

Implementing an FX2LP™-FPGA Interface

Author: Prajith Cheerakkoda

Associated Project: Yes

Associated Part Family: CY7C68013A/CY7C68014A/CY7C68015A

Software Version: None

Related Application Notes: [AN65209](#), [AN63620](#)

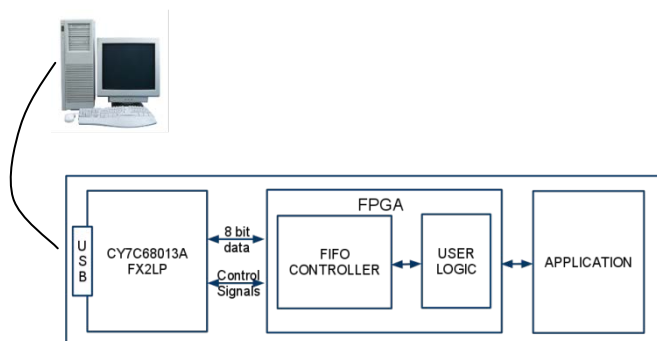
If you have a question, or need help with this application note, contact the author at prji@cypress.com

AN61345 provides a sample project to interface an FX2LP™ (CY7C68013A) with FPGA. The interface, described in a sample implementation, adds High-Speed USB connectivity to applications such as data acquisition, industrial control and monitoring, and image processing. Project provided with this Application Note is implemented and tested with Xilinx Virtex 5 and Spartan 3E FPGAs.

Introduction

An FX2LP™-FPGA interface is implemented to add High-Speed USB connectivity for FPGA based applications, such as data acquisition, industrial control and monitoring, and image processing. The FX2LP acts in Slave-FIFO mode and the FPGA acts as the master. This Application Note also gives a sample FX2LP firmware for Slave-FIFO implementation and a sample VHDL and Verilog project for FPGA implementation.

Figure 1. FX2LP-FPGA System



The endpoint FIFOs of FX2LP can be configured to act as a high-speed data path between a USB host and external data processing logic. The FX2LP gives two options for implementing this interface: general programmable interface (GPIF) mode and Slave-FIFO mode.

- **GPIF Mode:** In this mode, FX2LP acts as a master to an external system and generates all the necessary control signals to read and write data. GPIF mode is usually preferred when the external system is not intelligent enough to act as a master to FX2LP. In this

case, most of the complexity of the interface implementation resides in the FX2LP firmware.

- **Slave-FIFO Mode:** In this mode, the external data processing logic is intelligent enough to generate the necessary read and write control signals, and it can act as a master to FX2LP. Here, FX2LP is in the Slave-FIFO mode.

This Application Note describes the implementation of synchronous 8-bit Slave-FIFO on FX2LP. This Application Note includes Verilog and VHDL sample projects that shows how to interface to FX2LP's slave FIFO interface.

The project folder associated_project.zip includes following projects

1. Verilog code for data streaming (Implementation for Spartan 3E)*
2. Verilog code for data loopback (Implementation for Spartan 3E)*
3. VHDL code for data streaming (Implementation for Spartan 3E and Virtex 5)*
4. VHDL code for data loopback (Implementation for Spartan 3E and Virtex 5)*
5. FX2LP firmware for slave-FIFO implementation

*Note: Sample projects are implemented and tested in Xilinx Virtex 5 / Spartan 3E FPGAs. But codes provided with this Application Note are standard Verilog/VHDL codes. Hence, customers can use these files as sample codes for implementations to any FPGAs. Customers need to select correct FPGA device while synthesizing and implementing the project.

It is assumed that the reader has a familiarity with chapter 9 (Slave FIFOs) of the [EZ-USB Technical Reference](#)

Manual, Verilog/VHDL coding and FPGA synthesis and implementation tools.

Hardware Connections

The following figure illustrates the hardware connections required for interfacing the FX2LP to the FPGA.

Figure 2. Hardware Connections Diagram

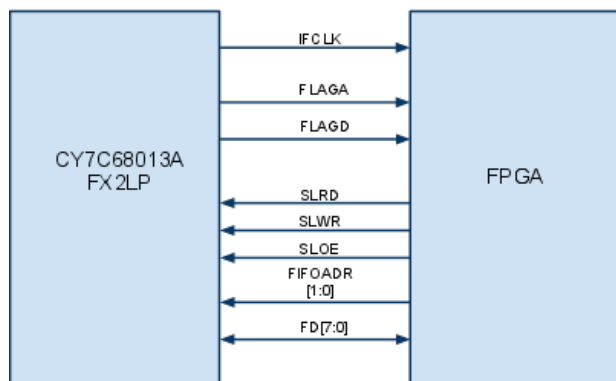


Table 1 shows the pins of FX2LP that are relevant for the implementation of the Slave-FIFO interface, as shown in Figure 2.

Table 1. Slave FIFO Pins

Pin Name	Description
SLRD	SLRD pin should be asserted by the master in order to read data from the FIFO.
SLWR	SLWR pin should be asserted by the master in order to write data to the FIFO.
SLOE	This is the enable signal for the FIFO's output driver.
FIFOADR[1:0]	These signals select the active endpoint.
FD[7:0]	The data bus.
FlagA/FlagB/ FlagC/FlagD	These flags are used by the FIFO to indicate status (Full, Empty, and Programmable).
IFCLK	The clock for the synchronous interface with 48 MHz and is generated by FX2LP.

Firmware Implementation

The FX2LP firmware was developed on Keil uVision 2.0 IDE, the evaluation version of the firmware is present in the [CY3684](#) DVK contents. This section describes the coding required for implementing the Slave-FIFO interface on the slave (FX2LP) and the master side (FPGA).

FX2LP Code Architecture

The firmware configures auto mode for both the IN and OUT endpoint FIFOs. This means that the packets are committed automatically from the peripheral domain to the USB domain for OUT transfers and vice versa for IN transfers. The 8051 CPU is not involved in committing packets. The Application note [AN4067 Endpoint FIFO Architecture of EZ-USB FX1/FX2](#) explains in detail about the auto and manual mode, and the different data domains. As we are using bulk transfer in this application, we can configure endpoint buffer as Bulk. But based on the end application we can configure endpoint buffer type as Interrupt, Control, or Isochronous in the descriptor file.

Because the Slave works in AUTO mode, no code is required for data transfer to and from the master, except for the initialization of the following registers (shown in Table 2).

Table 2. Slave-FIFO Configuration Registers

Register Name	Register Description
IFCONFIG	Configure the IFCONFIG register to place FX2LP in Slave-FIFO mode and to output the 48 MHz IFCLK to use it as the clock for the synchronous interface.
PINFLAGSAB/ PINFLAGSCD	Configure the FIFO Flags. FIFO Flags can be configured either to act in fixed mode or indexed mode. In indexed mode, Flag A, Flag B, and Flag C are automatically configured as the Empty, Full, and Programmable flags respectively, for whichever endpoint the FIFOADR [1:0] lines are pointing at a particular instant. In fixed mode, each of the four flags can be configured to act as Empty, Full, or Programmable for any of the endpoints by writing into the programmable flag registers.
EP2CFG/ EP6CFG	Configure EP2 as OUT, 512 bytes, quad-buffered endpoint and EP6 as IN, 512 bytes, quad-buffered endpoint.
EP2FIFOCFG/ EP6FIFOCFG	Configure the FIFO as byte wide and operating in AUTO mode
EP6AUTOINLENH/ EP6AUTOINLENL	When the number of bytes in the EP6 FIFO becomes equal to the value specified by these registers, the packet gets auto-committed. The value specified should be less than or equal to the maximum packet size specified in the endpoint descriptor.

As shown here, the code for initializing these registers are to be placed in TD_Init().

```

void TD_Init( void )
{ // Called once at startup

    CPUCS = 0x12; // CLKSPD[1:0]=10, for 48MHz operation, output CLKOUT
    IFCONFIG = 0xE3;
    // 48 MHz internal clock source, drive IFCLK, synchronous slave FIFO mode

    PINFLAGSAB = 0x08;          // FLAGA - EP2 Empty flag
    SYNCDELAY;
    PINFLAGSCD = 0xE0;          // FLAGD - EP6 FULL flag

    SYNCDELAY;

    PORTACFG |= 0x80;

    EP4CFG = 0x02;              //clear the valid bits on ep4 and ep8
    SYNCDELAY;
    EP8CFG = 0x02;
    SYNCDELAY;
    EP2CFG = 0xA0;              // OUT, 512-bytes, 4x, bulk
    SYNCDELAY;
    EP6CFG = 0xE0;              // IN, 512-bytes, 4x, bulk

    SYNCDELAY;
    FIFORESET = 0x80;           // activate NAK-ALL to avoid race conditions
    SYNCDELAY;                 // see TRM section 15.14
    FIFORESET = 0x02;           // reset, FIFO 2
    SYNCDELAY;                 //
    FIFORESET = 0x04;           // reset, FIFO 4
    SYNCDELAY;                 //
    FIFORESET = 0x06;           // reset, FIFO 6
    SYNCDELAY;                 //
    FIFORESET = 0x08;           // reset, FIFO 8
    SYNCDELAY;                 //
    FIFORESET = 0x00;           // deactivate NAK-ALL

    // handle the case where we were already in AUTO mode...
    // ...for example: back to back firmware downloads...
    SYNCDELAY;                 //
    EP2FIFOCFG = 0x00;          // AUTOOUT=0, WORDWIDE=0

    // core needs to see AUTOOUT=0 to AUTOOUT=1 switch to arm endp's

    SYNCDELAY;                 //
    EP2FIFOCFG = 0x10;          // AUTOOUT=1, WORDWIDE=0

    SYNCDELAY;                 //
    EP6FIFOCFG = 0x0C;          // AUTOIN=1, ZEROLENIN=1, WORDWIDE=0, INFM = 1;

    SYNCDELAY;

}

```

FPGA Code Architecture

The main function of the HDL code is to monitor the Full and Empty flags of the Slave FIFO, and then read and writes into the flags accordingly.

Two HDL projects are implemented to show how to configure an FPGA to act as a master to an FX2LP Slave-FIFO. Xilinx ISE Design Suite is used for code development.

Data Loopback

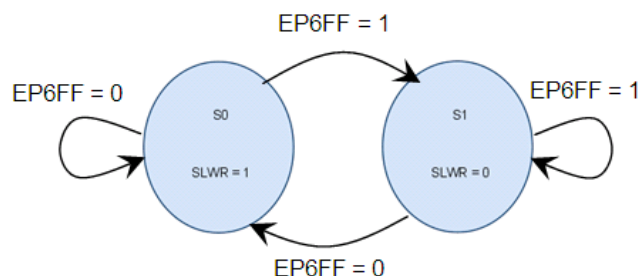
Any data written into the EP2 FIFO of FX2LP is read by the FPGA, incremented and written into the EP6 FIFO. The associated project folder, *Loopback.zip* (attached with the application note) contains this project.

Data Streaming

The FPGA monitors the Full flag of EP6 and continuously writes incrementing data into the flag. While writing data into the EP6 FIFO, the FPGA pauses the writing as soon as the Full flag gets asserted, and resumes the writing when the flag gets deasserted. The associated project folder, *Streamin.zip* contains this project.

State Diagrams

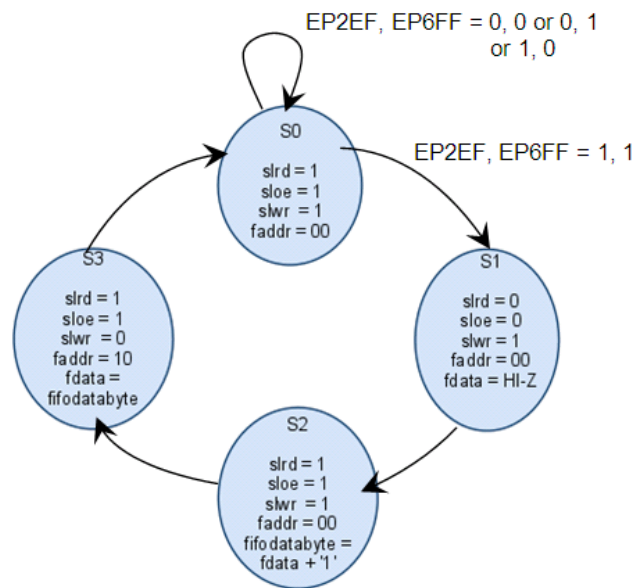
Figure 3. Data Streaming



There are two distinct states in this state machine: S0 and S1. S0 is the idle state, where SLWR stays high. As long

as the EP6 Full flag stays low (asserted), the state machine remains in S0. After the Full flag goes high, there is a transition from state S0 to S1. S1 is the active state for the SLWR signal, which gets asserted. The state machine stays in S1 until the EP6 Full flag gets asserted again.

Figure 4. Data Loopback



State S0 is the idle state in which the SLRD, SLOE, and SLWR signals are high (deasserted). The state machine stays in S0 until both EP2 FIFO Empty flag and EP6 FIFO Full flags are deasserted. Even if there is data in the EP2 FIFO, unless there is space in EP6, the state machine does not move to S1. In S1, both SLRD and SLOE are asserted and the data bus is placed in so that FX2LP can drive it. In S2, the data bus is read and the received data is incremented. In S3, the FADDR pins select EP6 and the SLWR signal is asserted to write the incremented data into the EP6 FIFO.

The following VHDL code snippet shows the state machine implementation for data streaming:

```

process(clk)
variable fdatawe : natural := 0;
variable fifodatabyte : STD_LOGIC_VECTOR(7 downto 0) := "00000000";
begin
    if(rising_edge(clk)) then

        case MasterState(3 downto 0) is

            when A =>

                sloe_i <= '1';      -- IDLE STATE
                faddr_i <= "10";
                slrd_i  <= '1';
                slwr_i  <= '1';
        
```

```

MasterState <= E;
fdatawe := 0;
gstate_i <= "0001";

when E =>

    faddr_i <= "10";
    slrd_i  <= '1';
    sloe_i <= '1';
    if (flagd = '1') then          -- if Full flag is in a deasserted state

        slwr_i <= '0';              -- assert slave write control signal
        fdatawe := 0 ;
        fdata <= fifodatabyte;
        fifodatabyte := fifodatabyte + '1';
        MasterState <= E;          -- stay in state E
    else
        slwr_i <= '1';
        MasterState <= A;          -- when Full flag gets asserted, move to state A
    end if;

    gstate_i <= "0110";

when others =>

    faddr_i <= "00";

    slrd_i  <= '1';                --if an undefined state move to IDLE
    sloe_i <= '1';
    slwr_i <= '1';
    gstate_i <= "1000";
    MasterState <= A;
end case;
end if;
end process;
end rtl;

```

The following VHDL code snippet implements data loopback:

```

process(clk)

variable fifodatabyte : STD_LOGIC_VECTOR(7 downto 0);

begin
    if(rising_edge(clk)) then

        case MasterState(3 downto 0) is

            when A =>

                sloe_i <= '1';      -- IDLE STATE
                faddr_i <= "00";
                slrd_i  <= '1';
                slwr_i  <= '1';
                MasterState <= B;
                gstate_i <= "0001";

            when B =>

```

```
faddr_i <= "00";
if (( flaga = '1' ) AND (flagd = '1') ) then
    --IF EP2 is not empty and EP6 is not full and there is data waiting to be written
    into EP6,
    --assert SLRD to read a byte from EP2
    slrd_i <= '0';
    sloe_i <= '0';
    fdata <= "ZZZZZZZZ";
    MasterState <= C;
else
    slrd_i <= '1';
    sloe_i <= '1';
    MasterState <= A;
end if;
slwr_i <= '1';
gstate_i <= "0010";

when C =>

    fifodatabyte := fdata + '1';    --data processing state

    faddr_i <= "00";
    slrd_i <= '1';
    slwr_i <= '1';
    sloe_i <= '1';

    gstate_i <= "0100";
    MasterState <= D;

when D =>

    fdata <= fifodatabyte;    --place the data on the data bus
    faddr_i <= "10";
    slrd_i <= '1';
    slwr_i <= '1';
    sloe_i <= '1';
    gstate_i <= "0101";
    MasterState <= E;

when E =>

    faddr_i <= "10";
    slrd_i <= '1';
    sloe_i <= '1';
    if (flagd = '1') then
        slwr_i <= '0';    --assert slave write control signal
    else
        slwr_i <= '1';
    end if;

    gstate_i <= "0110";
    MasterState <= A;

when others =>

    faddr_i <= "00";

    slrd_i <= '1';
    sloe_i <= '1';
    slwr_i <= '1';    --if an undefined state move to IDLE
```

```

        gstate_i <= "1000";
        MasterState <= A;
      end case;
    end if;
  end process;
end rtl;

```

The following Verilog code snippet implements data loopback:

```

module fpga_master(fdata, faddr, gstate, slrd, slwr, flagd, flaga, clk, sloe);

  inout [7:0]      fdata; // FIFO data lines.
  output [1:0]     faddr; // FIFO select lines
  output [2:0]     gstate; // gstate
  output          slrd; // Read control line
  output          slwr; // Write control line

  input           flagd; // EP6 Full Flag
  input           flaga; // EP2 Empty Flag
  input           clk; // Interface Clock
  output          sloe; // Slave Output Enable Control

  reg [1:0]       faddr_i;
  reg [7:0]       fdata_i;
  reg [7:0]       temp;
  reg            slrd_i;
  reg            slwr_i;

  reg [3:0]       MasterState = 4'b0000; //Counter to sequence the fifo signals.

  reg            sloe_i;
  reg [2:0]       gstate_i;

  parameter [3:0] A = 4'b0000;
  parameter [3:0] B = 4'b0001;
  parameter [3:0] C = 4'b0010;
  parameter [3:0] D = 4'b0011;
  parameter [3:0] E = 4'b0100;
  parameter [3:0] F = 4'b0101;
  parameter [3:0] G = 4'b0110;
  parameter [3:0] H = 4'b0111;

  assign slrd = slrd_i;
  assign slwr = slwr_i;

  assign faddr = faddr_i;
  assign sloe = sloe_i;
  assign fdata = fdata_i;
  assign gstate = gstate_i;

  always @(posedge clk)
  begin: test_proc
    case (MasterState[3:0])
      A :
        begin
          sloe_i <= 1'b1;

```

```

        faddr_i <= 2'b00;           //IDLE STATE
        slrd_i <= 1'b1;
        slwr_i <= 1'b1;
        MasterState <= B;
        gstate_i <= 3'b001;
    end

    B :
    begin
        faddr_i <= 2'b00;
        if (flaga == 1'b1) //IF EP2 is not empty and EP6 is not full and there is
data waiting to be written into EP6
        begin
            if (flagd == 1'b1)
                begin
                    slrd_i <= 1'b0; //assert SLRD and SLOE to read
a byte from EP2
                    sloe_i <= 1'b0;
                    fdata_i <= 8'bz; //place fdata bus in high-
impedance state so that FX2LP can drive it
                    MasterState <= C;
                end
            end
        else
            begin
                slrd_i <= 1'b1;
                sloe_i <= 1'b1;
                MasterState <= A; //else go back to IDLE
            end
        end
        slwr_i <= 1'b1;
        gstate_i <= 3'b010;
    end

    C :
    begin
        temp <= fdata + 1'b1; //read the fdata bus and increment
        faddr_i <= 2'b00;
        slrd_i <= 1'b1;
        slwr_i <= 1'b1;
        sloe_i <= 1'b1;

        MasterState <= D;
        gstate_i <= 3'b011;
    end

    D :
    begin
        fdata_i <= temp; //place the incremented data on the fdata bus
        MasterState <= E;
        gstate_i <= 3'b011;
    end

    E :
    begin
        faddr_i <= 2'b10;
        slrd_i <= 1'b1;
        sloe_i <= 1'b1;
    end

```

```

        if (flagd == 1'b1)
            begin
                slwr_i <= 1'b0;    //assert slave write control signal
            end
            else
            begin
                slwr_i <= 1'b1;
            end
            MasterState <= A;
            gstate_i <= 3'b101;
        end
    default :
        begin
            faddr_i <= 2'b11;

            slrd_i <= 1'b1;
            sloe_i <= 1'b1;
            slwr_i <= 1'b1;
            MasterState <= A; //if an undefined state move to IDLE
            gstate_i <= 3'b110;
        end
    endcase
end
endmodule

```

The following Verilog code snippet shows the state machine implementation for data streaming:

```

module fpga_master(fdata, faddr, slrd, slwr, flagd, flaga, clk, sloe);

    inout [7:0]    fdata; // FIFO data lines.
    output [1:0]   faddr; // FIFO select lines
    output         slrd;  // Read control line
    output         slwr;  // Write control line

    input          flagd; // EP6 Full Flag
    input          flaga; // EP2 Empty Flag
    input          clk;   // Interface Clock
    output         sloe;  // Slave Output Enable Control

    reg [7:0] fdata_i = 8'b00000000;

    reg [1:0] faddr_i;

    reg      slrd_i;
    reg      slwr_i;

    reg [3:0] MasterState; //Counter to sequence the FIFO signals

    reg      sloe_i;

    parameter [3:0] A = 4'b0000;
    parameter [3:0] B = 4'b0001;
    parameter [3:0] C = 4'b0010;
    parameter [3:0] D = 4'b0011;
    parameter [3:0] E = 4'b0100;

```

```

parameter [3:0] F = 4'b0101;
parameter [3:0] G = 4'b0110;
parameter [3:0] H = 4'b0111;

assign slrd = slrd_i;
assign slwr = slwr_i;

assign faddr = faddr_i;
assign sloe = sloe_i;
    assign fdata = fdata_i;

always @(posedge clk)
begin: test_proc

    case (MasterState[3:0])
        A :
            begin
                sloe_i <= 1'b1;
                faddr_i <= 2'b10;    //      IDLE STATE
                slrd_i <= 1'b1;
                slwr_i <= 1'b1;
                MasterState <= E;

            end

        E :
            begin
                faddr_i <= 2'b10;
                slrd_i <= 1'b1;
                sloe_i <= 1'b1;
                if (flagd == 1'b1) //if Full flag is in a deasserted state
                begin
                    //assert slave write control signal
                    slwr_i <= 1'b0;
                    fdata_i <= fdata_i + 8'b00000001;
                    MasterState <= E; //stay in state E
                end
                else
                begin
                    faddr_i <= 2'b00;
                    slwr_i <= 1'b1;
                    MasterState <= A; //when Full flag gets asserted, move to state A
                end
            end

        default :
            begin
                faddr_i <= 2'b00;

                slrd_i <= 1'b1;
                sloe_i <= 1'b1;
                slwr_i <= 1'b1;
                MasterState <= A; //if an undefined state move to IDLE
            end
    endcase
end

```

```

        end
    endcase
end
endmodule

```

FPGA Device Utilization (Virtex 5: xc5vlx220)

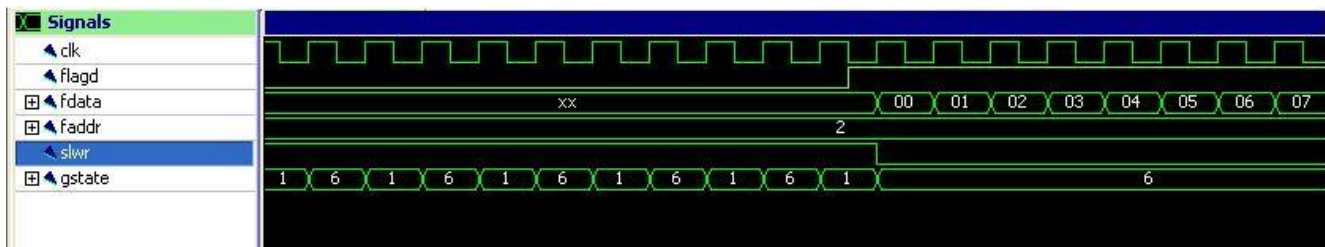
Device Utilization Summary:

Number of BUFs	1 out of 32	3%
Number of External IOBs	20 out of 800	2%
Number of LOCed IOBs	16 out of 20	80%
Number of Slice Registers	28 out of 138240	1%
Number used as Flip Flops	28	
Number used as Latches	0	
Number used as LatchThrus	0	
Number of Slice LUTs	22 out of 138240	1%
Number of Slice LUT-Flip Flop pairs	30 out of 138240	1%

Usage of FX2LP to implement the High-Speed USB interface saves valuable FPGA resources as compared to the usage of a USB IP core and PHY.

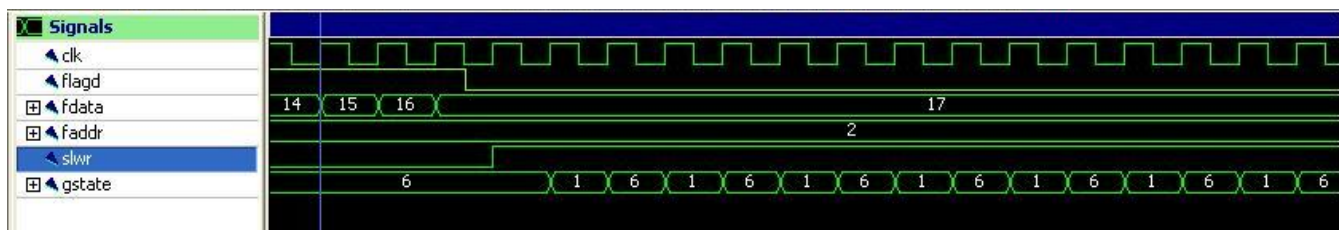
Simulation Waveforms

Figure 5. Data Streaming – Waveform Front End



This waveform shows the SLWR assertion when FLAGD, or the FULL flag, is deasserted.

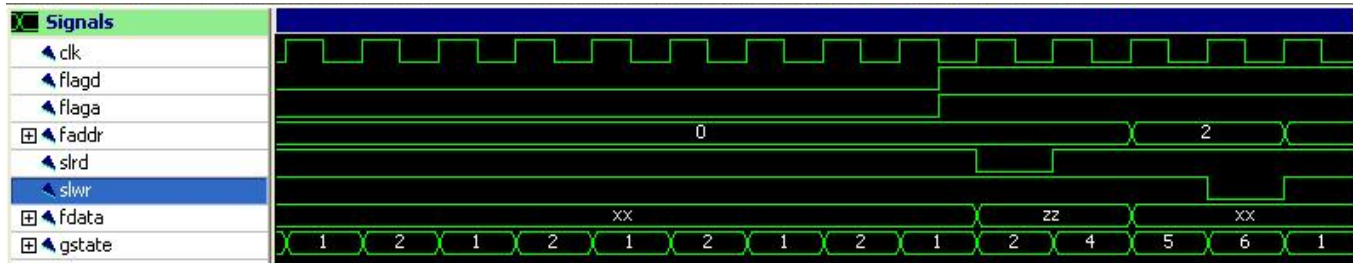
Figure 6. Data Streaming Back End



After the Full flag gets deasserted, SLWR gets deasserted only on the next rising edge of the clock. One additional byte is written into the FIFO on this clock edge and causes data loss. To take care of the loss, you can set the bit INFM in register

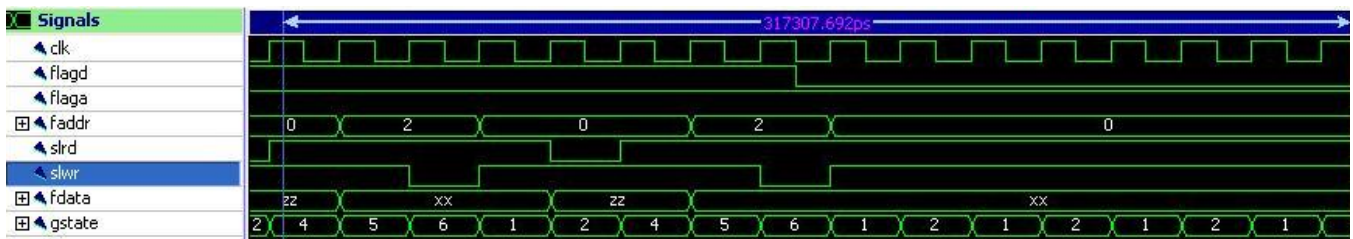
EP6FIFOCFG. When the INFM bit is set, the Full flag is asserted after the second-to-last byte is written into the FIFO. This ensures that only the correct number of bytes is written into the FIFO.

Figure 7. Data Loopback Front End



When both FLAGD (EP6 FF) and FLAGA (EP2 EF) are in the deasserted state, SLRD and SLWR are asserted alternately.

Figure 8. Data Loopback Back End



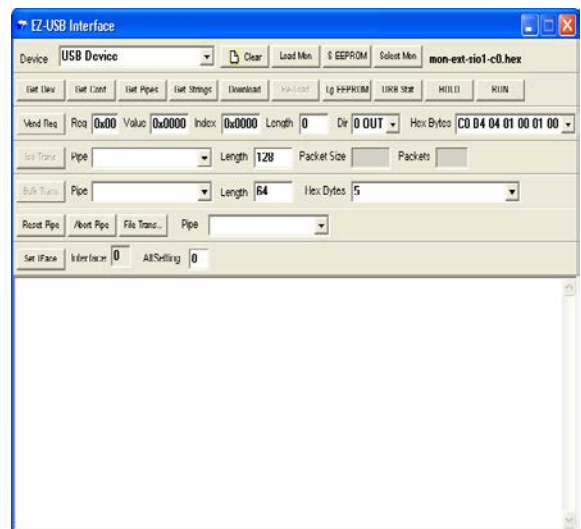
As soon as one of the flags is asserted, the SLWR-SLRD cycles end.

Testing the Project

You can either download firmware to FX2LP using CyConsole host application available in the SuiteUSB3.4 SDK; or write a script file to download the firmware automatically when the device is plugged in. [AN50963](#) explains the implementation of scripting for automatic firmware download.

For testing purposes, follow these steps to manually download the firmware to FX2LP using CyConsole, a development host application included in the SuiteUSB3.4 SDK.

- Connect the board to the PC. It enumerates with the default internal descriptors.
- Use the CyUSB.inf file inside the Drivers folder to bind with the device. For help on binding the driver, refer to *Cypress CyUsb.sys Programmer's Reference* inside the Drivers folder. [AN61465](#) contains detailed explanations on the generation of an .inf file.
- Start the CyConsole host application. In the **Options** menu, select **EZ-USB Interface**. The following window is displayed



- Click the **Download** button and navigate to *slave.hex* file in the associate project folder. Wait for the firmware to re-enumerate and reconnect as a Cypress EZ-USB device.
- A window is displayed prompting you to bind the driver. Use the appropriate *CyUSB.inf* file inside the Drivers folder to bind.

After the firmware download and enumeration of the device with VID/PID 0x04B4/1003, configure the

FPGA for loopback operation. The example implements a loopback on endpoints EP2OUT and EP6IN. Data sent from host to EP2 is read by the FPGA, incremented and written to the EP6 endpoint FIFO.

- For verifying loopback operation, in the **Bulk Trans** field select **pipe** as **0: Endpoint 2 OUT EP2OUT**, enter the number of bytes to transfer in the **Length** field (512 bytes is default length in this project) and then click the **Bulk Trans** button to perform the specified data transfer. Now read the EP6IN buffer in similar way to verify the incremented data written into it.

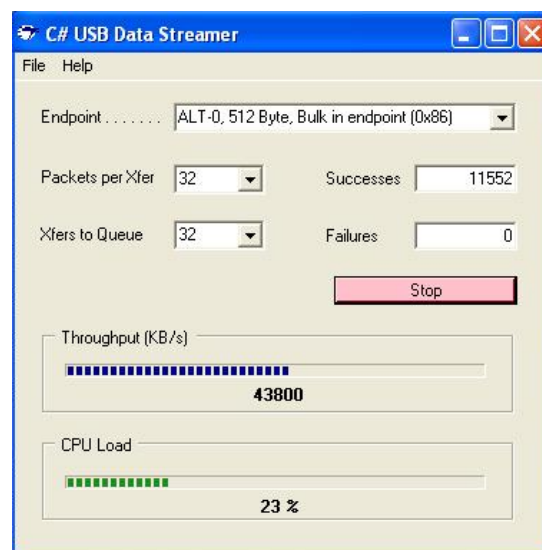
Throughput Measurement

The FPGA is configured with the code for data streaming and the interface throughput was measured with Cypress Streamer application, which is included in [SuiteUSB 3.4](#). A throughput of around 43 MB/s is measured on an Intel IHC9 host controller running Windows XP SP3 with the device bound to *CyUSB.sys* driver (version 3.4.2). More information on the throughput evaluation of a streaming device is present in the application note [AN4053 - Streaming Data Through Isochronous/Bulk Endpoints on EZ-USB FX2™ and EZ-USB FX2LP™](#).

A throughput of approximately 43 Mbytes/second* is measured.

*Note: Throughput value for real applications will vary depends on firmware overhead and FPGA overheads.

Figure 9. Data Streamer Application



Summary

This application note describes how to setup an FPGA-FX2LP interface with FX2LP configured in the Slave-FIFO mode. The associated projects contain the firmware for initializing FX2LP in Slave-FIFO mode as well as the HDL code for configuring an FPGA to act as master to FX2LP.

About the Author

Name: Prajith Cheerakkoda
 Title: Applications Engineer
 Contact: prji@cypress.com

Document History

Document Title: AN61345 – Implementing an FX2LP™-FPGA Interface

Document Number: 001-61345

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2896051	HRID	04/30/10	New Application note.
*A	3028509	HRID	09/13/10	Details on the PC specification on which the throughput measurement screenshot was captured. Link added to the application note detailing throughput measurement AN4053 Streaming Data Through Isochronous/Bulk Endpoints on EZ-USB FX2™ and EZ-USB FX2LP. Addition of FPGA details which were used for the project implementation in abstract.
*B	3049785	HRID	10/06/10	Post to external web.
*C	3187032	HRID	03/03/2011	Included details of software tools used and the associated projects.
*D	3539468	PRJI	03/01/2012	Updated the document with Virtex5 FPGA. Updated template. Modified abstract and major rewrite.
*E	3697826	PRJI	08/01/2012	Updated the document with verilog code for spartan-3E FPGA.
*F	3732422	PRJI	09/03/2012	Updated the associated project files.

Worldwide Sales and Design Support

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

Products

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

PSoC® Solutions

psoc.cypress.com/solutions

[PSoC 1](#) | [PSoC 3](#) | [PSoC 5](#)

Cypress Developer Community

[Community](#) | [Forums](#) | [Blogs](#) | [Video](#) | [Training](#)

Technical Support

cypress.com/go/support

EZ-USB FX2LP is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.



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

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

© Cypress Semiconductor Corporation, 2010-2012. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

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

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

Use may be limited by and subject to the applicable Cypress software license agreement.