW when there is no data in the DMA buffer.

For details on the Slave FIFO interface implementation, see [AN65974, Designing with the EZ-USB FX3 Slave FIFO Interface](#).

5 GPIF II Designer Tool

Now that the Slave FIFO interface requirements are defined, let's learn how to use GPIF II Designer to implement the GPIF II design. GPIF II Designer is a graphical tool that uses state machine entry and menus to configure the FX3 GPIF II interface.

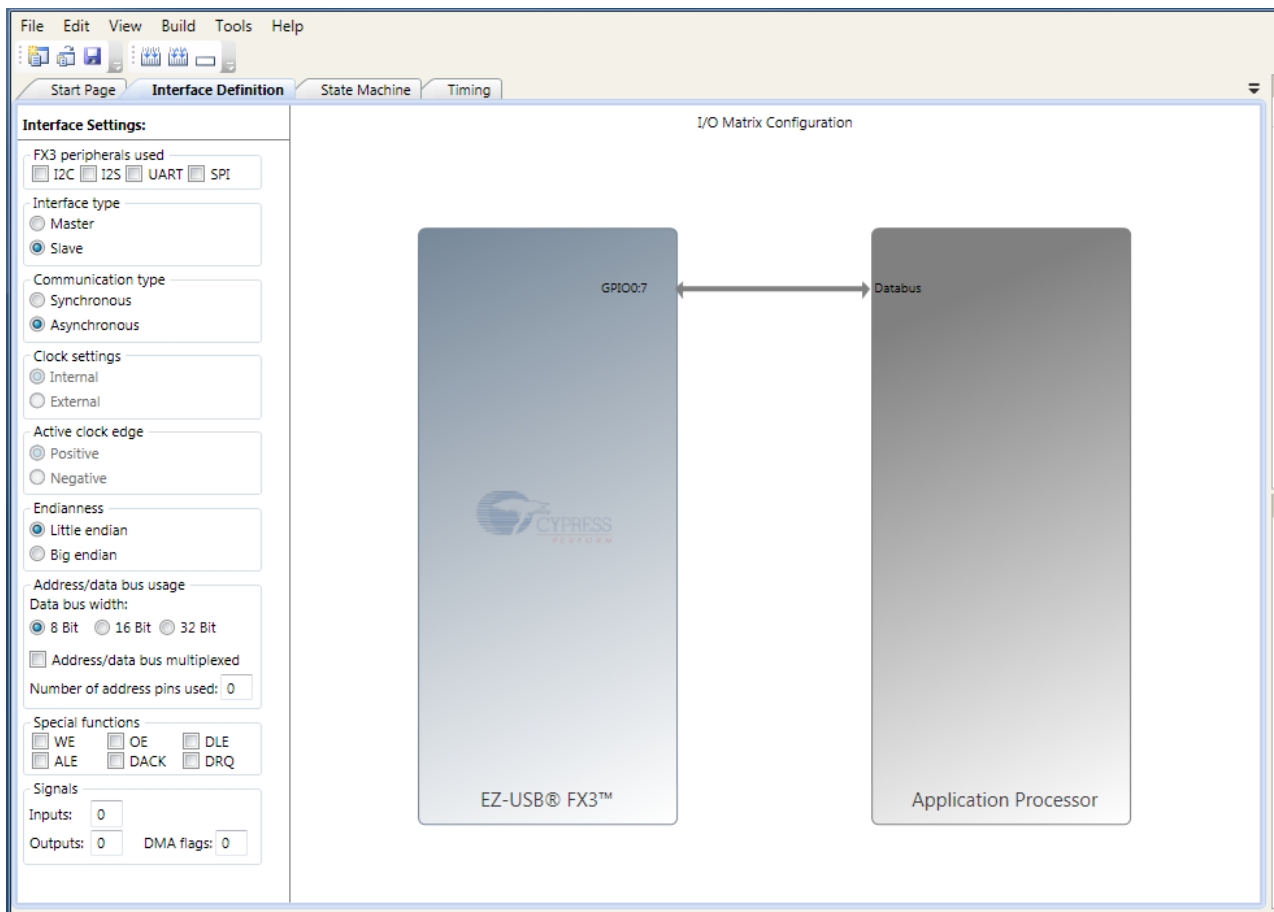
By using GPIF II Designer, you can select from a library of Cypress pre-built interfaces or you can create a GPIF II interface from scratch. Cypress supplies industry standard interfaces, such as asynchronous and synchronous Slave FIFO, asynchronous and synchronous ADMUX (Address/Data multiplexed RAM), and asynchronous SRAM. Designers who already have one of these pre-defined interfaces in their system can select an interface, choose from a set of standard parameters, such as bus width (x8, 16, x32), endian-ness and clock settings; and compile the interface.

To design a custom interface using this tool, you must first select your pin configuration and standard parameters available on the "Interface Definitions" tab. Then, select the "State Machine" tab to design a state machine using configurable actions. Once complete, the GPIF II Designer compiles the interface into a C header file, suitable for inclusion in an FX3 project.

GPIF II is a part of [FX3 SDK installation](#). It is also available as a standalone installation file that you can download from the following location on the [Cypress webpage](#).

[Figure 8](#) shows a portion of the GPIF II Designer screen for a new project.

Figure 8. GPIF II Designer Tool



5.1 Interface Definition

The first step is to configure the GPIF II outside-world interface by filling out the entries in the “Interface Settings” tab (Figure 8). As you select and deselect options, the center panel changes to reflect a “living schematic” of the interface. This saves you the trouble of figuring out the FX3 pin mapping, because the FX3 signals are labeled in the FX3 block.

1. Which FX3 serial peripherals will be used by your overall application? Selecting a box allows GPIF II Designer to avoid pins that may be assigned to other FX3 peripherals, such as the SCL and SDA pins if “I2C” is selected.
2. Will FX3 act as a master or slave of the interface? A master initiates transfers and drives the address bus if one is implemented. When you select master or slave, the Action List (right panel) automatically updates to show the available choices for the selection.
3. Does this interface use a clock? If yes, then choose “Synchronous.” An example of an asynchronous interface is a static RAM that has an address and data bus, read and write strobes, but no clock. A read operation, for example, occurs by asserting the address, then asserting chip enable and read strobes. The data comes out a propagation time after the read strobe, with no reference to a clock.
4. Do you want FX3 GPIF II to drive the interface clock? If yes, then internal; otherwise, it’s external. This section is dimmed out if “Synchronous” is not selected in question 3.
5. When do you want GPIF II to sample interface signals? On positive edge or on negative edge? This section is dimmed out if “Synchronous” is not selected in question 3.
6. Endianness of your device? Little endian or big endian. “Endianness” refers to byte ordering in multi-byte integers—most significant (big) or least significant (little) byte first.
7. Data bus width of the external device connected to FX3: 8-bit, 16-bit, or 32-bit.

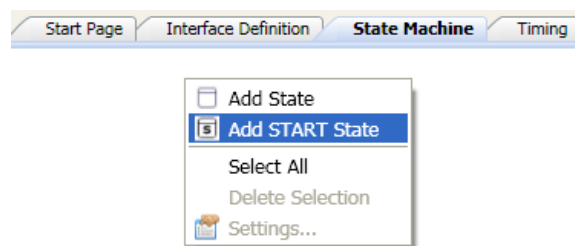
8. Does the external device require address lines? This would be true if it needs to select between FX3 resources, such as sockets. Are address and data buses multiplexed?
9. "Special features" refer to unique hardware pins and configurations.
10. **WE:** The WE special function can be connected only to GPIO_18. This connection ensures that the assertion of the GPIO_18 line will result in the setting up the data bus direction for ingress path (into the FX3 device).
OE: You can use this feature only with GPIO_19. With this function, the assertion of GPIO_19 can directly change the data bus direction for egress path (out of the FX3 device).
DLE: The Data Latch Enable (DLE) function uses the GPIO_18 line's de-assertion to latch the data line for a few additional nanoseconds. Use this feature while the FX3 device is reading the data lines for ingress path at the de-assertion edge of GPIO_18.
11. How many signals need to be monitored by FX3, and how many need to be controlled by FX3? Also, how many DMA flags do you need in your application? As you add inputs or outputs, the state machine designer automatically adds them as choices to be tested (inputs) or asserted (outputs) in any state.

5.2 State Machine Implementation

After the Interface Definition is done, the next step is to implement a state machine. To help you, view the pictures shown below.

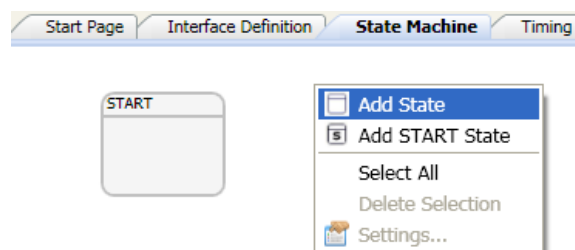
Every state machine should have a START state. You can add a START state by right-clicking and selecting "Add START State". The START state cannot have any incoming transition, and the transition equation is fixed to 'LOGIC_ONE' for the transition originating from the START state.

Figure 9. Adding START State



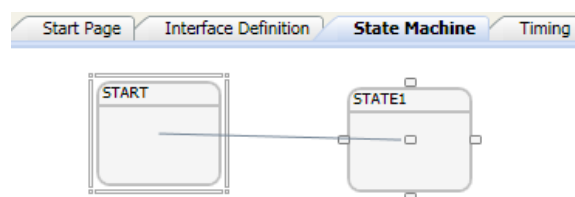
You can add a state by right-clicking and selecting "Add State". Figure 10 shows this step.

Figure 10. Adding a State



You can connect two states, as Figure 11 shows. The transition shown in Figure 11 is going from START to STATE1.

Figure 11. Connecting Different States

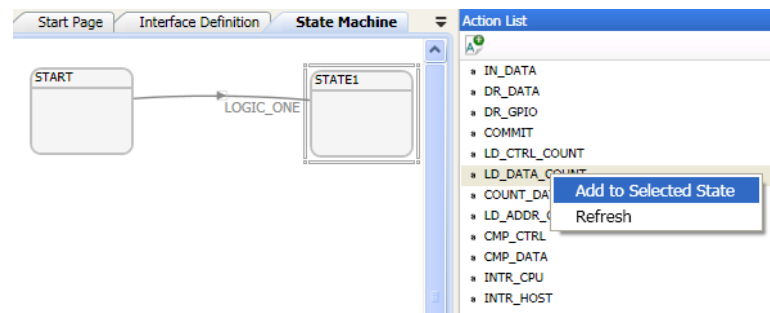


An action needs to be added to the state based on the application requirement. The list of actions required to implement a master Slave FIFO interface are detailed in the following sections.

Figure 12 shows you how to add an action to a state. After selecting the state box, right-click on one of the choices in the “Action List” panel and chose “Add to Selected State”. Alternatively, you can drag the action into a selected state box. Hovering the mouse over each Action entry causes a hint to pop up that describes what the action does. To remove an action from a state, right-click the name text inside the state box, then select “Delete Action”.

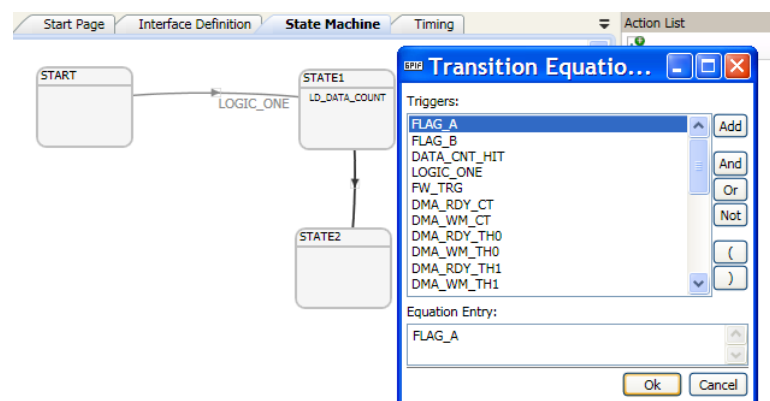
Note that more than one action can be assigned to a STATE box. GPIF II Designer issues an error message if you try to include conflicting operations in the same state. One example would be if you try to load and increment the data counter in the same state.

Figure 12. Adding an Action to a State



You get a list of events available for that state after double-clicking on the transition line. Transition equations are added as shown in Figure 13.

Figure 13. Adding a Transition Equation



5.3 GPIF II Actions

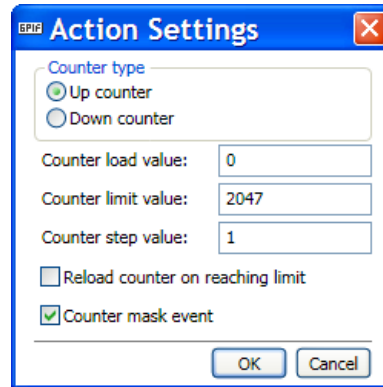
Each state in a GPIF II state machine can be programmed to perform one or more GPIF II actions. Actions performed in a state can be programmed to occur once or on every clock edge until the state transitions to another state. GPIF II actions used in master state machine implementations are described here. For a description of all GPIF II actions, go to *Help* available on GPIF II Designer and click on *Topics* or press F1. Select the *Contents* tab in the newly opened *GPIF II Designer* window and browse to the *GPIF II actions* subsection in the *Programming GPIF II State machine* section. You can view the same details from the [GPIF II Designer](#) web page.

5.3.1 LD_DATA_COUNT (Load Data Counter)

This action configures the Data counter when it is added to a particular state. After it is configured, the LD_DATA_COUNT action cannot be repeated with different values.

You can take an equivalent action by using the `CyU3PGpifInitDataCounter()` function in the FX3 firmware API in the FX3 firmware. The FX3 API guide gives function parameter information. Note that this action should be added to a state in the GPIF II state machine even if you are configuring the Data counter with the `CyU3PGpifInitDataCounter()` function. Otherwise, the `DATA_CNT_HIT` event will not be available in the transition equations list. By using this API, you can change the data counter value during run time.

Figure 14. LD_DATA_COUNT Action Settings



The following parameters are associated with this action:

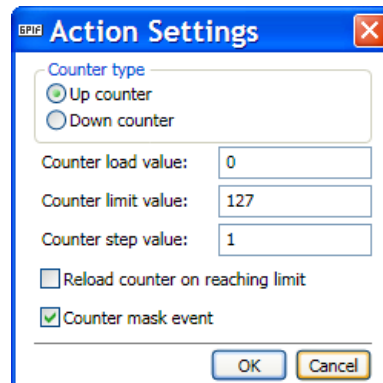
- *Counter type (Up / Down).* Up is selected in this case.
- *Counter load value:* The initial count loaded when this action is performed. It is loaded with 0 in this case.
- *Counter limit value:* The count value at which the event should be generated. It is loaded with 2047 for reading 8 kbytes of data using a 32-bit data bus. This value can be calculated with the help of a simple formula:

$$\text{Counter limit value} = (\text{Number of Bytes to read} / \text{Data bus width in Bytes}) - 1^*$$
 *1 is subtracted only if the counter load value is 0.
- *Reload counter on reaching limit:* The count load value can be reloaded when the count limit value is hit by checking this parameter box. Otherwise, it acts as a one-time counter.
- *Counter mask event:* If the box is unchecked, a firmware event is generated when the counter limit is reached.
- *Counter step value:* The counter step value that is be added / subtracted each time the `COUNT_DATA` action is used. This value must be greater than zero to activate the `DATA_CNT_HIT` event.

5.3.2 LD_ADDR_COUNT (Load Address Counter)

This action is similar to `LD_DATA_CNT`.

Figure 15. LD_ADDR_COUNT Action Settings



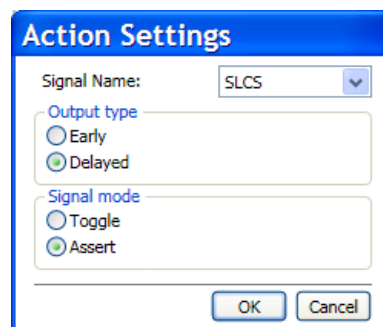
The following parameters are associated with this action:

- *Counter type (Up / Down)*: Counter can be configured to count in ascending or descending order.
- *Counter load value*: The initial count loaded when this action is performed.
- *Counter limit value*: The count value at which the event should be generated. It is loaded with 127 for reading 512 bytes of data over a 32-bit data bus.
- *Reload counter on reaching limit*: The count load value can be reloaded when the count limit value is hit by checking this parameter box.
- *Counter mask event*: If the box is unchecked, a firmware event is generated when the counter limit is reached.
- *Counter step value*: Counter step value that should be added or subtracted each time the COUNT_ADDR action is used.

5.3.3 DR_GPIO (Drive GPIO)

The DR_GPIO action drives a GPIO pin HIGH, LOW, or toggles it. The action takes place after an “Assertion Delay”, which is specified as two or three cycles for synchronous operation, or 25 ns or 30 ns for asynchronous operation. The GPIO driven using the action will be de-asserted during the transition to the next state. That means if you want to assert a particular GPIO to HIGH or LOW across all states in the state machine, you need to add that action in every state. If you don't want to add it in every state, then add this action in the first state and select Toggle mode.

Figure 16. DR_GPIO Action Settings



The following parameters are associated with this action:

- *Signal name*: User-defined alphanumeric string to indicate the signal can be entered here. This name will appear on the state machine canvas.
- *Output type*: This parameter controls the delay of the output signal being driven. The delay is 25 ns in asynchronous mode or two clock cycles in synchronous mode when the *Early* parameter box is checked. The delay will be 30 ns in asynchronous mode or three clock cycles in synchronous mode when the *Delayed* parameter box is checked.
- *Signal mode*: In Toggle mode, the signal value gets toggled when the containing state is exited. In Assert mode, the signal value is driven to the assertion polarity specified for the output. To select the assertion polarity, switch to the Interface Definition tab, and double-click on the signal name in the Application Processor (for example, “OUTPUT0”). This brings up a dialog that lets you set initial pin value and active-HIGH or active-LOW polarity. Selecting active-LOW, for example, drives the pin low when the “Assert” action executes.

Notes:

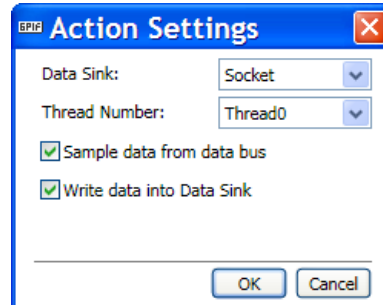
1. The delayed output and signal mode settings are global for each signal, and they cannot be changed every time the action is used in a different state. The tool will generate a configuration file that corresponds to the last settings that were made for each output signal.
2. “Repeat actions until next transition” in the State Settings dialog box has no effect on the behavior of the DR_GPIO action. The DR_GPIO action is repeated for every clock cycle (the clock being the interface clock or the FX3 internal clock). If “Toggle” is selected, then that GPIO toggles every clock while in the containing state.

5.3.4 IN_DATA (Input Data)

The IN_DATA action samples data from the data bus and moves it to the destination specified. Destination can be a DMA channel or the firmware application. *CyU3PGpifReadDataWords()* API is used get data into firmware if *Data Sink* is selected as *Register*. Refer to [FX3 API guide](#) to get more details of this API. Note that the option for directly selecting one of the four destination threads is available only when the destination is a DMA channel with the addressing mode set to *Thread selected by State machine (Number of address lines =0)*.

It is possible to latch only the data from the data bus and not store it in the selected destination, or to store previously latched data into the selected destination. These options are made available to satisfy certain protocols where the data on the bus may lead a strobe signal that indicates data availability.

Figure 17. IN_DATA Action Settings



The following parameters are associated with this action:

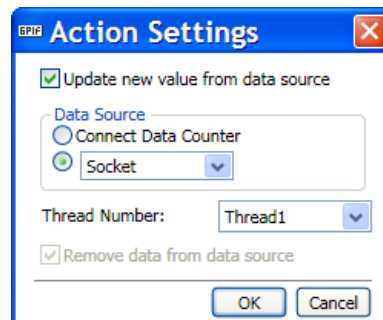
- *Data Sink* – Register/ Socket/ PPRegister
- *Thread Number* – Thread number with which the data sink is associated. Selectable from Thread0 to 3. (This feature is available when there is no address line to select the thread number or master mode is enabled.)
- *Sample data from data bus* – This option samples data from the data bus but does not write it to the specified data sink.
- *Write data into Data Sink* – This option is checked when the user wants to push the sampled data into the specified Data Sink.

These two choices make it possible to sample the data and write the data to its destination.

5.3.5 DR_DATA (Drive Data)

The DR_DATA action drives data onto the data bus from the specified source. The source can be a DMA channel or the firmware application. *CyU3PGpifWriteDataWords()* API is used to write data from firmware if *Data Source* is selected as *Register*. Refer to [FX3 API guide](#) to get more details about this API. Note that the option for selecting the source as a thread is available only when the source is a DMA channel with its addressing mode set to *Thread selected by State machine (Number of address lines =0)*.

Figure 18. DR_DATA Action Settings



The following parameters are associated with this action:

- *Update new value from data source* – This option updates the data bus with the data word present in the Data Source and removes the word from the specified Source.
- *Data Source* – This option helps the user to choose between data counter and register/ Socket / PPRegister.
- *Thread Number* – Thread number with which the data source is associated. Selectable from Thread0 to 3 This choice is available only if there is no address line to select the thread number. This means the “Number of Address Pins Used” field in the Interface Definition tab is set to 0.
- *Remove Data from the Data Source*: When this option is disabled. the source data is not discarded, so it can be used in another state.

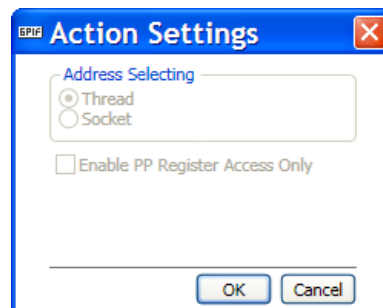
Note: “Update new value from data source” flag overrides the “Repeat actions until next transition” flag on the State Settings dialog box. This means if “Update new value from data source” checkbox is selected, data will be updated on the bus in every clock cycle when GPIF II is in that state even if the “Repeat actions until next transition” flag is unchecked.

5.3.6 IN_ADDR (Input Address)

The IN_ADDR action causes the GPIF hardware to sample the value from the address bus and use it to select a DMA thread or a socket. If 0 address bits are specified in the Interface Definition tab, the Address Selecting choices are dimmed out.

When the address bus width is less than or equal to two bits, the address can only select one of four DMA threads. If the address is between three and five bits wide, it can select either a DMA thread or a specific socket. The “Address Selecting” parameter shown in the dialog above is used to make this selection.

Figure 19. IN_ADDR Action Settings



The following parameter is associated with this action:

- Address Selecting – Thread or Socket

Action choices automatically (and instantly) change as you change Interface Definition values. For example, if you change “Number of address pins used” to 0, the IN_ADDR choice disappears from the Action List.

5.3.7 COUNT_DATA, COUNT_ADDR

Update Data counter value by the step value configured through the LD_DATA_COUNT action. The DATA_CNT_HIT trigger will become true if this update results in the count reaching the specified limit. COUNT_ADDR takes the same action with the address counter.

There are no parameters associated with this action.

5.3.8 INTR_CPU

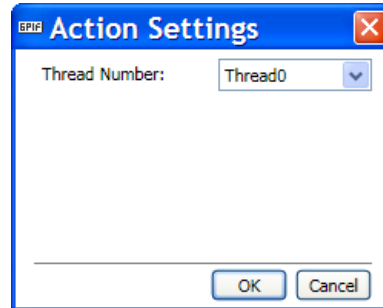
Interrupts the on-chip CPU to generate CYU3P_GPIF_EVT_SM_INTERRUPT event, which is handled by the firmware application. The INTR_PENDING event is available in the transition equations list if we select INTR_CPU action in a state. If there are actions that you need to perform before the INTR_PENDING signal is cleared, use the CyU3PGpifRegisterSMIntrCallback function. This callback is called before the interrupt is cleared by the CPU.

There are no parameters associated with this action.

5.3.9 COMMIT

COMMIT the data packet / buffer on the selected Ingress DMA channel (DMA channel to read data into FX3 device). The buffer will be transferred to the other side of the pipe. This action is typically used to force buffer / packet end using a state machine.

Figure 20. COMMIT Action Settings



The following parameters are associated with this action:

- *Thread Number* – Thread0 to 3 (Available only when number of address bits is configured 0 or master mode is enabled).

5.4 GPIF Events

The triggers that cause GPIF II state machine transitions are Boolean expressions formed using trigger variables. The following table captures events generated as a result of the GPIF II actions that are used in master state machine implementation.

Table 3. GPIF II Events Description

GPIF II event	Action Causing the Event	Description
Input signal name	None; this is an external event	Name associated with any GPIO configured as input can be used as a trigger in a transition equation.
LOGIC_ONE	None	Unconditionally moves to the next state.
DATA_CNT_HIT	COUNT_DATA	This trigger is true when the "Counter limit value" in the action setting is reached. This trigger is generated as a result of a COUNT_DATA action
ADDR_CNT_HIT	COUNT_ADDR	This trigger is true when the Counter limit value" in the action setting is reached. This trigger is generated as a result of a COUNT_ADDR action
DMA_RDY_CT, DMA_RDY_TH0, DMA_RDY_TH1	IN_DATA DR_DATA	This trigger is true when the DMA is ready to send or receive data.

6 GPIF II Master State Machine Implementation

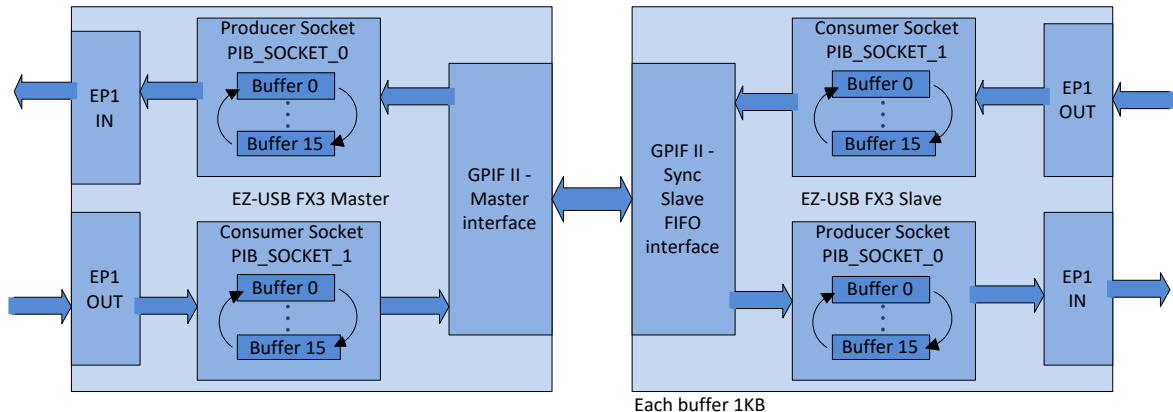
This section uses actions and events described above to implement the master state machine that sends and retrieves data from the synchronous Slave FIFO that is implemented by a second FX3 DVK.

The master state machine decides the read or write operation based on the flags coming from the slave FX3 device.

On the slave side, FLAGA is configured to indicate the readiness of Thread 1 (Thread_1_DMA_Ready), and FLAGB is configured to indicate the readiness of Thread 0 (Thread_0_DMA_Ready). Therefore, FLAGA indicates the availability of data on the slave side and FLAGB indicates the availability of a free buffer on the slave side.

Two DMA channels are created in the master firmware to do bi-directional data transfers. To transfer data from the USB side to GPIF II of FX3, PIB socket 1 is configured as a consumer socket and USB socket is configured as a producer. To transfer data from the GPIF II side to USB, PIB socket 0 is configured as a producer socket and USB socket is configured as a consumer (see [Figure 21](#)).

Figure 21. PIB Sockets of FX3 Slave and Master



Master FX3 can drive the address over address bus with the help of a socket/register/address counter. In this application, the address is required to select a thread on the slave FX3. To make the implementation easier, two address lines (A1 and A0) are configured as GPIOs, and we are driving these GPIOs to address a thread of slave FX3.

The clock is driven from the master FX3 to the slave FX3, and the data bus width is 32-bit. Control signals needed for the Slave FIFO interface are driven from the GPIF II of master FX3.

6.1 FX3 GPIF II Master Complete State Machine

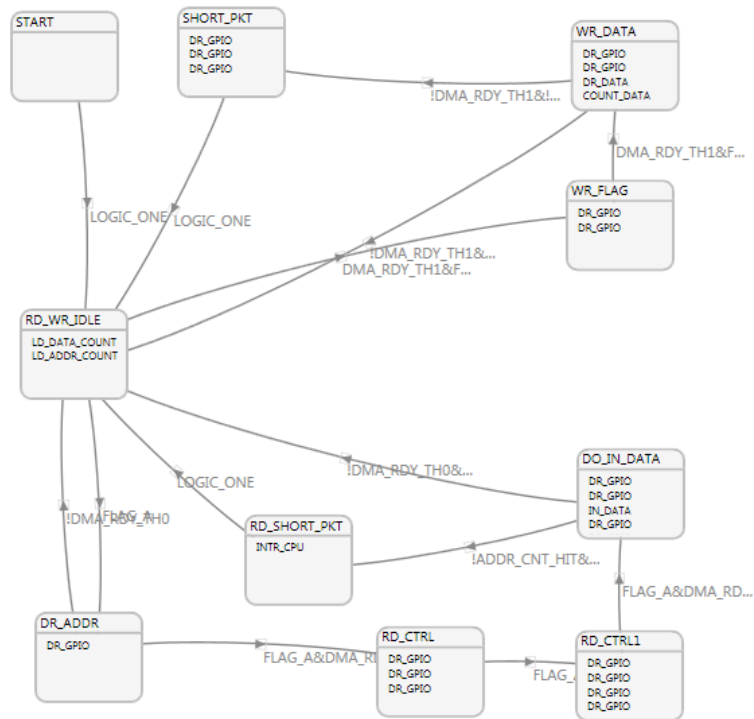
The GPIF II master state machine is shown in Figure 22. The master state machine stays in the RD_WR_IDLE state when there is no data transfer between the master FX3 and the slave FX3. Data and address counters are loaded in this state. In the attached example project, the data counter and the address counter are loaded with 511 (multiples of packet size, 512 in USB 2.0, 1024 in USB 3.0) to count the data that has been written into or read from the buffer

If transition equations are not clear from Figure 22, Figure 23, and Figure 24, refer to the attached GPIF projects.

This state (RD_WR_IDLE) continuously checks for FLAGA or (DMA_RDY_TH1 & FLAG_B) events to become true so that it can branch to the read or write part of the state machine. FLAGA becomes HIGH when there is some data in the DMA buffer of consumer socket 1 on the slave FX3. Therefore, FLAGA indicates that there is some valid data on the slave, and the master has to start the read operation. The master must check for a free buffer before driving any control signals required for a read operation. DMA_RDY_TH0 is used to determine the buffer status.

To do a write operation, there should be valid data in the DMA buffer of consumer socket 1 on the master FX3. To find out this condition, use DMA_RDY_TH1. In addition, there should be a free buffer on the slave side. This condition is verified with the help of FLAGB.

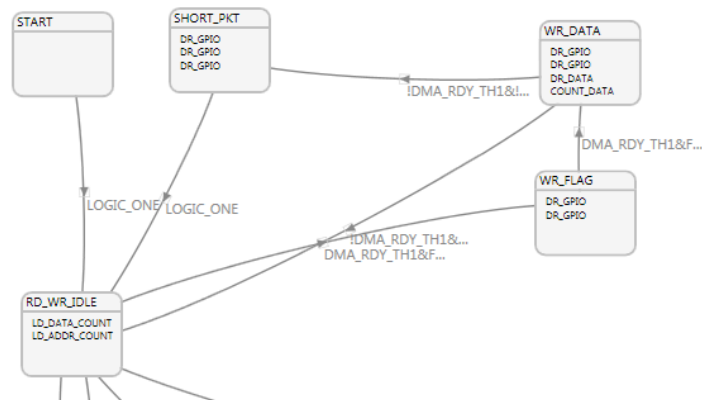
Figure 22. GPIF II Master State Machine



6.2 FX3 Master Write to FX3 Slave

The part of the state machine where FX3 master writes data to slave FX3 is shown in [Figure 23](#). See [Figure 6](#) for a list of signals that needs to be driven from the master FX3 to write data to the Slave FIFO.

Figure 23. Write Part of Master State Machine



The master state machine moves from the RD_WR_IDLE state to the WR_FLAG state when there is some data on the master side DMA buffer and there is a free buffer on the slave side. DMA_RDY_TH1 indicates the availability of data on the master side, and FLAGB indicates the availability of a free buffer on the slave side. In this state, the master drives the control signals SLCS# and SLWR# before it moves to the WR_DATA state. In this state, the master drives data and the control signals SLCS# and SLWR#.

You can allocate multiple buffers to a particular DMA channel when you configure the channel (when using CyU3PDmaChannelCreate API in firmware, refer to the [FX3 API guide](#) for more details on APIs). 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 512 bytes have been allocated to a DMA channel, the full FLAG will indicate full when 512 bytes have been written into the first buffer. The FLAG will continue to indicate full until the DMA channel has switched to the second buffer. The time required for the DMA channel to switch to the next buffer is not deterministic, although it is typically a few microseconds.

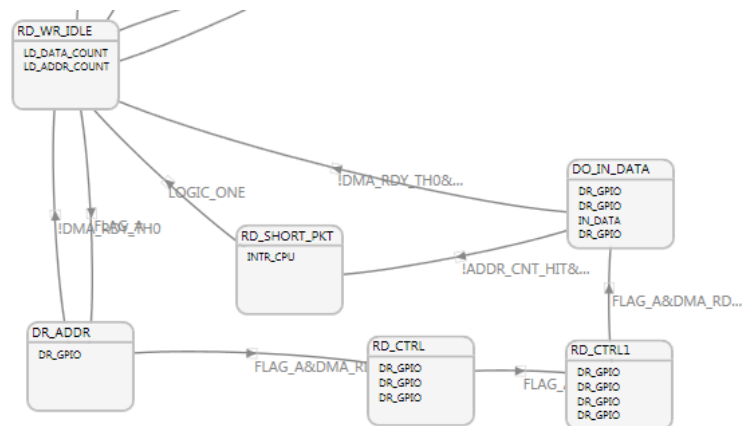
The master state machine goes to the RD_WR_IDLE state and reloads the address counter during this buffer switching delay. The ADDR_CNT_HIT event is used to find this buffer-switching condition. The DMA_RDY_TH1 flag cannot be used for this purpose because there is an internal delay of one clock cycle to update this flag. The state machine goes back to the WR_DATA state when the next buffer becomes available.

A short packet is identified when there is no more data in the DMA buffer and the ADDR_CNT_HIT event is not generated. PKTEND# and SLWR# signals are driven along with the short packet.

6.3 FX3 Master Read from FX3 Slave

The part of the state machine where the FX3 master reads data from slave FX3 is shown in [Figure 24](#). For a list of signals that needs to be driven from the master FX3 to read data to the Slave FIFO, see [Figure 5](#).

Figure 24. Read Part of Master State Machine



The master state machine moves from the RD_WR_IDLE state to the DR_ADDR state when there is some data on the slave side DMA buffer. In this state, the master drives the address lines. A0 is driven HIGH to address thread 1 of slave FX3. Then it checks if there is any free buffer and moves to the RD_CTRL state based on DMA buffer availability. In this state, the master drives the address and control signals SLCS# and SLOE#. Then it goes to the next state and drives the SLRD# signal in addition to the control signals that are already driven. Then it goes to the DO_IN_DATA state, where it continuously reads data by driving all of the required control signals.

As explained in the write part of the state machine, there will be some buffer-switching delay. It goes to the RD_WR_IDLE state during this delay, reloads the address counter, and returns to the DO_IN_DATA state when the next buffer is ready.

A short packet is identified when there is no more data in the DMA buffer of slave FX3 and the ADDR_CNT_HIT event is not generated. Then, the state machine moves to the RD_SHORT_PKT state and calls INT_CPU action. This generates an interrupt to FX3 firmware, and there we are wrapping up the DMA channel. Look for CyFxApplnGPIFEventCB() in the attached firmware.

Note: You can also use a COMMIT action instead of an INT_CPU action. However, you also need to have an IN_DATA action in addition to COMMIT; otherwise it will produce a Zero Length Packet (ZLP). Therefore, if you are replacing the INT_CPU action with COMMIT and IN_DATA actions, make sure that you reduce one count in the LD_DATA_COUNTER action settings.

Note: Use OUT_REG_VALID and IN_REG_VALID events when you are doing read and write operations over registers. (For this, select "Data Source" in IN_DATA or DR_DATA actions as "Registers").

If you plan to write some data at a particular address location on the peripheral device, then write that address and data to egress registers (registers to send data out of the FX3 device) in the firmware, and then you can monitor for those events in the state machine to check whether there is some valid data.

However, when you transfer a lot of data using DMA buffers, you need to check for DMA-ready flags. DMA-ready flags notify you whenever there is some valid data in the buffers.

7 FX3 Master Firmware Implementation

Apart from the GPIF-II state machine, FX3 firmware involves creation of DMA channels, starting them for data transfer and setting the values of the data and the address counter. Two AUTO DMA channels are created between the USB socket and the GPIF socket in order to transfer data. Each DMA channel consists of sixteen buffers, 2048 bytes each if connected to a USB 3.0 port (512 bytes if connected to a USB 2.0 port). The `CyFxAplnSetPibDIIParameters()` API is used to set the correct clock phase when operated in a master mode. Also, the data count registers are initialized to 511 (2047 bytes) if connected to a USB 3.0 port and 127 (512 bytes) if connected to a USB 2.0 port. When we get an interrupt from the GPIF-II state machine through the `INTR_CPU` action, we wrap-up the current buffer and commit it for transferring it to the USB in the `CyFxAplnGPIFEventCB()` API.

8 Hardware Connections

The GPIF II master design is tested using a second FX3 DVK board programmed to act as the synchronous slave interface. Interface connections between two FX3 chips are shown in [Figure 25](#). The FX3 back-to-back setup is shown in [Figure 26](#) using two FX3 SuperSpeed Explorer Kits (CYUSB3KIT-003). Two SuperSpeed Explorer boards are connected such that their GPIF interfaces are also connected. The interposing connectors (Samtec part number SSW-120-03-G-D) serve two functions; physically keeping the two boards apart such that the USB 3.0 connector does not interfere. Some pins of the interposing connector should be clipped off the interposing connectors so that the voltage pins and the low speed peripherals pins should not be shorted between the two boards. [Figure 27](#) shows which pins should be clipped off on the interposing connector before plugging them on the FX3 board.

Note: If you are using CYUSB3KIT-001, please refer to the [Appendix: Hardware Setup Using FX3 Development Kit \(CYUSB3KIT-001\)](#) for the hardware set-up information.

Figure 25. Hardware Connections

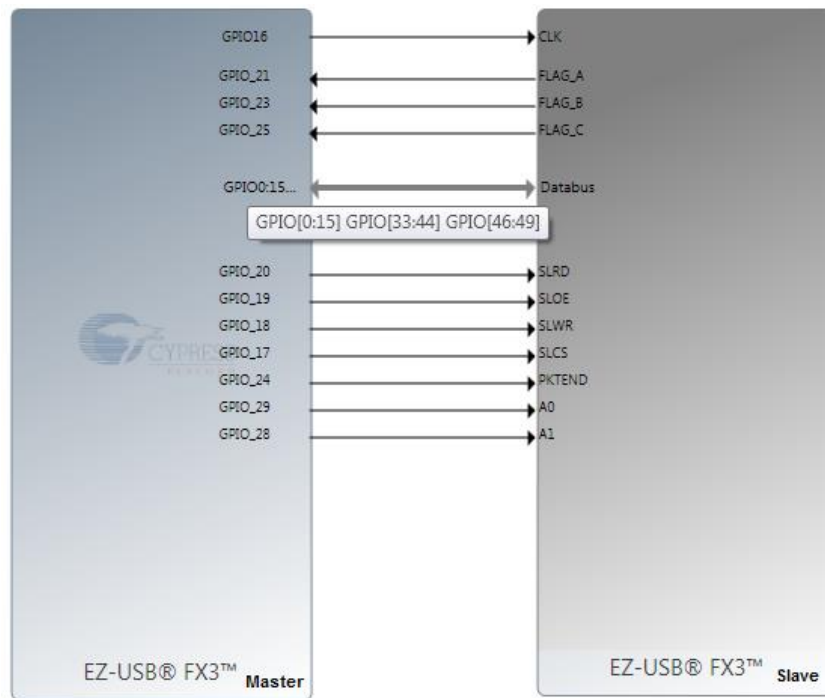


Figure 26. FX3 Back-to-Back Setup Using CYUSB3KIT-003

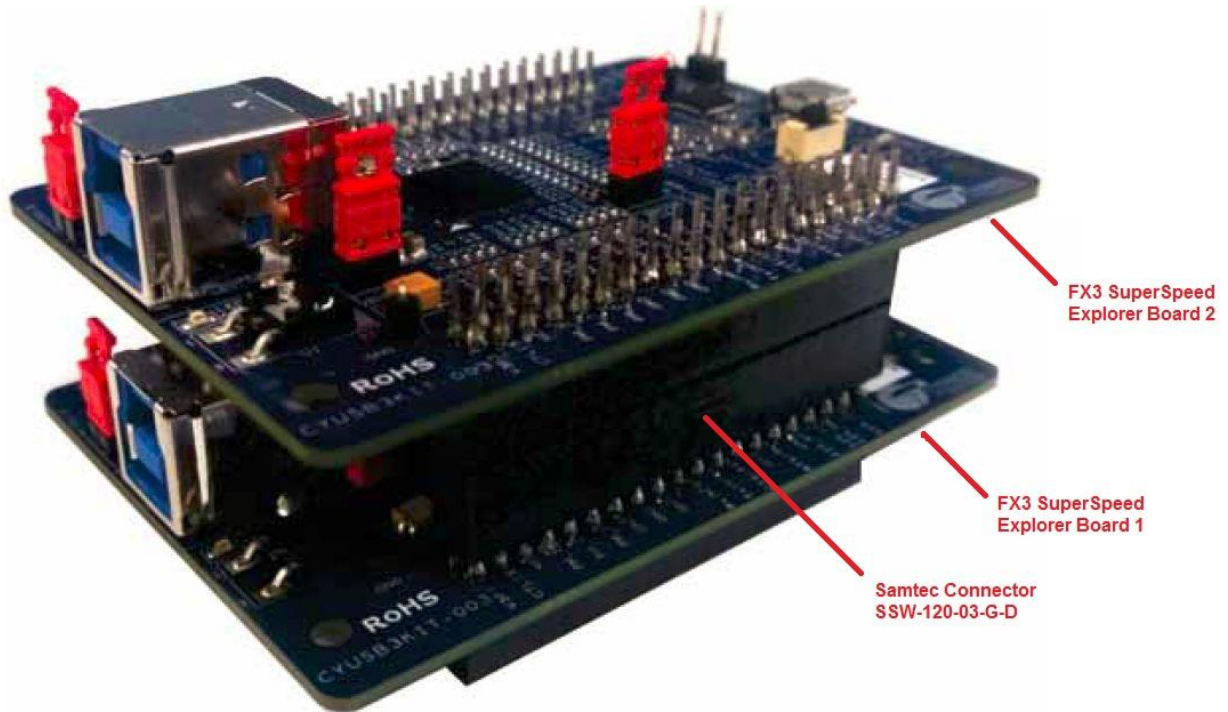


Figure 27. Clip the white pins on the interposing connectors



9 Steps to Run the Demo

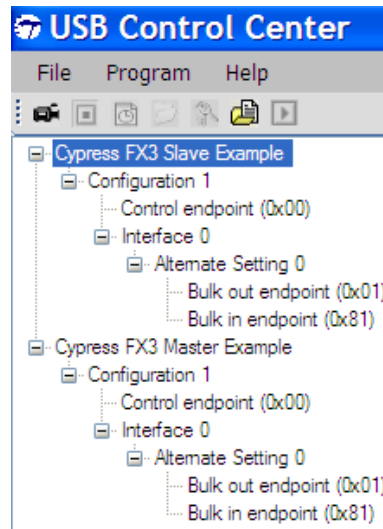
Firmware source code and GPIF II state machines for both slave and master FX3s are attached to this application note. This section shows you the steps to run a demo using the files attached to this application note.

1. Connect the two FX3 DVKs using the interconnection board. Refer to [Figure 26](#) in the hardware connections section.
2. Connect the two FX3 DVKs to a PC using USB 3.0 cables. Then, both devices enumerate as Cypress bootloader devices.
3. Use the USB Control Center to download firmware images into RAM of the two FX3 devices.

Note: Download the pre-built firmware image for the slave device (Autoslave.img in the bin folder of the attachment) before downloading the firmware image for the master device (Automaster.img in the bin folder of the attachment).

4. You can observe the enumeration of slave and master device, as shown in [Figure 28](#). The slave FX3 device uses a VID of 0x4B4 and a PID of 0x00F2. The master FX3 uses a VID of 0x4B4 and a PID of 0x00F4.

Figure 28. Master and Slave FX3 Devices



5. Each of these devices has one Bulk OUT endpoint and one Bulk IN endpoint. Transfer data to the Bulk OUT endpoint of slave FX3 and read back the same data from Bulk in the endpoint of master FX3. Click on the Transfer File-OUT button and browse to the 8192_count.hex (provided in the bin folder of the attachment). This step transfers 8 kbytes of data to the Bulk OUT endpoint of slave FX3, as [Figure 29](#) shows. [Figure 30](#) shows the data that has been read from the Bulk in endpoint of master FX3.

Figure 29. Transferring Data to Bulk OUT Endpoint of Slave FX3

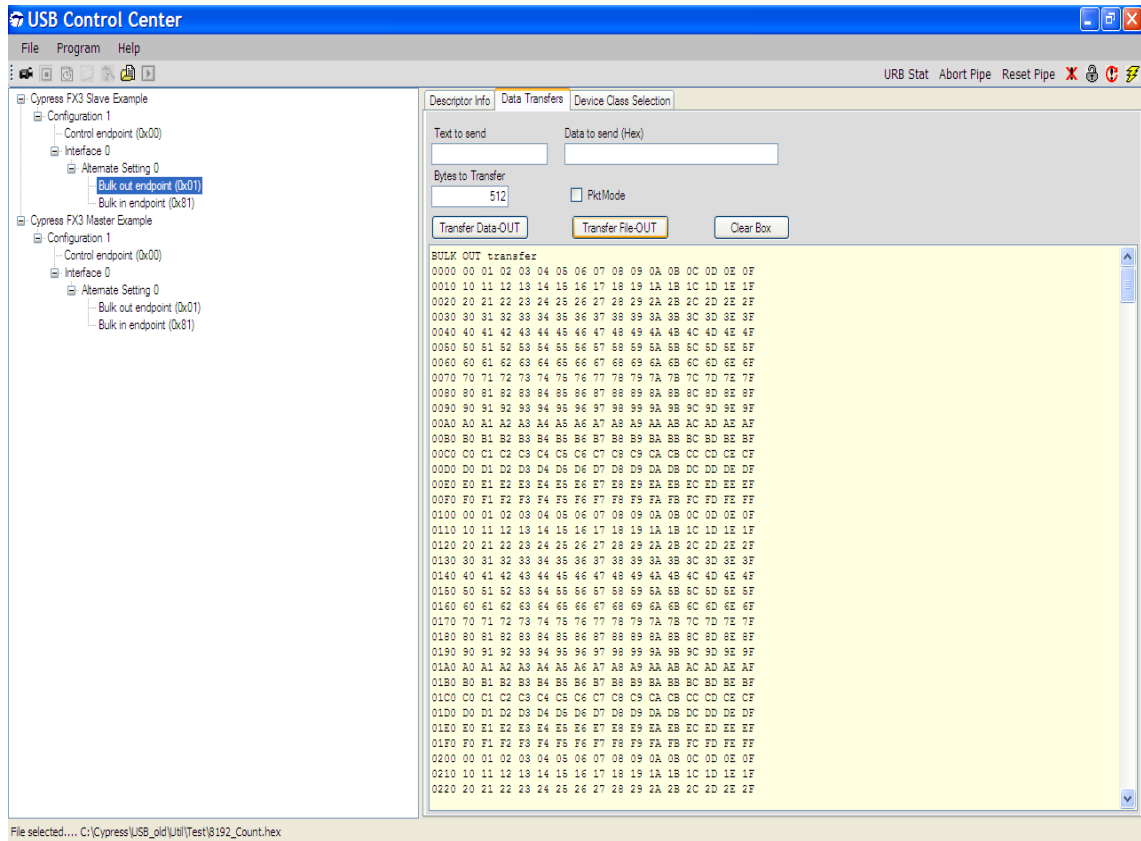
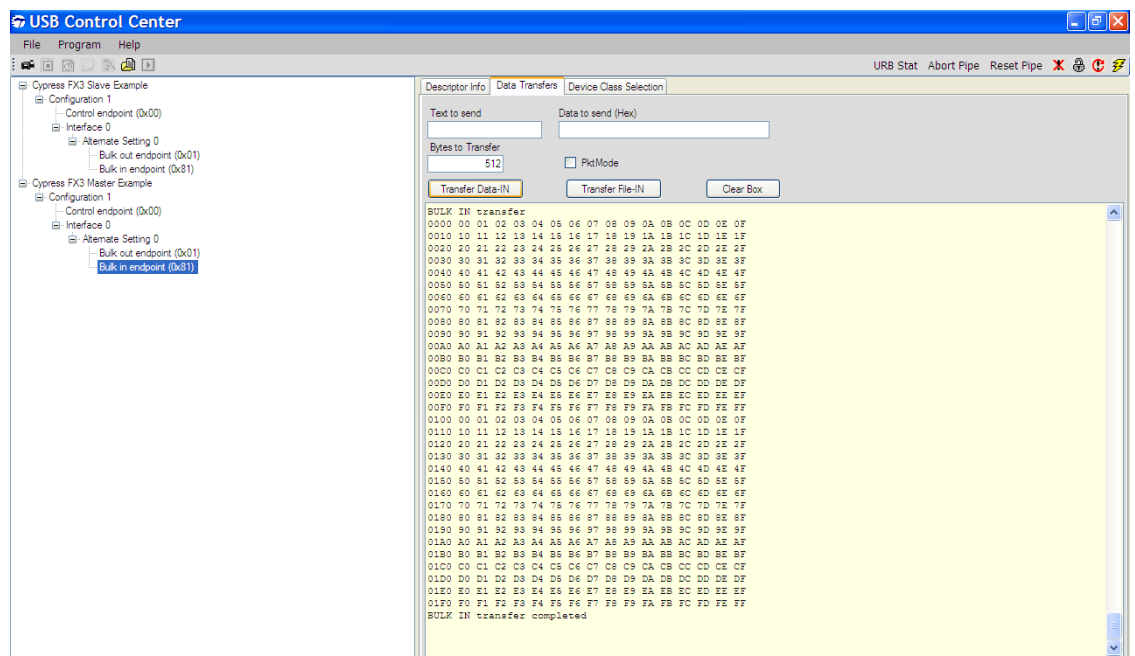


Figure 30. Reading Data from Bulk IN Endpoint of Master FX3



6. You can also transfer some short packets from one side to the other. [Figure 31](#) shows the transfer of 18 bytes to Bulk OUT endpoint of master FX3. [Figure 32](#) shows the data that has been read from the Bulk IN endpoint of slave FX3. You can observe an extra two bytes of 0s because of the 32-bit data bus.

Figure 31. Transferring a Short Packet to Bulk OUT Endpoint of Master FX3

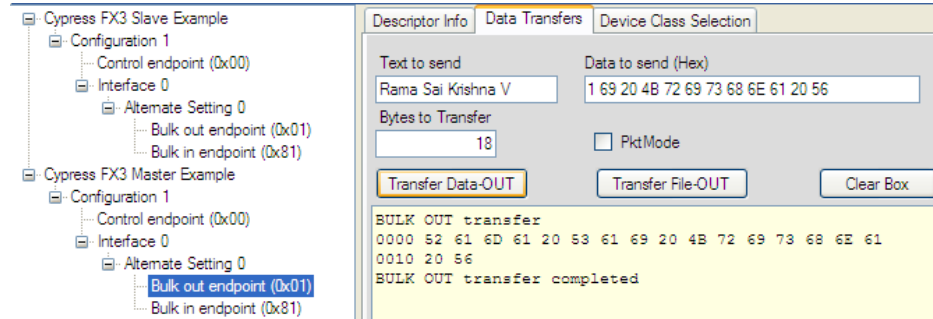
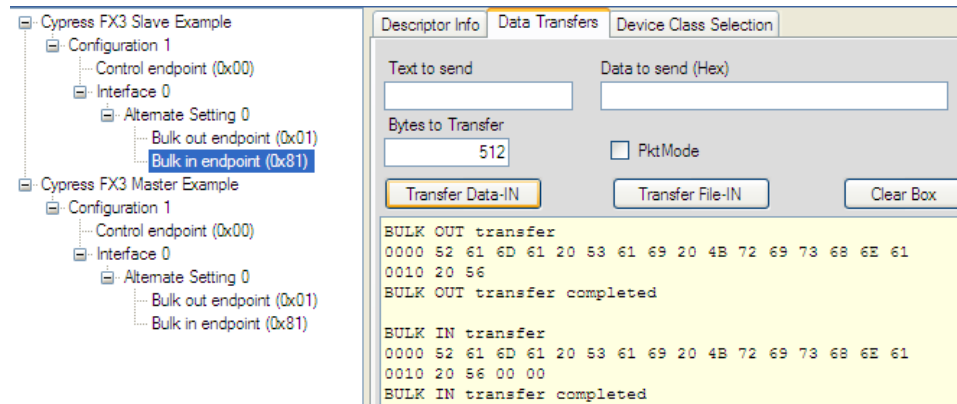


Figure 32. Reading a Short Packet from Bulk IN Endpoint of Slave FX3



7. [Figure 33](#) shows the transfer of 8 bytes to Bulk OUT endpoint of master FX3. [Figure 34](#) shows the data that has been read from the Bulk IN endpoint of slave FX3.

Figure 33. Transferring a Short Packet to Bulk OUT Endpoint of Master FX3

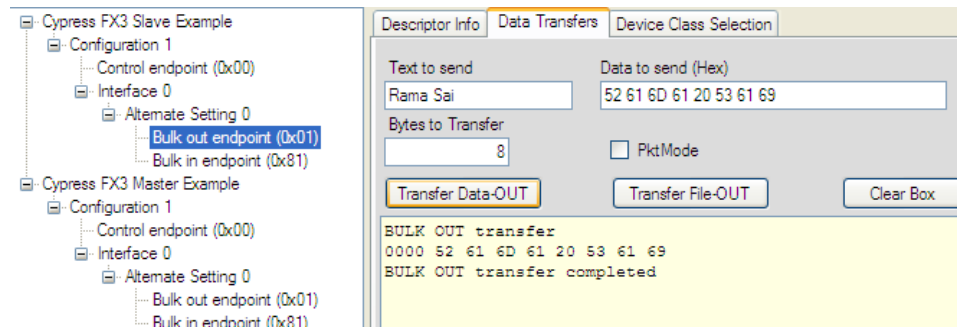
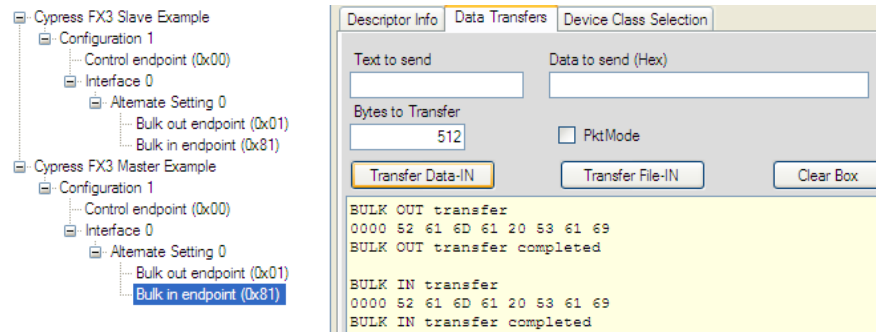


Figure 34. Reading a Short Packet from Bulk IN Endpoint of Slave FX3



10 Associated Project Files

Table 4 describes the files attached to this application note.

Table 4. Description of the Files in the Attachment to this Application Note

File/Folder	Description
FX3 Firmware	This folder contains the following folders: <i>AutoSlave</i> – FX3 firmware source files for slave FX3 <i>AutoMaster</i> – FX3 firmware source files for master FX3
GPIF II projects	This folder contains the following folders: <i>master_read_write_sync.cydsn</i> – GPIF II project for master FX3 <i>sync_slave_fifo_2bit_editable_latestGPIF.cydsn</i> – GPIF II project for slave FX3
bin	This folder contains the following files: <i>AutoSlave.img</i> – FX3 firmware image file for slave FX3 <i>AutoMaster.img</i> – FX3 firmware image file for master FX3. These image files can be downloaded into FX3 DVKs to see a quick demo of this application. <i>8192_count.hex</i> – File with 8192 bytes of incremental data. This is used to do file transfer of 8KB on an OUT endpoint of FX3.

11 Summary

This application note described all the details you need to understand to implement a master interface compatible with a synchronous Slave FIFO interface using the GPIF II Designer tool of FX3. You can use this design as a starting point to develop a GPIF II master state machine according to the application requirements.

12 Related Application Notes

[AN65974 – Designing with the EZ-USB® FX3 Slave FIFO Interface](#)

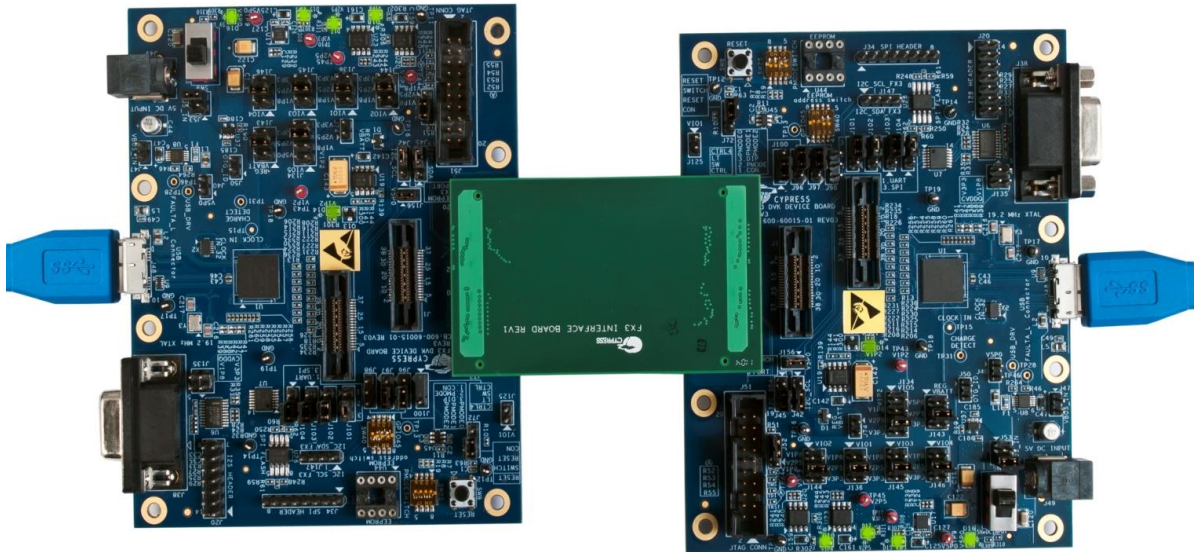
About the Author

Name: Rama Sai Krishna
 Title: Applications Engineer Sr.

Appendix: Hardware Setup Using FX3 Development Kit (CYUSB3KIT-001)

Figure 35 shows the hardware setup using two FX3 Development Kits (CYUSB3KIT-001) connected back-to-back.

Figure 35. FX3 Back-to-Back Setup Using Rev3 DVKs



Note: Make sure that jumper J100 is connected between posts 1 and 2 on both FX3 DVKs. This configures CTRL[4] (GPIO[21]) as FLAGA. Contact Cypress to get the schematics of the interconnection board between two FX3 DVKs.

Document History

Document Title: AN87216 - Designing a GPIF™ II Master Interface

Document Number: 001-87216

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	4078764	RSKV	07/29/2013	New application note.
*A	4212892	RSKV	12/06/2013	Removed references to the GPIF II videos on the Cypress web page.
*B	5022028	MDDD	12/14/2015	Added FX3 GPIF Master Timing Parameters. Updated FX3 GPIF Master State machine. Updated Figures with CYUSB3KIT-003. Added Appendix for CYUSB3KIT-001 hardware set-up. Updated to new template.
*C	5559025	MDDD	01/20/2017	Added More Information. Updated to new template. Completing Sunset Review.
*D	5774574	AESATP12	06/15/2017	Updated logo and copyright.
*E	6535653	HPPC	3/31/2020	Updated "Figure 3" to "Figure 5" in " Synchronous Slave FIFO Read Sequence " section. Completing Sunset review.

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

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

[PSoC 1](#) | [PSoC 3](#) | [PSoC 4](#) | [PSoC 5LP](#) | [PSoC 6 MCU](#)

Cypress Developer Community

[Community](#) | [Code Examples](#) | [Projects](#) | [Videos](#) | [Blogs](#)
| [Training](#) | [Components](#)

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.



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

© Cypress Semiconductor Corporation, 2013-2020. This document is the property of Cypress Semiconductor Corporation and its subsidiaries ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress shall have no liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. CYPRESS DOES NOT REPRESENT, WARRANT, OR GUARANTEE THAT CYPRESS PRODUCTS, OR SYSTEMS CREATED USING CYPRESS PRODUCTS, WILL BE FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (collectively, "Security Breach"). Cypress disclaims any liability relating to any Security Breach, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from any Security Breach. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. "High-Risk Device" means any device or system whose failure could cause personal injury, death, or property damage. Examples of High-Risk Devices are weapons, nuclear installations, surgical implants, and other medical devices. "Critical Component" means any component of a High-Risk Device whose failure to perform can be reasonably expected to cause, directly or indirectly, the failure of the High-Risk Device, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from any use of a Cypress product as a Critical Component in a High-Risk Device. You shall indemnify and hold Cypress, its directors, officers, employees, agents, affiliates, distributors, and assigns harmless from and against all claims, costs, damages, and expenses, arising out of any claim, including claims for product liability, personal injury or death, or property damage arising from any use of a Cypress product as a Critical Component in a High-Risk Device. Cypress products are not intended or authorized for use as a Critical Component in any High-Risk Device except to the limited extent that (i) Cypress's published data sheet for the product explicitly states Cypress has qualified the product for use in a specific High-Risk Device, or (ii) Cypress has given you advance written authorization to use the product as a Critical Component in the specific High-Risk Device and you have signed a separate indemnification agreement.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.