



CYPRESS

EZ-USB[®] FX2[™] GPIF Primer

Abstract

This primer first introduces the underlying architecture of the EZ-USB[®] FX2[™] before jumping into basic GPIF concepts, giving you a solid understanding of how the GPIF fits into the overall data path.

A methodology is then presented for developing GPIF applications. This allows you to get a grasp of what the key pieces that comprise a complete GPIF application solution are, and a framework for going about connecting the GPIF to the external slave device. Applying this methodology prevents you from being overwhelmed by the complex nature of the GPIF and guarantees a high rate of success.

Two end-to-end design examples are discussed in detail within the scope of creating GPIF waveform descriptors, GPIF firmware programming, and waveform analysis. By revealing the implementation details of these examples, you will become more familiar with the inner workings of the GPIF and should be able to apply the techniques presented to other practical applications. Debug tips are also presented where applicable, the largest tip (as evident by the ample waveforms presented) being that a logic analyzer is essential for any GPIF debug session.

The FX2 GPIF is a highly configurable and flexible piece of hardware that allows you to get the most out of your USB 2.0 design, and fits in applications that need an external mastering device for the exchange of information. This primer serves as an excellent guide for USB 2.0 peripheral interfacing with the FX2 GPIF.

1.0 Introduction

To achieve the maximum sustained throughput in USB 2.0 High-speed designs, the physical interconnect should never be the primary bottleneck in the system. EZ-USB FX2's GPIF (General Programmable Interface) provides a highly configurable and flexible glueless peripheral interface that allows the highest possible bandwidth to be achieved over the physical layer. However, along with this flexibility comes added complexity, and so starting on the right foot has never been more important. To help you get started on your own GPIF designs, this primer sheds some light on the architecture and implementation of FX2 GPIF, and discusses application usage models and debugging strategies. Two end-to-end examples are also discussed to reinforce GPIF concepts and provide you with concrete design examples. For the best learning experience, use this primer in conjunction with the FX2 Technical Reference Manual.

2.0 Understanding FX2 is Key to Understanding GPIF

2.1 Overview

EZ-USB FX2 offers a highly flexible and configurable Full-speed and High-speed USB peripheral function that's designed to achieve the maximum USB 2.0 High-speed bandwidth. FX2 accomplishes this through its auto-transfer and peripheral interface architecture. The GPIF engine is one of the vehicles for the auto-transfer architecture, and is used to gluelessly move data between FX2, your external slave device, and the USB host. The following sections briefly touch upon the silicon architecture and implementation of FX2, and lay the groundwork for understanding GPIF concepts discussed later on.

2.2 Why an Auto-Transfer Architecture?

In order to achieve the maximum USB 2.0 High-speed bandwidth, the CPU (in this case an enhanced 8051) should never be directly involved in moving the payload data from the external slave device to the USB host (and vice versa). The CPU would clearly be the largest bottleneck in a High-speed design. So instead an Auto-Transfer mode was invented, whereby the payload data is "auto-committed" from the USB host to the external slave device and likewise in the other direction. The GPIF engine uses this Auto-Transfer mode to move data to and from the external slave device. *Figure 1* and *Figure 2* show a system level view of the data flow for both the IN and OUT directions.

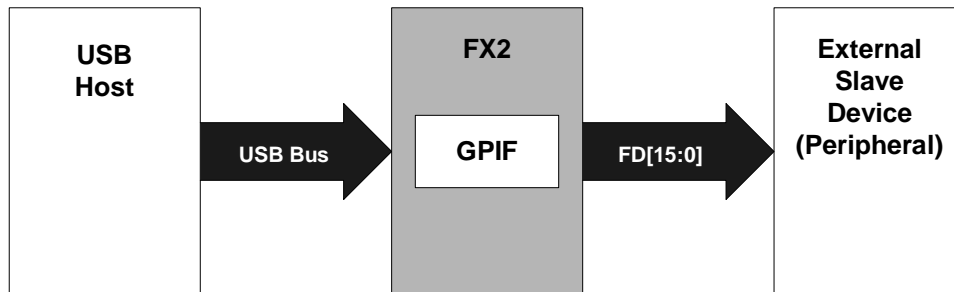


Figure 1. System Level View of Data Flow in the OUT Direction

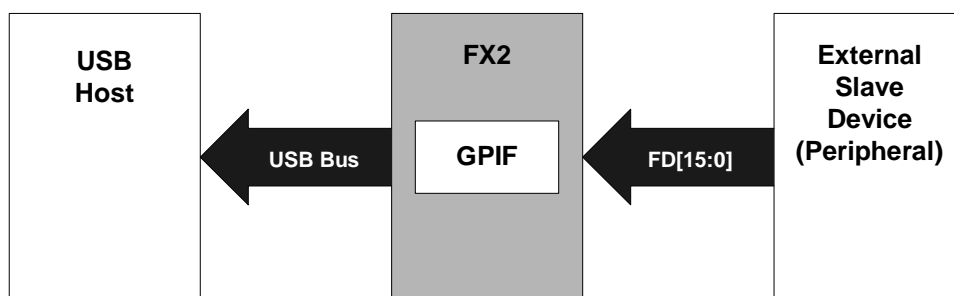


Figure 2. System Level View of Data Flow in the IN Direction

2.3 Silicon Features that Facilitate the Auto-Transfer Architecture

2.3.1 Endpoint FIFOs

In FX2, the USB endpoints share the same physical memory as the FIFO buffers. This is the very backbone of what allows the Auto-Transfer mechanism to work. Anytime the USB host sends a packet of data, the packet is stuffed into an available Endpoint FIFO buffer (the multiple buffering schemes available in FX2 assure that no NAKs appear on the USB side and no wait states appear on the interface side). The data packet is then ready to be distributed to the external device. Conversely, a data packet read from the external device can be automatically made available (or “committed”) to the USB host. FX2 can be configured to allow the CPU to manually commit the data packets (called Manual Mode) or allow the data packets to be committed automatically (called Auto Mode) to either the USB or peripheral domain. *Figure 3* and *Figure 4* show the maximum buffering scheme (4x) in relation to the overall data path the packets follow.

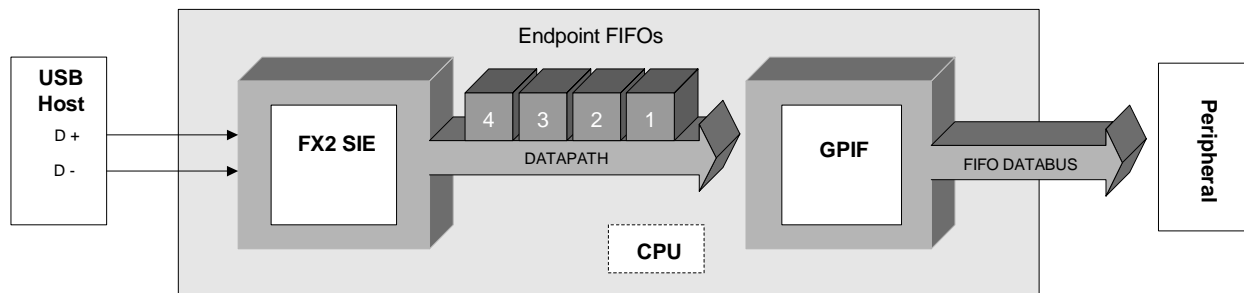


Figure 3. Auto-Transfer Data Flow in the OUT Direction

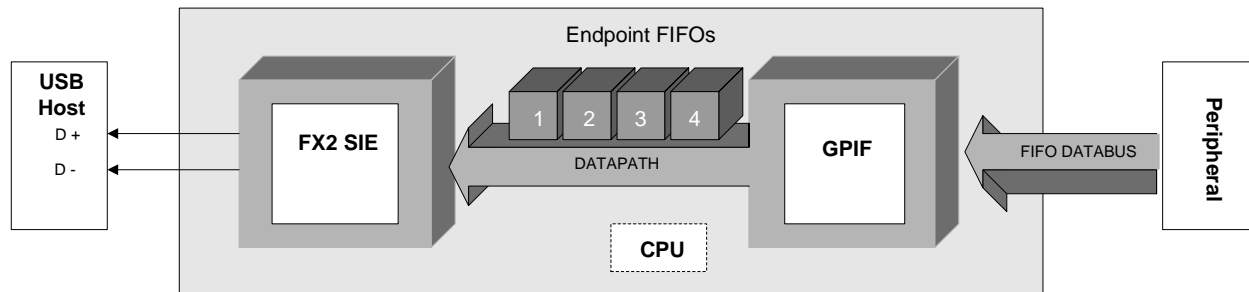


Figure 4. Auto-Transfer Data Flow in the IN Direction

2.3.2 CPU is the Traffic Cop

The CPU in FX2 does not participate in moving the payload data, but it does play a very important role nonetheless. The CPU configures and defines how the physical interface operates, sets up the endpoint configurations, triggers GPIF transfers, and can be allowed to manually commit the data packets to either the USB or peripheral domain. It can also monitor the status of the external world, thus giving it the capability of regulating GPIF transfers.

3.0 GPIF Application Usage

This section introduces the GPIF feature from a hardware standpoint and discusses how to design with GPIF from the ground up. As the name suggests, the physical connection or interface to the external peripheral is highly configurable. This allows FX2 to become the solution for many of the parallel interfaces that exist. With the ability to run the physical interface at a maximum rate of 48 MHz in 8- or 16-bit mode, the bottleneck in a USB 2.0 system should never be the actual GPIF interconnect itself. One of the objectives of this primer is to guide you down a path to get the most out of your GPIF designs in terms of performance over the physical interface. Thus, the general thought process for implementing successful GPIF designs is also discussed, allowing you to understand and use a design flow methodology for current and future GPIF implementations. Debugging strategies are provided where applicable to guide you on the best approaches for tackling GPIF design problems.

3.1 Overview

The GPIF is a key feature of FX2 used to master an external peripheral without any external glue logic. The GPIF is basically a controllable state machine that allows you to generate waveforms required by external peripheral read/write cycle timing. The CPU's role is minimal; it only loads the micro code that defines the waveform behaviors into designated FX2 on-chip RAM space, and manages how the waveforms are triggered in the application firmware.

3.2 Physical Interconnect

The GPIF has an interconnect which features a configurable 8- or 16-bit data bus, control outputs, and ready inputs. *Figure 5* shows what signals are available to you.

GPIF Signal Name Descriptions (FX2 in GPIF Master Mode)

IFCLK (bi-directional)

IFCLK (interface clock) is the reference clock for all GPIF operations. It can be either an input or output signal and its behavior is determined by the system requirements. As an output signal, IFCLK can be driven by FX2 at either 30 MHz or 48 MHz. As an input signal, its range is 5–48 MHz.

GPIFADR[8:0] (output only)

The GPIF can use GPIFADR[8:0] to provide address lines for peripherals that need them. These are output only signals and the value presented on this bus can be auto-incremented.

FD[15:0] (bi-directional)

This is the data bus used by GPIF operations and is the conduit for payload data transferred between FX2 and the external peripheral. It can be configured to operate as an 8- or 16-bit interface and can be tri-stated if the system requires it. In 16-bit mode, FD[7:0] represents the first byte in the endpoint FIFO and FD[15:8] represents the second byte in the endpoint FIFO.

CTL[5:0] (output only)

These are control outputs that can provide signals required by the external peripheral such as read/write strobes, enables, etc.

RDY[5:0] (input only)

These are ready input signals that can monitor status outputs from the external peripheral such as FIFO status flags, data available, etc. The GPIF has the ability to use these signals as decision point qualifiers for wait-state generation or inputs into a branch condition.

GSTATE[2:0] (output only)

These are debug output signals that represent the states executed in a GPIF waveform. These are typically connected to a logic analyzer.

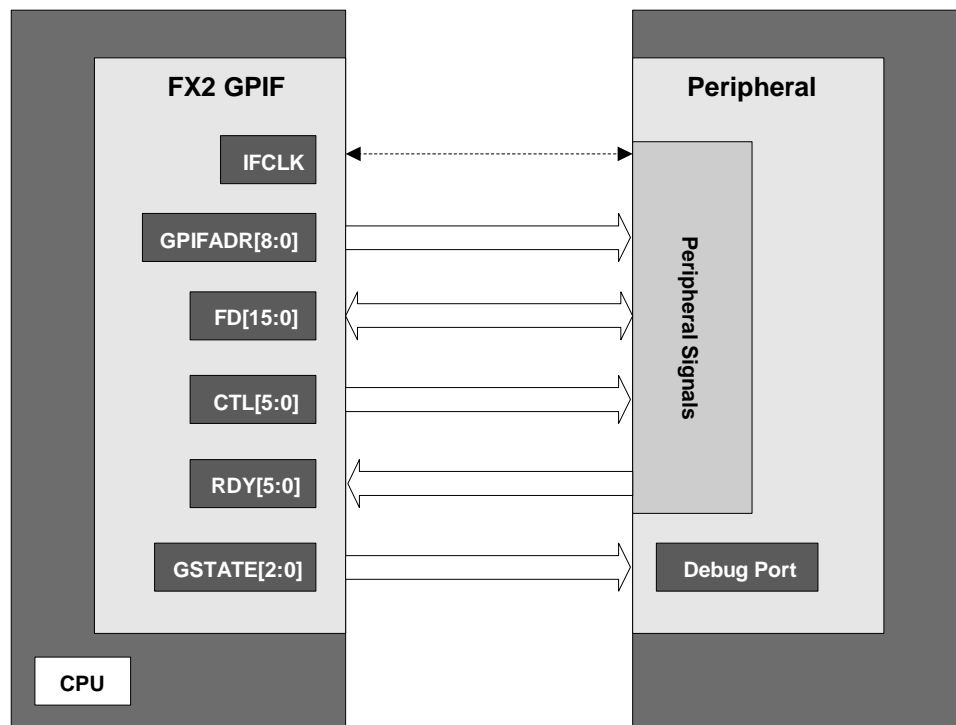


Figure 5. GPIF Interconnect Diagram

3.3 GPIF Application Design Flow Methodology (GADFM)

The following section presents a design flow methodology for GPIF developers wishing to maximize their “first time success” experience. The design examples that are discussed later on in this primer will often refer back to these design principles.

Design GPIF Interconnect

Using section 3.2, you should have a pretty good basis for determining how FX2 is going to be connected to your peripheral device using GPIF. This is also a good time to start collecting the FX2 data sheet and Technical Reference Manual, as well as the peripheral's data sheet as a minimum. You can then start by asking yourself the following questions:

- *Is the data path going to be 8- or 16-bit?*

This decision is often dictated by what size data path the peripheral offers. If it has a 16-bit data path, use it to maximize the bandwidth over the physical interface.

Endianness and bit nomenclature may also come into play here so watch how those lines are wired.

- *Will an external IFCLK be used or an internal 30- or 48-MHz clock source?*

This decision is often based on how flexible the peripheral is in terms of its own operating modes. For example, if it can accept a 30- or 48-MHz clock input, then the internal 30- or 48-MHz clock source can be used to serve as the clock input for the slave device.

- *Does the peripheral require any address lines during its read/write cycles?*

If the peripheral requires any lines to be addressed during a read/write cycle operation, then GPIFADR[8:0] can be used (or port I/O pins).

- *How many control lines does the peripheral need?*

Designate GPIF control outputs from CTL[5:0]. The peripheral may require read/write signals, chip selects, and other control inputs during a read/write cycle. Determine what they are and allocate CTL[5:0] appropriately.

- *How many inputs does the GPIF need to monitor from the peripheral?*

Determine how many status signals need to be monitored during a read/write cycle. Identify what they are and allocate RDY[5:0] appropriately.

Note: Not all FX2 package types have the complete set of GPIF signals available so you have to be mindful of this when determining the GPIF interconnect. You also need to consider what other signals may be used (such as port I/O) to interface the FX2 to the peripheral (e.g., reset signals, other address lines). For instance, the 56-pin package does not bring out GPIFADR[8:0]. Therefore, if the peripheral needs some sort of address scheme before a GPIF read/write cycle is initiated, port I/O pins must be used instead (see discussion of TI DSP example in section 4.2 for more details).

Use Firmware Frameworks

Before embarking on any FX2 firmware project, let alone a GPIF one, it is strongly recommended to start with a *Firmware Frameworks* based Keil uVision2 project. Not only will this ensure that the integration and test phase will be smoother, it will also help Cypress USB Applications Support know that you're starting from a well-known and well-tested firmware base. Any of our firmware examples will be Frameworks based so you can start with one of those, or start by copying the contents of C:\Cypress\USB\Target\Fw\FX2 to a new sub-directory. This will allow you to start with a "clean" firmware base. Starting with a Firmware Frameworks project also allows you to concentrate on your application code as the USB protocol servicing has been handled already (see the file called fw.c and the development kit documentation for more details).

There are two main parts to a complete GPIF Application Solution (GAS):

1. The higher-level firmware that configures the GPIF and launches the transfers (including the rest of your application code).
2. The GPIF waveform descriptors that implement the physical bus timing.

Part 1 normally consists of five files (fw.c, periph.c (you can rename this file), dscr.a51, ezusb.lib, usbjmbt.obj) which makes up the Keil uVision 2 *Firmware Frameworks* project. Part 2 is typically a self-contained source file (gpif.c) that contains the GPIF waveform descriptors and is added to the Keil project. The GPIF Designer, a user mode application supplied with the development kit software, generates this output source file.

Implement GPIF Waveform Descriptors using the GPIF Designer

As mentioned above, the GPIF Designer generates the GPIF waveform descriptors that implement the physical bus timing. The GPIF Designer can implement multiple waveform behaviors, which can then be assigned to four waveform types: Single Write, Single Read, FIFO Write, and FIFO Read. The assignment is made through the GPIFWFSELECT register setting. Each descriptor is 32 bytes long and resides in a special GPIF waveform descriptor area in on-chip memory space, once loaded by the CPU.

It is your responsibility to create these GPIF waveform descriptors that will in turn execute single or FIFO read/write transactions to the peripheral. You have seven states or intervals (S0-S6) to work with before having to terminate the transaction naturally by branching to a "special" IDLE state (S7) (*Figure 6* shows an example of a simple waveform broken down into the GPIF state transitions). Once the GPIF waveforms are ready to be exercised by the CPU, the reading or writing of specific GPIF "trigger" registers determine at any one time which of the four waveform types gets executed by the GPIF engine.

The usage details of the GPIF Designer will become clear in the discussion of the two design examples later in this document. It is important to note at this point that the first set of waveforms you should implement are the single read/write transaction waveforms. This will allow you to flesh out the physical interconnect and get data moving back and forth among the USB host, FX2, and the peripheral. It's always wise to start by implementing single read/write waveforms before leaping into the more conceptually difficult FIFO read/write transactions, because getting comfortable with GPIF is essential to a successful design. Taking the time to absorb the learning curve up front will pay large dividends in the end.

Implement Single Read/Write Transactions

The main goal of implementing single read/write transactions is to be able to confirm that the physical interconnect is sound between the GPIF and the peripheral, and that basic data movement can be achieved in the final system. It is possible to get an entire design working, even if only single transactions are utilized. Another advantage of performing this stage first in the GPIF development cycle is that all areas of the system (hardware, firmware, software) can be verified within a relatively short period of time (compared to jumping into FIFO transactions first).

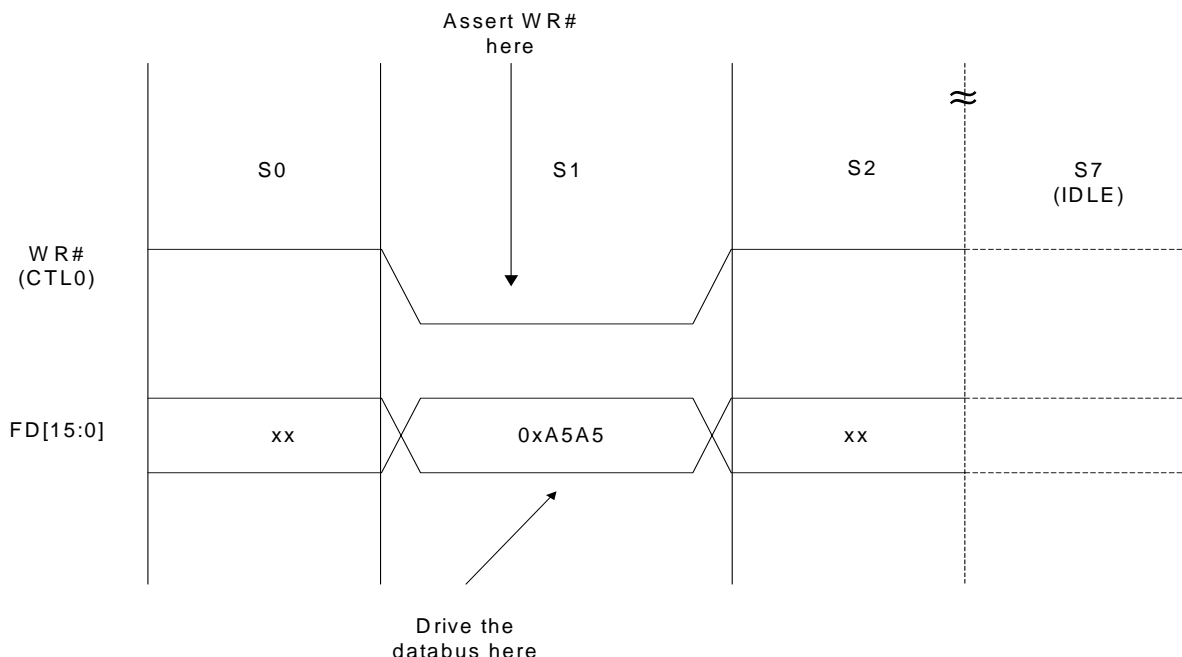


Figure 6. A Simple Waveform Broken Down into GPIF States

Even though single transactions are the easiest of GPIF waveforms to implement, it may take several iterations of integration and test of the GAS project before you are happy with the initial design, since this stage can be also considered as the “fleshing out” stage. There may be hardware-timing issues that pop up, hardware connection issues, firmware code flow issues, driver issues, etc., but this is the right time to become aware of these types of issues.

Implement FIFO Read/Write Transactions

Once you are satisfied with the single transaction implementation, the next logical step is to implement GPIF FIFO Read/Write transactions to achieve higher bandwidth. It is highly recommended that the FIFO Write waveforms be implemented and tested first.

FIFO Write transactions are always the easier to implement out of the two types. You can perform a FIFO Write transaction and verify the integrity of the data on the peripheral side, and if the data looks good then implementing FIFO Reads is the next step. This is an important accomplishment because it will rule out the FIFO Write code as the culprit if problems arise with the FIFO Read waveform in a loop back scenario; you will never know which side is truly at fault if each operation is not implemented and tested independently.

At this development stage, there may also be several iterations of integration and test before you are happy with the overall performance of the design.

Optimize if Necessary

In generating the GPIF waveforms, you may initially decide to be generous with the physical bus timing. So, there may be room for improvement to cut down the cycle time for each GPIF transaction and still meet the timing parameters required by the peripheral. The design may also be revised to improve firmware code efficiency and overall firmware code flow at this stage.

Summary

This section has discussed a GPIF application development flow that uses a systematic approach to help ensure a successful GPIF design the first time. At this point, you should have a good feel for the amount of effort involved and an approach to tackle a GPIF design from start to finish. *Figure 7* summarizes this section in a GADFM Flow Diagram. The sections ahead present the two design examples that will dive into the hardware, firmware, and software necessary to implement a practical GPIF application.

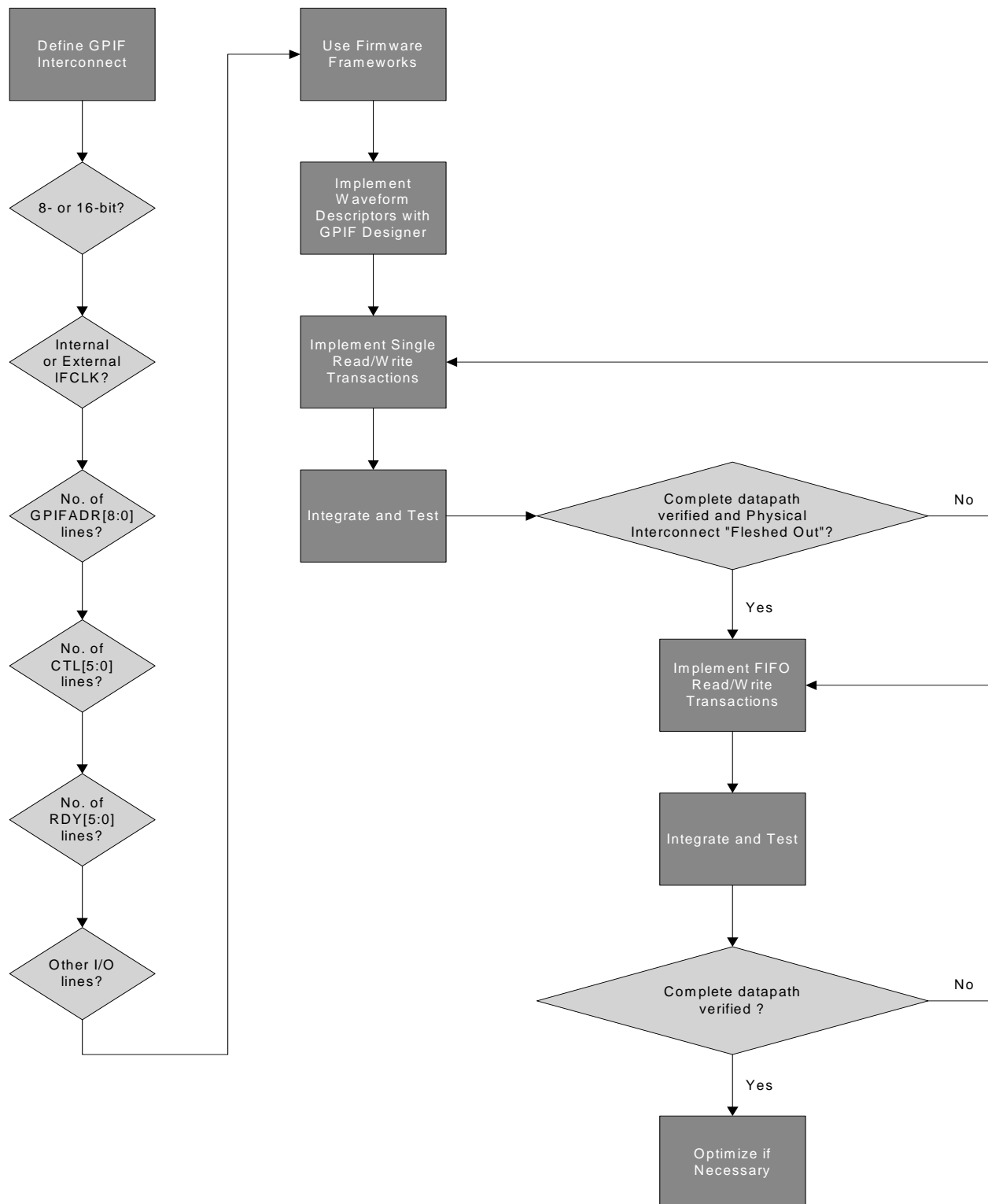


Figure 7. GADFM Flow Diagram

4.0 Design Examples

The following sections present two design examples aimed at giving you enough information to start designing your own GPIF applications. The first example demonstrates how to interface with an external synchronous 16K deep FIFO from Cypress and perform a bulk loopback function, and the second example shows how to interface with a TI DSP and access the host port interface (HPI) port. The goal of these examples is to shed light on the GPIF concepts previously discussed and dive into the implementation details such that you will become familiar with how the GPIF can be used in a practical application.

4.1 16-bit Interface to an External Synchronous FIFO CY7C4625-15AC

4.1.1 Overview

The end objective of this example is to be able to perform a bulk loopback function with the external FIFO. The FX2 will write data out FD[15:0] to the external FIFO and read data back from FD[15:0] (the outputs from the external FIFO Q[15:0] are also connected to FD[15:0]). The bulk transfers can be exercised by using the EZ-USB Control Panel or bulkloop.exe utility supplied with the EZ-USB development kit software.

4.1.2 Hardware Connections

Table 1 discusses the definition of the GPIF interconnect, which is shown below in Figure 8.

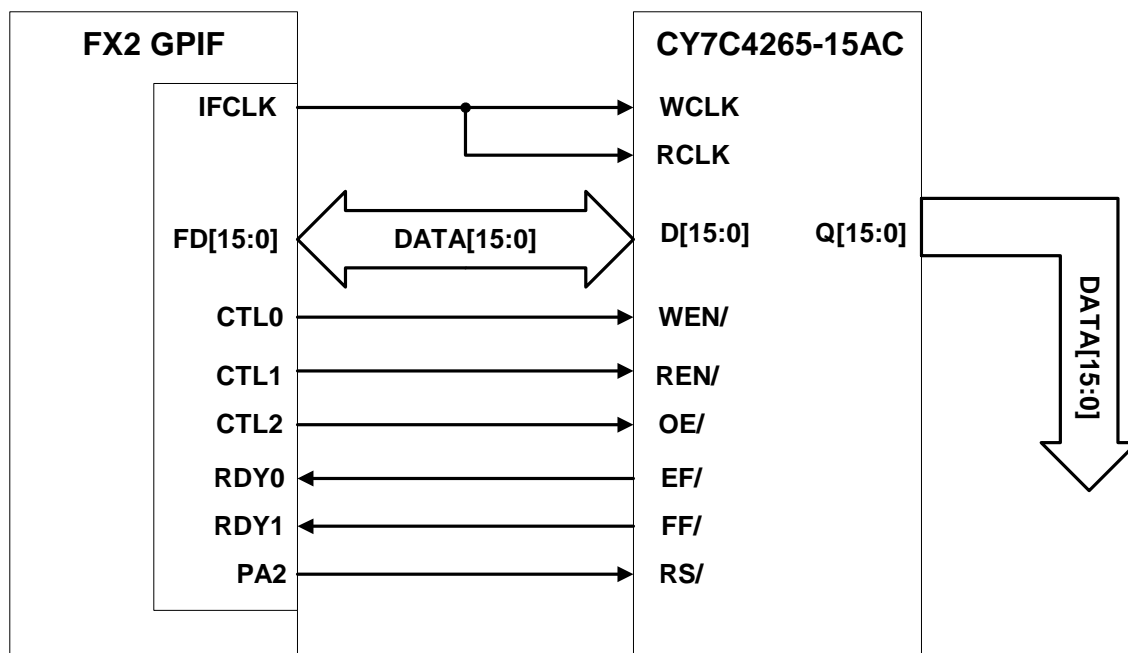


Figure 8. GPIF Interconnect to External Synchronous FIFO (CY4625-15AC)

Table 1. Assignment of FX2 GPIF Signals to CY4625-15AC Signals

FX2 GPIF Signals	CY4625-15AC Signals	Description
IFCLK	WCLK, RCLK	IFCLK is connected to the write and read clock inputs (WCLK, RCLK) of the external FIFO. Data is clocked into the external FIFO on every rising edge of WCLK while WEN is asserted. Likewise, new data is presented on Q[15:0] on every rising edge of RCLK while REN and OE are asserted. The external FIFO can accept an input clock frequency of up to 66.7 MHz so it can handle the incoming IFCLK frequency (either 30 MHz or 48 MHz)
FD[15:0]	D[15:0], Q[15:0]	The GPIF data bus (FD[15:0]) is connected to the external FIFO's input data bus (D[15:0]) to allow for word-wide operations. The output data bus of the external FIFO (Q[15:0]) is also connected to the GPIF data bus to allow the FX2 to read back the data contents. In order to ensure that bus contention will never occur, the OE signal must be manipulated appropriately
CTL0	WEN/	CTL0 is connected to the write enable line (\overline{WEN}) of the external FIFO. While WEN is held LOW, data is written into the external FIFO on every rising edge of WCLK
CTL1	\overline{REN}	CTL1 is connected to the read enable line (\overline{REN}) of the external FIFO. While REN and OE are held LOW, new data is presented on Q[15:0] on every rising edge of RCLK
CTL2	\overline{OE}	CTL2 is connected to the output enable line (\overline{OE}) of the external FIFO. While REN and OE are held LOW, new data is presented on Q[15:0] on every rising edge of RCLK
RDY0	\overline{EF}	RDY0 is connected to the empty flag (\overline{EF}) of the external FIFO. \overline{EF} is asserted if the external FIFO is empty. The GPIF can use this to regulate data transfers when reading from the external FIFO
RDY1	\overline{FF}	RDY1 is connected to the full flag (\overline{FF}) of the external FIFO. \overline{FF} is asserted if the external FIFO is full. The GPIF can use this to regulate data transfers when writing to the external FIFO
PA2	\overline{RS}	PA2 is connected to the reset signal of the external FIFO. PA2 is not part of the GPIF interconnect but is still part of the overall system design. PA2 is used as an I/O pin to reset the external FIFO to a known state before GPIF data transfers commence

The assignment of CTLx and RDYn lines is optimized for the FX2 56-pin package. The CTLx lines are used as input strobes into the external FIFO, and the status outputs from the external FIFO (EF and FF) are used to monitor under run and over run conditions. The basic rule of thumb is: one should never read from an empty FIFO or write to a full FIFO.

The external FIFO was mounted onto an FX2 development board by using the prototype board supplied with the development kit. The external FIFO was placed on a 64-pin TQFP package surface mount adapter (available from Twin Industries at www.twinhunter.com) and piggybacked on top of the prototype board. *Figure 9* shows a snapshot of the actual hardware set-up. For full hardware specifications on the external FIFO, its data sheet can be downloaded from the Cypress website. For a pin-out list for the prototype board connection to the FX2 development board and a full schematic for the external FIFO prototype board, see the software contents available with this primer.



Figure 9. Hardware Set-up

4.1.3 Application-specific Data Flow

Now that the GPIF interconnect has been presented, it's important to understand the overall data flow for this design example. Endpoint 2 OUT (EP2OUT) is used as the source endpoint for GPIF writes to the external FIFO, and Endpoint 6 IN (EP6IN) is used as the sink endpoint for GPIF reads from the external FIFO. Remember that the IN and OUT directions are USB host-centric, therefore EP2OUT contains the data packets sent by the USB host (in this case the PC) and EP6IN contains the data packets sent to the USB host. *Figure 10* and *Figure 11* show the data flow models for this particular example.

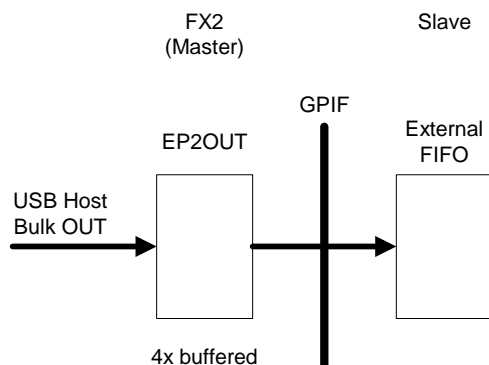


Figure 10. Data Flow Model in the OUT Direction

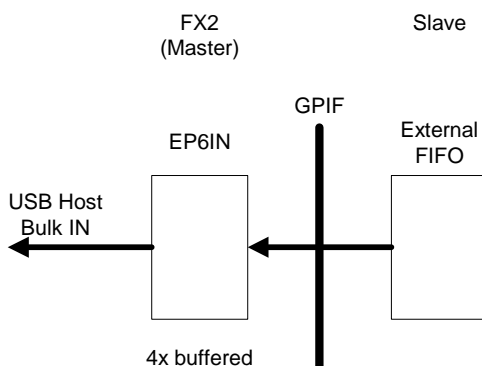


Figure 11. Data Flow Model in the IN Direction

Manual Mode versus Auto Mode

FX2 endpoints can operate in two modes, Manual (AUTOIN/AUTOOUT = 0) or Auto (AUTOIN/AUTOOUT = 1). Manual mode makes the CPU responsible for committing the USB packet to the peripheral domain and vice versa. In order to maximize the USB 2.0 bandwidth, Auto mode should be used. This allows USB packets to be committed automatically to the peripheral domain and vice versa by completely removing the CPU from the data path. This example demonstrates the use of Auto mode.

Two Examples in One

The FIFO example is really two examples in one because two versions of the firmware are discussed. Sections 4.1.4–4.1.6 present a version that uses GPIF single transactions to read and write to the external FIFO, which then sets the stage for sections 4.1.7–4.1.9. Sections 4.1.7–4.1.9 discuss a version that uses GPIF FIFO transactions and the endpoints in auto mode, thus maximizing the USB 2.0 bandwidth. This two-phased approach is in line with the methodology presented in section 3.3, and by understanding the two approaches, you should be able to discern what it takes to move from a simple working example to an example that utilizes the full USB 2.0 bandwidth capabilities of the FX2.

4.1.4 Creating Single Transaction GPIF Waveform Descriptors using the GPIF Designer

As previously mentioned, the simplest set of GPIF waveforms to produce is the single read and write waveforms. Performing this function first will not only enhance your initial experience with the design, but also allows the design to be in fairly good shape, fairly quickly. Of course, for high bandwidth applications the natural migration is then to create waveforms that use FIFO read and write transactions, but the implementation of single transaction waveforms is the right place to start. This section discusses the single transaction waveforms implemented using the GPIF Designer, and the next section covers the firmware that triggers them.

The GPIF Designer makes creating GPIF waveform descriptors easy. Rather than having to know each bit of the waveform descriptor opcode bytes in detail to create a waveform, the GPIF Designer allows you to “draw” each waveform and export the waveform descriptors to a self-contained file, typically called gpif.c. When you open GPIF Designer, it will present you with a block diagram view of the physical interconnect as shown in *Figure 12*. You can then use the block diagram view to name the individual

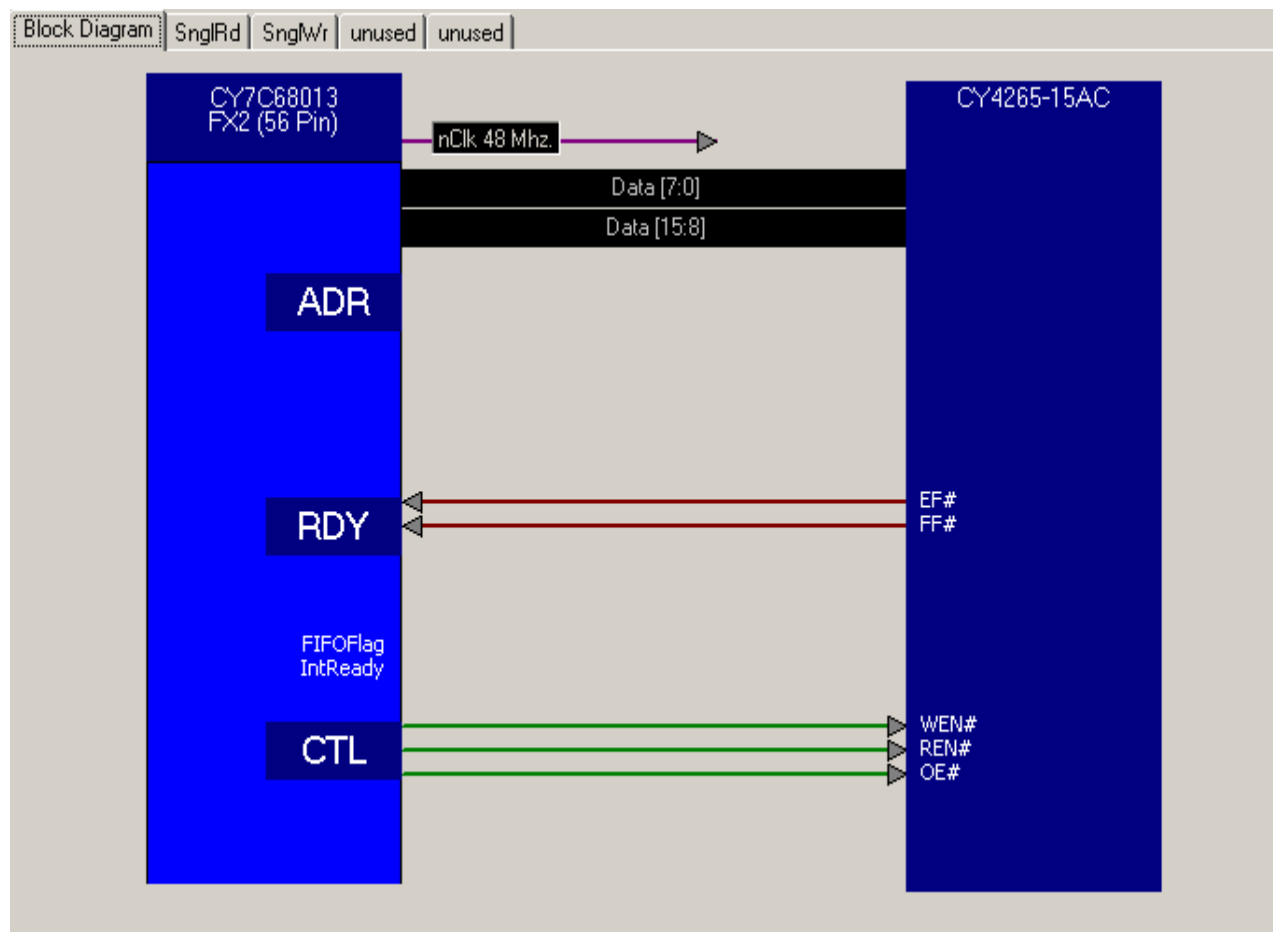


Figure 12. GPIF Designer Block Diagram View

CTLx lines and RDYn signals. These names propagate into the waveform tabs, allowing you to personalize each waveform and determine which signals are being manipulated. You can also use the block diagram to configure the clock properties of IFCLK, select different package types, and label the external slave device.

We can see that the naming conventions are consistent with the hardware set-up. The single write waveform (waveform 1) is shown below in *Figure 13*.

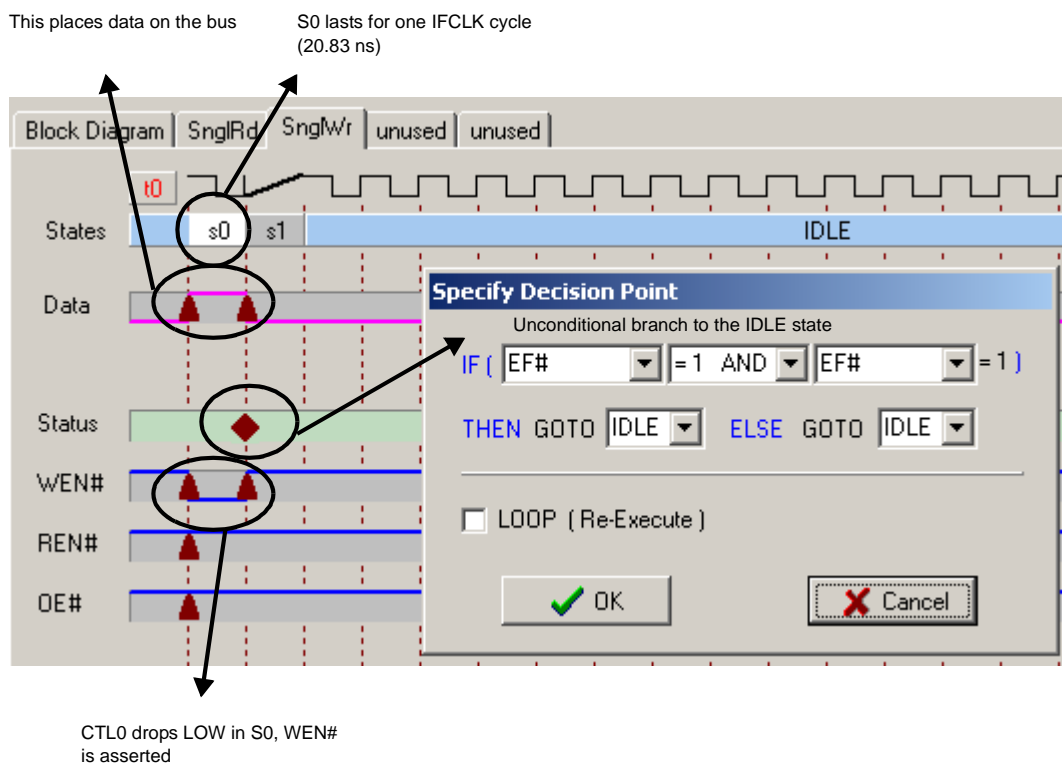


Figure 13. Single Write Waveform in GPIF Designer

For the single write waveform, data is written to the external FIFO in S0 by making CTL0 a logic LOW (\overline{WEN} is asserted) and placing data on the bus (Activate Data) for one IFCLK cycle (Wait 1). At 48 MHz, one IFCLK cycle is 20.83 ns. With the IFCLK output inverted, this provides enough set-up and hold time for the data.

S1 is a decision-point state that forces an unconditional branch to the IDLE state, which terminates the waveform (no activity occurs in the IDLE state). A decision point state allows you to pick, at most, two terms to evaluate a logical expression. Based on the results of that evaluation, you can control the next state the waveform goes to. See the FX2 Technical Reference Manual for more information on decision point states. Also in S1, CTL0 is a logic HIGH (\overline{WEN} is de-asserted), and the data bus is tri-stated (De-activate Data).

Every time a single write waveform is initiated, the GPIF engine will cycle through S0, S1, and then S7.

The single read waveform (waveform 0) is very similar to the single write waveform. The single read waveform is shown in *Figure 14*.

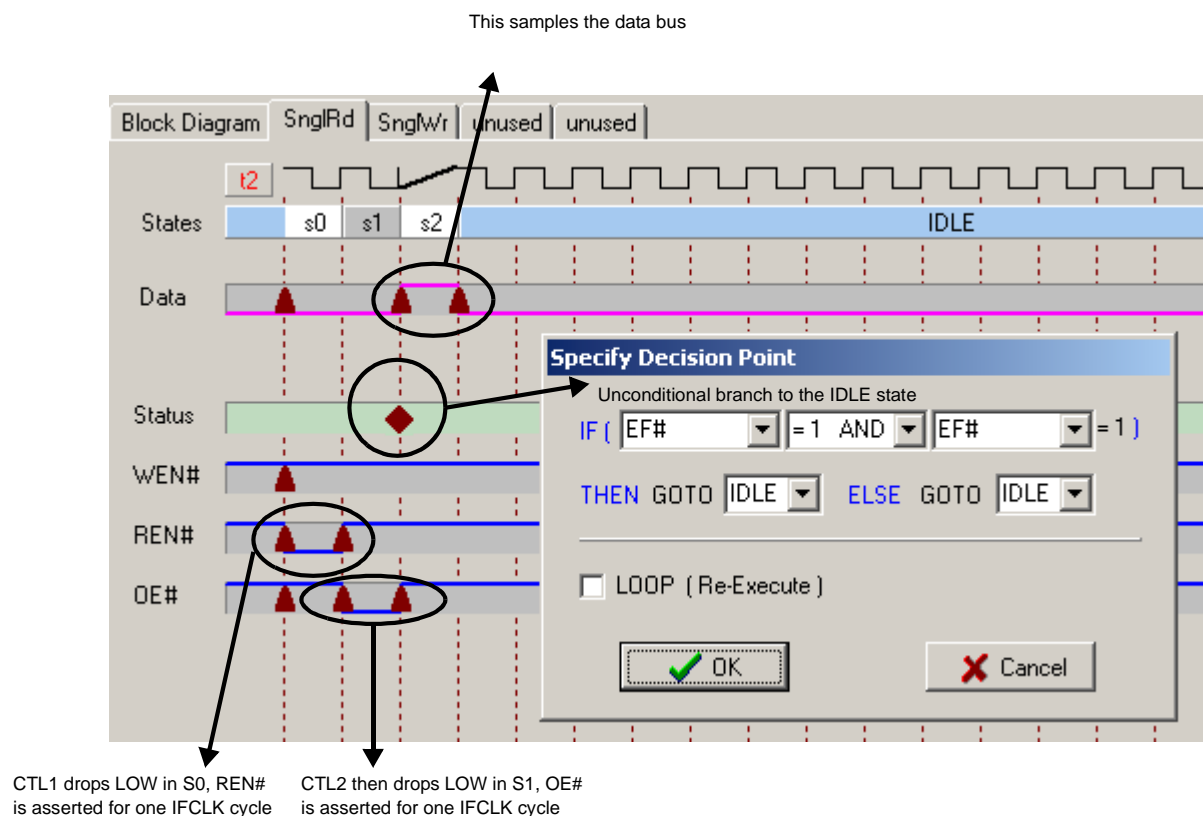


Figure 14. Single Read Waveform in GPIF Designer

For the single read waveform, CTL1 starts off as a logic LOW in S0 ($\overline{\text{REN}}$ asserted) for one IFCLK cycle. This is to account for a t_{ENS} set-up time for the external FIFO before OE (CTL2) is asserted. S1 then asserts OE, and in S2 the data bus is sampled (Activate Data) and an unconditional branch to the IDLE state is taken to terminate the waveform (no activity occurs in the IDLE state).

Note that the data bus is sampled in S2 when it would be tempting to sample it in S1. At the beginning of S1, the data is not yet available from the external FIFO, therefore the GPIF has to "catch" the data at the beginning of S2. This is why the data bus is sampled in S2 instead of S1.

Every time a single read waveform is initiated, the GPIF engine will cycle through S0, S1, S2, and then S7. Notice that waveforms 2 and 3 are unused for the single transaction example, but will be used later for the FIFO transaction example.

4.1.5 Firmware Programming for GPIF Single Transactions

After the single transaction waveforms are implemented in the GPIF Designer, the next step is to integrate the USB portion of the overlying firmware with the GPIF Designer output to perform write and read operations to and from the external FIFO. To do this a Firmware Frameworks project was copied and the code that performs the external FIFO operations was added to the `TD_Poll()` function within `FX2_to_extsyncFIFO.c` (note that `periph.c` was renamed to something more meaningful here). Endpoint and GPIF register initialization is performed in the `TD_Init()` function, which is also within `FX2_to_extsyncFIFO.c`. When you open up the Keil uVision2 project for the FIFO example, *Figure 15* shows the list of files that should be seen in the Project Window:

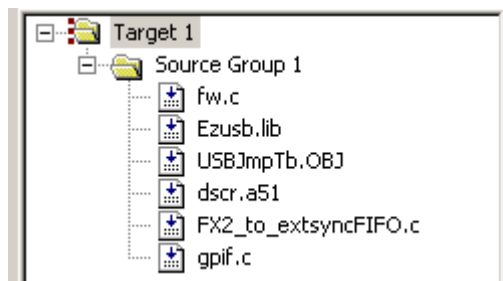


Figure 15. Files Shown in Project Window

`fw.c` – Firmware Frameworks which handles USB requests and calls the task dispatcher `TD_Poll()`

`Ezusb.lib` – collection of functions that handle suspend, resume, I²C operations, etc.

`USBJmpTb.OBJ` – interrupt vector jump table for USB (INT2) and GPIF/Slave FIFO (INT4) interrupt sources

`Dscr.a51` – device descriptor tables for the FIFO example which report EP2OUT and EP6IN as the available endpoints for the FX2 device

`FX2_to_extsyncFIFO.c` (renamed from `periph.c`) – main user application code where `TD_Poll()` and `TD_Init()` can be found. You will mainly modify this particular file and *will not need to touch fw.c*

`Gpif.c` – file which contains the GPIF waveform descriptor tables that implement the Single/FIFO GPIF transaction waveform behaviors.

TD_Init()

The first task at hand is to set up the endpoints appropriately for this example. The following code switches the CPU clock speed to 48 MHz (since at power-on default it is 12 MHz), and sets up EP2 as a Bulk OUT endpoint, 4x buffered of size 512, and EP6 as a Bulk IN endpoint, also 4x buffered of size 512. This set-up utilizes the maximum allotted 4-KB FIFO space. It also sets up the FIFOs for manual mode, word-wide operation, and goes through a FIFO reset and arming sequence to ensure that they are ready for data operations.

```
// set the CPU clock to 48 MHz
CPUCS = ((CPUCS & ~bmCLKSPD) | bmCLKSPD1);
SYNCDelay;

EP2CFG = 0xA0;           // EP2OUT, bulk, size 512, 4x buffered
SYNCDelay;
EP4CFG = 0x00;           // EP4 not valid
SYNCDelay;
EP6CFG = 0xE0;           // EP6IN, bulk, size 512, 4x buffered
SYNCDelay;
EP8CFG = 0x00;           // EP8 not valid
SYNCDelay;

EP2FIFOCFG = 0x01;       // manual mode, disable PKTEND zero length send, word ops
SYNCDelay;
EP6FIFOCFG = 0x01;       // manual mode, disable PKTEND zero length send, word ops
SYNCDelay;

FIFORESET = 0x80;        // set NAKALL bit to NAK all transfers from host
SYNCDelay;
FIFORESET = 0x02;        // reset EP2 FIFO
SYNCDelay;
FIFORESET = 0x06;        // reset EP6 FIFO
SYNCDelay;
FIFORESET = 0x00;        // clear NAKALL bit to resume normal operation
SYNCDelay;

// out endpoints do not come up armed
// since EP2OUT is quad buffered we must write dummy byte counts four times

EP2BCL = 0x80;           // arm EP2OUT by writing byte count w/skip.
SYNCDelay;
EP2BCL = 0x80;
SYNCDelay;
EP2BCL = 0x80;
SYNCDelay;
EP2BCL = 0x80;
SYNCDelay;

GpifInit(); // initialize GPIF registers
```

TD_Init then calls the function GPIFInit() which resides in gpif.c. GPIFInit() is where the loading of the GPIF waveform descriptor table into on-chip memory takes place and other GPIF registers get setup. At any one time only four waveforms can be loaded. If more than four waveforms are required to describe the operation of the physical interface, you will have to manually load another set of four waveforms (beyond the scope of this primer). An important register, IFCONFIG, also gets set up here to define how the physical interface operates. *Table 2* shows the set-up of the IFCONFIG register for this example.

Table 2. IFCONFIG Register Bit Settings for FIFO Example

Bit Position	Bit Name	Setting for Example
IFCONFIG.7	IFCLKSRC	IFCLKSRC is set to 1 to run the GPIF using the internal clock source
IFCONFIG.6	3048MHZ	3048MHZ is set to 1 to run the internal clock source for the GPIF at 48 MHz
IFCONFIG.5	IFCLKOE	IFCLKOE is set to 1 to turn on the IFCLK output to drive the WCLK and RCLK inputs of the external FIFO
IFCONFIG.4	IFCLKPOL	IFCLKPOL is set to 1 to invert the IFCLK output to the external FIFO. This allows enough set-up time for the external FIFO
IFCONFIG.3	ASYNC	The ASYNC setting is a don't care in GPIF master mode, since GPIF always references IFCLK (internal or external)
IFCONFIG.2	GSTATE	GSTATE is set to 1 to turn on the debug outputs of the state machine. PE[2:0] displays the states the GPIF engine cycles through during each transaction (Note: PE[2:0] are only available on the 100- and 128-pin packages)
IFCONFIG[1:0]	IFCFG[1:0]	IFCFG[1:0] are set to 10 to put the FX2 part into GPIF mode. Otherwise, FX2 defaults to port I/O mode

The next thing TD_Init() does is reset the external FIFO by pulsing PA2 (RS/). This ensures that the external FIFO is at a ground-zero state before commencing data operations. The following code does the trick:

```
// reset the external FIFO

OEA |= 0x04;    // turn on PA2 as output pin
IOA |= 0x04;    // pull PA2 high initially
IOA &= 0xFB;    // bring PA2 low
EZUSB_Delay (1); // keep PA2 low for ~1ms, more than enough time
IOA |= 0x04;    // bring PA2 high
```

A vendor command was also set up in the DR_VendorCmnd() function so that the host could reset the external FIFO at any time (i.e., by performing an IN vendor request of 0xB2 from the EZ-USB Control Panel).

Triggering GPIF Single Transactions

In order for the data transfers to occur across the physical interface, the CPU needs to trigger the GPIF waveforms by accessing the registers XGPIFSGLDATH, XGPIFSGLDATLX, and XGPIFSGLDATLNOX.

In order to trigger a GPIF Single Word Write transaction, you write to the XGPIFSGLDATH and XGPIFSGLDATLX in the following manner:

```
XGPIFSGLDATH = <word_value> >> 8;
XGPIFSGLDATLX = <word_value>;    // trigger GPIF
```

This effectively sets up the MSB and LSB of the word value to be transferred, and the act of writing to the XGPIFSGLDATLX register fires off the Single Word Write transaction. To make things a little easier to follow in TD_Poll(), the following function was defined which accepts a word value as an input argument and performs the GPIF Single Word Write transaction.

```
void GPIF_SingleWordWrite( WORD gdata )
{
    while( !( GPIFTRIG & 0x80 ) )    // poll GPIFTRIG.7 Done bit
    {
        ;
    }

    // using registers in XDATA space
    XGPIFSGLDATH = gdata;
    XGPIFSGLDATLX = gdata >> 8;    // trigger GPIF Single Word Write
                                    // transaction
}
```

This function also checks to see if the GPIF is in the IDLE state (GPIFTRIG.7 is set if the GPIF is IDLE) before it launches the transaction. You must always ensure the GPIF is in the IDLE state before launching any GPIF transaction. Note the access

sequence to the single transaction registers since the endpoint buffer is organized as a FIFO. This sequence ensures that the first byte in the endpoint buffer is written out FD[7:0], and the second byte is written out FD[15:8].

In order to trigger a GPIF Single Word Read transaction, the firmware performs a dummy read from the XGPIFSGLDATX register. The word value just read will be contained in the registers XGPIFSGLDATH and XGPIFSGLDATLNOX.

Again, to make things a little easier to follow in TD_Poll(), the following function was defined that accepts a word pointer for the destination variable as an input argument and performs the GPIF Single Word Read transaction:

```
void GPIF_SingleWordRead( WORD xdata *gdata )
{
    static BYTE g_data = 0x00;    // dummy variable

    while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 Done bit
    {
        ;
    }

    // using register in XDATA space
    g_data = XGPIFSGLDATLX;        // dummy read to trigger GPIF
    // Single Word Read transaction

    while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 Done bit
    {
        ;
    }

    // using register(s) in XDATA space, retrieve word just read from ext. FIFO
    *gdata = ( ( WORD )XGPIFSGLDATLNOX << 8 ) | ( WORD )XGPIFSGLDATH;
}
```

This function first checks to see if the GPIF is IDLE and then performs a dummy read from XGPIFSGLDATLX to fire off the GPIF Single Word Read transaction. Then, just as a precaution, another check is performed before accessing the registers that contain the word value.

TD_Poll()

The TD_Poll() function is where the main application code resides. The firmware here calls GPIF_SingleWordWrite() to send data from EP2OUT and calls GPIF_SingleWordRead to read data into EP6IN.

Code that Handles USB OUT Transfers

```
if(!(EP2468STAT & bmEP2EMPTY) && (EXTFIFONOTFULL))
{
    // if host sent data to EP2OUT AND external FIFO is not full,

    Tcount = (EP2BCH << 8) + EP2BCL; // load transaction count with EP2 byte count
    Tcount /= 2;                     // divide by 2 for word wide transaction
    Source = (WORD *)&EP2FIFOBUF;
    for( i = 0x0000; i < Tcount; i++ )
    {
        // transfer data from EP2OUT buffer to external FIFO
        GPIF_SingleWordWrite (*Source);
        Source++;
    }
    EP2BCL = 0x80; // re-arm EP2OUT
}
```

The first thing the OUT handling code does is check to see if the host sent data to EP2OUT, and if the external FIFO is not full by accessing the GPIFREADYSTAT register (the CPU can check the states of the RDY signals by accessing the GPIFREADYSTAT register; EXTFIFONOTFULL is a macro for GPIFREADYSTAT & bmBIT1).

If both conditions are met, the word variable *Tcount* is set up appropriately. Since each GPIF Single Word Write transaction sends an entire word to the external FIFO, the number of transactions is always half the number of bytes actually contained within the endpoint buffer.

A for loop then calls the GPIF_SingleWordWrite function and indexes through the endpoint buffer values, sending data out to the external FIFO one word at a time. The last step then is to re-arm the endpoint buffer so that the next USB data packet can be accepted.

Code that Handles USB IN Transfers

```

if(in_enable) // if IN transfers are enabled,
{
    if(!(EP2468STAT & bmEP6FULL) && (EXTFIFONOTEMPTY))
    {
        // if EP6IN is not full AND there is data in the external FIFO,

        Destination = (WORD *)(&EP6FIFOBUF);
        for( i = 0x0000; i < Tcount; i++ )
        {
            // transfer data from external FIFO to EP6IN buffer
            GPIF_SingleWordRead (Destination);
            Destination++;
        }
        Tcount *= 2;           // multiply by 2 to obtain byte count value
        EP6BCH = MSB(Tcount);
        SYNCDELAY;
        EP6BCL = LSB(Tcount); // arm EP6IN to send data to the host
        SYNCDELAY;
    }
}

```

An IN vendor command of 0xB3 is set up to enable the IN transfers to occur. The *in_enable* flag makes it possible for you to test each read and write operation independently. Otherwise, after the OUT handling code, the IN is processed immediately. By lock stepping the code, you can capture each read/write operation relatively easily on the logic analyzer (useful for debugging purposes).

If the *in_enable* flag is set, the code will fall through and check if the EP6IN endpoint buffer is not full and if the external FIFO is not empty (EXTFIFONOTEMPTY is a macro for GPIFREADYSTAT & bmBIT0).

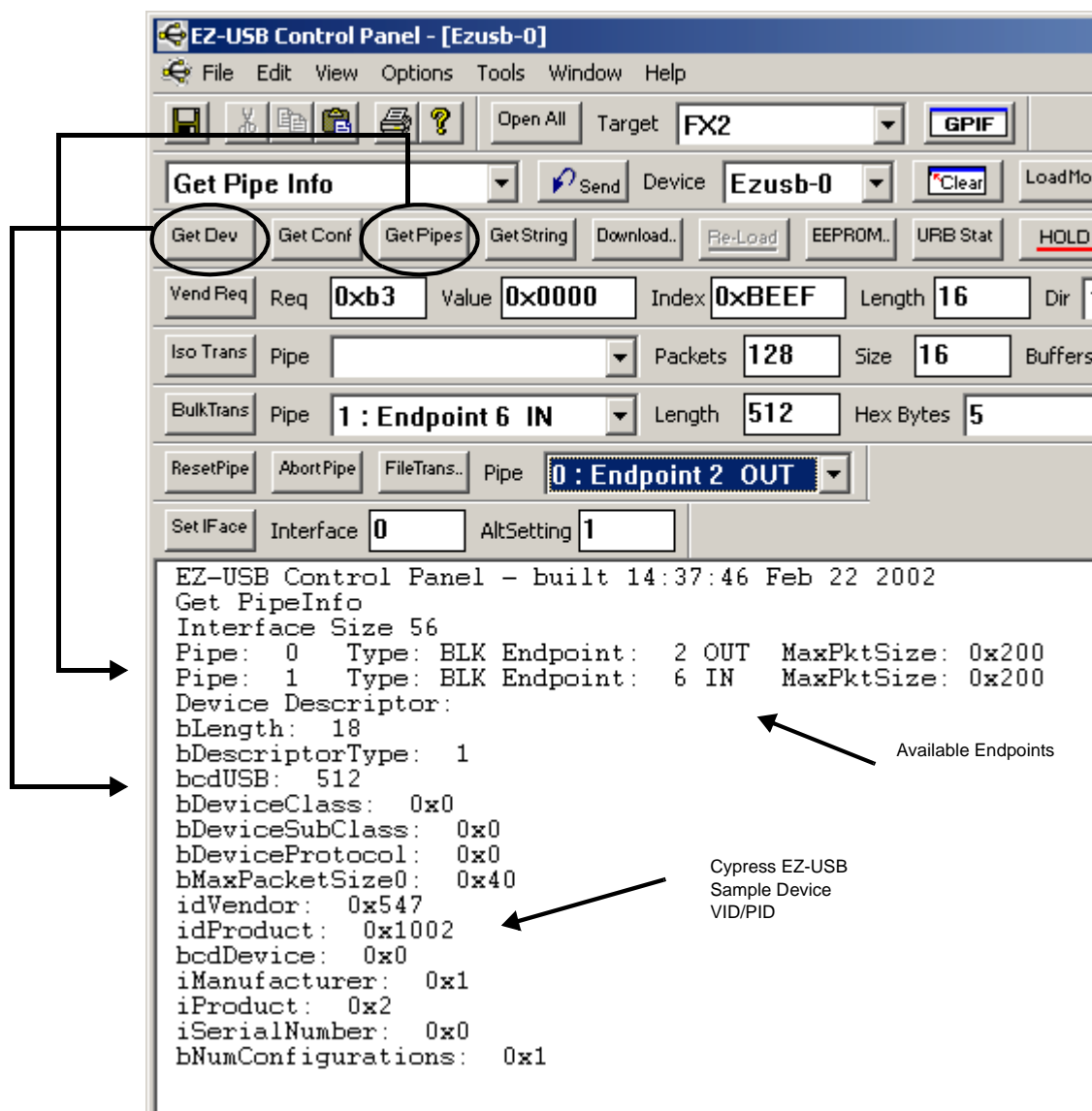
If both conditions are met, a for loop then calls the GPIF_SingleWordRead function and indexes through the endpoint buffer values, receiving data from the external FIFO one word at a time. The last step then is to arm the endpoint buffer to send the data packet to the host. Since each GPIF Single Word Read transaction receives an entire word from the external FIFO, the number of bytes to send to the host is always twice the number of transactions.

4.1.6 Running the Example for GPIF Single Transactions

Now that you understand how this single transaction example works, the bulk loopback function can be exercised by performing the steps discussed in this section.

Step 1: Download the firmware using the EZ-USB Control Panel

- Unzip the "FX2_to_extsyncFIFO GPIF Single Transactions.zip" package in the C:\Cypress\USB\Examples\FX2 directory.
- After you plug in the FX2 board, launch the EZ-USB Control Panel and ensure that the selected target is FX2.
- Then, press the "Download" button and select the FX2_to_extsyncFIFO.hex file. The FX2 board reenumerates as a Cypress EZ-USB Sample Device and LED0 should come up flashing.
- Perform a "Get Pipes" and "Get Dev" to verify one more time that the firmware is up and running. You should then see the screen shown below:



The screenshot shows the EZ-USB Control Panel - [Ezusb-0] window. The 'Get Pipes' button is circled, and an arrow points from it to the output window. The output window displays the following information:

```

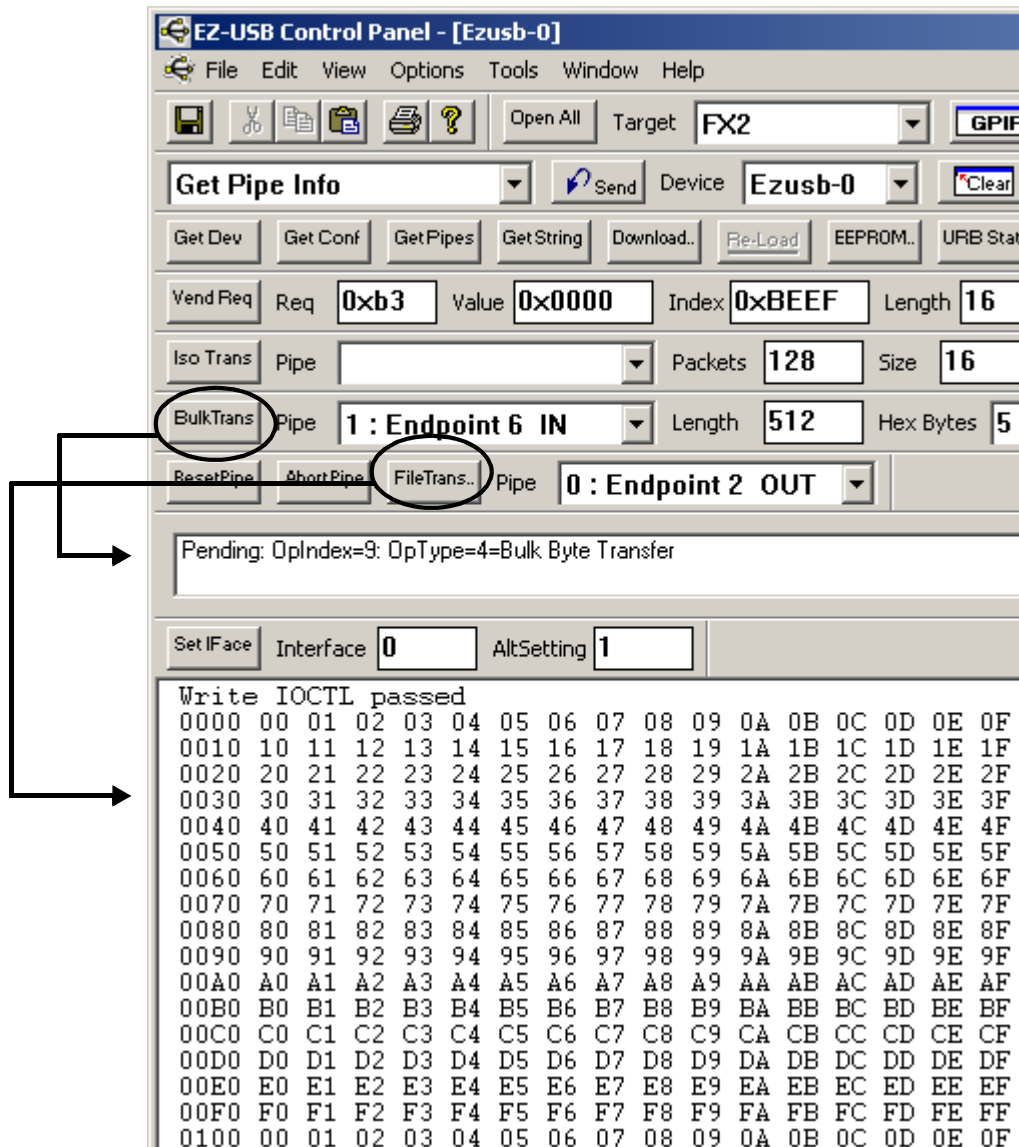
EZ-USB Control Panel - built 14:37:46 Feb 22 2002
Get PipeInfo
Interface Size 56
Pipe: 0 Type: BLK Endpoint: 2 OUT MaxPktSize: 0x200
Pipe: 1 Type: BLK Endpoint: 6 IN MaxPktSize: 0x200
Device Descriptor:
bLength: 18
bDescriptorType: 1
bcdUSB: 512
bDeviceClass: 0x0
bDeviceSubClass: 0x0
bDeviceProtocol: 0x0
bMaxPacketSize0: 0x40
idVendor: 0x547
idProduct: 0x1002
bcdDevice: 0x0
iManufacturer: 0x1
iProduct: 0x2
iSerialNumber: 0x0
bNumConfigurations: 0x1
  
```

Annotations in the image include:

- An arrow pointing from the 'Get Pipes' button to the output window.
- An arrow pointing from the 'Get Dev' button to the output window.
- An arrow pointing from the 'Get Pipes' button to the output window.
- An arrow pointing from the output window to the text "Available Endpoints".
- An arrow pointing from the output window to the text "Cypress EZ-USB Sample Device VID/PID".

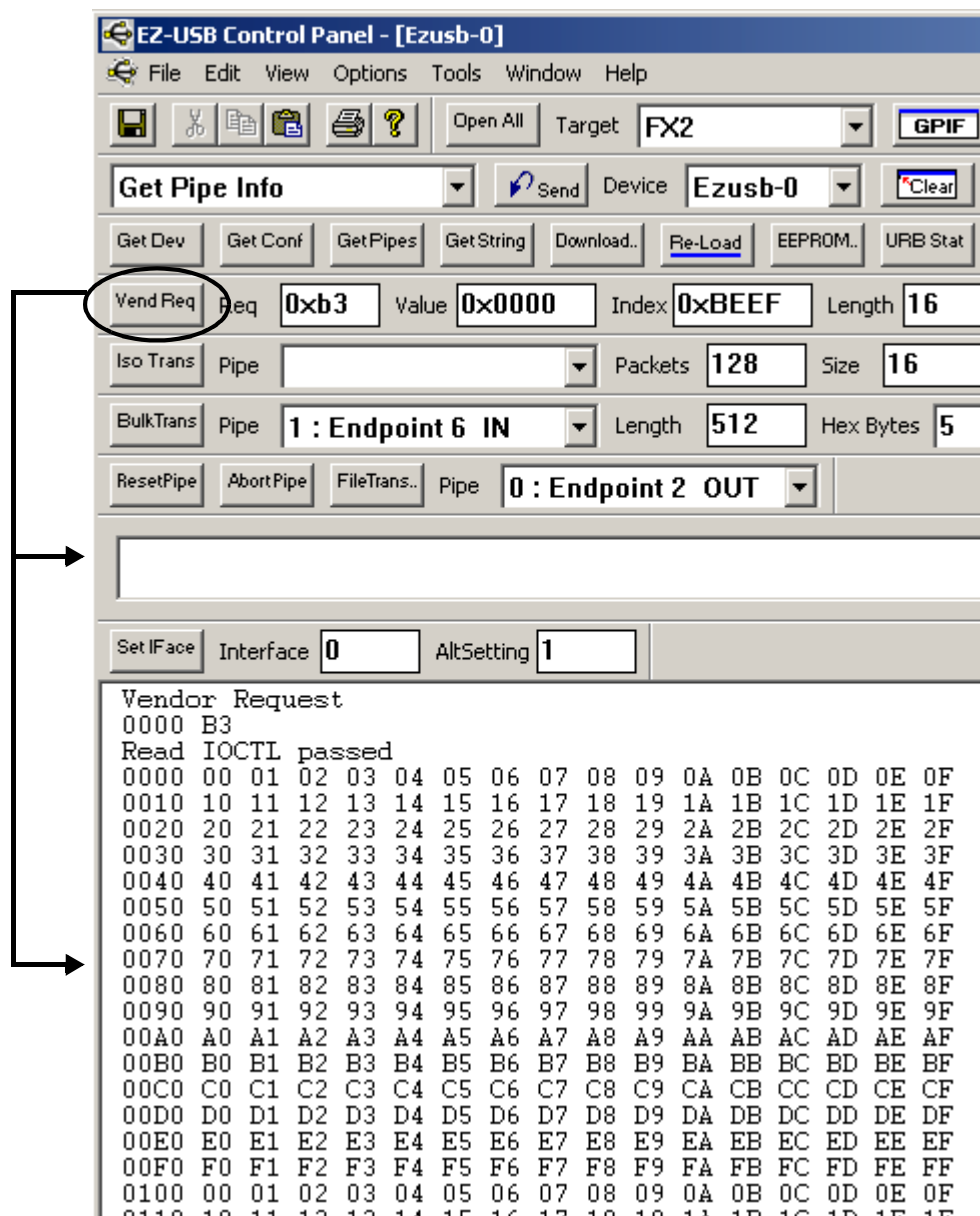
Step 2: Set up Bulk IN transfer and send 512 bytes to the external FIFO

- On the same line as the “BulkTrans” button, select Endpoint 6 IN as the “Pipe” and specify a “Length” of 512 bytes. Then click the “BulkTrans” button. This will setup a bulk IN transfer of 512 bytes to read that amount from the external FIFO. Select View -> Pending Ops to see the pending bulk IN transfer.
- On the same line as the “FileTrans..” button, select Endpoint 2 OUT as the “Pipe”. Press the “FileTrans..” button and select the 512_count.hex file. Click on “Open” and this action will send out 512 bytes out to the external FIFO (ramp test data).
- Even though 512 bytes have been written into the external FIFO, the IN transfer is not processed. This is because the *in_enable* flag in the firmware has not yet been set to TRUE.



Step 3: Complete IN transfer to read back 512 bytes from the external FIFO

In order to complete the pending IN transfer and read back 512 bytes from the external FIFO, the *in_enable* flag must be set to TRUE (remember that this allows the INs to be processed in the TD_Poll routine). To set the flag, on the same line as the “Vend Req” button, enter a value of 0xB3 in the “Req” field. Then click the “Vend Req” button. You should now see the 512 bytes read back from the external FIFO displayed in the window.



EZ-USB Control Panel - [Ezusb-0]

File Edit View Options Tools Window Help

Open All Target **FX2** **GPIF**

Get Pipe Info Send Device **Ezusb-0** Clear

Get Dev Get Conf Get Pipes Get String Download.. Re-Load EEPROM.. URB Stat

Vend Req Req **0xb3** Value **0x0000** Index **0xBEEF** Length **16**

Iso Trans Pipe Packets **128** Size **16**

BulkTrans Pipe **1 : Endpoint 6 IN** Length **512** Hex Bytes **5**

ResetPipe AbortPipe FileTrans.. Pipe **0 : Endpoint 2 OUT**

Set IFace Interface **0** AltSetting **1**

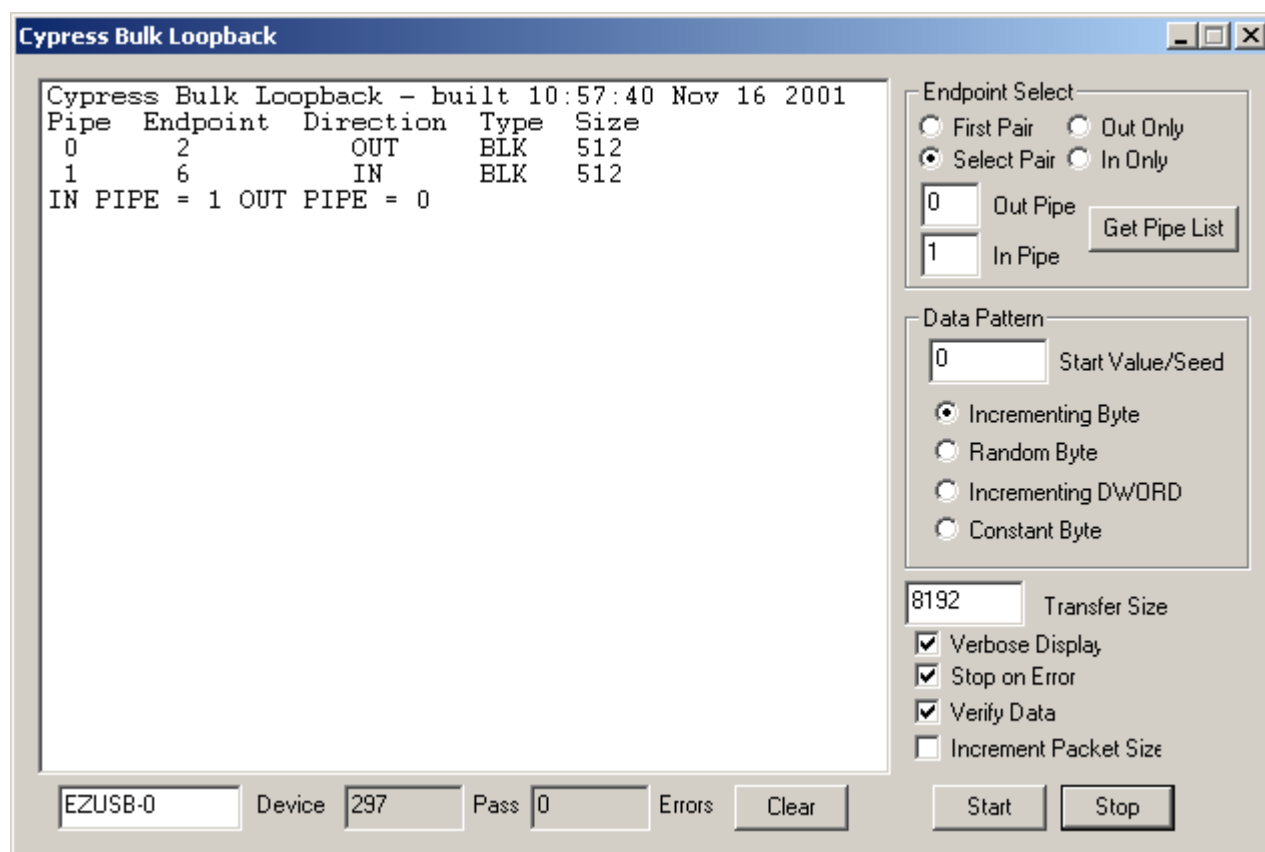
```

Vendor Request
0000 B3
Read IOCTL passed
0000 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0010 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
0020 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
0030 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
0040 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
0050 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
0060 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
0070 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
0080 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
0090 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
00A0 A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
00B0 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
00C0 C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
00D0 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
00E0 E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
00F0 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0100 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0110 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
  
```

Step 4: Exercise bulk loopback function

The bulk loopback function can be exercised by running the bulkloop.exe utility supplied with the EZ-USB development kit software. After downloading the firmware, launch the bulkloop.exe utility found in the C:\Cypress\USB\Bin subdirectory.

Set up the parameters according to the following screen:



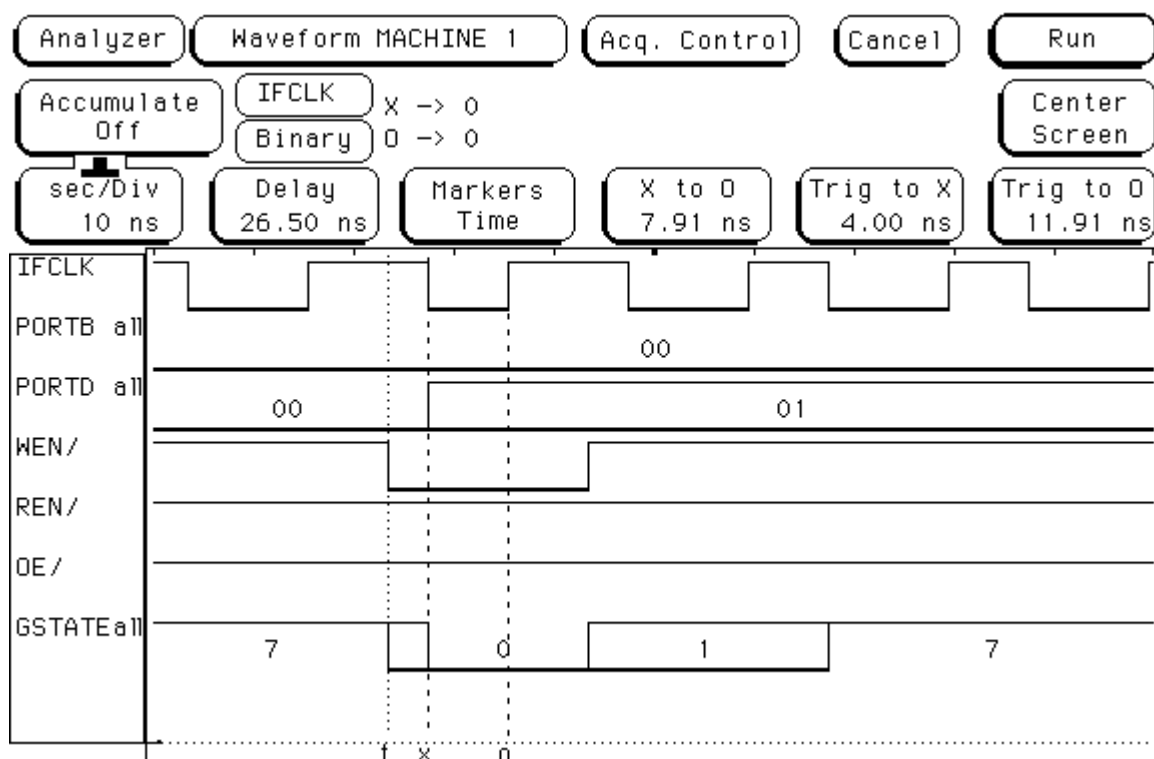
Pipe	Endpoint	Direction	Type	Size
0	2	OUT	BLK	512
1	6	IN	BLK	512

IN PIPE = 1 OUT PIPE = 0

Prior to clicking the “Start” button to commence the bulk loopback transfers, you should perform the 0xB3 vendor IN request to set the *in_enable* flag to TRUE. By clicking the “Start” button, you should see the “Pass” counter increment as each loop back transfer is exercised. Clicking on the “Stop” button will end the loopback transfers. The data values are also checked by the bulkloop utility on each pass, so you should see the “Error” count increment if any data value does not match. The application will also stop on any error if the “Stop on Error” checkbox is selected.

Debug Tip:

While running this single transaction example and at any time during GPIF development, you are strongly encouraged to connect a logic analyzer to the relevant signals on the development kit board headers. Monitoring the GPIF bus transactions aids debug sessions tremendously, and is essential for anyone seriously interested in writing GPIF firmware. The next topic presents the waveforms you should see on the logic analyzer as the single transaction example is run. An HP1660C Logic Analyzer was used to capture the waveforms.

Logic Analyzer Waveforms for Single Transaction Example
Single Write Waveform


The waveform above shows the timing generated by the GPIF engine for the Single Write waveform as defined by the GPIF Designer. All the essential signals are presented here, including GSTATE[2:0], which displays the states the GPIF engine cycles through as it performs the Single Write transaction.

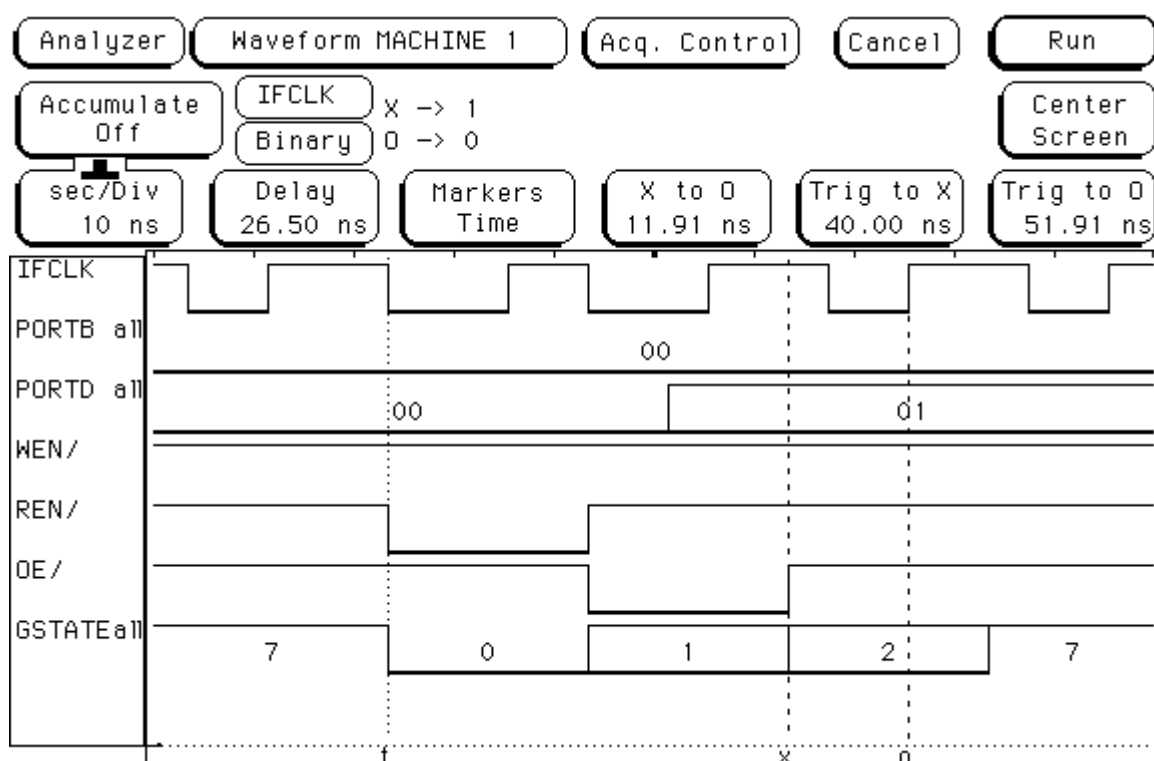
Debug Tip:

Bringing out the GSTATE signals to the logic analyzer headers allows you to correlate between the waveforms generated by the GPIF Designer and the actual waveforms generated on the physical interface. This also aids the debugging process because you can see the immediate effect of changing the waveform behavior in the GPIF Designer.

As expected from the GPIF Designer output, S0 places the data on the bus (PORTB is FD[7:0] and PORTD is FD[15:8]), and asserts CTL0 (connected to the external FIFO's WEN line). This effectively writes the 16-bit data value into the external FIFO. Note here that enough data setup time to the rising edge of IFCLK is provided, since the minimum data set-up time for the external FIFO is 4 ns (see CY7C4265 data sheet). S1 is a decision point state that unconditionally branches to the IDLE state to terminate the transaction. Without the unconditional branch, the GPIF engine would sequentially move through the remaining states until the IDLE state (S7) is reached.

For every word written out in a bulk OUT transfer, you should see the GPIF engine cycle through S0, S1, and S7. To capture the waveform, trigger the logic analyzer on the falling edge of CTL0. A sampling rate of 4 ns will give you the same resolution shown in the waveform above.

Single Read Waveform



The waveform above shows the timing generated by the GPIF engine for the Single Read waveform as defined by the GPIF Designer. All the essential signals are presented here, including GSTATE[2:0], which displays the states the GPIF engine cycles through as it performs the Single Read transaction.

As expected from the GPIF Designer output, S0 asserts CTL1 (connected to the external FIFO's $\overline{\text{REN}}$ line), S1 asserts CTL2 (connected to the external FIFO's $\overline{\text{OE}}$ line), and S2 samples the data bus (PORTB is FD[7:0] and PORTD is FD[15:8]). This effectively reads the 16-bit data value from the external FIFO. Note here that enough data set-up time to the rising edge of IFCLK is provided, since the minimum data set-up time for the GPIF is 9.2 ns (see FX2 data sheet). S2 is a decision point state that unconditionally branches to the IDLE state to terminate the transaction. Without the unconditional branch, the GPIF engine would sequentially move through the remaining states until the IDLE state (S7) is reached.

For every word read out from the external FIFO in a bulk IN transfer, you should see the GPIF engine cycle through S0, S1, S2, and S7. To capture the waveform, trigger the logic analyzer on the falling edge of CTL1. A sampling rate of 4 ns will give you the same resolution shown in the waveform above.

4.1.7 Creating FIFO Transaction GPIF Waveform Descriptors using the GPIF Designer

A fully working external FIFO example using GPIF Single transactions has already been discussed, but the bandwidth achieved is minuscule. This is because there is a lot of firmware overhead involved in launching GPIF Single transactions. With GPIF FIFO transactions, the GPIF engine directly handles bursts of data, so a higher bandwidth over the physical interface is achievable.

Introducing the Flow State Feature of the GPIF

In order to efficiently handle bursts of data and meet burst access timing to the external FIFO, the flow state feature of the GPIF was utilized for the FIFO transaction example. The flow state feature makes its debut in the FX2 GPIF and is a mechanism that allows the GPIF to efficiently throttle data on and off the bus by using an independent set of RDYn logic (flow logic) that is separate from the decision point RDYn logic. Since the flow state feature is an advanced mode of the GPIF, not every application will need to use the flow state. However, handling bursts of data to and from an external FIFO shows the simplest application of the flow state. One very advanced application of the flow state is in the generation of UDMA waveforms for the FX2 mass storage reference design firmware.

In any GPIF waveform, there can only be one flow state, but it can be any of the available non-idle states (S0–S6). The flow state behavior is controlled by a set of registers that are specific to the flow state feature (see the FX2 Technical Reference Manual for flow state register details). One can think of the flow state as being “orthogonal” to one of the GPIF waveform’s states, but it is still the regular decision point logic that is responsible for determining when the flow state should be exited and the normal GPIF waveform behavior continues.

Another property of the flow state is that it can be programmed to perform a different set of CTLx logic than what is described in the GPIF waveform descriptors themselves. This brings the level of autonomy to another notch. The idea behind the GPIF FIFO Read and Write descriptor programming is to have the read and write control lines assert for the duration of the transaction, thereby allowing data to be moved on every edge of IFCLK. Therefore, a 16-bit interface running at 48 MHz would yield an effective burst data rate of 96 MB/s over the GPIF interface.

The main difference between this FIFO transaction version and the single transaction version is that waveforms 2 and 3 are used (FIFORd and FIFOWr waveforms, respectively) instead of waveforms 0 and 1. RDY5 is used as the GPIF transaction count (GPIF TC) internal expiration flag (TCXpire). The GPIF TC is what is used in the waveform’s decision point logic to determine when to exit out of the flow state and terminate the waveform.

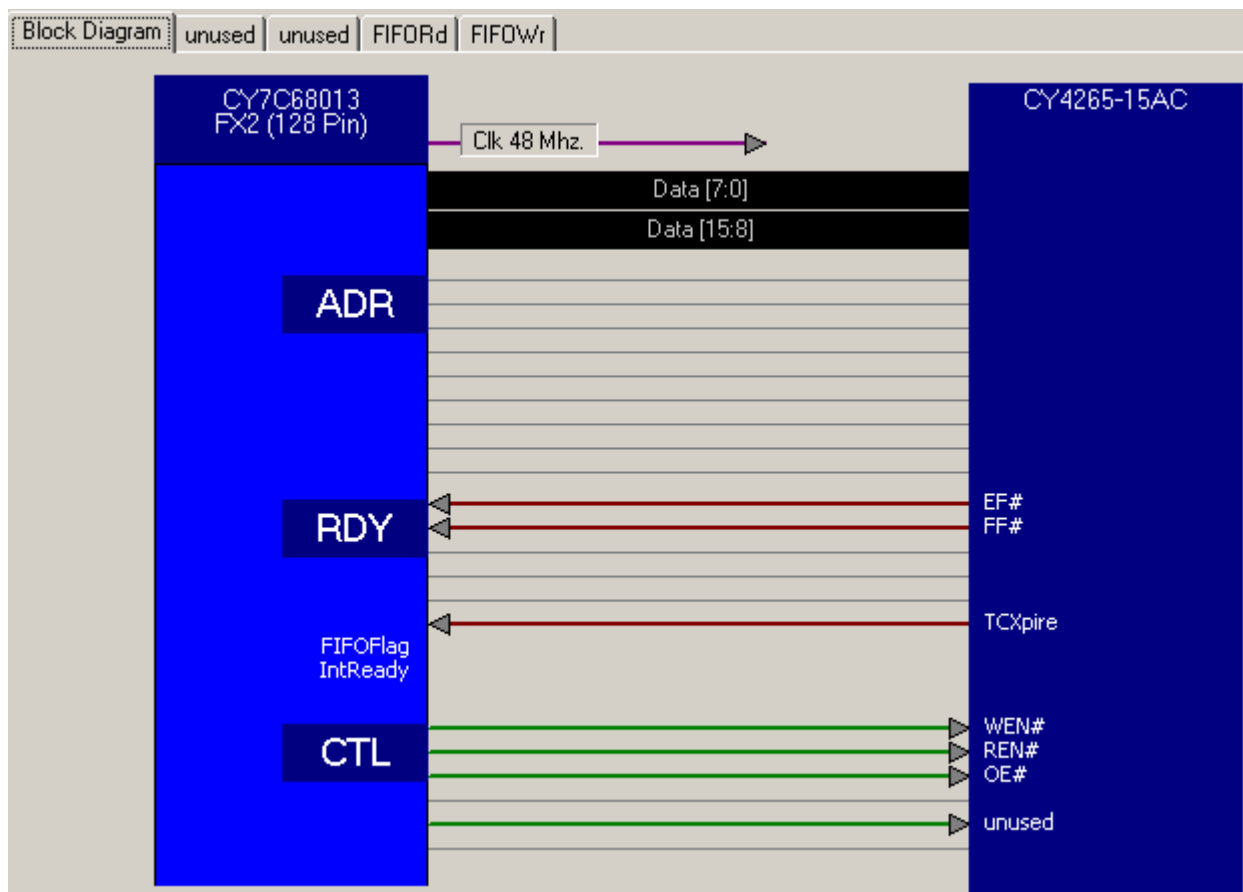


Figure 16. GPIF Designer Block Diagram View

Figure 16 shows the set-up of the block diagram and the naming conventions of the CTLx and RDYn signals (same as the single transaction example). Figure 17 below shows waveform 3, which characterizes the behavior of the FIFO Write waveform.

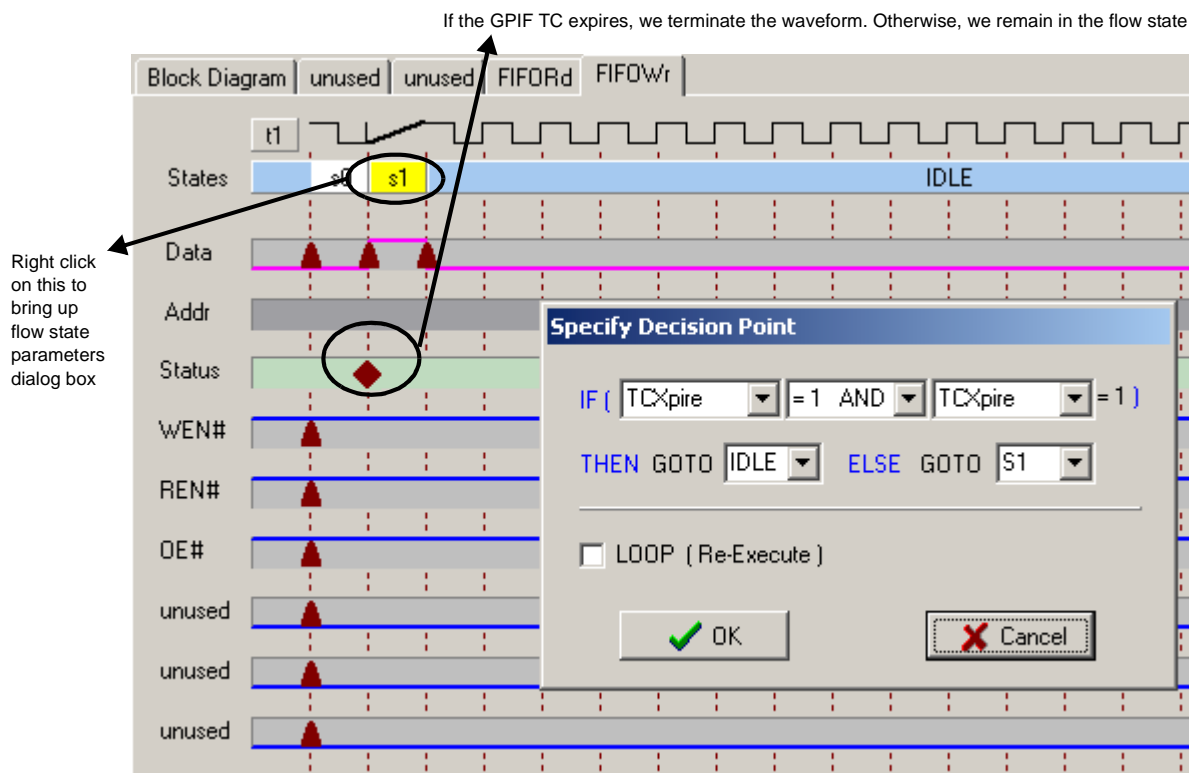


Figure 17. FIFO Write Waveform in GPIF Designer

In this FIFO Write waveform (waveform 3) we see that S0 is a period of inactivity, followed by S1 which is designated as the flow state. The decision point logic in S1 looks at the GPIF TC to determine when to terminate the waveform by branching to the IDLE state. As previously mentioned, the flow logic in S1 then takes over to throttle data on and off the bus and manipulate the CTLx lines. The flow state registers are set up by selecting the various flow state parameters, accessed by right clicking on the S1 state trace.

In order to set up the flow state for both FIFO reads and writes, a set of global GPIF and flow state registers are first initialized. The values are taken from a FlowStates[36] array in gpif.c, generated by GPIF Designer.

```
EP2GPIFFLGSEL = 0x01; // For EP2OUT, GPIF uses EF flag
SYNCDELAY;
EP6GPIFFLGSEL = 0x02; // For EP6IN, GPIF uses FF flag
SYNCDELAY;

// global flowstate register initializations

FLOWLOGIC = FlowStates[19]; // 0011 0110b - LFUNC[1:0] = 00 (A AND B), TERMA/B[2:0]=110 (FIFO Flag)
SYNCDELAY;
FLOWSTB = FlowStates[22]; // 0000 0100b - MSTB[2:0] = 100 (CTL4), not used as strobe
SYNCDELAY;
GPIFHOLDAMOUNT = FlowStates[26]; // hold data for one half clock (10ns) assuming 48MHz IFCLK
SYNCDELAY;
FLOWSTBEDGE = FlowStates[24]; // move data on both edges of clock
SYNCDELAY;
FLOWSTBHPERIOD = FlowStates[25]; // 20.83ns half period
SYNCDELAY;
```

The set-up is such that when FIFO Write transactions are launched from EP2OUT, the GPIF uses EP2's empty flag (EF) as the FIFO Flag, and when FIFO Read transactions are launched into EP6IN, the GPIF uses EP6's full flag (FF) as the FIFO Flag. Subsequently, the flow logic is set up to use the FIFO Flag to throttle data on and off the bus, so the flow state mechanism actually uses EP2EF and EP6FF status to know when to keep writing to the data bus or keep reading from the data bus, respectively.

Although CTL4 (unused) is not used in the application, we take advantage of the fact that the flow state can use any of the CTLx lines as a data strobe. At a 48-MHz IFCLK, CTL4 is toggled at a half period of 20.83 ns. Since the flow state is also programmed to move data on both edges of the data strobe, this allows us to nicely align the data values with the rising edge of IFCLK and achieve a 96-MB/s burst rate over the physical interface. Note that although CTL4 is not physically exposed on the 56-pin package, the flow state logic can still be set up to use it as a data strobe.

Let's also examine the flow state register set-up that is specific to FIFO Writes:

```
void Setup_FLOWSTATE_Write ( void )
{
    FLOWSTATE = FlowStates[18]; // 1000 0001b - FSE=1, FS[2:0]=001
    SYNCDELAY;
    FLOWEQ0CTL = FlowStates[20]; // CTL0 = 0 when flow condition equals zero (data flows)
    SYNCDELAY;
    FLOWEQ1CTL = FlowStates[21]; // CTL0 = 1 when flow condition equals one (data does not flow)
    SYNCDELAY;
}
```

Here we designate S1 to be the flow state and define the state of CTL0 when the flow condition equals zero (data flows) and when the flow condition equals one (data does not flow). Remember that the state of the flow condition is determined by the state of EP2EF. So when the EP2 FIFO contains data (EP2 is not empty) the flow condition equals zero, the flow state drops CTL0 LOW (WEN# is asserted), and data is placed on FD[15:0].

Figure 18 below shows waveform 2, which characterizes the behavior of the FIFO Read waveform.

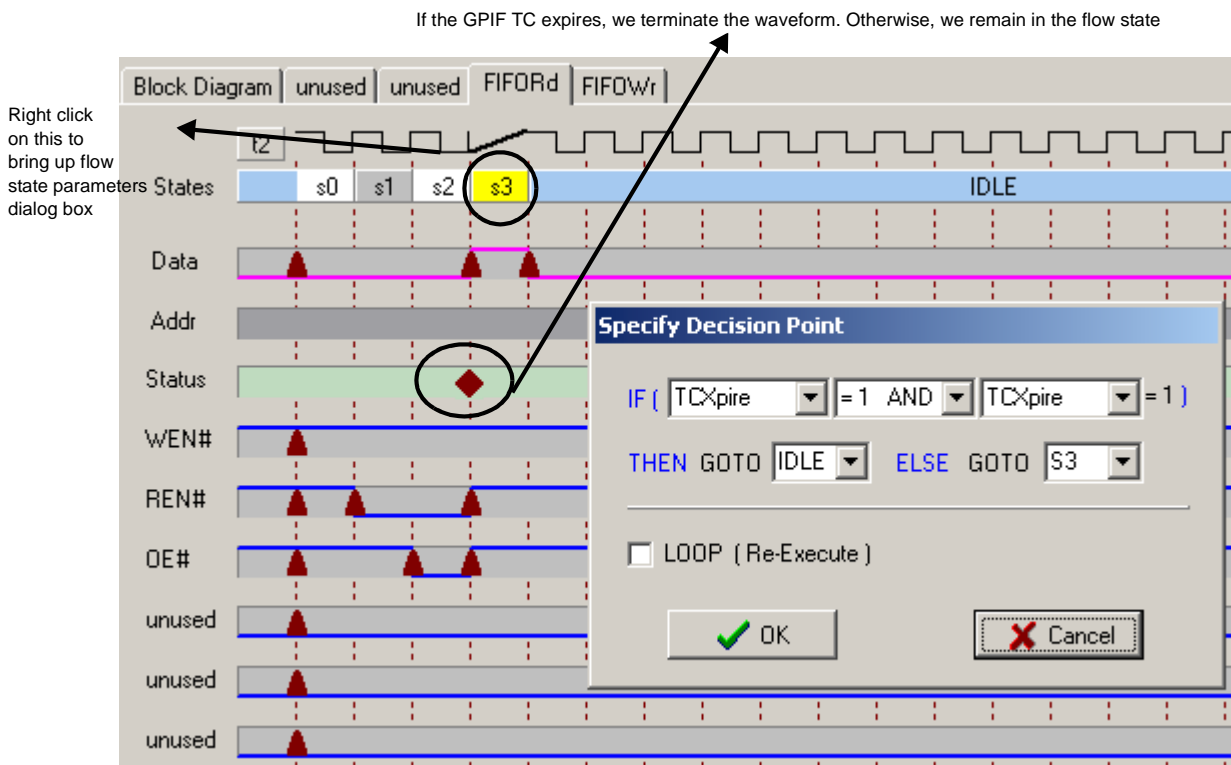


Figure 18. FIFO Read Waveform in GPIF Designer

In this FIFO Read waveform (waveform 2) S0 is a period of inactivity, then S1 and S2 sets up the "front porch" of the burst transfer, followed by S3 which is designated as the flow state. The decision point logic in S3 looks at the GPIF TC to determine when to

terminate the waveform by branching to the IDLE state. As previously mentioned, the flow logic in S3 then takes over to throttle data reads from the bus and manipulate the CTLx lines.

Let's examine the flow state register set-up that is specific to FIFO Reads:

```
void Setup_FLOWSTATE_Read ( void )
{
    FLOWSTATE = FlowStates[27]; // 1000 0011b - FSE=1, FS[2:0]=003
    SYNCDELAY;
    FLOWEQ0CTL = FlowStates[29]; // CTL1/CTL2 = 0 when flow condition equals zero (data flows)
    SYNCDELAY;
    FLOWEQ1CTL = FlowStates[30]; // CTL1/CTL2 = 1 when flow condition equals one (data does not flow)
    SYNCDELAY;
}
```

Here we designate S3 to be the flow state and define the state of CTL1 and CTL2 when the flow condition equals zero (data flows) and when the flow condition equals one (data does not flow). Remember that the state of the flow condition is determined by the state of EP6FF. So when the EP6 FIFO has room for data (EP6 is not full) the flow condition equals zero, the flow state drops CTL1 and CTL2 LOW (REN and OE are asserted), and data is read from FD[15:0].

Since there is a different flow state register set-up for FIFO read and write operations, the firmware has to call Setup_FLOWSTATE_Read() before launching a GPIF FIFO read transaction, and call Setup_FLOWSTATE_Write() before launching a GPIF FIFO write transaction.

Now that you understand how the GPIF FIFO read and write waveforms were programmed and set up, the firmware programming for GPIF FIFO transactions can be discussed.

4.1.8 Firmware Programming for GPIF FIFO Transactions

In moving from GPIF Single transactions to GPIF FIFO transactions, the only major difference really lies in the `TD_Poll()` code. The basic underlying architecture of the example remains the same. In this section, the basic principles of launching a FIFO transaction are introduced. Following that is a discussion of the `TD_Poll()` code that triggers the GPIF FIFO transactions.

Triggering GPIF FIFO Transactions

For triggering GPIF FIFO transactions, we reiterate the concept of the GPIF transaction count (or TC for short). Analogous to the *Tcount* variable in the single transaction example, the TC is a value the GPIF engine uses to determine how many times to go through a FIFO waveform. Another mechanism you can use to tell the GPIF when to stop the waveform is the FIFO Flag. However, the simplest and most often used method is the TC method, which is presented here.

For example, if you wished to burst out 512 bytes of data from the EP2OUT endpoint, the TC value would be set to 512 (for byte-wide operation) or 256 (for word-wide operation). The GPIF engine then decrements the TC value on every push or pop of the FIFO. When the TC value reaches zero, the waveform is complete (a waveform completion is signified by the GPIFDONE bit being set in the GPIFTRIG register). A decision point state can use the TC value as an internal flag to determine whether or not to branch to the IDLE state. GPIFREADYCFG.5 must be set to allow the GPIF engine to use the RDY5 signal as an internal TC expiration flag.

The act of triggering a GPIF FIFO transaction is actually very simple. Writing to the R/W bit in the GPIFTRIG register sets the direction of the transaction. If $R/W = 1$, a FIFO Read transaction gets triggered when the GPIFTRIG register is accessed. If $R/W = 0$, a FIFO Write transaction gets triggered instead.

For example, to trigger a GPIF FIFO Read transaction to EP6IN use the following line of code:

```
GPIFTRIG = GPIFTRIGRD | GPIF_EP6; // launch GPIF FIFO Read
                                     // transaction to EP6IN
```

To trigger a GPIF FIFO Write transaction from EP2OUT use the following line of code:

```
GPIFTRIG = GPIF_EP2; // launch GPIF FIFO Write transaction from
                       // EP2OUT
```

GPIFTRIGRD, GPIF_EP6, and GPIF_EP2 are bit masks to set the appropriate bits in the GPIFTRIG register. By setting the EP[1:0] bits in the GPIFTRIG register to valid options of 0, 1, 2, or 3 (in order of the endpoints 2, 4, 6, and 8), this specifies which endpoint should be used in the transaction. Source or sink direction is implied by whether the endpoint is an IN or an OUT endpoint.

TD_Init()

```
// set the CPU clock to 48 MHz
CPUCS = ((CPUCS & ~bmCLKSPD) | bmCLKSPD1);
SYNCDelay;

EP2CFG = 0xA0;    // EP2OUT, bulk, size 512, 4x buffered
SYNCDelay;
EP4CFG = 0x00;    // EP4 not valid
SYNCDelay;
EP6CFG = 0xE0;    // EP6IN, bulk, size 512, 4x buffered
SYNCDelay;
EP8CFG = 0x00;    // EP8 not valid

FIFORESET = 0x80; // set NAKALL bit to NAK all transfers from host
SYNCDelay;
FIFORESET = 0x02; // reset EP2 FIFO
SYNCDelay;
FIFORESET = 0x06; // reset EP6 FIFO
SYNCDelay;
FIFORESET = 0x00; // clear NAKALL bit to resume normal operation
SYNCDelay;

EP2FIFOCFG = 0x01; // allow core to see zero to one transition of auto out bit
SYNCDelay;
EP2FIFOCFG = 0x11; // auto out mode, disable PKTEND zero length send, word ops
SYNCDelay;
EP6FIFOCFG = 0x09; // auto in mode, disable PKTEND zero length send, word ops
SYNCDelay;

GpifInit();        // initialize GPIF registers

EP2GPIFFLGSEL = 0x01; // For EP2OUT, GPIF uses EF flag
SYNCDelay;
EP6GPIFFLGSEL = 0x02; // For EP6IN, GPIF uses FF flag
SYNCDelay;

// global flowstate register initializations

FLOWLOGIC = FlowStates[19];    // 0011 0110b - LFUNC[1:0] = 00 (A AND B), TERMA/B[2:0]=110 (FIFO Flag)
SYNCDelay;
FLOWSTB = FlowStates[22];    // 0000 0100b - MSTB[2:0] = 100 (CTL4), not used as strobe
SYNCDelay;
GPIFHOLDAMOUNT = FlowStates[26]; // hold data for one half clock (10ns) assuming 48MHz IFCLK
SYNCDelay;
FLOWSTBEDGE = FlowStates[24];    // move data on both edges of clock
SYNCDelay;
FLOWSTBHPERIOD = FlowStates[25]; // 20.83ns half period
SYNCDelay;

// reset the external FIFO

OEA |= 0x04;    // turn on PA2 as output pin
IOA |= 0x04;    // pull PA2 high initially
IOA &= 0xFB;    // bring PA2 low
EZUSB_Delay(1); // keep PA2 low for ~1ms, more than enough time
IOA |= 0x04;    // bring PA2 high
```

The initialization code in *TD_Init()* remains pretty much the same as the single transaction version. The main differences lie in the setup of the *EPxFIFOCFG* and flow state registers. To maximize the USB 2.0 bandwidth, the endpoints are placed into auto mode (*AUTOOUT/AUTOIN* = 1). Note that bits 1 and 0 of the *REVCTL* register are not set. Therefore, it is necessary to first set *AUTOOUT* = 0, then set *AUTOOUT* = 1. The FX2 needs to see a 0 to 1 transition of the *AUTOOUT* bit to automatically arm the endpoint buffers.

TD_Poll()

Code that handles USB OUT Transfers

```

if( GPIFTRIG & 0x80 )           // if GPIF interface IDLE
{
    if ( ! ( EP24FIFOFLGS & 0x02 ) ) // if there's a packet in the peripheral domain for EP2
    {
        if ( EXTIFONOTFULL )       // if the external FIFO is not full
        {
            if(enum_high_speed)
            {
                SYNCDELAY;
                GPIFTCB1 = 0x01;      // setup transaction count (512 bytes/2 for word wide -> 0x0100)
                SYNCDELAY;
                GPIFTCB0 = 0x00;
                SYNCDELAY;
            }
            else
            {
                SYNCDELAY;
                GPIFTCB1 = 0x00;      // setup transaction count (64 bytes/2 for word wide -> 0x20)
                SYNCDELAY;
                GPIFTCB0 = 0x20;
                SYNCDELAY;
            }
        }

        Setup_FLOWSTATE_Write(); // setup FLOWSTATE registers for FIFO Write operation
        SYNCDELAY;
        GPIFTRIG = GPIF_EP2;      // launch GPIF FIFO WRITE Transaction from EP2 FIFO
        SYNCDELAY;

        while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 GPIF Done bit
        {
            ;
        }
        SYNCDELAY;
    }
}
}

```

The first thing the OUT handling code does is it checks to see if the GPIF is IDLE. If so, it checks to see if there is at least one packet in the peripheral domain for EP2. Since EP2 is placed into auto mode, the firmware does not need to check if the host sent a USB packet. The USB packets are automatically committed to be used by the GPIF engine. Therefore, the firmware's job is to check if at least one packet has been committed to the peripheral domain.

Then, if the external FIFO is not full, the TC value is set up for word-wide operation. The TC value is a 32-bit register field, but for this application only the lower 16-bit fields are necessary. Since each GPIF FIFO Write transaction sends 512 bytes to the external FIFO over a 16-bit interface, the number of transactions is always half the number of bytes actually contained within the endpoint buffer. The appropriate TC value is set up for either high-speed (256) or full-speed (32) operation.

The appropriate flow state registers are then set up for the FIFO Write transaction, and a write to the GPIFTRIG register with the appropriate bits triggers the transaction from EP2OUT. The code then waits for the transaction to complete before exiting out of the "if" nest.

Code that handles USB IN Transfers

```

if(in_enable)                // if IN transfers are enabled
{
    if ( GPIFTRIG & 0x80 )    // if GPIF interface IDLE
    {
        if ( EXTIFIFONOTEMPTY ) // if external FIFO is not empty
        {
            if ( !( EP68FIFOFLGS & 0x01 ) ) // if EP6 FIFO is not full
            {
                if(enum_high_speed)
                {
                    SYNCDELAY;
                    GPIFTCB1 = 0x01;        // setup transaction count (512 bytes/2 for word wide -> 0x0100)
                    SYNCDELAY;
                    GPIFTCB0 = 0x00;
                    SYNCDELAY;
                }
                else
                {
                    SYNCDELAY;
                    GPIFTCB1 = 0x00;        // setup transaction count (64 bytes/2 for word wide -> 0x20)
                    SYNCDELAY;
                    GPIFTCB0 = 0x20;
                    SYNCDELAY;
                }
            }

            Setup_FLOWSTATE_Read(); // setup FLOWSTATE registers for FIFO Read operation
            SYNCDELAY;
            GPIFTRIG = GPIFTRIGRD | GPIF_EP6; // launch GPIF FIFO READ Transaction to EP6 FIFO
            SYNCDELAY;

            while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 GPIF Done bit
            {
                ;
            }

            SYNCDELAY;
        }
    }
}
}
}

```

Just as in the single transaction firmware, if the *in_enable* flag is not set, the code will just sit there and not process the INs.

If the *in_enable* flag is set, the code will fall through and check if the GPIF interface is IDLE. It then goes on to check if the external FIFO is not empty. If the external FIFO has data, the code then determines if EP6 is not full. We can interpret the not full condition to mean that there is room for at least one more max packet sized data packet, since we are moving max packet sized data packets every time.

If EP6 is not full, the TC value is set up for word-wide operation. The appropriate TC value is set up for either high-speed (256) or full-speed (32) operation. The flow state registers are then set up for the FIFO Read transaction, and a write to the GPIFTRIG register with the appropriate bits triggers the transaction to fill the EP6 FIFO. The code then waits for the transaction to complete. Since EP6 is placed into auto mode, there is no need to explicitly write a byte count value to indicate how many bytes to send to the host. FX2 uses the EP6AUTOINLENH/L register values set at enumeration time in the *DR_SetConfiguration()* function for the auto commit size.

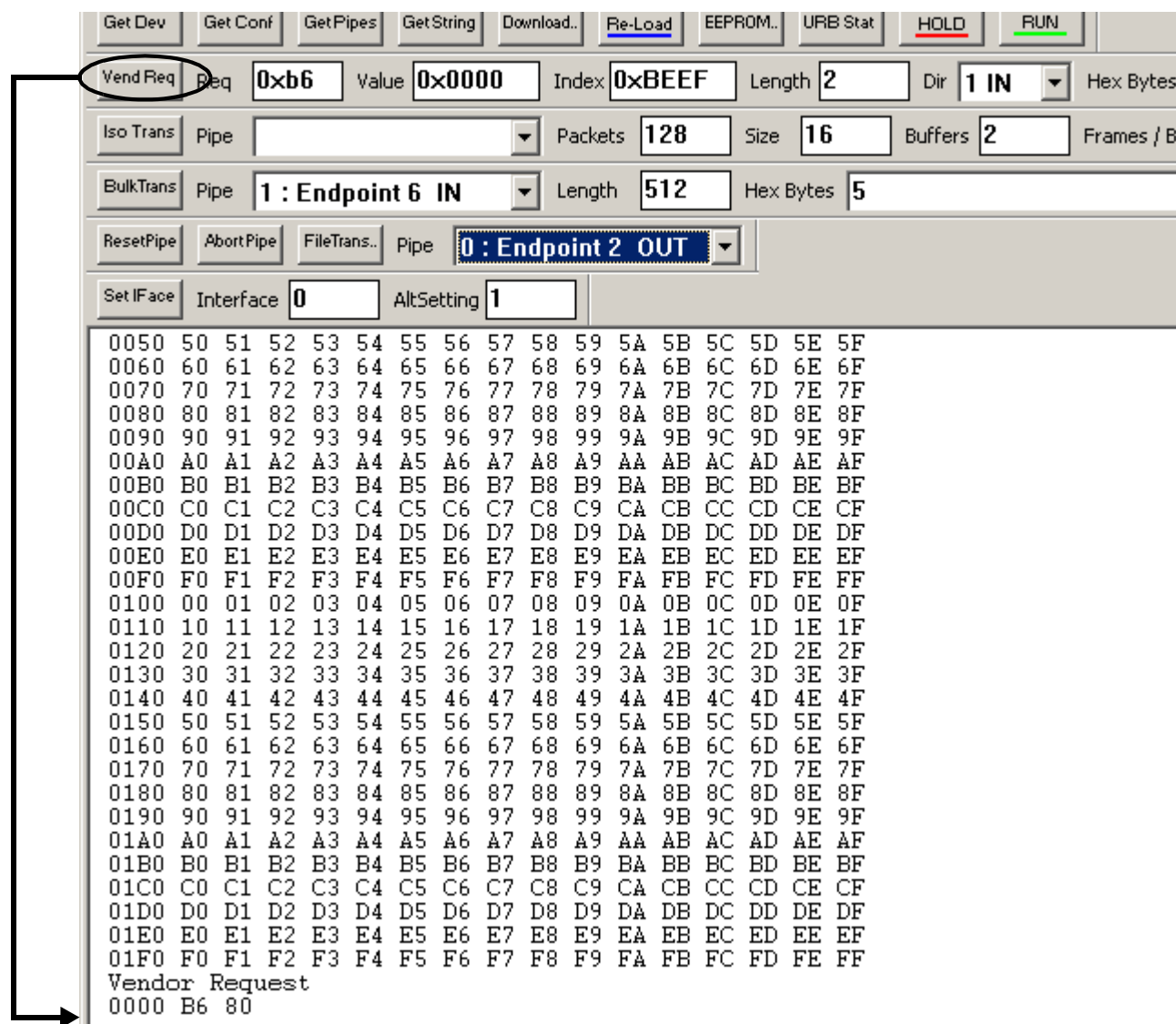
4.1.9 Running the Example for GPIF FIFO Transactions

The procedure for running the FIFO transaction example is essentially the same as the single transaction example. Going through steps 1 through 4 of section 4.1.6 will allow you to run the FIFO transaction example as well. For running this version of the example, unzip the “FX2_to_extsyncFIFO GPIF FIFO Transactions Auto mode.zip” package instead.

A couple of differences to note are that LED0 will no longer flash when the code is downloaded, and that a few more vendor commands were added for debug purposes. The LED0 code was taken out of TD_Poll() to optimize the firmware execution for FIFO transactions.

Debug Tip:

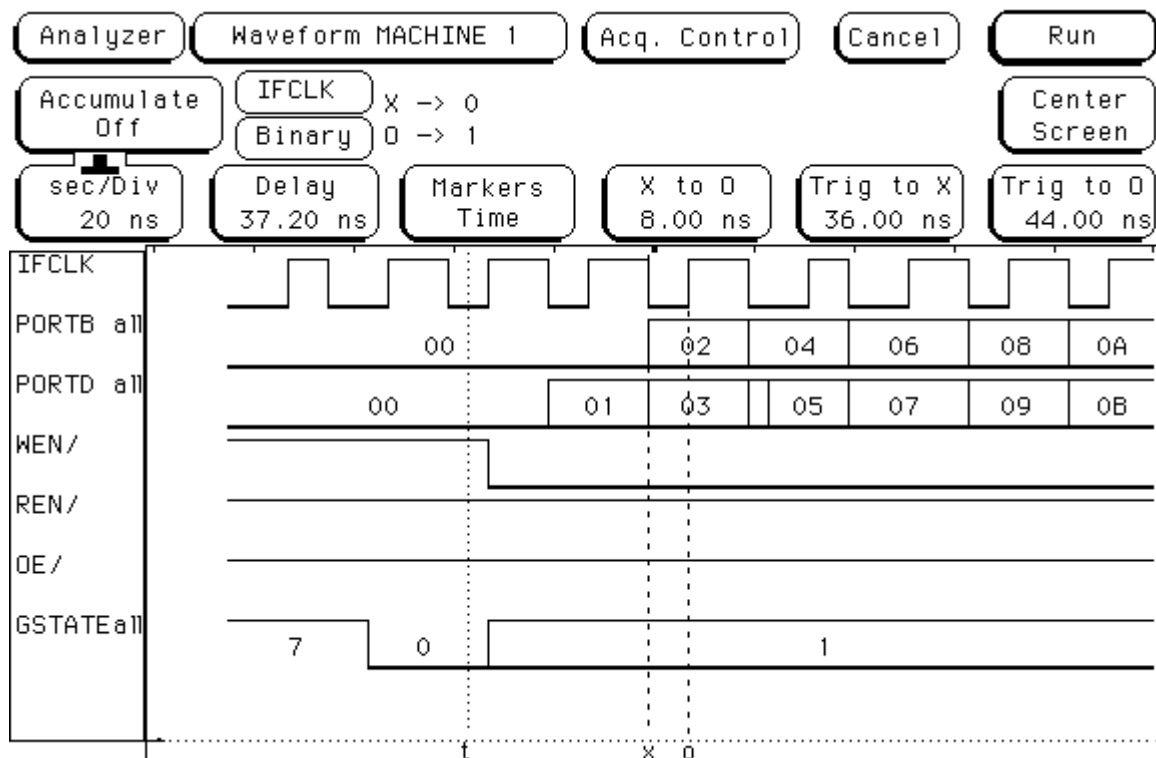
The use of vendor commands is a “cheap” way to add more debug functionality to the code without incurring unnecessary “printf” statements. With the use of vendor commands, the Keil debugger is not necessary for determining the state of the GPIF engine after a firmware event. For example, the vendor command 0xB6 was added to the FIFO transaction firmware to read back the status of the GPIF engine. The vendor command returns the 0xB6 IN request with the value of the GPIFTRIG register. If the GPIF engine has completed a FIFO read or write transaction, the GPIFDONE bit is set, returning a value of 0x80. The screenshot below shows what you should see in the EZ-USB Control Panel window.



Logic Analyzer Waveforms for FIFO Transaction Example

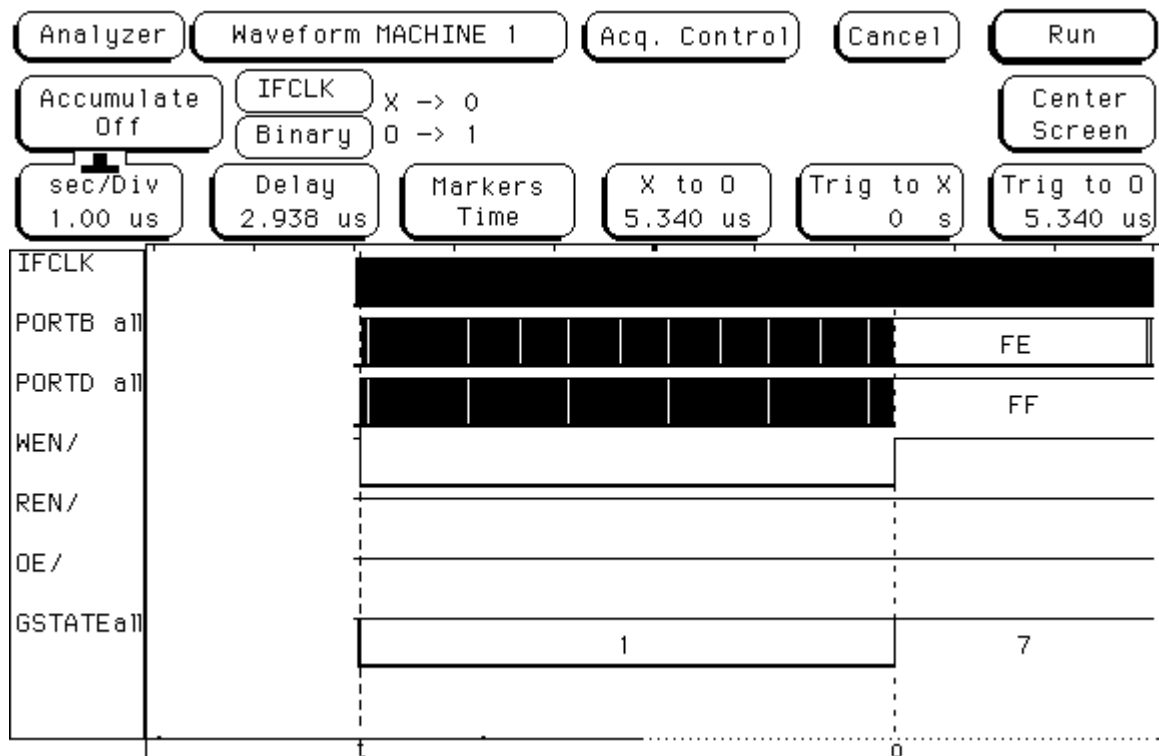
This section presents the waveforms you should see on the logic analyzer as the FIFO transaction example is run. Again, an HP1660C Logic Analyzer was used to capture the waveforms.

FIFO Write Waveform: Close-up view of the “front porch”



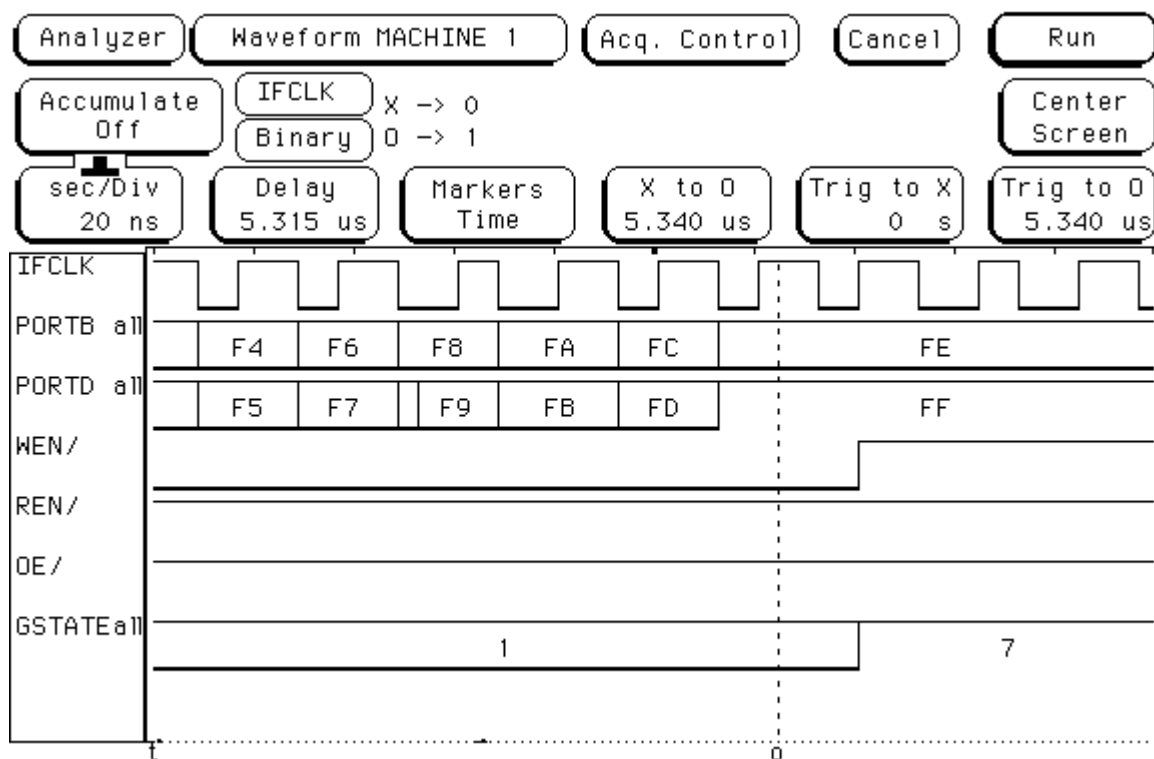
This trace shows that the 4-ns data set-up time for the external FIFO is satisfied using the X to 0 marker as an indicator. The word consisting of data values 0x02 and 0x03 is written into the external FIFO on the rising edge of IFCLK (the external FIFO's WCLK). While WEN/ is held LOW, consecutive words are written into the external FIFO on every rising edge of IFCLK. Notice that the GSTATE bus reflects the state of the GPIF engine as it's progressing through the GPIF FIFO Write waveform. S0 is a period of inactivity for 1 IFCLK cycle (20.83 ns), and S1 is the flow state and is active for the entire duration of the data burst phase.

FIFO Write Waveform: Time taken to transfer 512 bytes to the external FIFO



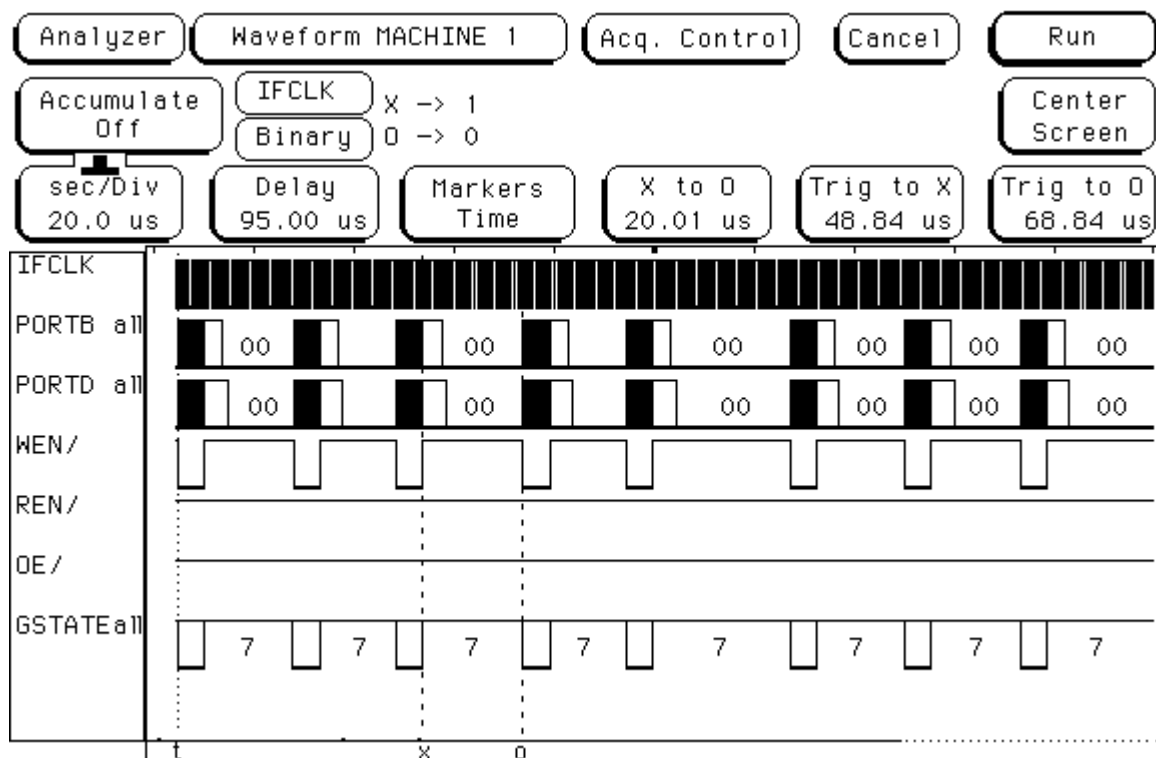
This trace shows how long it takes to write a burst of 512 bytes (256 words) into the external FIFO. At a burst rate of 96 MB/s (one word every IFCLK period), this results in a time of approximately 5.3 microseconds to transfer a payload of 512 bytes. This zoomed out view allows us to see that indeed the GPIF FIFO Write waveform remains in the flowstate until it is done transferring 512 bytes, at which point it then transitions to the IDLE state (S7).

FIFO Write Waveform: Close-up view of the “back porch”



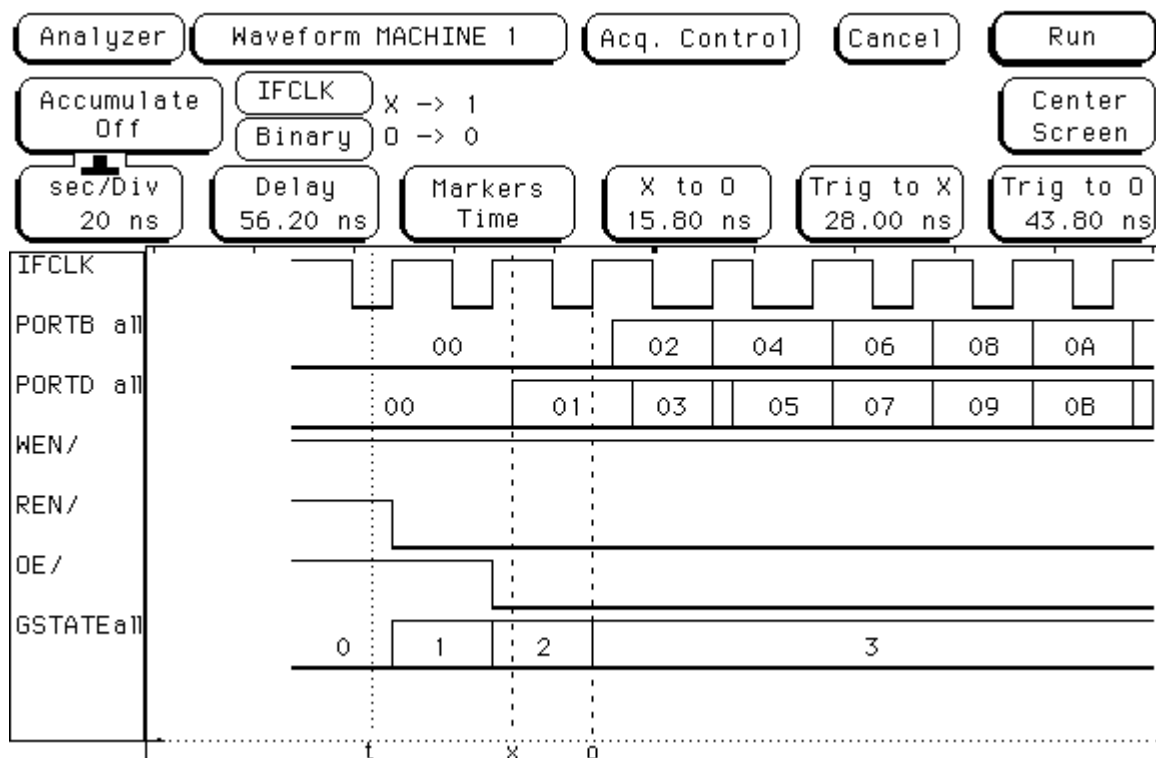
Here we see the back end of the 512-byte transfer at a zoomed in level. The last word in the packet consists of data values 0xFE and 0xFF (the end of our ramp test data). Note that a repeated word at the end is not clocked in as the set-up time for the WEN/ line is not met prior to the IFCLK edge.

FIFO Write Waveform: Inter-packet transfer time



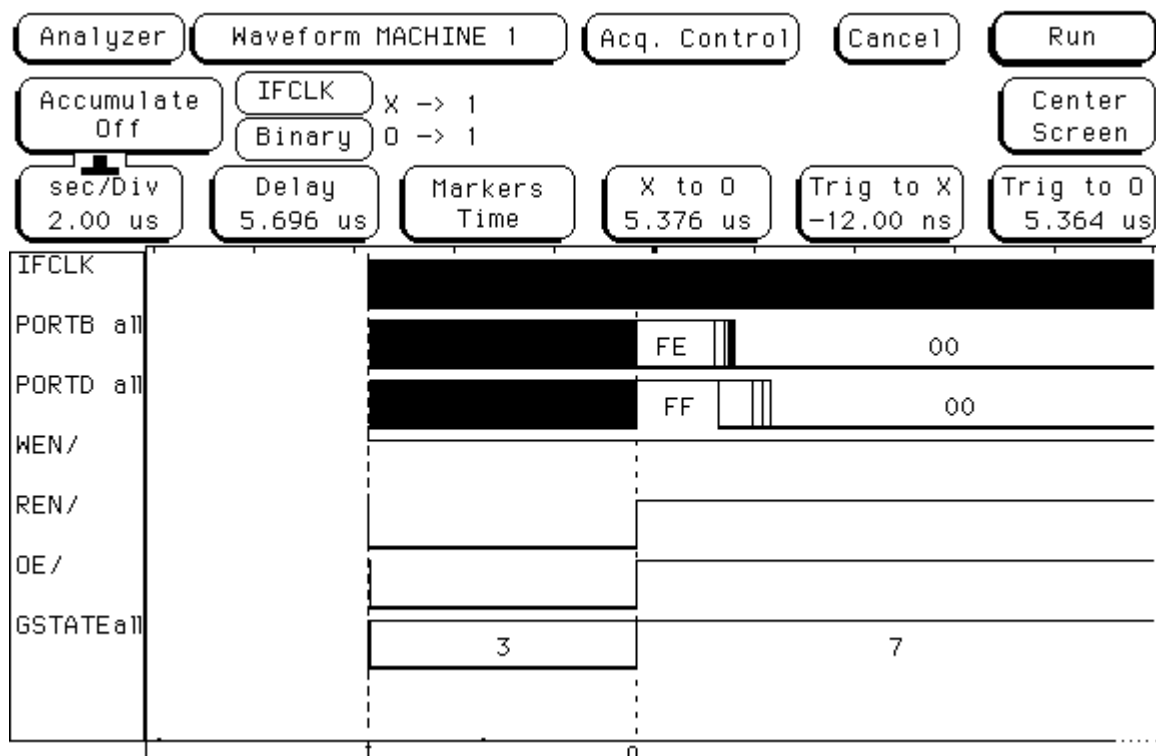
In this trace we examine the inter-packet transfer time between consecutive OUTs sent by the host. Notice that the FX2 has approximately 20 microseconds to spare before it has to burst out the next OUT packet. This means that the host, not the FX2, is limiting the transfer speed.

FIFO Read Waveform: Close-up view of the “front porch”



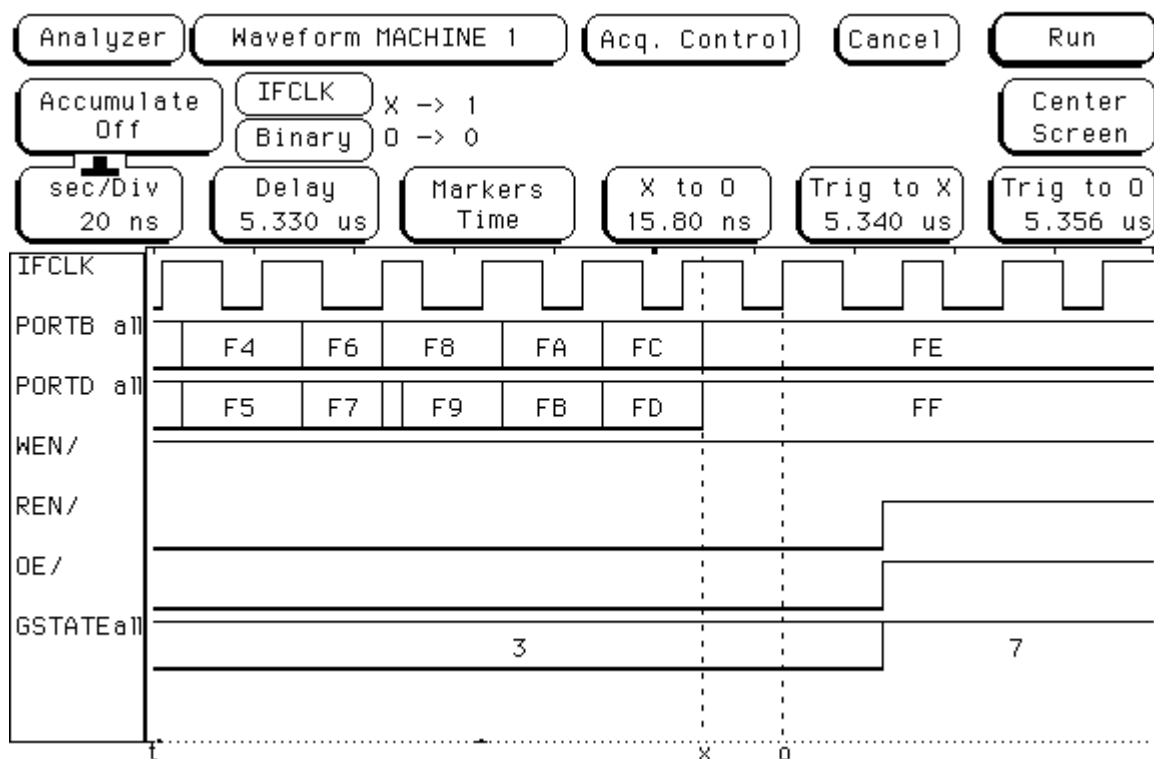
This trace shows that the 9.2-ns data set-up time for the GPIF is satisfied using the X to 0 marker as an indicator. The word consisting of data values 0x00 and 0x01 is read from the external FIFO on the rising edge of IFCLK (the external FIFO's RCLK). While REN is held LOW, consecutive words are read from the external FIFO on every rising edge of IFCLK. Notice that the GSTATE bus reflects the state of the GPIF engine as it's progressing through the GPIF FIFO Read waveform. S0 is a period of inactivity for 1 IFCLK cycle (20.83 ns). In S1, the REN is asserted since the external FIFO requires that the REN be set up t_{ENS} before the OE line is asserted. S2 asserts the OE line, and S3 is the flow state and is active for the entire duration of the data burst phase.

FIFO Read Waveform: Time taken to read 512 bytes from the external FIFO



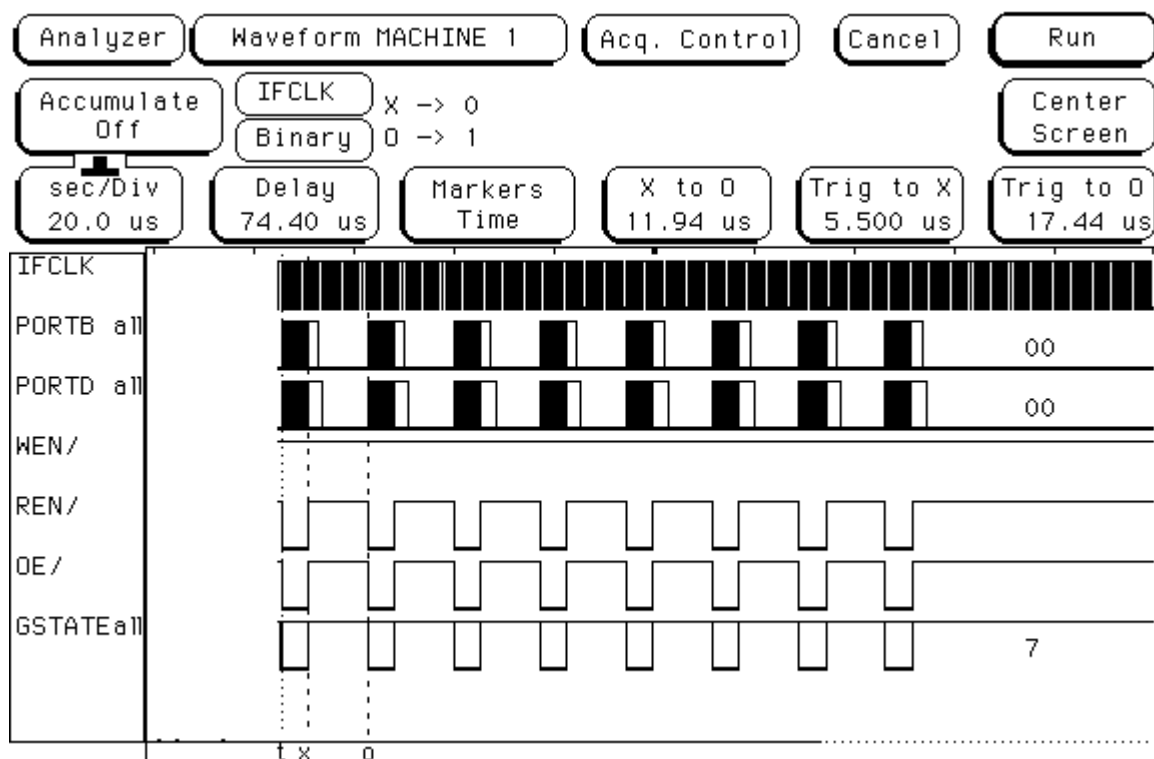
This trace shows how long it takes to read a burst of 512 bytes from the external FIFO. At a burst rate of 96 MB/s (one word every IFCLK period), this results in a time of approximately 5.3 microseconds to transfer a payload of 512 bytes. This zoomed out view allows us to see that indeed the GPIF FIFO Read waveform remains in the flowstate until it is done transferring 512 bytes, at which point it then transitions to the IDLE state (S7).

FIFO Read Waveform: Close-up view of the “back porch”



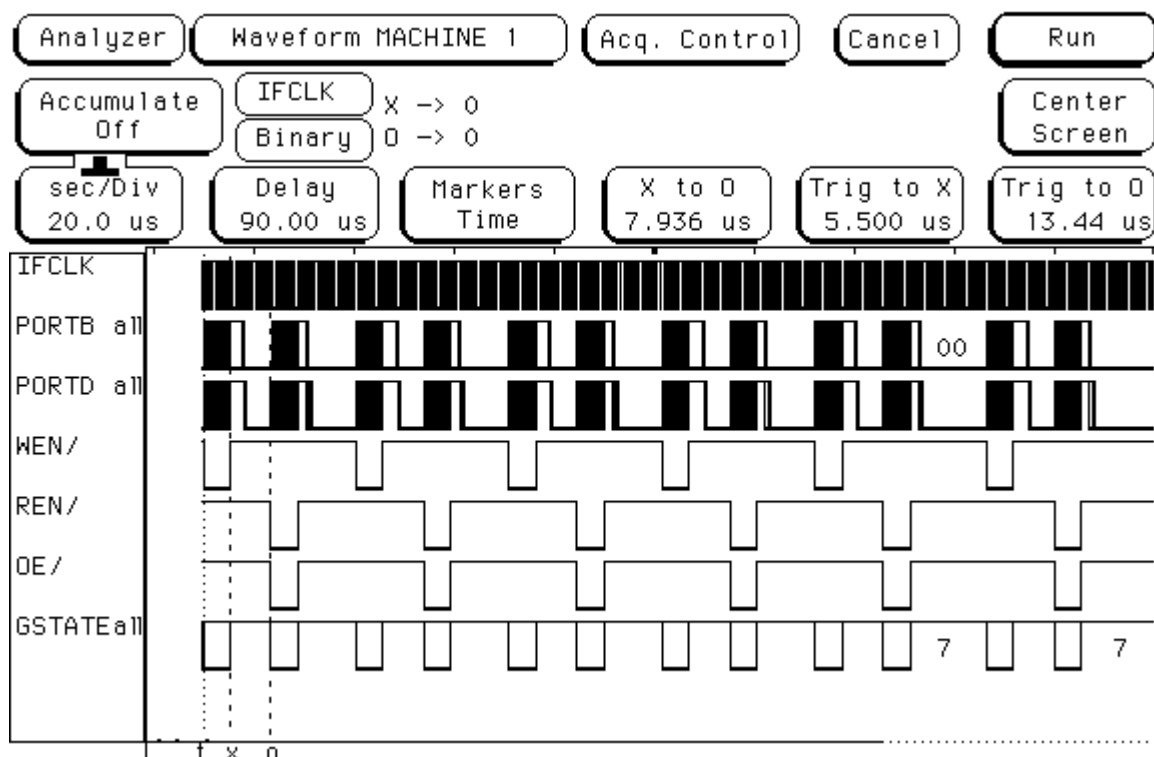
Here we see the back end of the 512-byte transfer at a zoomed in level. The last word in the packet consists of data values 0xFE and 0xFF (the end of our ramp test data). Note that a repeated word at the end is not clocked in as the setup time for the REN line is not met prior to the IFCLK edge.

FIFO Read Waveform: Inter-packet transfer time



In this trace we examine the inter-packet transfer time between consecutive INs requested by the host. Notice that the FX2 has approximately 20 microseconds to spare before it has to fulfill the next IN request. This means that the host, not the FX2, is limiting the transfer speed.

Bulk loopback waveform: FIFO Reads and Writes



You will observe the above waveform when the bulkloop utility is exercised. This trace shows activity that includes both reads and writes to the external FIFO. We notice here that the host judiciously schedules INs and OUTs. No favoritism is shown to either type of transfer.

Summary

This design example of a 16-bit interface to an external synchronous FIFO has illustrated many GPIF programming fundamentals, such as determining GPIF hardware connections, creating GPIF single and FIFO waveform descriptors using the GPIF Designer, and launching GPIF single and FIFO transfers in firmware. You should now have a firm grasp of what it takes to create a full-featured GPIF application solution, and how to go from a simple set of firmware that utilizes GPIF single transactions, to a more complex and robust application that uses GPIF FIFO transactions. Also, by now you should be aware that the logic analyzer is a GPIF programmer's best friend. Let's extend the basic toolset you should already have by presenting a more complex design example in the next section. The next section discusses interfacing to a Texas Instruments (TI) DSP's HPI port.

4.2 Interfacing to a TI 5416 DSP via the Host Port Interface (HPI)

4.2.1 Background on the TI 5416 DSP and Overview

The Texas Instruments TI 5416 fixed-point DSP finds its home in many mid-range DSP applications. It is supported by the TI 5416 DSK, which proved very attractive for this example because it exposes the HPI on one of the three expansion headers of the DSK board. The HPI allows a host processor access to the internal RAM of the 5416, thereby enabling the transfer of data between the host processor and the 5416.

By interfacing the FX2 to the 5416's HPI, developers of embedded audio and imaging applications can easily add a high-speed USB port. The FX2 can also bootstrap the 5416 DSP code via the HPI. In this example, you will be shown how to interface the FX2 to the 5416 HPI using the GPIF to accomplish two things: (1) read and write to the internal RAM block of the 5416, and (2) bootstrap the 5416 by downloading the DSP code from the PC. For detailed information about how the HPI block of the 5416 works, please refer to the TI documentation mentioned in the references section at the end of this document. Note that the information presented here may also be applicable to other TI DSPs that expose the HPI port.

4.2.2 Hardware Connections

Table 3 discusses the definition of the GPIF interconnect, which is shown below in Figure 19.

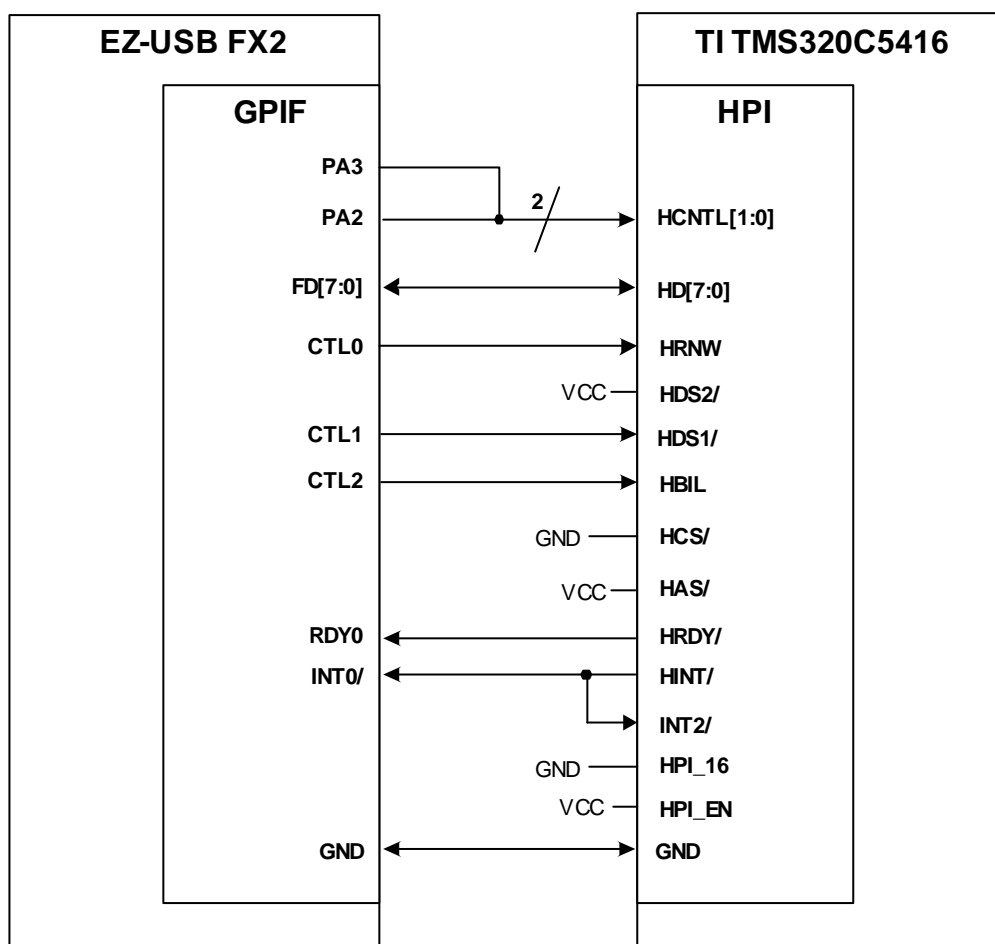


Figure 19. GPIF Interconnect to TI 5416 HPI

Table 3. Assignment of FX2 GPIF Signals to TI 5416 HPI Signals

FX2 GPIF Signals	TI 5416 HPI Signals	Description
PA3, PA2	HCNTL[1:0]	Port pins PA3 and PA2 are used to provide address lines to select either the HPIC, HPIA, or HPID registers of the HPI. The FX2 reads and writes data by accessing these registers
FD[7:0]	HD[7:0]	The lower portion of the GPIF data bus (FD[7:0]) is connected to the HPI data bus (HD[7:0]). The FX2 uses this connection for exchange of information between itself and the HPI
CTL0	HRNW	CTL0 is connected to the HRNW signal of the HPI. If HRNW is a 1, this indicates a read access to the HPI. If HRNW is a 0, this indicates a write access to the HPI
CTL1	HDS1	CTL1 is connected to the HDS1 $\overline{}$ strobe of the HPI. The falling edge of HDS1 $\overline{}$ marks the beginning of the HPI access, and samples the value of HRNW, HCNTL[1:0], and HBIL. The rising edge of HDS1 $\overline{}$ marks the end of the HPI access.
CTL2	HBIL	CTL2 is connected to the HBIL signal of the HPI. A complete HPI access consists of a two byte transfer. If HBIL is 0, this indicates to the HPI that the first byte is being transferred. To indicate to the HPI that the second byte is being transferred, HBIL must be 1
RDY0	HRDY $\overline{}$	RDY0 is connected to the HRDY $\overline{}$ signal of the HPI. HRDY $\overline{}$ is low when the HPI is completing the internal portion of a complete HPI access. Another access to the HPI must not be performed until the internal portion of the transfer is complete. This signal can be monitored by the GPIF
INT0 $\overline{}$	HINT $\overline{}$	The INT0 $\overline{}$ interrupt signal on the FX2 is connected to the HINT $\overline{}$ output of the HPI. When the DSP is reset, the HINT $\overline{}$ will be asserted. The DSP can also use this as a general purpose interrupt to the FX2

On the DSP side, HPI_EN is tied to V_{CC} to enable the HPI port, and HPI_16 is tied to ground to make the HPI operate in 8-bit mode (HPI-8). The HPI can operate in 16-bit mode (HPI-16) if the 5416's external memory interface (EMIF) is not used. For most DSP applications, the EMIF will already be used for memory expansion. Using the HPI in 8-bit mode also simplifies the GPIF interface. The HINT $\overline{}$ signal is also tied to the 5416's INT2 pin to allow the FX2 to be able to bootstrap the DSP code.

HCS $\overline{}$ is tied to ground to allow continuous access to the HPI, and HAS $\overline{}$ and HDS2 $\overline{}$ are tied to V_{CC} since they are not necessary for this interface (due to the flexibility of the GPIF interface).

The assignment of CTLx and RDYn lines was again optimized for the FX2 56-pin package. The connection between the TI 5416 DSK board and the FX2 development board was accomplished through the use of a ribbon cable set. The TI 5416 DSK board exposes headers that require breakout panels (available from www.dspsglobal.com) for prototyping purposes. The ribbon cables connect between a breakout panel installed on the DSK board's P3 and the FX2 prototype board mounted onto the FX2 development board. *Figure 20* shows a snapshot of the actual hardware set-up.

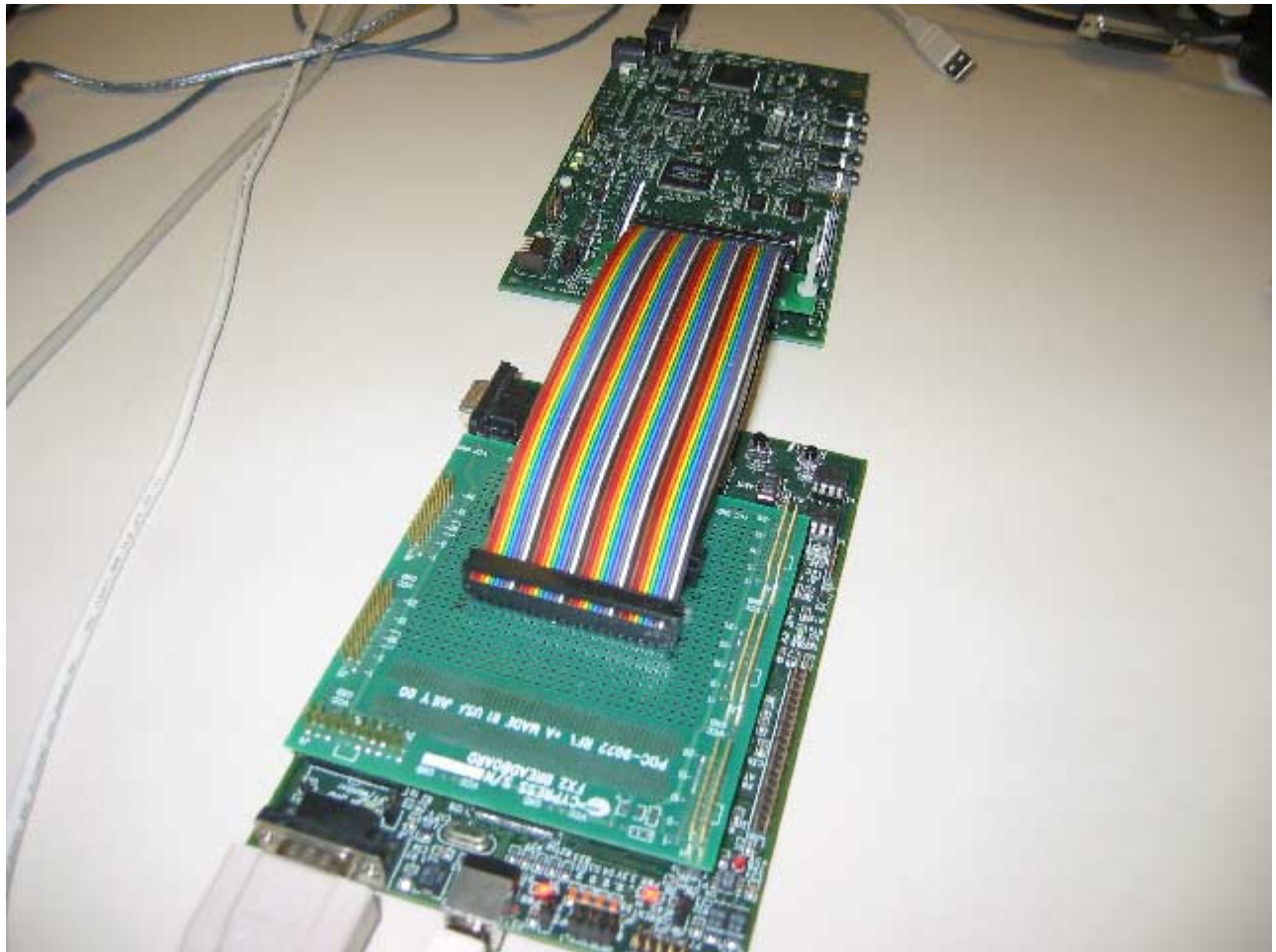


Figure 20. Hardware Set-up

4.2.3 Application-specific Data Flow

Now that the GPIF interconnect has been presented, it's important to understand the overall data flow for this design example. EP2OUT (4x buffered) is the source endpoint used for data writes to the HPI and EP6IN (4x buffered) is the sink endpoint used for data reads from the HPI. EP0, the FX2's control endpoint, is used for writes to the HPI control (HPIC) and HPI address (HPIA) registers.

Before a data read or write can commence to and from a specific address in the DSP, the HPIA needs to be set up with the appropriate source or destination address. The HPIC also needs to be set up to set the BOB (byte order bit) bit to 1, which allows the first byte of transfer to be the LSB and the second byte of the transfer to be the MSB (as organized in the DSP memory). Since the 5416 supports an extended address scheme, the XPHIA bit in the HPIC register needs to be set if FX2 wants to access the upper seven bits of the HPIA register. The XPHIA bit also needs to be set if proper auto-increment of the address is to occur when consecutive data read and write accesses are made.

Figure 21 and Figure 22 show the data flow models for this example. The most common method of operation would be to send a control OUT transfer once with the source or destination address, followed by a bulk transfer to send or receive a contiguous block of data. A control OUT transfer can then be sent again to start a block access from a different address.

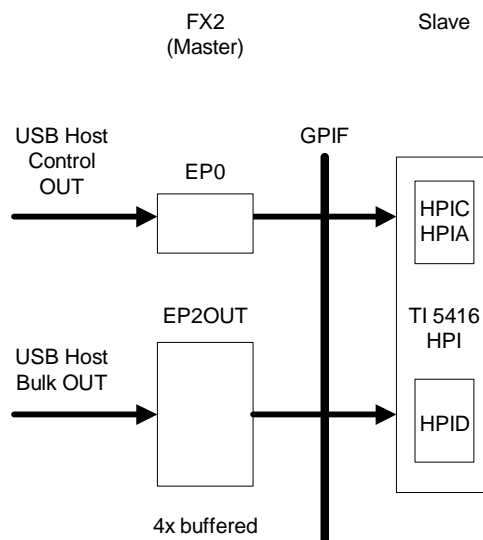


Figure 21. Data Flow Model in the OUT Direction

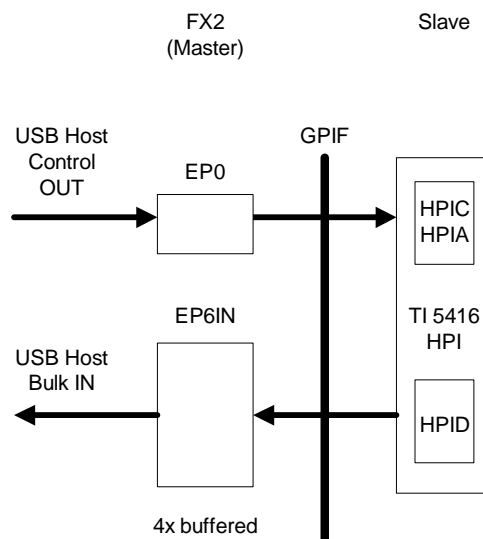


Figure 22. Data Flow Model in the IN Direction

GPIF single transactions are used to write out the data from EP0 to the HPIC and HPIA registers. GPIF FIFO transactions are used for data reads and writes using EP6IN and EP2OUT in auto mode, respectively.

4.2.4 Creating GPIF Waveform Descriptors using the GPIF Designer

In order to design the GPIF waveform descriptors for this example, it is first important to understand a little bit about how the TI DSP's HPI protocol works. Each HPI transfer is a two-byte sequence. The meaning of the first byte and second byte depends on how the BOB bit is set in the DSP's HPIC register. In our example, the BOB bit is set to 1, which means that the first byte of the HPI transfer is going to be the LSB and the second byte of the transfer is going to be the MSB (as organized in the DSP memory).

The example uses GPIF single write transactions for writing to the HPIC and HPIA registers, GPIF FIFO Write transactions for writing data into the HPI RAM, and GPIF FIFO Read transactions for reading data from the HPI RAM. Writing to the HPIC and HPIA registers is a special case that requires two separate waveform behaviors to describe the entire HPI transfer. One waveform behavior describes the timing and control logic for the first byte of the HPI transfer, and another describes the timing and control logic for the second byte of the HPI transfer.

In all of the GPIF waveforms, CTL0-2 are manipulated according to the HPI protocol and the HPI8 Mode Timing Requirements as outlined in the 5416 data sheet. In the text to follow, we will discuss how CTL0-2 were manipulated in the GPIF waveforms to describe the HPI protocol. *Figure 23* shows the block diagram for the DSP example.

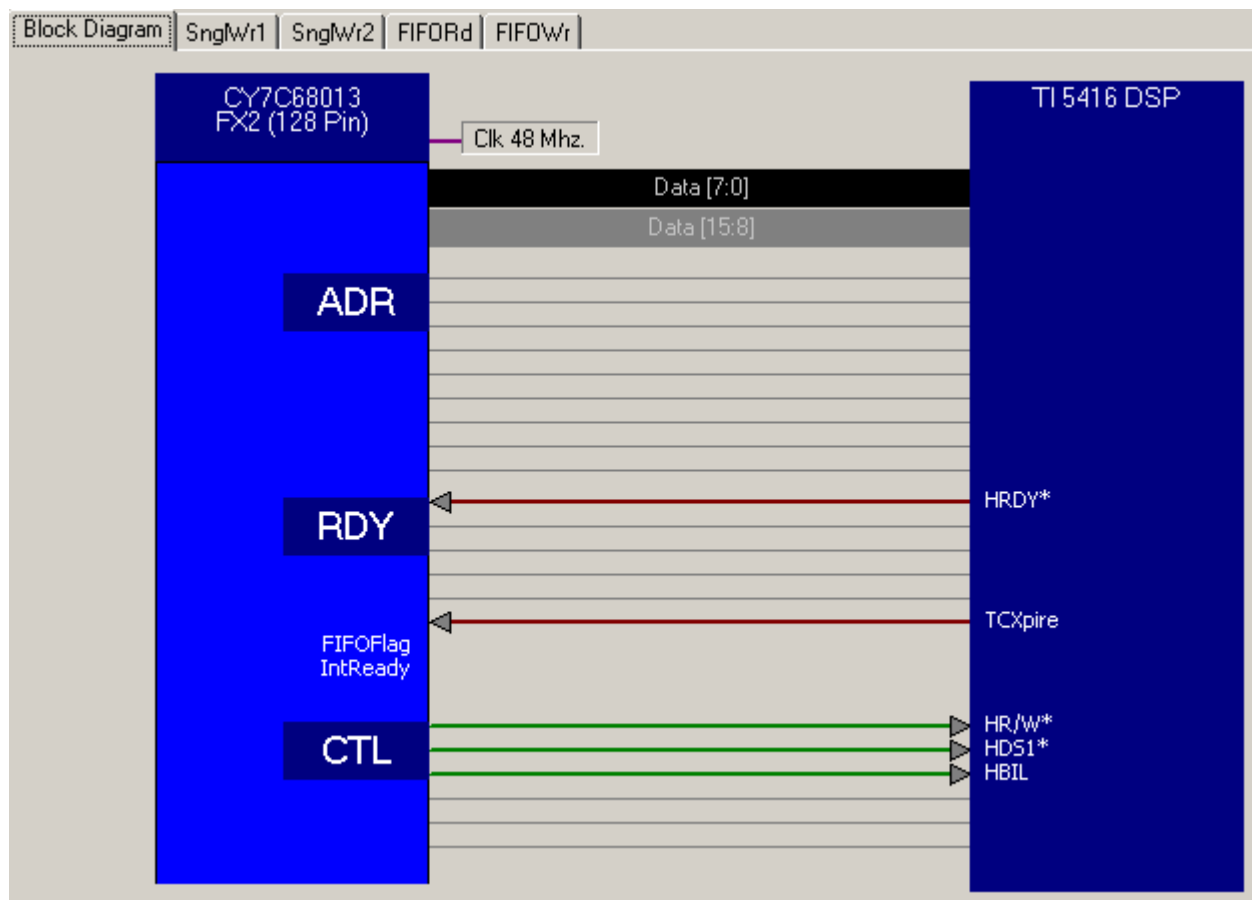


Figure 23. GPIF Designer Block Diagram View

Figure 24 below shows waveform 0, which characterizes the behavior of the waveform called SnglWr1. SnglWr1 describes the HPI protocol required to write the first byte of an HPI transfer.

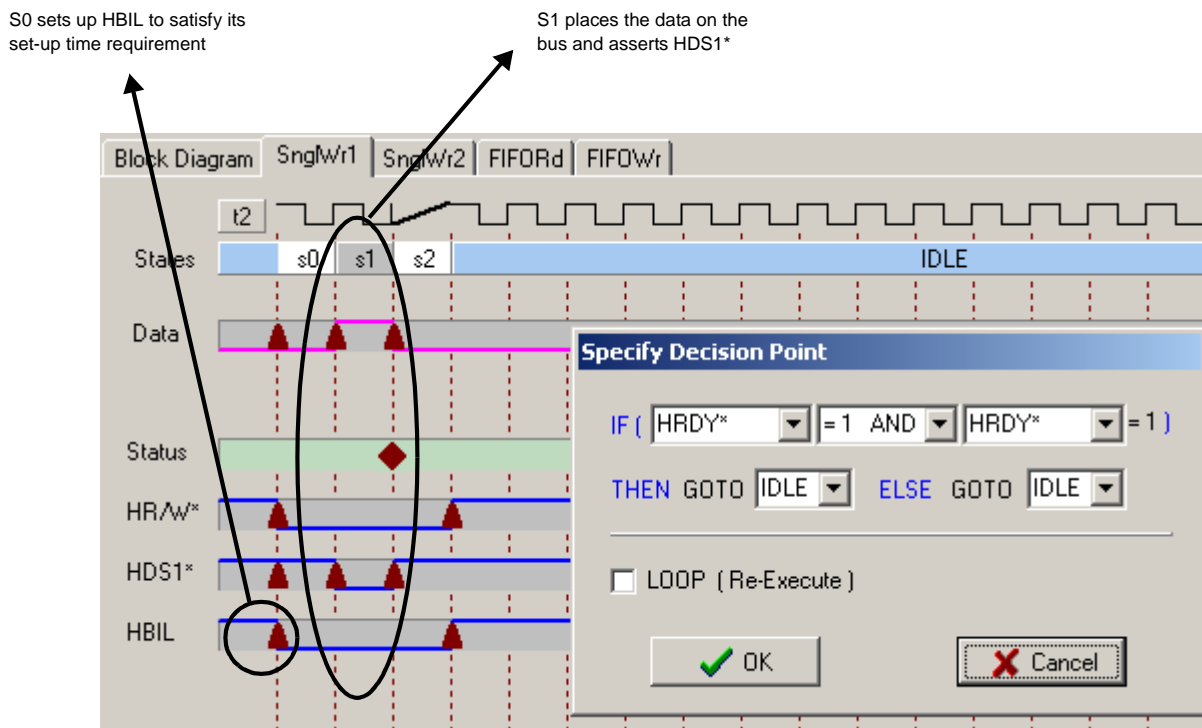


Figure 24. SnglWr1 Waveform in GPIF Designer

Since SnglWr1 describes an HPI write operation, $\overline{\text{HR/W}}$ (CTL0) is held LOW throughout the entire transfer (S0-S2). HBIL (CTL2) is dropped LOW in S0 to signify that the first byte is being transferred. This is done before HDS1 (CTL1) is asserted in S1, in order to satisfy the set-up time requirement for HBIL before HDS1 can be made LOW. Since S0 is active for 20.83 ns (Wait 1 at 48-MHz IFCLK), this satisfies the set-up time requirement of 6 ns for HBIL. Since there is also a hold time requirement for HBIL, to simplify matters, HBIL is actually held LOW throughout the active portion of the entire waveform.

By looking at the HPI8 Timing Requirements in the 5416 data sheet, it becomes apparent that any of the strobe widths or set-up and hold times are well under 20.83 ns. Therefore, one can assume that a state need only last at maximum 20.83 ns (Wait 1). In S1, data is also placed on the bus (Activate Data). In S2, HDS1 is deasserted, thus ending this portion of the HPI transfer. S2 also unconditionally branches to the IDLE state to terminate the waveform.

The waveform that describes the second portion of the HPI transfer is very similar to SnglWr1, and is shown in Figure 25.

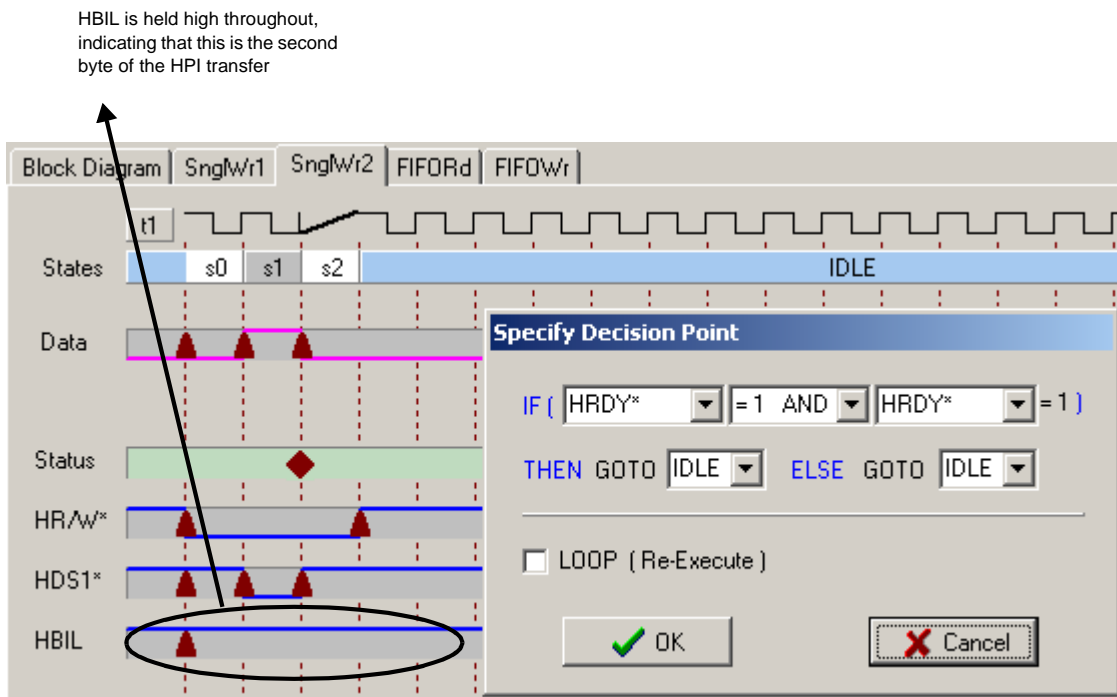


Figure 25. SnglWr2 Waveform in GPIF Designer

Again, since SnglWr2 also describes an HPI write operation, $\overline{\text{HR}/\text{W}}$ is held LOW throughout the entire active portion of the waveform (S0–S2). The main difference between SnglWr2 and SnglWr1 is the state of HBIL; HBIL is HIGH throughout S0–S2. This signifies to the HPI that the second byte of the HPI transfer is being transmitted. S2 unconditionally branches to the IDLE state to terminate the waveform.

To recap, the SnglWr1 and SnglWr2 waveforms are used for GPIF single write accesses, which allow us to write to the DSP's HPIC/HPIA registers. The GPIF engine allows you to select which of these waveforms are triggered by a GPIF single write access, via the GPIFWFSELECT register. Consecutive GPIF single write accesses using the waveforms SnglWr1 and SnglWr2 are made to describe the entire HPI transfer protocol. The details of this are described in the firmware programming section (4.2.5).

To create the GPIF FIFO read and write accesses that handle writing to and from the HPI data RAM, the attributes of the SnglWr1 and SnglWr2 waveforms can be combined to form each of the GPIF FIFO read and write waveforms. *Figure 26* shows the GPIF FIFO write waveform.

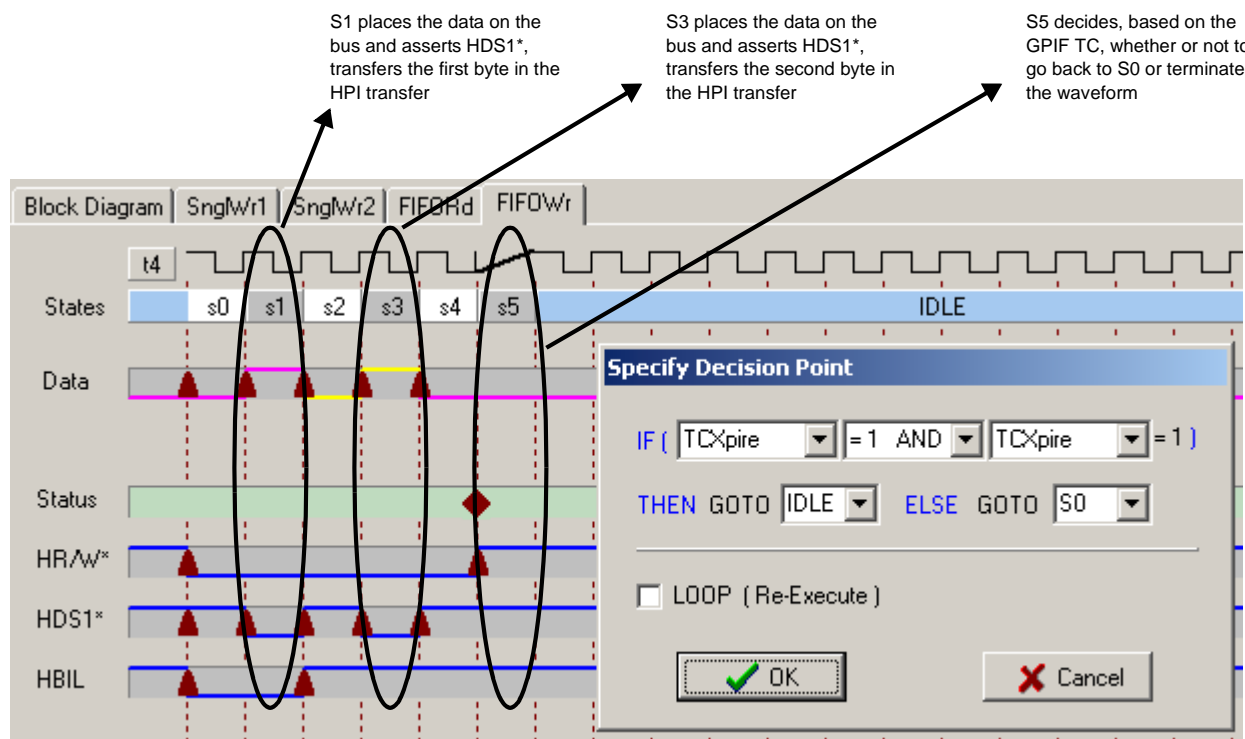


Figure 26. FIFOWr Waveform in GPIF Designer

Waveform 3 (FIFOWr) describes an entire HPI write transfer. S0 drives both $\overline{\text{HR/W}}$ and HBIL LOW for 20.83 ns, then S1 asserts $\overline{\text{HDS1}}$ and drives the data bus to present the first byte in the EP2 FIFO, effectively writing out the first byte of the HPI transfer to the HPI RAM. S2 then increments the FIFO pointer using Next FIFO data, deasserts $\overline{\text{HDS1}}$, and drives HBIL HIGH to tell the HPI the second byte of the transfer is coming. S3 asserts $\overline{\text{HDS1}}$ again and drives the data bus to present the second byte in the EP2 FIFO, effectively writing out the second byte of the HPI transfer to the HPI RAM.

The waveform then traverses to S5, a decision point state that examines the GPIF TC to determine whether or not to branch to the IDLE state. If the GPIF TC has not yet expired, the waveform will then branch back to S0 to actuate another HPI transfer. Otherwise, the waveform branches to the IDLE state and terminates.

The FIFO read waveform is quite similar in nature and is shown in *Figure 27*.

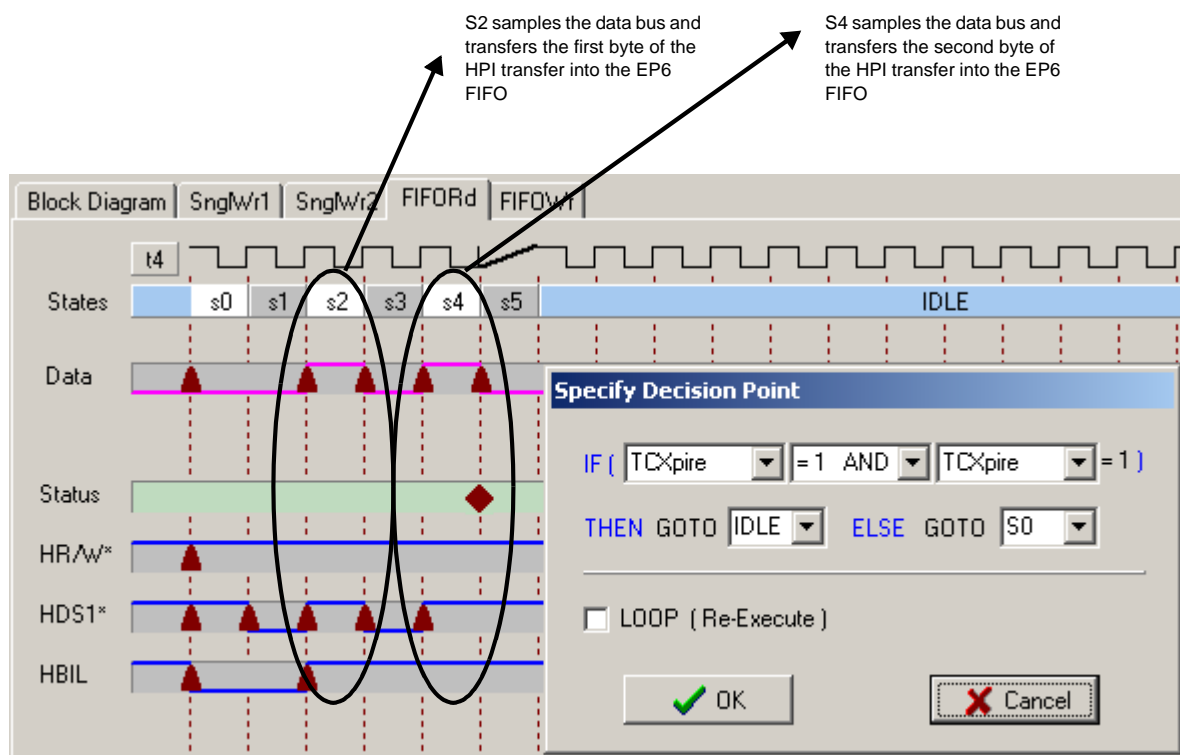


Figure 27. FIFORd Waveform in GPIF Designer

Waveform 2 (FIFORd) describes an entire HPI read transfer. $\overline{\text{HR}}/\overline{\text{W}}^*$ is driven HIGH throughout the waveform to tell the HPI that this is a read operation. In S0, HBIL is driven LOW for 20.83 ns to satisfy the set-up time requirement for HBIL, then S1 asserts HDS1, which tells the HPI to present the first byte of the HPI read transfer onto the data bus. The data is not presented until 10 ns later, therefore it is correct to only sample the databus in S2 and not in S1. By sampling the databus in S2, the first byte is read into the FX2's EP6 FIFO. For a GPIF FIFO read transaction, an Activate Data also advances the FIFO pointer, so a Next FIFO data is not necessary.

S2 also drives HBIL HIGH to tell the HPI the second byte of the transfer is expected. S3 asserts $\overline{\text{HDS1}}$ again and S4 samples the data bus to read the second byte into the EP6 FIFO.

The waveform then traverses to S5, a decision point state that examines the GPIF TC to determine whether or not to branch to the IDLE state. If the GPIF TC has not yet expired, the waveform will then branch back to S0 to actuate another HPI read transfer. Otherwise, the waveform branches to the IDLE state and terminates.

Now that you understand how the GPIF waveforms are programmed and set up for the DSP example, the firmware programming can be discussed.

4.2.5 Firmware Programming

After the GPIF waveforms are implemented in the GPIF Designer, the next step is to integrate the USB portion of the overlying firmware with the GPIF Designer output to perform write and read operations to and from the HPI. To do this a Firmware Frameworks project was copied and the code that performed the HPI operations was added to the TD_Poll() function within FX2_to_TI5416_HPI.c (note that `periph.c` was renamed to something more meaningful here). Endpoint and GPIF register initialization is performed in the TD_Init() function, which is also within FX2_to_TI5416_HPI.c. When you open up the Keil uVision2 project for the DSP example, Figure 28 shows the list of files that should be seen in the Project Window:

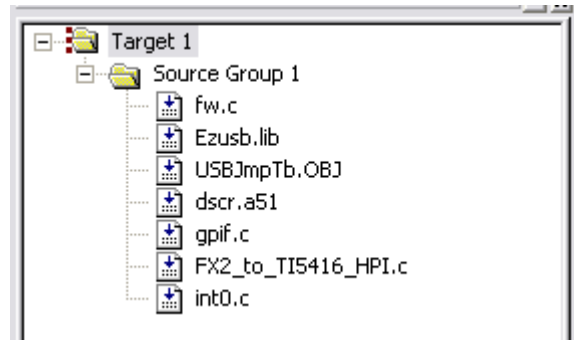


Figure 28. Files Shown in Project Window

`fw.c` – Firmware Frameworks, which handles USB requests and calls the task dispatcher TD_Poll()

`Ezusb.lib` – collection of functions that handle suspend, resume, I²C operations, etc.

`USBJmpTb.OBJ` – interrupt vector jump table for USB (INT2) and GPIF/Slave FIFO (INT4) interrupt sources

`Dscr.a51` – device descriptor tables for the DSP example which report EP2OUT and EP6IN as available endpoints for the FX2 device. EP1IN and EP1OUT are also reported although they are not used in this example. These low bandwidth endpoints may be used for general purpose should the need arise in an application

`Gpif.c` – file that contains the GPIF waveform descriptor tables that implement the single/FIFO GPIF transaction waveform behaviors.

`FX2_to_TI5416_HPI.c` (renamed from `periph.c`) – main user application code where TD_Poll() and TD_Init() can be found. You will mainly modify this particular file and *will not need to touch fw.c*

`int0.c` – file that contains the interrupt service routine for servicing the INT0/ interrupt

TD_Init()

The first task at hand is to set up the endpoints appropriately for this example. The following code switches the CPU clock speed to 48 MHz (since at power-on default it is 12 MHz), and sets up EP2 as a Bulk OUT endpoint, 4x buffered of size 512, and EP6 as a Bulk IN endpoint, also 4x buffered of size 512. This set-up utilizes the maximum allotted 4-KB FIFO space. It also sets up the FIFOs for auto mode, byte wide operation, and goes through a FIFO reset and arming sequence to ensure that they are ready for data operations. EP1IN and EP1OUT are also set up in case the application needs them, although they are not used by this example. Note that although EP1IN and EP1OUT are reported to have a max packet size of 512 bytes, this is strictly for compliance reasons. You should never transfer more than 64 bytes using EP1IN and EP1OUT.

```
// set the CPU clock to 48 MHz
CPUCS = ((CPUCS & ~bmCLKSPD) | bmCLKSPD1);
SYNCDelay;

EP1OUTCFG = 0xA0; // always OUT, valid, bulk
EP1INCFG = 0xA0; // always IN, valid, bulk
SYNCDelay;
EP2CFG = 0xA0; // EP2OUT, bulk, size 512, 4x buffered
SYNCDelay;
EP4CFG = 0x00; // EP4 not valid
SYNCDelay;
EP6CFG = 0xE0; // EP6IN, bulk, size 512, 4x buffered
SYNCDelay;
EP8CFG = 0x00; // EP8 not valid
SYNCDelay;

FIFORESET = 0x80; // set NAKALL bit to NAK all transfers from host
SYNCDelay;
FIFORESET = 0x02; // reset EP2 FIFO
SYNCDelay;
FIFORESET = 0x06; // reset EP6 FIFO
SYNCDelay;
FIFORESET = 0x00; // clear NAKALL bit to resume normal operation
SYNCDelay;

EP2FIFOCFG = 0x00; // allow core to see zero to one transition of auto out bit
SYNCDelay;
EP2FIFOCFG = 0x10; // auto out mode, disable PKTEND zero length send, byte ops
SYNCDelay;
EP6FIFOCFG = 0x08; // auto in mode, disable PKTEND zero length send, byte ops
SYNCDelay;
EP1OUTBC = 0x00; // arm EP1OUT by writing any value to EP1OUTBC register

GpifInit(); // initialize GPIF registers

PORTACFG = bmBIT0; // PA0 takes on INT0/ alternate function
OEA |= 0x0C; // initialize PA3 and PA2 port i/o pins as outputs

EX0 = 1; // Enable INT0/ interrupt
IT0 = 1; // Detect INT0/ on falling edge
```

TD_Init then calls the function GPIFInit() which resides in gpif.c. GPIFInit() is where the loading of the GPIF waveform descriptor table into on-chip memory takes place and other GPIF registers get set up. An important register, IFCONFIG, also gets set up here to define how the physical interface operates. *Table 4* shows the set-up of the IFCONFIG register for this example.

Finally, TD_Init sets up PA3 and PA2 as outputs and enables the INT0/ functionality.

Table 4. IFCONFIG Register Bit Settings for DSP Example

Bit Position	Bit Name	Setting for Example
IFCONFIG.7	IFCLKSRC	IFCLKSRC is set to 1 to run the GPIF using the internal clock source
IFCONFIG.6	3048MHZ	3048MHZ is set to 1 to run the internal clock source for the GPIF at 48 MHz
IFCONFIG.5	IFCLKOE	IFCLKOE is set to 1 to turn on the IFCLK output for debug purposes on the logic analyzer
IFCONFIG.4	IFCLKPOL	IFCLKPOL is set to 0 and bears no significance to this example
IFCONFIG.3	ASYNC	The ASYNC setting is a don't care in GPIF master mode, since GPIF always references IFCLK (internal or external)
IFCONFIG.2	GSTATE	GSTATE is set to 1 to turn on the debug outputs of the state machine. PE[2:0] displays the states the GPIF engine cycles through during each transaction (Note: PE[2:0] are only available on the 100- and 128-pin packages)
IFCONFIG[1:0]	IFCFG[1:0]	IFCFG[1:0] are set to 10 to put the FX2 part into GPIF mode. Otherwise, FX2 defaults to port I/O mode

Writing to the HPIC and HPIA registers

The firmware implements a 0xB6 vendor OUT command and a 0xB7 vendor OUT command to write to the HPIC and HPIA registers, respectively. The following code writes to the HPIC register:

```
case VX_B6: // write to HPIC register
{
    EP0BCL = 0;                // re-arm EP0
    while(EP01STAT & bmEP0BSY); // wait until EP0 is available to be accessed by CPU
    while(!HPI_RDY);           // wait for HPI to complete internal portion of previous transfer
    IOA = bmHPIC;               // select HPIC register
    GPIFWFSELECT = 0x1E;        // point to waveforms that write first byte of HPI protocol
    GPIF_SingleByteWrite(EP0BUF[0]); // write LSB of DSP address
    GPIFWFSELECT = 0x4E;        // point to waveforms that write second byte of HPI protocol
    GPIF_SingleByteWrite(EP0BUF[1]); // write MSB of DSP address

    break;
}
```

The code first re-arms the EP0 buffer to accept the next packet from the host. It then waits until the EP0 buffer is available for the CPU to access. Before addressing the HPIC register, the code checks to see if the HPI is ready to accept another transfer. The HPIC register is then addressed by writing a 0 to both PA3 and PA2.

The technique of changing the waveform index in the GPIFWFSELECT register to point to different waveforms is useful for handling protocols such as the HPI, where more than one waveform behavior is required to describe a complete read/write cycle. The GPIFWFSELECT register is first configured to point to the SnglWr1 waveform which writes the first byte of the HPI protocol. This is now the waveform that gets triggered when a GPIF single write access occurs. The first byte in EP0BUF is then written out to the HPI. The GPIFWFSELECT register is then configured to point to the SnglWr2 waveform which writes the second byte of the HPI protocol. This is now the waveform that gets triggered when a GPIF single write access occurs. The second byte in EP0BUF is then written out to the HPI. Because the HPI protocol requires that both first and second bytes in the HPI transfer should be of equal value when writing to the HPIC register, the host should send down two bytes of equal value when performing the vendor OUT request of 0xB6.

The following code writes to the HPIA register:

```
case VX_B7: // write to HPIA register
{
    EP0BCL = 0;                // re-arm EP0
    while(EP01STAT & bmEP0BSY); // wait until EP0 is available to be accessed by CPU
    while(!HPI_RDY);           // wait for HPI to complete internal portion of previous transfer
    IOA = bmHPIA;               // select HPIA register
    GPIFWFSELECT = 0x1E;        // point to waveforms that write first byte of HPI protocol
    GPIF_SingleByteWrite(EP0BUF[0]); // write LSB of DSP address
    GPIFWFSELECT = 0x4E;        // point to waveforms that write second byte of HPI protocol
    GPIF_SingleByteWrite(EP0BUF[1]); // write MSB of DSP address

    break;
}
```

This code re-arms the EP0 buffer to accept the next packet from the host. It then waits until the EP0 buffer is available for the CPU to access. Before addressing the HPIA register, the code checks to see if the HPI is ready to accept another transfer. The HPIA register is then addressed by writing a 1 to PA3 and a 0 to PA2.

The GPIFWFSELECT register is then configured to point to the SnglWr1 waveform which writes the first byte of the HPI protocol. This is now the waveform that gets triggered when a GPIF single write access occurs. The first byte in EP0BUF is then written out to the HPI. The GPIFWFSELECT register is then configured to point to the SnglWr2 waveform, which writes the second byte of the HPI protocol. This is now the waveform that gets triggered when a GPIF single write access occurs. The second byte in EP0BUF is then written out to the HPI. When the host performs the vendor OUT request of 0xB7, the first byte contains the LSB of the DSP address, and the second byte contains the MSB of the DSP address.

Performing Data Writes and Reads to and from the HPI

The code in TD_Poll() handles USB OUT transfers (Data Writes to the HPI) and USB IN transfers (Data Reads from the HPI).

Code that handles USB OUT Transfers

```
if( GPIFTRIG & 0x80 )           // if GPIF interface IDLE
{
    if ( ! ( EP24FIFOFLGS & 0x02 ) ) // if there's a packet in the peripheral domain for EP2
    {
        IOA = bmHPID_AUTO;          // select HPID register with address auto-increment
        while(!HPI_RDY);            // wait for HPI to complete internal portion of previous transfer

        SYNCDELAY;
        GPIFTCB1 = EP2FIFOBCH;       // setup transaction count with number of bytes in the EP2 FIFO
        SYNCDELAY;
        GPIFTCB0 = EP2FIOBCL;
        SYNCDELAY;
        GPIFTRIG = GPIF_EP2;         // launch GPIF FIFO WRITE Transaction from EP2 FIFO
        SYNCDELAY;

        while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 GPIF Done bit
        {
            ;
        }
        SYNCDELAY;
    }
}
```

The OUT handling code checks to see if the GPIF is IDLE. If so, it checks to see if there is at least a packet in the peripheral domain for EP2. Since EP2 is placed into auto mode, the firmware does not need to check if the host sent a USB packet. The USB packets are automatically committed to be used by the GPIF engine. Therefore, the firmware's job is to check if at least one packet has been committed to the peripheral domain.

The TC value is then simply setup with the number of bytes in the EP2 FIFO. This allows the GPIF to handle packet sizes less than 512 but greater than zero. The TC value is a 32-bit register field, but for this application only the lower 16-bit fields are used. A write to the GPIFTRIG register with the appropriate bits triggers the transaction from EP2OUT. The code then waits for the transaction to complete before exiting out of the "if" nest.

Code that handles USB IN transfers

```

if(in_enable)                // if IN transfers are enabled,
{
    if(Tcount)                // if Tcount is not zero
    {
        if( GPIFTRIG & 0x80 )    // if GPIF interface IDLE
        {
            if( !( EP6FIFOFLGS & 0x01 ) )    // if EP6 FIFO is not full
            {
                IOA = bmHPID_AUTO;    // select HPID register with address auto-increment
                while(!HPI_RDY);    // wait for HPI to complete internal portion of previous transfer

                SYNCDELAY;
                GPIFTCB1 = MSB(Tcount);    // setup transaction count with Tcount value
                SYNCDELAY;
                GPIFTCB0 = LSB(Tcount);
                SYNCDELAY;

                GPIFTRIG = GPIFTRIGRD | GPIF_EP6; // launch GPIF FIFO READ Transaction to EP6IN
                SYNCDELAY;

                while( !( GPIFTRIG & 0x80 ) )    // poll GPIFTRIG.7 GPIF Done bit
                {
                    ;
                }

                SYNCDELAY;

                xFIFOBCH_IN = ( ( EP6FIFOBCH << 8 ) + EP6FIFOBCL ); // get EP6FIFOBCH/L value

                if( xFIFOBCH_IN < 0x0200 )    // if pkt is short,
                {
                    INPKTEND = 0x06;    // force a commit to the host
                }
                Tcount = 0;    // set Tcount to zero to cease reading from DSP HPI RAM
            }
        }
    }
}

```

If the *in_enable* flag is not set via vendor IN command 0xB3, the code will just sit there and not process the INs. If the *in_enable* flag is set, and if *Tcount* is not zero (read along for a further explanation of the *Tcount* variable), the code will fall through and check if the GPIF interface is IDLE. It then determines if EP6 is not full, implying that it has room for at least one more data packet.

If EP6 has room for at least one more data packet, the TC value is set up with a user defined value of *Tcount*. The value of *Tcount* is set up prior to the IN transfer and can be done so by executing the vendor OUT command 0xB5. For example, if you wanted to read 4 KB worth of information from the HPI, you would send a value of 0x1000 using the vendor OUT command 0xB5. This requires that you have a priori knowledge of how much data is going to be read.

A write to the GPIFTRIG register with the appropriate bits triggers the transaction to fill the EP6 FIFO. The code then waits for the transaction to complete. Since EP6 is placed into auto mode, there is no need to explicitly write a byte count value to indicate how many bytes to send to the host. FX2 uses the EP6AUTOINLENH/L register values set at enumeration time in the *DR_SetConfiguration()* function for the auto commit size.

However, to handle packets sizes less than 512 (the last packet of the transfer will typically be short), the code checks to see if it needs to write to the INPKTEND register with a 0x06 to commit the short packet to the host. The *Tcount* variable is then set to zero to prevent from entering the loop again. Now we see the relevance of the upper if statement that checks for *Tcount*. If the statement was not present, the code would continuously fill EP6IN as the host requests IN after IN. This complicates matters on the host side because when the next transfer is started, 'stray' buffers from the previous transfer would be retrieved by the host. By checking if *Tcount* is greater than zero, you would be forced to set the *Tcount* variable to the desired value to start another transfer.

Handling *INT0*/

The *HINT*/ output is connected to the *INT0* interrupt on the FX2. This allows the 5416 to perform software handshaking with the FX2 when necessary. The *INT0* interrupt is handled via the following ISR:

```
void int0_isr (void) interrupt 0
{
    hpi_int = TRUE; // HPI interrupted the FX2
    EX0 = 0;        // disable INT0/ interrupt, let foreground re-enable it
}
```

The ISR sets a global flag *hpi_int* which is monitored by *TD_Poll()*. The code in *TD_Poll* then clears the *hpi_int* flag and turns on LED1 on the FX2 development board:

```
if (hpi_int)
{
    hpi_int = FALSE; // clear HPI interrupt flag
    EX0 = 1;         // enable INT0 interrupt again
    LED_On (bmBIT1); // turn on LED1 to alert user HPI interrupt occurred
}
```

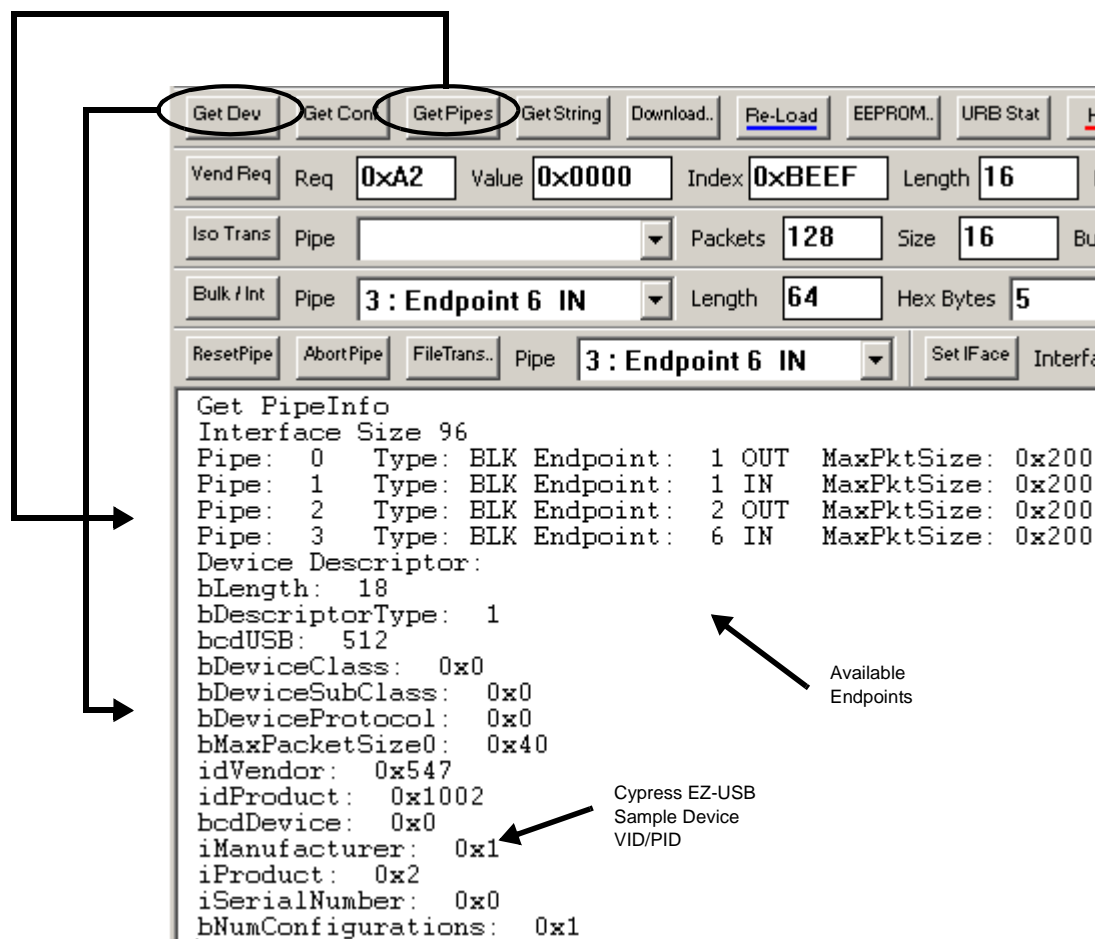
A vendor IN command of 0xB2 is provided so that you can turn off LED1.

4.2.6 Running the DSP Example

Now that you have a good idea of how this DSP example works, we can now attempt to read and write to the HPI and bootload the DSP code.

Step 1: Download the firmware using the EZ-USB Control Panel

- Unzip the “FX2_to_TI5416_HPI GPIF FIFO Transactions Auto Mode.zip” package in the C:\Cypress\USB\Examples\FX2 directory.
- Plug in the 5416 DSK board and launch the Code Composer Studio™ for the 5416.
- After you plug-in the FX2 board, launch the EZ-USB Control Panel and ensure that the selected target is FX2.
- Then, press the “Download” button and select the FX2_to_TI5416_HPI.hex file. The FX2 board renumerates as a Cypress EZ-USB Sample Device and LED0 should come up flashing.
- Perform a “Get Pipes” and “Get Dev” to verify one more time that the firmware is up and running. You should then see the screen shown below:



Step 2: Perform a write to the HPI

As an example, this step demonstrates how to write to address 0x7000 of the DSP via the HPI.

- Launch Code Composer Studio and pull up the "Memory" window at 0x7000. This will allow you to monitor if the writes occurred correctly.
- Perform a vendor OUT command of 0xB6 to write to the HPIC register using data values 0x01 0x01. This sets the BOB to 1 in the HPIC register so that the first byte of future HPI transfers is the LSB and the second byte is the MSB (as organized in DSP memory)

Vend Req	Req	0xB6	Value	0x0000	Index	0xBEEF	Length	2	Dir	0 OUT	Hex Bytes	01 01
----------	-----	------	-------	--------	-------	--------	--------	---	-----	-------	-----------	-------

- Perform a vendor OUT command of 0xB7 to write to the HPIA register with the address of 0x7000. Notice the ordering of the data in the hex bytes field since the BOB bit has been set to 1.

Vend Req	Req	0xB7	Value	0x0000	Index	0xBEEF	Length	2	Dir	0 OUT	Hex Bytes	00 70
----------	-----	------	-------	--------	-------	--------	--------	---	-----	-------	-----------	-------

- Perform another vendor OUT command of 0xB6 to write to the HPIC register using data values 0x11 0x11. This keeps the BOB bit set at 1 and sets the XPHIA bit. Setting the XPHIA bit does two things: 1) it allows the address to auto-increment for consecutive data accesses to HPI and 2) the next write to the HPIA register contains the extended address value.

Vend Req	Req	0xB6	Value	0x0000	Index	0xBEEF	Length	2	Dir	0 OUT	Hex Bytes	11 11
----------	-----	------	-------	--------	-------	--------	--------	---	-----	-------	-----------	-------

- Perform another vendor OUT command of 0xB7 to write the extended address value into the HPIA register. The value is 0x00 0x00 since the address 0x7000 lies within the first page of DSP memory.

Vend Req	Req	0xB7	Value	0x0000	Index	0xBEEF	Length	2	Dir	0 OUT	Hex Bytes	00 00
----------	-----	------	-------	--------	-------	--------	--------	---	-----	-------	-----------	-------

- Select Endpoint 2 OUT as the "Pipe", press the "FileTrans .." button and select the 512_count.hex file. Click on "Open" and this action will send out 512 bytes of ramp data to the HPI. Since the auto-increment mode is used for data writes to the HPI, the first data value will appear at 0x7001 since the HPI pre-increments the address prior to the data write. If you wanted the first data value to appear at 0x7000, the address to write to the HPIA register would have been 0x6FFF.

You could have also used the "Bulk Trans" button to send a specific data pattern to the HPI.

BulkTrans	Pipe	2 : Endpoint 2 OUT	Length	512	Hex Bytes	5
ResetPipe	AbortPipe	FileTrans..	Pipe	2 : Endpoint 2 OUT		
Set IFace	Interface	0	AltSetting	1		

```

Write IOCTL passed
0000 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0010 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
0020 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
0030 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
0040 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
0050 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
0060 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
0070 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
0080 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
0090 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
  
```

The screen on the next page shows what you will see in the Memory window of Code Composer Studio, indicating that indeed the HPI write was successful.

Memory (Data: Hex - C Style)				
0x7000:	0x2CE3	0x0100	0x0302	0x0504
0x7004:	0x0706	0x0908	0x0B0A	0x0D0C
0x7008:	0x0F0E	0x1110	0x1312	0x1514
0x700C:	0x1716	0x1918	0x1B1A	0x1D1C
0x7010:	0x1F1E	0x2120	0x2322	0x2524
0x7014:	0x2726	0x2928	0x2B2A	0x2D2C
0x7018:	0x2F2E	0x3130	0x3332	0x3534
0x701C:	0x3736	0x3938	0x3B3A	0x3D3C
0x7020:	0x3F3E	0x4140	0x4342	0x4544
0x7024:	0x4746	0x4948	0x4B4A	0x4D4C
0x7028:	0x4F4E	0x5150	0x5352	0x5554
0x702C:	0x5756	0x5958	0x5B5A	0x5D5C
0x7030:	0x5F5E	0x6160	0x6362	0x6564
0x7034:	0x6766	0x6968	0x6B6A	0x6D6C

Step 3: Perform a Read from the HPI

To read from a specific address in the DSP, you should perform steps a. through e. as listed in Step 2. For HPI auto-increment reads, the DSP post-increments the address for a data read, so you can exactly specify the address to read from. In this instance, we will read back the data values previously written in step 2. We will read 768 bytes to illustrate that the FX2 firmware can handle short packets (since 768 is not an even divisor of 512).

- a. Perform a 0xB5 vendor OUT command to set the *Tcount* variable to 768 (0x0300). This sets up the GPIF to read 768 bytes from the HPI.

Vend Req	Req	0xb5	Value	0x0000	Index	0xBEEF	Length	2	Dir	OUT	Hex Bytes	03 00
----------	-----	------	-------	--------	-------	--------	--------	---	-----	-----	-----------	-------

- b. Pend an IN request for 768 bytes on Endpoint 6 IN.
- c. Perform a 0xB3 vendor IN command to set the *in_enable* flag to TRUE. 768 bytes worth of data should now be displayed on the EZ-USB Control Panel window. Note that the data values are the same as the ones previously written by the HPI write, thus proving that the read was a success.

Vend Req	Req	0xb3	Value	0x0000	Index	0xBEEF	Len
Iso Trans	Pipe		Packets	128	Size		
BulkTrans	Pipe	3 : Endpoint 6 IN	Length	768	Hex		
ResetPipe	AbortPipe	FileTrans..	Pipe	2 : Endpoint 2 OUT			
<div style="border: 1px solid black; height: 30px; width: 100%;"></div>							
Set IFace	Interface	0	AltSetting	1			
<pre> Vendor Request 0000 01 01 Vendor Request 0000 00 70 Vendor Request 0000 11 11 Vendor Request 0000 00 00 Vendor Request 0000 03 00 Vendor Request 0000 B3 Read IOCTL passed 0000 E3 2C 00 01 02 03 04 05 06 07 08 09 0A 0B 0010 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 0020 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 0030 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 0040 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 0050 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 0060 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 0070 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 0080 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 0090 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 00A0 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB 00B0 AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB 00C0 BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB </pre>							

Data value at
address 0x7000

Step 4: Bootloading the DSP code

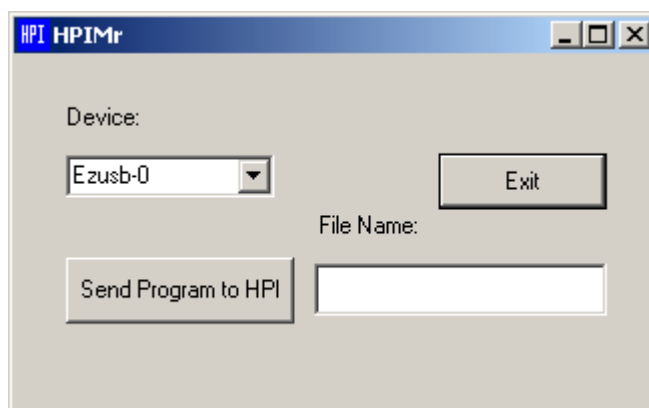
A host processor can download the DSP code via the HPI and “bootload” it by writing the entry point of the program to locations 0x7E and 0x7F in DSP data memory. This is a common method used to bootload a TI DSP. To force the 5416 to use the HPI as the bootload method, the HINT and INT2 pins are tied together. The host processor just needs to then download the code, write the entry point, and the DSP program starts running. For simplicity's sake, this example chose to use the LED code supplied in the 5416 DSK software. Once the bootload is complete, the user LED0 on the DSK board starts flashing.

The bootloading techniques used in this example are very similar to those explained in the TI application note SPRA382. Code Composer Studio generates a .out executable file that follows the Common Object File Format (COFF). This .out file cannot be simply downloaded to the DSP. The program and data sections need to be extracted by running a COFF Hex Extraction Utility on the .out file. This utility is available with the SPRA382 application note. After typing in the command “coff_both -out led.out”, the utility creates a hex listing that is called led.out.c.

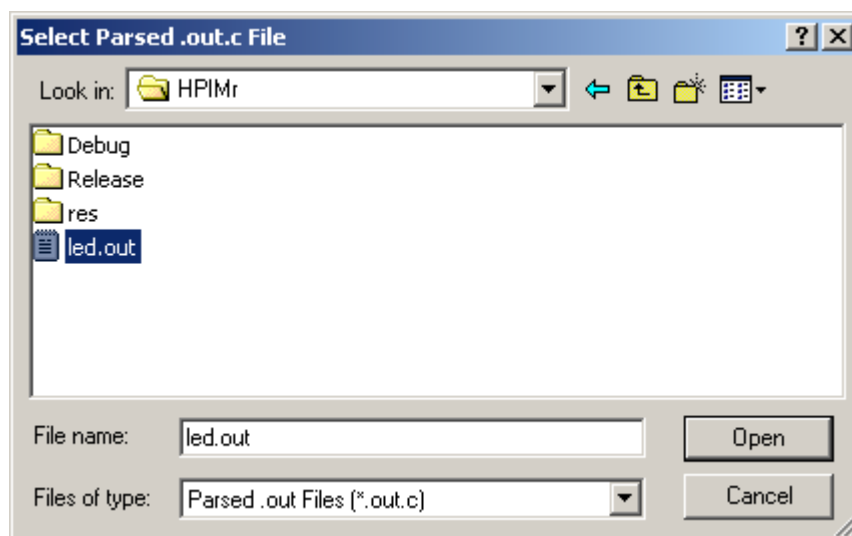
A PC application created in MS VC++ 6.0 called HPIMr (short for HPI Manager) reads in this hex listing, determines the start address of each section, and writes each section of code/data to the FX2's Endpoint 2 OUT. Prior to writing each block of code/data, the sequence of 0xB6 and 0xB7 vendor OUT commands are performed to write to the HPIC/HPIA registers (similar to Step 2). The source code for the HPIMr utility is provided if you are interested in seeing how this is done. This utility can also serve as another host application example.

These are the steps you should follow to download the LED example:

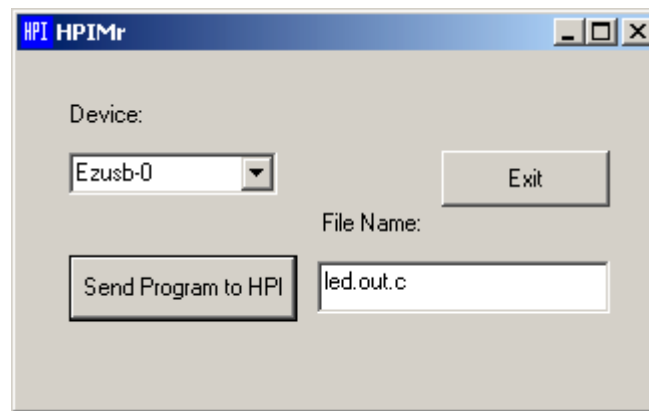
- a. Plug in the 5416 DSK board, plug in the FX2 development board.
- b. Download the FX2_to_TI5416_HPI.hex firmware.
- c. Run the HPIMr.exe utility. You should see the following screen if an FX2 development board is plugged in.



- d. Next, click on the “Send Program to HPI” button and select the led.out.c file. Click on “Open”.



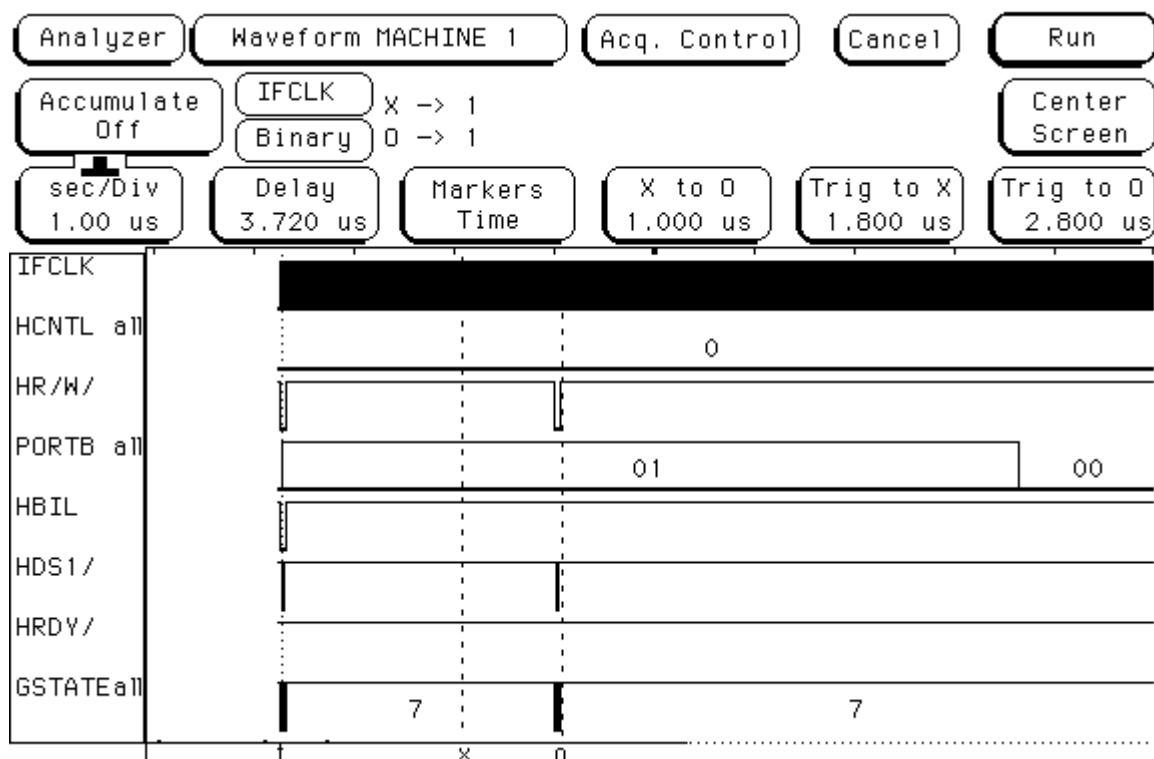
You should now see the user LED0 flashing on the 5416 DSK board.



Logic Analyzer Waveforms for DSP Example

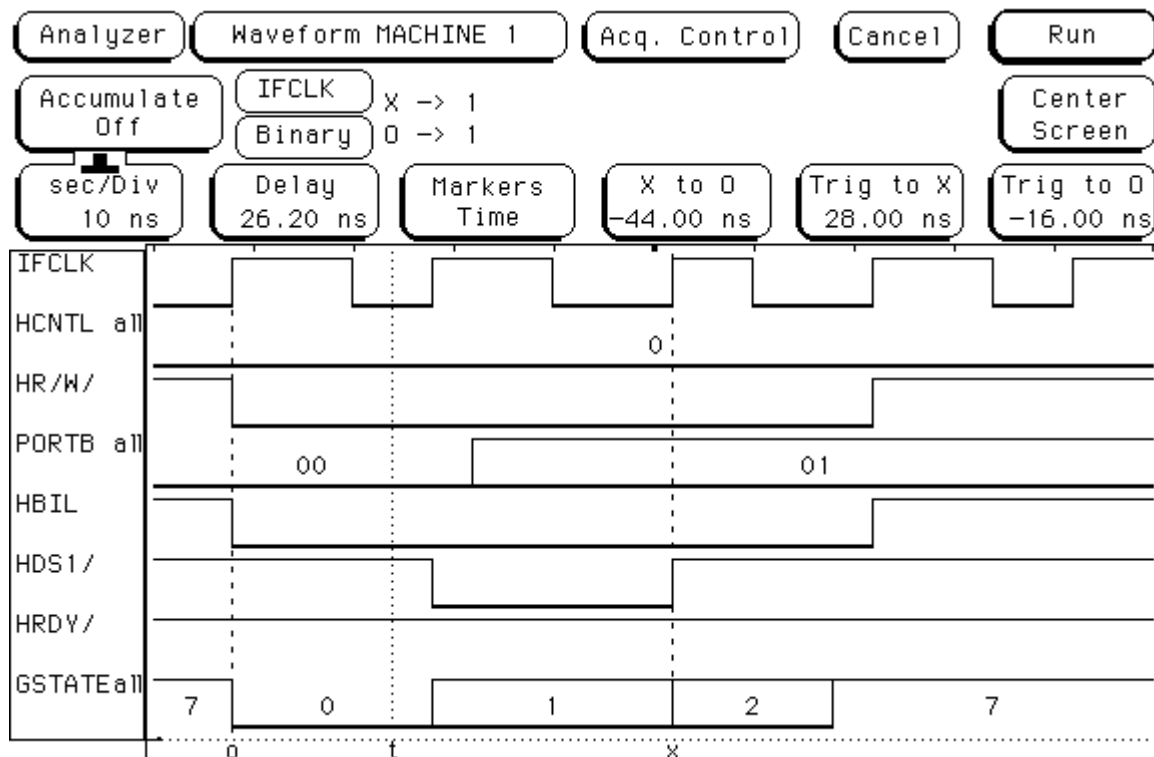
This section presents the waveforms you should see on the logic analyzer as the DSP example is run. An HP1660C Logic Analyzer was used to capture the waveforms.

Writing to the HPIC register: Zoomed out view



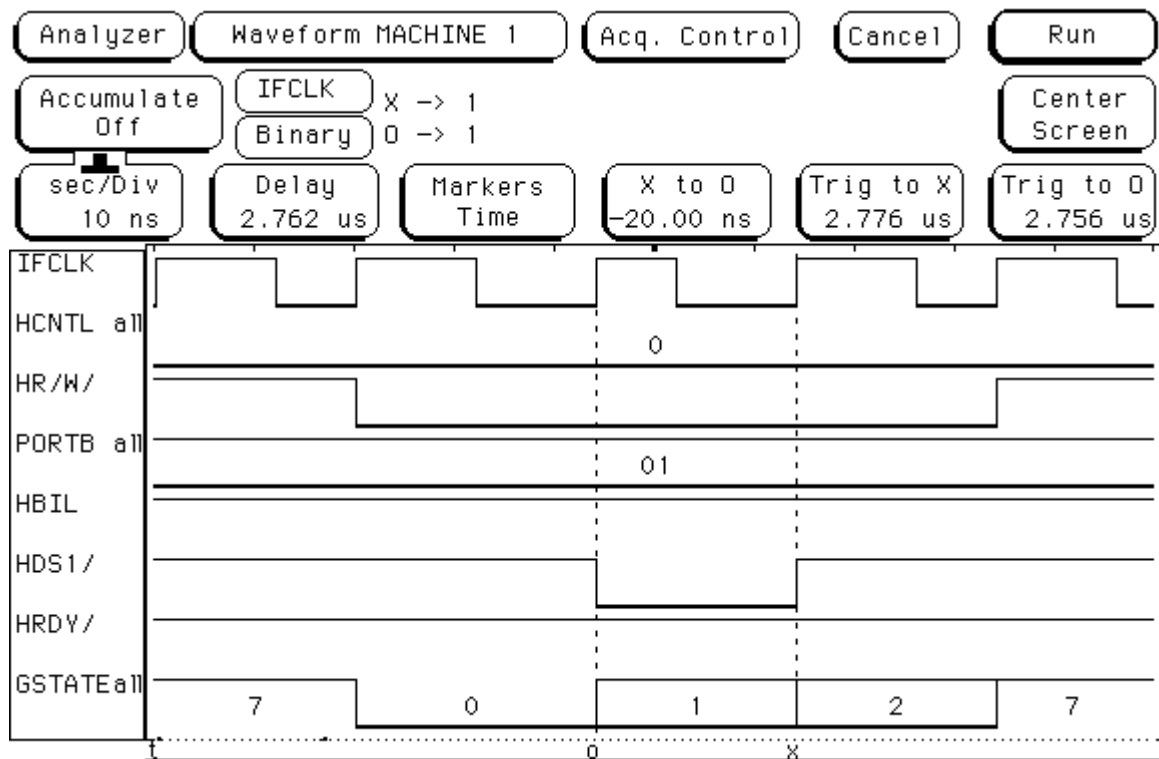
This trace shows what you should see when a 0xB6 vendor OUT command is performed to write to the HPIC register (HCNTRL = 00). The first byte of the HPI transfer is followed by IDLE time, where the FX2 firmware is switching to the GPIF single write waveform that writes the second byte of the HPI transfer. A similar waveform can be observed when a 0xB7 vendor OUT command is performed to write to the HPIA register.

Writing to the HPIC register: Close-up view of 1st byte of HPI transfer



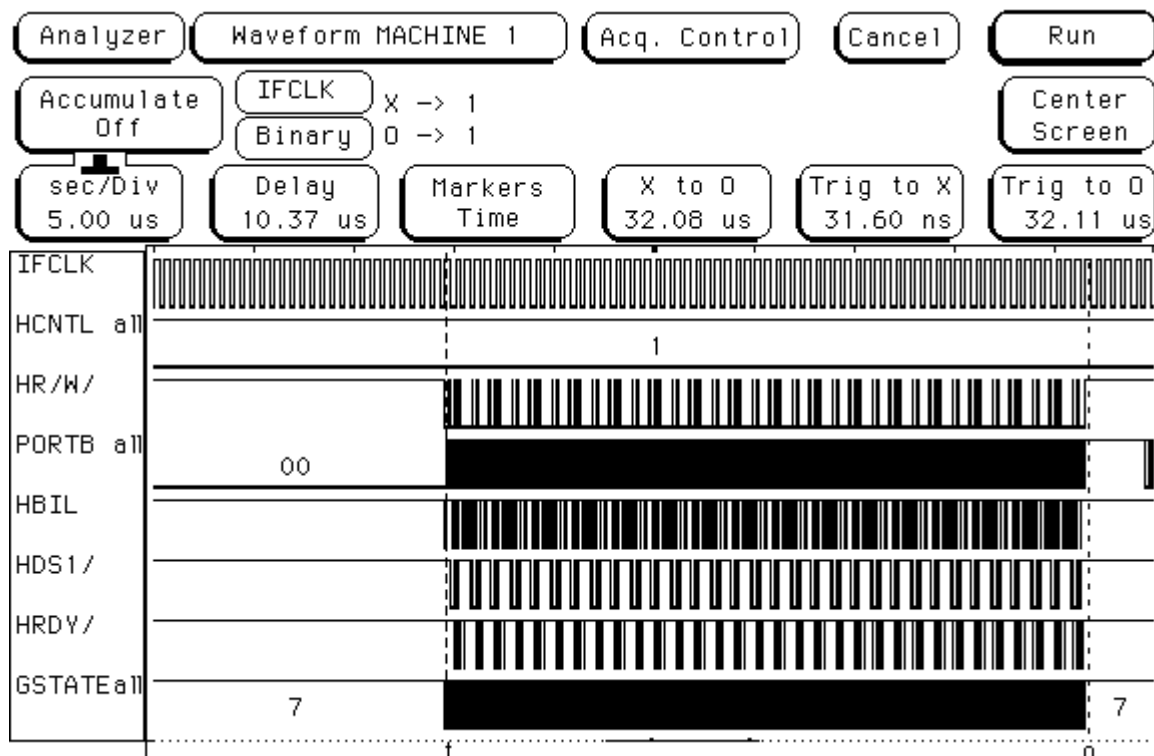
Here we see a close-up view of the first byte of a HPI write to the HPIC register. A write operation is signified by $\overline{\text{HR/W}}$ being driven LOW. $\overline{\text{HBIL}}$ is also driven LOW to signify that this is the first byte of the HPI transfer. The timing shown here adheres to the HPI8 Mode Timing Requirements in the 5416 data sheet. A similar waveform can be observed when a 0xB7 vendor OUT command is performed to write to the HPIA register.

Writing to the HPIC register: Close-up view of 2nd byte of HPI transfer



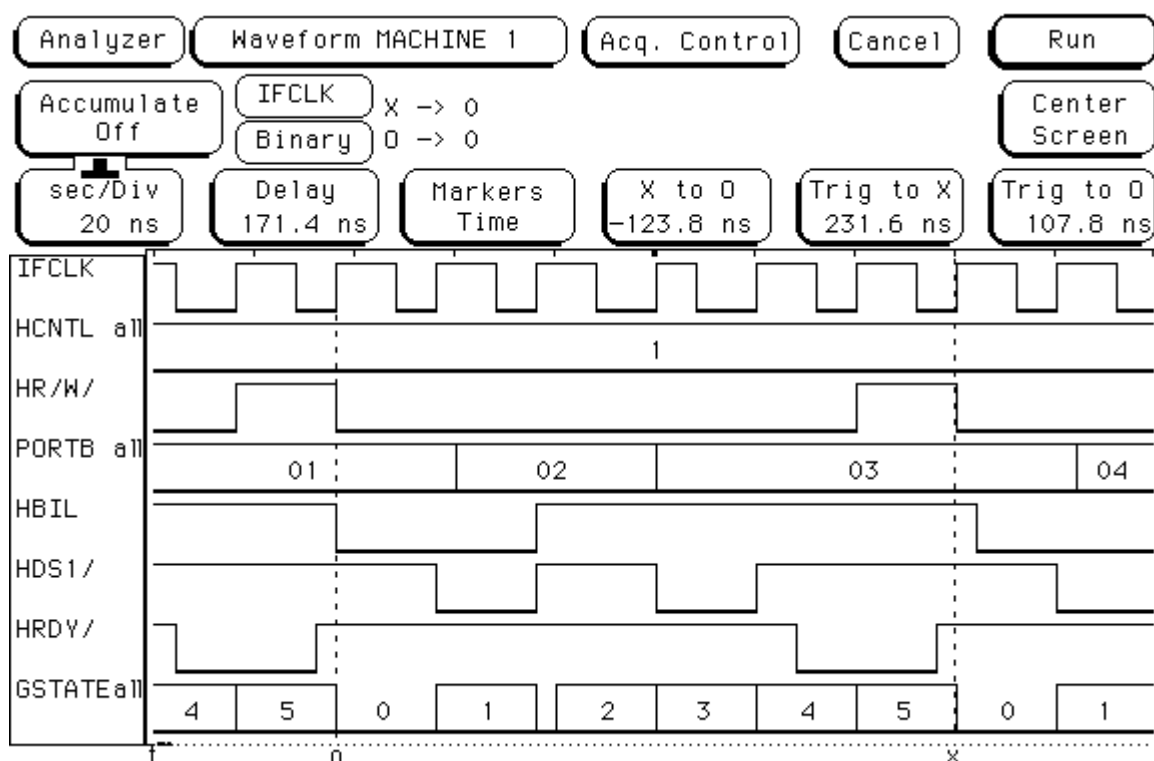
Here we see a close-up view of the second byte of a HPI write to the HPIC register. A write operation is signified by $\overline{\text{HR/W}}$ being driven LOW. HBIL is now driven HIGH to signify that this is the second byte of the HPI transfer. The timing shown here adheres to the HPI8 Mode Timing Requirements in the 5416 data sheet. A similar waveform can be observed when a 0xB7 vendor OUT command is performed to write to the HPIA register.

FIFO Write to HPI: Zoomed out view



This is a snapshot of the activity when the FX2 performs a burst write to the HPI using GPIF FIFO Write transactions. A similar waveform can be observed when the FX2 performs a burst read from the HPI using GPIF FIFO Read transactions.

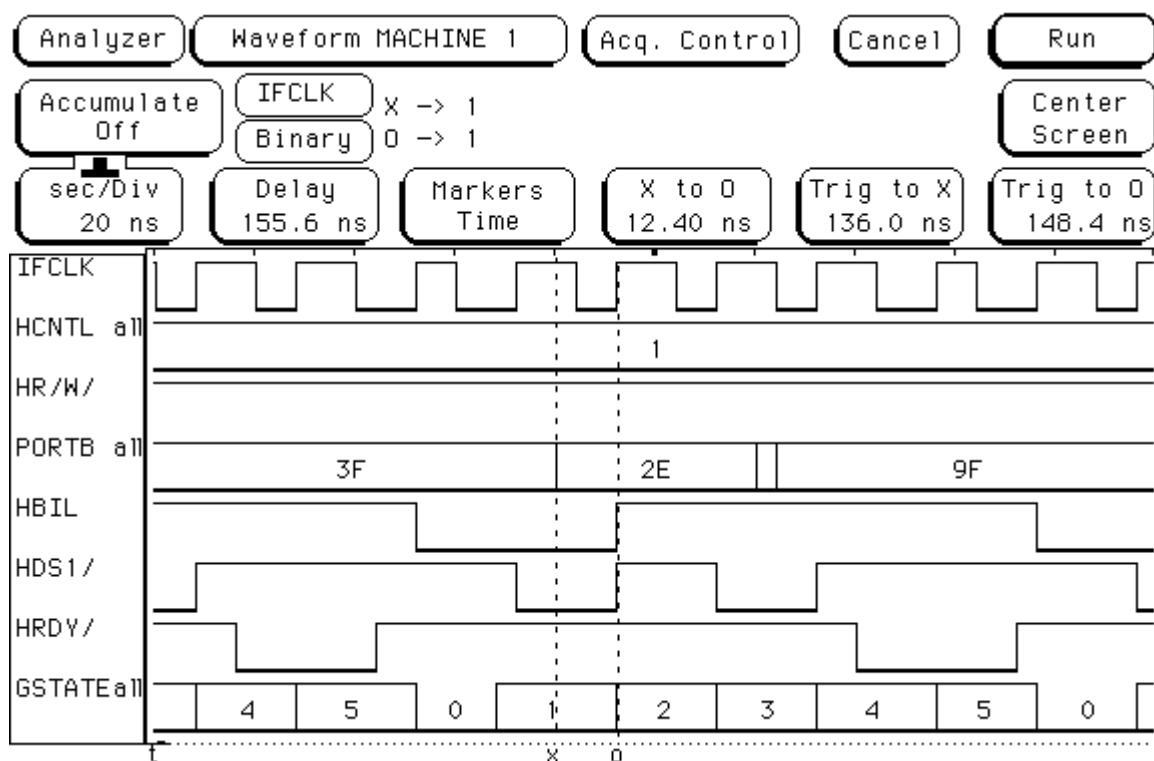
FIFO Write to HPI: Close-up view



Here we see a close-up view of a write to the HPI using GPIF FIFO Write transactions. S0–S5 marks the duration of a complete HPI write access. Since it is a write, the $\overline{\text{HR/W/}}$ signal is driven LOW for the duration of the transfer. In S0–S1, the HBIL signal is driven LOW to transfer the first byte, and in S2–S3 the HBIL signal is driven HIGH to transfer the second byte. The timing here conforms to the HPI8 Mode Timing Requirements in the 5416 data sheet.

Notice that the $\overline{\text{HRDY}}$ signal goes LOW after the second byte is transferred. The next HPI write does not start until the $\overline{\text{HRDY}}$ is HIGH again (indicating that the HPI is ready for the next write). S4 and S5 allow enough time for the $\overline{\text{HRDY}}$ signal to become HIGH again. Thus it was not necessary to check for $\overline{\text{HRDY}}$ in the GPIF waveform logic before starting the next HPI write sequence. S5 is the decision point state that uses the GPIF transaction count expiration flag (TCXpire) to determine whether or not the data burst should continue.

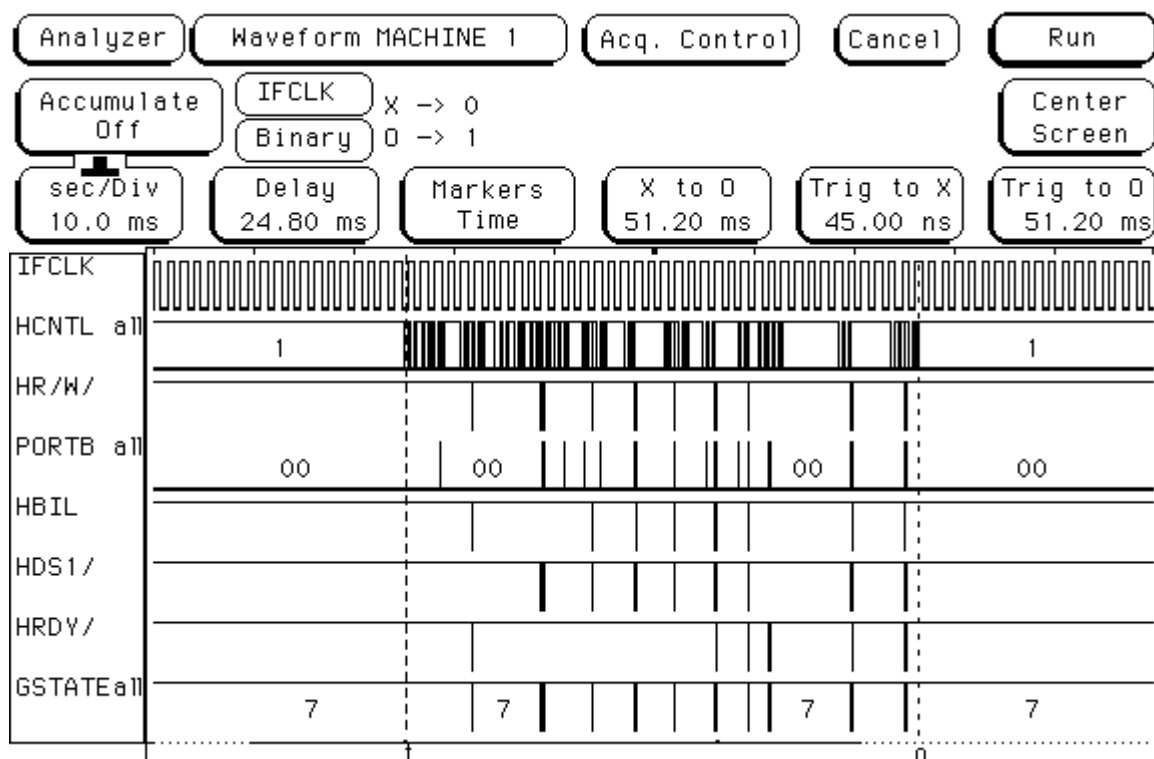
FIFO Read from HPI: Close-up view



Here we see a close-up view of a read from the HPI using GPIF FIFO Read transactions. S0–S5 marks the duration of a complete HPI read access. Since it is a read, the HR/W signal is driven HIGH for the duration of the transfer. In S0–S1, the HBIL signal is driven LOW to transfer the first byte, and in S2–S3 the HBIL signal is driven HIGH to transfer the second byte. The timing here conforms to the HPI8 Mode Timing Requirements in the 5416 data sheet.

Notice that the $\overline{\text{HRDY}}$ signal goes LOW after the second byte is transferred. The next HPI read does not start until the $\overline{\text{HRDY}}$ is HIGH again (indicating that the HPI is ready for the next read). S4 and S5 allow enough time for the $\overline{\text{HRDY}}$ signal to become HIGH again. Thus, it was not necessary to check for $\overline{\text{HRDY}}$ in the GPIF waveform logic before starting the next HPI read sequence. S5 is the decision point state that uses the GPIF transaction count expiration flag (TCXpire) to determine whether or not the data burst should continue.

Downloading DSP code



This is a snapshot of the activity when the FX2 downloads the DSP code to the 5416 DSK board.

Summary

This design example of an FX2 GPIF interface to a TI 5416 DSP's HPI has given you another concrete example of how the FX2 GPIF can be used in a practical application. This DSP example builds upon the external FIFO example, and extends your understanding of how to create GPIF waveform descriptors and program the GPIF to perform reads and writes over the physical interface. Because the HPI protocol is significantly more complex than that of an external FIFO, you can appreciate what it takes to create a full featured GPIF application, and apply the techniques presented here to solve applications that have slave devices with more sophisticated protocols.

4.3 Benchmark Set-up

Gateway E-4200 Desktop:

- Pentium® III 700 MHz, 384 MB RAM
- Microsoft Windows® 2000 Professional v5.00.2195, Service Pack 2
- NEC to PCI USB Enhanced Host Controller B1
- Microsoft USB 2.0 Drivers v5.1.2600.0
- Two USB 2.0 enabled ports
- Two EZ-USB FX2 Development Boards (Rev C)/Breadboards
- TI 5416 DSK Board (Rev C)
- HP1660C Logic Analyzer
- CATC Advisor USB Bus Analyzer

5.0 References

1. "CY7C68013 EZ-USB FX2 USB Microcontroller Data Sheet Rev. *B." *Cypress Semiconductor Corporation* 21 June 2002.
2. "The EZ-USB FX2 Technical Reference Manual Version 2.1." *Cypress Semiconductor Corporation* 2001.
3. "Universal Serial Bus Specification Revision 2.0." *USB Implementers' Forum* 27 April 2000.
4. "CY7C4255/CY7C4265 Data Sheet Rev. **." *Cypress Semiconductor Corporation* 20 November 2000.
5. "Bootloading the TMS320VC5402 in HPI Mode SPRA382" *Texas Instruments* April 2002.
6. "TMS320VC5416 Bootloader SPRA602D." *Texas Instruments* March 2002.
7. "TMS320C54x DSP Reference Set Volume 5: Enhanced Peripherals SPRU302." *Texas Instruments* June 1999.
8. "TMS320VC5416 Fixed-Point Digital Signal Processor Data Manual SPRS095H." *Texas Instruments* December 2001.
9. "TMS320VC5416 DSK Technical Reference Rev. A" *Spectrum Digital, Inc.* March 2002

EZ-USB is a registered trademark and FX2 is a trademark of Cypress Semiconductor Corporation.

Windows is a registered trademark of Microsoft Corporation.

Pentium is a registered trademark of Intel Corporation.

Code Composer Studio is trademark of Texas Instruments.

All other product or company names mentioned in the document may be the trademarks of their respective owners.

approved dsg 11/21/02