AN63787

EZ-USB FX2LP GPIF And Slave FIFO Configuration Examples Using An 8-Bit Asynchronous Interface

Author: Gayathri Vasudevan

Associated Project: Yes

Associated Part Family: CY7C68013A/14A/15A/16A

Software Version: Keil µVision 2

Related Documents: For a complete list, click here.

To get the latest version of this application note, or the associated project file, please visit http://www.cypress.com/AN63787.

More code examples? We heard you.

To access a variety of FX2LP code examples, please visit our **USB High-Speed Code Examples webpage**.

Are you looking for USB 3.0 peripheral controllers?

To access USB 3.0 product family, please visit our **USB 3.0 Product Family webpage**.

AN63787 discusses how to configure the general programmable interface (GPIF) and slave FIFOs of EZ-USB® FX2LP™, in both manual and auto modes, to implement an 8-bit asynchronous parallel interface. This application note is tested with two FX2LP development kits connected in a back-to-back setup, the first one acting in master mode and the second in slave mode.

Contents

1 Introduction

Cypress FX2LP is one of the most popular programmable high-speed USB controllers in the industry. The general programmable interface (GPIF) of FX2LP allows it to perform local bus mastering to external peripherals, implementing a wide variety of protocols. For example, EIDE/ATAPI, the printer parallel port (IEEE P1284), Utopia, and other interfaces are supported using the GPIF block of the FX2LP. In this example, it masters the slave FIFO interface of another FX2LP. This implementation uses the GPIF Designer (a utility Cypress provides to create GPIF waveform descriptors) to design the application-specific physical layer. The firmware is based on the Cypress FX2LP firmware frameworks. A hardware setup of two back-to-back FX2LP boards is used to test the firmware projects attached to this application note, one acting as a master and another as a slave.

2 FX2LP Architecture Overview

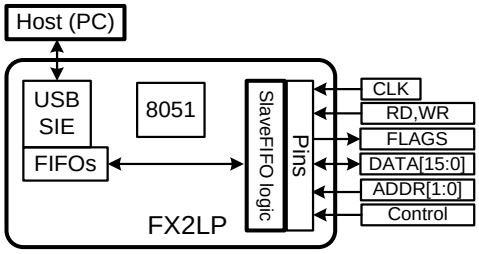
EZ-USB FX2LP is a flexible USB 2.0 peripheral controller, designed to handle the maximum USB 2.0 bandwidth. FX2LP optimizes USB throughput by providing the GPIF to implement a high-speed parallel interface to an external device. The GPIF moves data between FX2LP endpoint FIFOs and the GPIF interface. The following sections briefly describe the FX2LP architecture by showing the different possible configurable modes of FX2LP.

2.1 Slave FIFO Mode

In slave FIFO mode, dedicated FX2LP logic provides control and data signals to connect the USB endpoint FIFOs to an outside FIFO controller. In addition to the data bus and the FIFO select inputs, the interface provides the usual FIFO signals such as RD, WR, and FIFO flags, as shown in Figure 1.

For more information about the FX2LP Slave FIFO interface, see the “Slave FIFO” chapter in the EZ-USB Technical Reference Manual and AN61345 – Designing with EZ-USB FX2LP Slave FIFO Interface using FPGA, which presents a detailed design example.

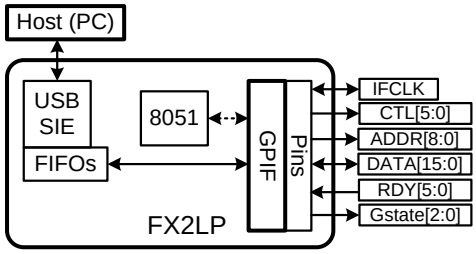
Figure 1. FX2LP Pins in Slave FIFO Mode



2.2 GPIF Mode

The GPIF is a programmable 8- or 16-bit parallel interface that reduces system costs by providing a glue less interface between the EZ-USB FX2LP and different types of external peripherals. When the GPIF is activated, the interface pins function as a master device to control external peripherals such as a RAM, FIFO, or external processor. Figure 2 shows the GPIF interface signals.

Figure 2. FX2LP in GPIF Mode



For more information about the GPIF, see the “GPIF” chapter in the EZ-USB Technical Reference Manual. AN66806 - Getting Started with EZ-USB® FX2LP™ GPIF introduces the GPIF unit and its graphical design tool called GPIF Designer and also includes an example demonstrating how to incorporate a USB connection into a GPIF design. AN57322 – Interfacing SRAM with FX2LP over GPIF discusses how to connect a Cypress CY7C1399B SRAM to FX2LP using an 8-bit asynchronous interface and GPIF auto mode.

This application note demonstrates the implementation of manual mode and auto mode operation in both the GPIF and slave FIFOs of FX2LP. It describes the manual mode and auto mode of operation using an FX2LP-to-FX2LP back-to-back setup, with one FX2LP operating in master (GPIF) mode and the other in slave mode:

- **Manual mode:** Two FX2LP chips are interfaced with each other; one of the chips functions in GPIF manual mode and the other in slave FIFO manual mode. In certain applications, the EZ-USB CPU must be present in the data path to interpret or modify the data before it is passed to the external device or host computer. In this section, a bidirectional parallel interface is implemented to show how the master CPU (FX2LP in GPIF manual mode) and slave CPU (FX2LP in slave FIFO manual mode) perform data modification.
- **Auto mode:** Two FX2LP chips are interfaced with each other; one of the chips functions in GPIF auto mode and the other in slave FIFO auto mode. The EZ-USB CPU is usually removed from the data path to maximize bandwidth. System Requirements

2.3 Hardware

This example uses two FX2LP Development Kit (CY3684) boards as the development and testing platforms. A detailed schematic of the development kit (DVK) is provided in the hardware folder, included in the attachment. More information about the board is available in the “Advanced Development Board” section of the EZ-USB Development Kit User Guide document, which is available at the following location:

C:\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Documentation (after DVK installation).

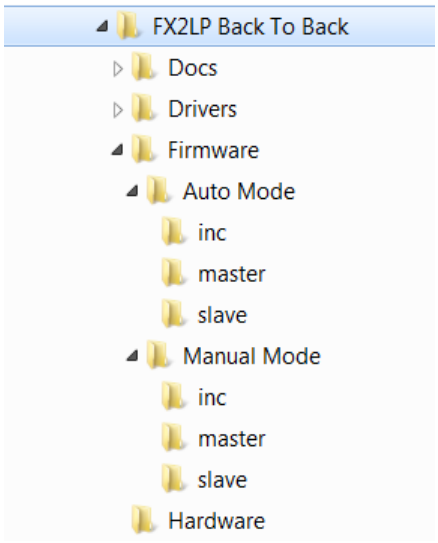
2.4 Software

- Control Center: Available through Suite USB 3.4.
- Keil µVision 2 IDE: The 4-KB evaluation version is available with the CY3684 DVK. For the full version, contact Keil.
- GPIF Designer: Available here.

3 Project Directory Structure

Figure 3 shows the folder structure of the attachment.

Figure 3. Folders in the Attachment



3.1 FX2LP Back-to-Back

This project folder is included with this application note and contains the following:

- Docs
- Drivers
- Firmware
- Hardware

3.2 Documentation

Folder path: *FX2LP Back To Back\Docs*

This folder contains the FX2LP datasheet and the Technical Reference Manual.

3.3 Drivers

Folder path: *FX2LP Back To Back\Drivers*

This folder contains the *CyUSB.inf* and *CyUSB.sys* files for FX2LP; use *CyUSB.inf* in the path shown in Table 1 for the operating system you are using.

Table 1. Paths for Various Operating Systems

Operating System	Folder Path
Windows XP 32-bit	wxp\x86
Windows XP 64-bit	wxp\x64
Windows 7 32-bit	wlh\x86
Windows 7 64-bit	wlh\x64
Windows Vista 32-bit	wlh\x86
Windows Vista 64-bit	wlh\x64

3.4 Firmware

Folder path: *Back To Back\Firmware*

This folder contains the following folders:

- Manual Mode
- Auto Mode

The Manual Mode and Auto Mode folders contain the master and slave subfolders. They also contain an inc folder containing the header files used in the projects. These header files remain common for master and slave projects, so the changes made to any of these header files are reflected in both the master and slave projects (a rebuild of both master and slave projects is necessary after any changes are made to any of the header files in the inc folder).

- Master

  Folder path: *FX2LP Back To Back\Firmware\Manual Mode\master* and *FX2LP Back To Back\Firmware\Auto Mode\master*

  This folder contains the firmware needed for the master FX2LP; *master.uv2* is the project file. This folder also includes the *master.hex* and *master.iic* files. You can either program the RAM of FX2LP by loading the *master.hex* file or program the large EEPROM by loading the *master.iic* file.

- Slave

  Folder path: *FX2LP Back To Back\Firmware\Manual Mode\slave* and *FX2LP Back To Back\Firmware\Auto Mode\slave*

  This folder contains the firmware needed for the slave FX2LP; *slave.uv2* is the project file. This folder also includes the *slave.hex* and *slave.iic* files. You can either program the RAM of FX2LP by loading the *slave.hex* file or program the large EEPROM by loading the *slave.iic* file.

- Inc

  Folder path: *FX2LP Back To Back\Firmware\inc*

  This folder contains the header files *fx2.h*, *fx2regs.h*, *fx2sdly.h* and *syncdly.h.*

4 FX2LP Firmware Project Files Structure

When you click to open the *master.uv2* or *slave.uv2* file, the Keil µVision 2 opens the respective FX2LP project. On the left side of the window appears a tree of the files that are part of the project (see Figure 4).

Table 2 identifies the files and explains their respective functions.

Figure 4. FX2LP Firmware Project Opened Using Keil IDE

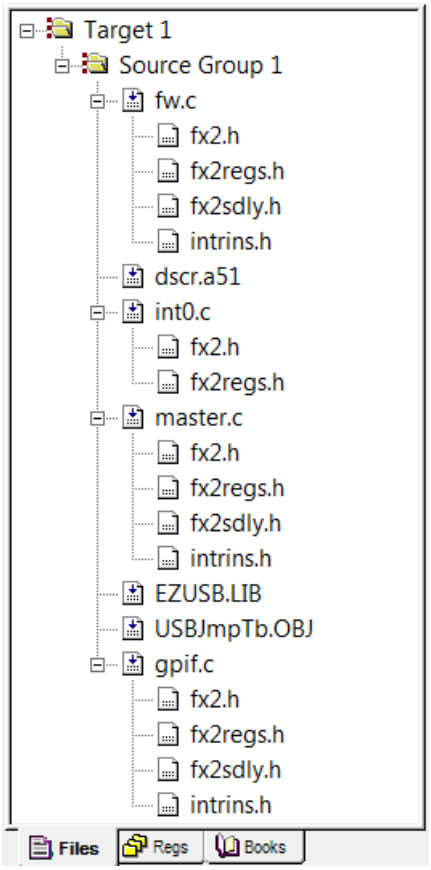


Table 2. FX2LP Firmware Project Files and Descriptions

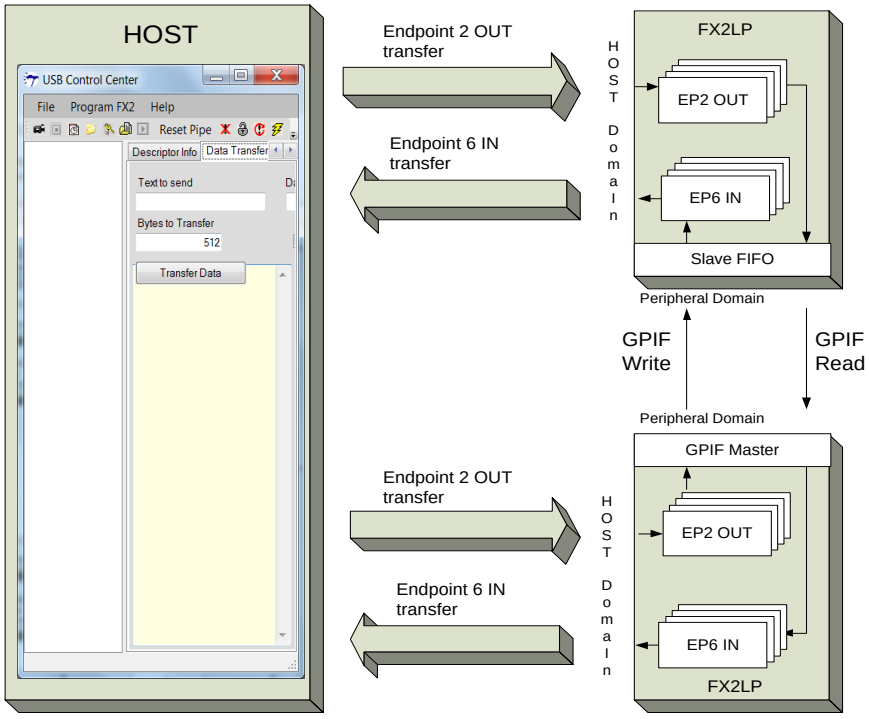
File	Description
fw.c	This file is part of the Cypress-written Firmware Frameworks, and it handles low-level USB detail. You should never need to modify this file. It contains the project's main function, which calls the `TD_Init` function once at startup and then repeatedly calls the `TD_Poll` function during operation. This endless loop also handles the CONTROL endpoint (EP0) SETUP packets. For `GET_DESCRIPTOR` requests, it uses the descriptor data that is supplied in *dscr.a51*. For other host requests, it makes calls into the application to handle various actions such as changing an interface's alternate setting.
dscr.a51	This is an 8051-assembly language module containing descriptor data for the user's particular USB device. This file contains *.db* (define byte) statements to list the descriptor table data.
int0.c	This file contains the interrupt service routine (ISR) for INT0# pin, used by the peripheral to issue the ZERO LENGTH PACKET (ZLP).
master.c/slave.c	This is the user application. The application code is written in this module with help from the USB Frameworks: User writes the `TD_Init` and `TD_Poll` functions to suit the custom application. *Fw.c* makes calls to specifically named functions in the code as it fields various device requests over endpoint 0. A Cypress code template (*peripheral.c*) saves the work of creating the functions by providing a code skeleton containing all the function stubs. The user supplies ISRs to handle the interrupts that the application uses. Most of these are simple acknowledgement housekeeping; you do not need to modify them if no further action is necessary.
EZUSB.LIB	This file contains the binary for the EZ-USB library functions. For more information, see the "EZ-USB Library" section in the EZ-USB Development Kit User Guide document, available at *C:\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Documentation*.
USBJmpTb.OBJ	This file contains the interrupt vector and the jump table for the USB interrupts. If this object file is linked to your project, enable autovectoring prior to enabling interrupts. The function names called by the jump table are located in the source file *USBJMPTB.A51*.
gpif.c	This file is generated from the GPIF designer and configures the GPIF for FX2LP. The sections of text that are marked as "DO NOT EDIT ..." should not be modified. In this case, the file is present in the FX2LP master project only because master is the one that uses GPIF interface signals to control the slave.
fx2.h, fx2regs.h, fx2sdly.h, intrins.h	These are the various header files included in both projects (master and slave). Their individual roles are explained inside each of these files.

For more information on FX2LP firmware frameworks, see the "EZ-USB Firmware Frameworks" section in the EZ-USB Development Kit User Guide document, available at *C:\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Documentation* (after DVK installation).

5 Block Diagram

Figure 5 illustrates the system-level block diagram demonstrating the functionality of this application.

Figure 5. System-Level Block Diagram



Two FX2LP chips are interfaced with each other: One of the chips functions in GPIF mode, and the other functions in slave FIFO mode. Both are configured to have the following endpoints:

- EP2 – BULK OUT, 512-byte, quad buffered
- EP6 – BULK IN, 512-byte, quad buffered

The data sent from the host to the EP2 OUT endpoint of the master is transferred to the EP6 IN endpoint of the slave. Similarly, the data transferred from the host to EP2 OUT of the slave is transferred to EP6 IN of the master. Thus, the data read from EP6 IN of the slave (master) is the same as that transferred to EP2 OUT of the master (slave).

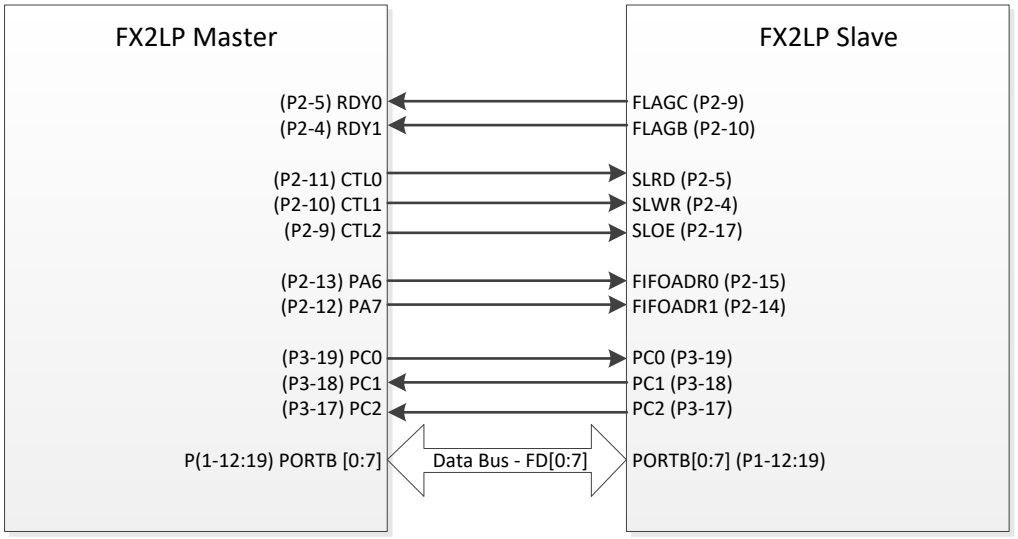
To demonstrate the difference between auto and manual modes, the data is modified inside both the master FX2LP and slave FX2LP in manual mode.

6 Hardware Connections

This section discusses the required hardware interconnect between the two FX2LP development kits.

Figure 6 shows the hardware connection between the two FX2LP DVKs for manual mode operation. These connections are made using the jumper wires, as there is no special interconnection board for this project.

Figure 6. Hardware Connections for Manual Mode (Both Master and Slave are of 128-pin Package)



Note: For auto mode operation, the PC0 and PC1 lines are not required, while all other hardware connections remain the same as those in manual mode, as shown in Figure 6.

In Figure 6, markings inside brackets denote the name of the headers in the FX2LP DVK. For example, P2-5 denotes PIN5 of PORT 2 of the FX2LP DVK. The data bus is eight bits wide, and the interface is asynchronous.

Hardware connections of the slave FX2LP and their respective uses

- Master pins:
  - CTL[5:0] are programmable control outputs that are used as strobes, read/write lines, or other outputs. Control signals (CTL0, CTL1, and CTL2) are used in this application and are tied to SLRD, SLWR, and SLOE of the slave.
  - RDY[5:0] are “ready” inputs that can be sampled and allow a transaction to wait (inserting wait states), continue, or repeat until the signal is at the appropriate level. This implementation uses RDY0 and RDY1 to control data flow. RDY0 is tied to FLAGC (EP2 empty flag) of the slave, and RDY1 is tied to FLAGB (EP6 full flag) of the slave.
  - The master FX2LP uses port A pins [6, 7] to drive the address of the slave FIFO being accessed by the master.
  - PC2 (SLAVEREADY) is an input to the master from the slave, indicating that the slave has now been reset and its firmware is ready. This is to prevent the master from reading in any garbage values in its EP6 FIFO, before or during the slave reset.
  - For manual mode operation, two additional lines are required for a handshake between the slave and master devices to implement the master-out, slave-in transfer. PC0 (`Txn_Over`) and PC1 (`Pkt_Committed`) are used for this purpose. More details on the use of these pins are given in later sections.
- Slave pins:
  - FLAGB and FLAGC are used to report the status of the slave FIFOs.
  - FLAGB – EP6FF (endpoint 6 full flag) indicates the state of “fullness” of the EP6 FIFO.
  - FLAGC – EP2EF (endpoint 2 empty flag) indicates the state of “emptiness” of the EP2 FIFO.
  - The slave FIFO control pins used are SLOE (Slave Output Enable), SLRD (Slave Read), SLWR (Slave Write), and FIFOADR[1:0] (FIFO Select).
  - The FIFOADR[1:0] pins select which of the four FIFOs is connected to the data bus (controlled by the external master).
  - PC2 is an output of the slave device that is made high after the slave has been reset and hence ready with its firmware.

FD[0:7]

This is port B, which is configured as the 8-bit data bus. If the WORDWIDE bit of any of the EPxFIFOCG registers (EPxFIFOCFG.0) is set, then port D is configured to be FD[8:15]. The OR of all four WORDWIDE bits of EPxFIFOCFG (x = 0:3) is what causes PORTD to be PORTD or FD[15:8]. The individual WORDWIDE bits indicate how data is to be passed for each individual endpoint. This implementation has an 8-bit interface.

7 GPIF Waveforms

The GPIF Designer utility is used to create the waveform descriptors to read from and write into the slave FX2LP. First, you must define the interface and then create the waveforms using the utility. After the interface is configured, create the read and write waveforms using the communication that takes place over the interface.

See AN66806 – Getting Started with EZ-USB FX2LP GPIF to get step-by-step instructions for using the Cypress GPIF Designer tool.

Both the GPIF read and write waveforms follow the logic of passing through N iteration (S1 - S2 -...- Idle), where N is the transaction count specified by loading into the registers GPIFTCB3:0 with the desired number of transactions.

For more information on registers, see chapter 15 of the EZ-USB Technical Reference Manual.

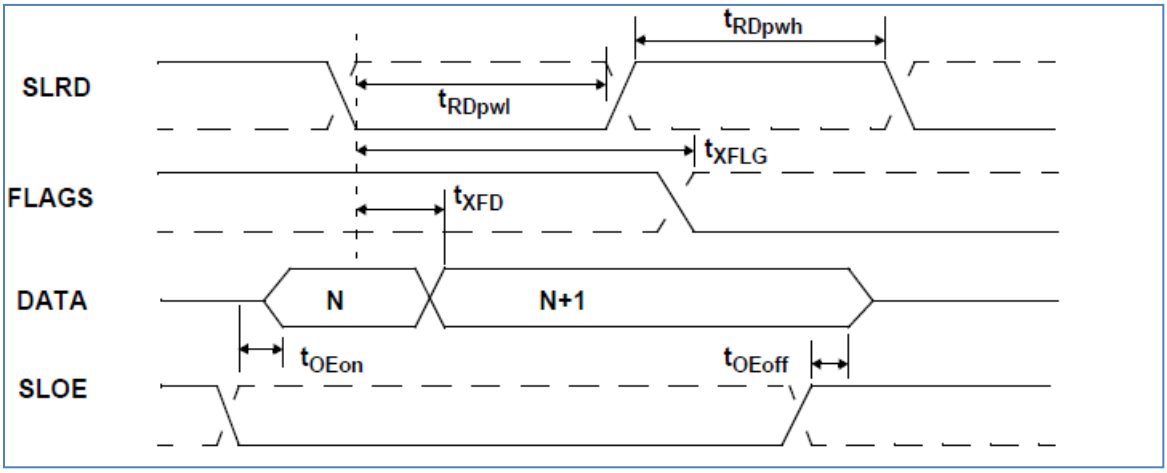
This application note uses two waveforms, one to read from and the other to write into the slave FX2LP; as shown in the screenshots on the following pages.

7.1 Read Waveforms

Read waveforms are designed to read data from the OUT endpoint (EP2) of the slave FX2LP into the IN endpoint (EP6) of the master FX2LP. They must satisfy the timing requirements of the various signals involved in the read cycle of the FX2LP. Figure 7 shows the timing diagram for FIFO Read, and

Figure 8 shows the waveform.

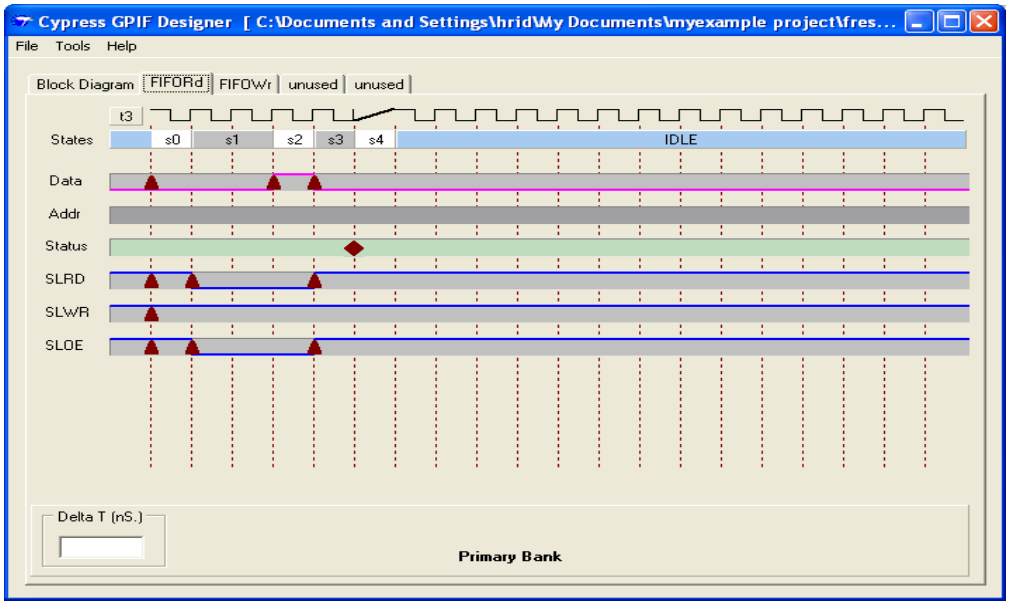
Figure 7. Slave FIFO Read Waveform with Timing Parameters



The following timing parameters must be met:

Parameter	Description	Min (ns)	Max (ns)
t_{RDpwl}	SLRD pulse width LOW	50	–
t_{RDpwh}	SLRD pulse width HIGH	50	–
t_{XFLG}	SLRD to FLAGS output propagation delay	–	70
t_{XFD}	SLRD to FIFO data output propagation delay	–	15
t_{OEon}	SLOE turn on to FIFO data valid	–	10.5
t_{OEoff}	SLOE turn off to FIFO data hold	–	10.5

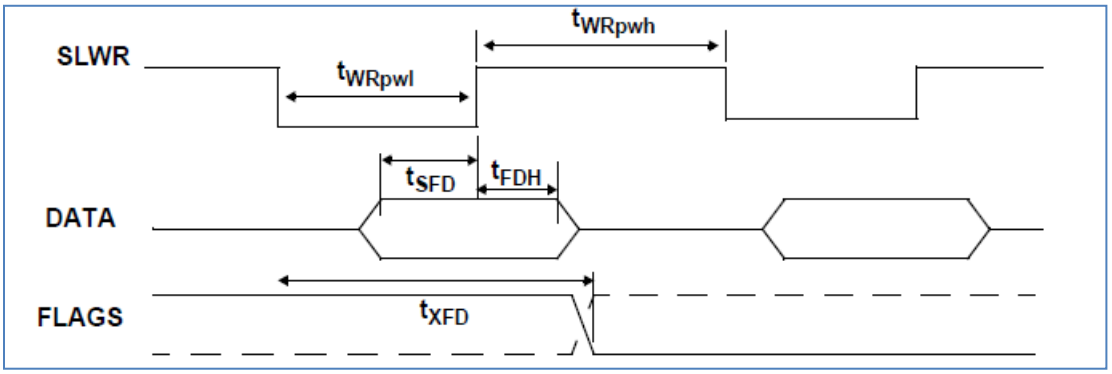
Figure 8. FIFO Read Waveform Made in GPIF Designer Utility



7.2 Write Waveforms

Write waveforms are designed to write data from the OUT endpoint (EP2) of the master FX2LP into the IN endpoint (EP6) of the slave FX2LP. They must satisfy the timing requirements of the various signals involved in the FX2LP write cycle. Figure 9 shows the timing diagram for FIFO Write, and Figure 10 shows the waveform.

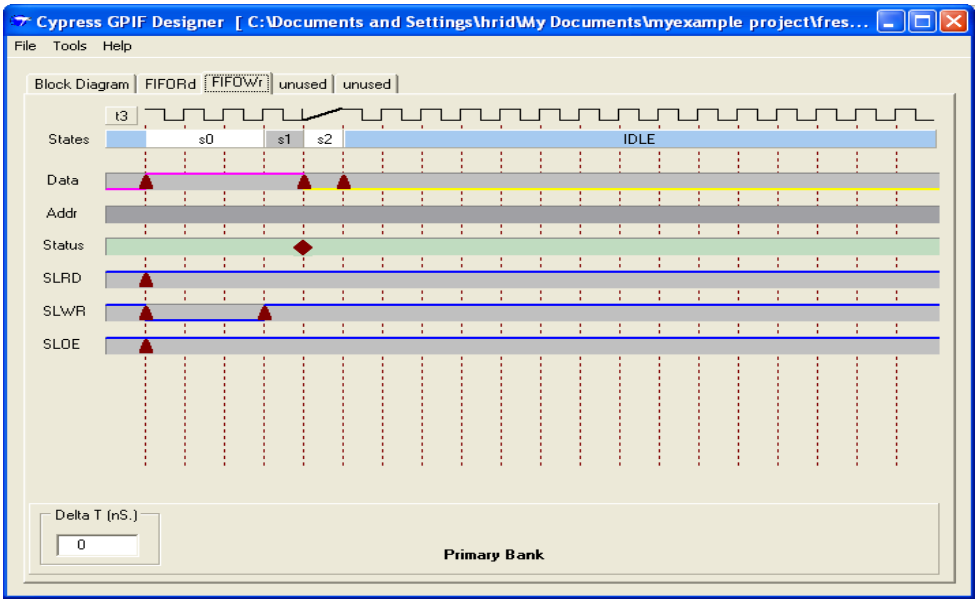
Figure 9. Slave FIFO Write Waveform with Timing Parameters



The following timing parameters must be met:

Parameter	Description	Min (ns)	Max (ns)
t_{WRpwl}	SLWR pulse width LOW	50	–
t_{WRpwh}	SLWR pulse width HIGH	70	–
t_{SFD}	SLWR to FIFO DATA setup time	10	–
t_{FDH}	FIFO DATA to SLWR hold time	10	–
t_{XFD}	SLWR to FLAGS output propagation delay	–	70

Figure 10. FIFO Write Waveform

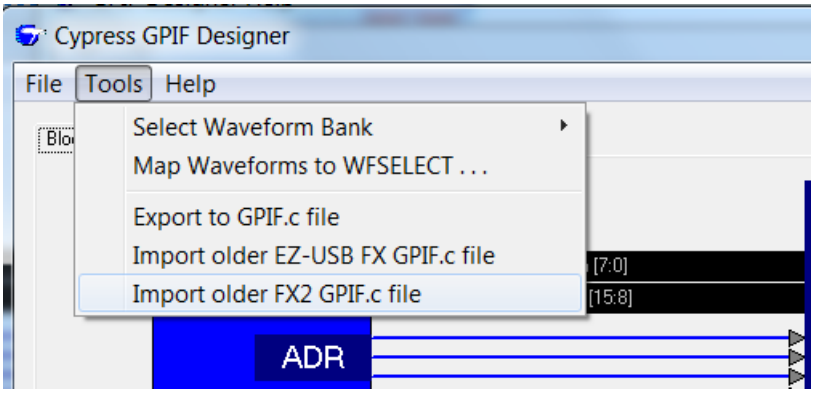


7.3 Import *gpif.c*

To see the waveforms shown in Figure 8 and Figure 10, import the *gpif.c* file that is included in the firmware project.

1. Open **GPIF Designer**.
2. Click the **Tools** tab. Select **Import older FX2 GPIF.c file** and point to the *gpif.c* file present in *FX2LP Back To Back\Firmware\Manual Mode\master* (see Figure 11). Now you should be able to see the slave FIFO read waveforms in the **FIFORd** tab and the slave FIFO write waveforms in the **FIFOWr** tab.

Figure 11. Importing the *gpif.c* File



8 FX2LP Manual Mode Operation in GPIF and Slave FIFO Configuration

8.1 Firmware

To view the code for the master, open *master.uv2* from *FX2LP Back To Back\Firmware\Manual Mode\master*. To view the code for the slave, open *slave.uv2* from *FX2LP Back To Back\Firmware\Manual Mode\slave*.

8.2 Configuring FX2LP in GPIF Manual Mode

- For detailed information on the registers used in the firmware, see "Chapter 15: Registers" of the EZ-USB Technical Reference Manual.
- Configure the REVCTL.0 bit to '1'. This is done so that both the IN and OUT packets can be edited, sourced, skipped, and committed.
- Set the REVCTL.1 bit to '1'; this disables the auto-arming of the OUT endpoints if it transitions from AUTOUT=0 to AUTOOUT=1.

```
REVCTL = 0x03;      // CPU can source and edit both IN and OUT packets
SYNCDELAY;
```

- Set EP2FIFOCFG and EP6FIFOCFG to '0'; this configures the endpoints in 8-bit manual mode.

```
EP2FIFOCFG = 0x00; // manual out mode, 8 bit data bus
SYNCDELAY;
EP6FIFOCFG = 0x00; // manual in mode, 8 bit data bus
SYNCDELAY;
```

- Configure PC0 (`Txn_Over`) as an output pin, PC1 (`Pkt_Committed`) as an input pin, and PC2 (`SLAVEREADY`) as an input pin.

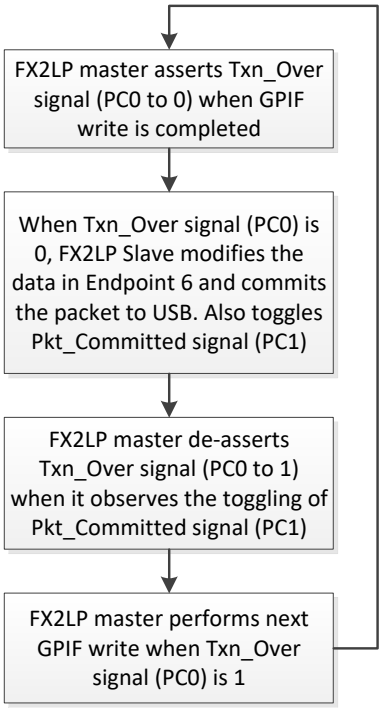
```
/* #define statements before TD_Init()*/
#define Txn_Over PC0
#define Pkt_Committed PC1
#define SLAVEREADY PC2

/* code snippets from TD_Init() */
PORTCCFG = 0x00; //configure port C as an I/O port
OEC= 0xF9;       /*Txn_Over configured as output, Pkt_Committed configured as input,
                 SLAVEREADY configured as input*/
PC0=1;
```

- To implement the slave FIFO in "manual in" mode, two lines are required to handshake between the slave and the master. Name the two lines `Txn_Over` and `Pkt_Committed`. `Txn_Over` is asserted by the master when a GPIF transaction is completed. The `Pkt_Committed` line is toggled by the slave whenever it commits a packet.
- `Txn_Over`, the deasserted state: `Txn_Over = 1` indicates that the master can start the next GPIF write transaction (writing from EP2 OUT of the master to EP6 IN of the slave).
- `Txn_Over`, the asserted state: `Txn_Over = 0` indicates that a GPIF write transaction has been completed by the master and that the slave can now start reading from its IN endpoint (EP6).
- `Pkt_Committed`: Every toggle of this signal means that the previous packet sent by the GPIF master has been processed and committed by the slave, and it is now ready to accept another packet.
- The `SLAVEREADY` input tells the master that the slave has been reset and its firmware is up and running.

The flowchart in Figure 12 shows the states of the handshake signals `Txn_Over` and `Pkt_Committed` during FX2LP master write to the slave FX2LP.

Figure 12. States of the Handshake Signals `Txn_Over` and `Pkt_Committed` during Master Write and Slave Read



8.2.1 GPIF Manual OUT Mode

- In the master, inside the `Td_Poll` function, continuous checking is done to see i the `Pkt_Committed` pin is toggled.

```
if(Pkt_Committed == ~b)
 {
  b = Pkt_Committed; /* store the current state of Pkt_Committed in
                        variable b so that the next toggle can be detected */
  Txn_Over = 1;
 }
```

- When a toggle is detected (that is, when the just-received packet in the IN endpoint of the slave is committed to its USB domain), Txn_Over is deasserted, indicating that the master can start the next GPIF transaction.

```
if( (!( EP2468STAT & 0x01 )) )    /*if EP2 not empty, modify packet and commit it to
                                    peripheral domain */
    {   SYNCDELAY;
        EP2FIFOBUF[0] = 0x01;      // editing the packet
        SYNCDELAY;
        EP2FIFOBUF[1] = 0x02;
        SYNCDELAY;
        EP2FIFOBUF[2] = 0x03;
        SYNCDELAY;
        EP2FIFOBUF[3] = 0x04;
        SYNCDELAY;
        EP2FIFOBUF[4] = 0x05;
        SYNCDELAY;
        EP2BCH = 0x02;
        SYNCDELAY;
        EP2BCL = 0x00;                          // commit edited pkt. to interface fifo
        SYNCDELAY;
    }
    if ( ! (EP24FIFOFLGS & 0x02) )
    {
        if((SLAVENOTFULL) && (Txn_Over == 1))
        {
           if( GPIFTRIG & 0x80 )          // if GPIF interface IDLE
           {
               PERIPH_FIFOADR0 = 0;       // FIFOADR[1:0]=10 - point to peripheral EP6
               PERIPH_FIFOADR1 = 1;
               SYNCDELAY;
               if(enum_high_speed)
               {

                   SYNCDELAY;
                   GPIFTCB1 =   0x02;       // setup transaction count 512
                   SYNCDELAY;
                   GPIFTCB0 =   0x00;
                   SYNCDELAY;
                }
                else
                {
                    SYNCDELAY;
                    GPIFTCB1 = 0x00;        // setup transaction count 64
                    SYNCDELAY;
                    GPIFTCB0 = 0x40;
                    SYNCDELAY;
                }
                SYNCDELAY;
                GPIFTRIG = GPIFTRIGWR | GPIF_EP2; /* launch GPIF FIFO WRITE Transaction
                                                     from EP2 */
                SYNCDELAY;
                while( !( GPIFTRIG & 0x80 ) ) // poll GPIFTRIG.7 GPIF Done bit
                {
                      ;
                }
                SYNCDELAY;
                Txn_Over = 0;   /*assert Txn_Over signal to indicate that packet has been
                                  transmitted */
           }
       }
    }
```

- If there is a packet in the USB domain of the EP2 OUT endpoint, the first five bytes of the packet are modified and then committed.
- Then, if there is space in the EP6 IN endpoint of the slave FX2LP and if the Txn_Over is not asserted, then the GPIF write transaction is triggered.

- GPIF Write: Writing from EP2 OUT of the master FX2LP to EP6 IN of the slave FX2LP.
- After that transaction is over (determined by polling the “done” bit in the GPIFTRIG register), `Txn_Over` is asserted to indicate to the slave that one transaction is over and it can start reading that packet.

8.2.2 GPIF Manual IN Mode

Firmware required to perform GPIF read in manual mode is as follows.

```
if(SLAVEREADY)                    //checking if slave firmware is ready i.e if PC2=1
{
if ( GPIFTRIG & 0x80 )          // if GPIF interface IDLE - triggering gpif IN transfers
{
  PERIPH_FIFOADR0 = 0;
  PERIPH_FIFOADR1 = 0;          // FIFOADR[1:0]=00 - point to peripheral EP2
  SYNCDELAY;
  if ( SLAVENOTEMPTY )                // if slave is not empty
  {
    if ( !( EP68FIFOFLGS & EP6FULL ) )// if EP6 FIFO is not full
    {
      if(enum_high_speed)
      {
        SYNCDELAY;
        GPIFTCB1 = 0x02;             // setup transaction count 512
        SYNCDELAY;
        GPIFTCB0 = 0x00;
        SYNCDELAY;
       }
     else
       {
        SYNCDELAY;
        GPIFTCB1 = 0x00;              // setup transaction count 64
        SYNCDELAY;
        GPIFTCB0 = 0x40;
        SYNCDELAY;
       }

    GPIFTRIG = GPIFTRIGRD | GPIF_EP6; // launch GPIF FIFO READ Transaction to EP6 FIFO
    SYNCDELAY;
```

- In GPIF manual IN mode, check the slave’s “empty” flag to verify that there is a packet to be read from the slave device. If the empty flag is deasserted, then a FIFO read transaction to read from the slave is initiated.
- GPIF Read: Reading from EP2 OUT of the slave FX2LP into EP6 IN of the master FX2LP.
- The GPIFTRIG.7 bit is continuously polled to wait until the GPIF transaction has ended.

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

      EP6FIFOBUF[ 4 ] = 0x05; //edit the five bytes before committing
      EP6FIFOBUF[ 3 ] = 0x04;
      EP6FIFOBUF[ 2 ] = 0x03;
      EP6FIFOBUF[ 1 ] = 0x02;
      EP6FIFOBUF[ 0 ] = 0x01;
      SYNCDELAY;
      SYNCDELAY;
      EP6BCH = 0x02;            //committing the packet
      SYNCDELAY;
      EP6BCL = 0x00;
      SYNCDELAY;
    }
}
```

- When the transaction ends, the first five bytes of the packet are modified and then committed to the host domain by writing the number of bytes to be committed into the EP6BCH/BCL registers.
- To flush out any garbage values that the master might read into its EP6 FIFO during the slave reset, EP6 is reset until the time that the PC2 signal is not asserted high by the slave.

```
// This will keep resetting Master EP6FIFO until slave firmware starts to run
else  {
          FIFORESET = 0x80;  // set NAKALL bit to NAK all transfers from host
          SYNCDELAY;
          FIFORESET = 0x06;  // reset EP2 FIFO
          SYNCDELAY;
          FIFORESET = 0x00;  // reset EP6 FIFO
          SYNCDELAY;
          }
```

8.3 Configuring FX2LP in Slave FIFO Manual Mode

The `TD_Init` function configures endpoint 2 as an OUT endpoint and endpoint 6 as an IN endpoint, both functioning in 8-bit manual mode.

8.3.1 Slave FIFO Manual IN Mode

The piece of code in the FX2LP slave firmware (manual mode) that modifies the data received in the slave FIFO before committing to USB is as follows:

```
PC2=1;         //making it 1 to show that slave firmware has started running
if (PC0 == 0 &&(!(EP68FIFOFLGS & 0x02))) /*if (PC0/Txn_Over = Zero) meaning master is
                                    done writing to Slave and EP6 FIFO is not empty*/
{
      EP6FIFOBUF[ 507 ] = 0x05; //edit the last five bytes before committing
      EP6FIFOBUF[ 508] = 0x04;
      EP6FIFOBUF[ 509 ] = 0x03;
      EP6FIFOBUF[ 510 ] = 0x02;
       EP6FIFOBUF[ 511 ] = 0x01;
       SYNCDELAY;
       SYNCDELAY;
       EP6BCH = 0x02;            //committing the packet
       SYNCDELAY;
       EP6BCL = 0x00;
       SYNCDELAY;
       PC1 = ~PC1;            //toggle PC0 to indicate that the buffer has been passed
       while(  PC0 != 1);        //wait for PC0 to become high again.
}
```

- PC2 has been asserted high to tell the master device that the slave has been reset and its firmware has started running.
- In the `Td_Poll` function, PC0 (which is tied to the `Txn_Over` of the master FX2LP) is continuously checked to verify that `Txn_Over` is asserted to indicate the end of a FIFO write transaction from the master. This check is necessary; otherwise, the data in the IN FIFO of the slave FX2LP can get committed to the USB domain before the entire packet is transferred into it from the master. When this happens, the host will see data being split into several smaller packets.
- When the data packet is committed, the slave toggles the PC1 (which is tied to the `Pkt_Committed` line of the master FX2LP) to let the master device know that the packet is committed and the next packet transaction can now start. The master also deasserts the `Txn_Over` line when it finds that the `Pkt_Committed` line has been toggled.
- When PC0 (`Txn_Over`) is asserted and when EP6 IN is not empty, then the last five bytes of the packet are modified in the slave FX2LP and then committed to the host domain.

8.3.2 Slave FIFO Manual OUT Mode

The slave FIFO manual OUT mode firmware checks if the empty flag of the OUT endpoint is deasserted. If it is deasserted, then the last five bytes of the packet in the EP2 OUT endpoint of the slave FX2LP are edited, and then the packet is committed to the peripheral domain.

```
if( !( EP2468STAT & 0x01 ) )//if EP2 not empty, modify packet and commit it to peripheral
side
        {     SYNCDELAY;
              EP2FIFOBUF[511] = 0x01; // editing the packet
              SYNCDELAY;
              EP2FIFOBUF[510] = 0x02;
              SYNCDELAY;
              EP2FIFOBUF[509] = 0x03;
              SYNCDELAY;
              EP2FIFOBUF[508] = 0x04;
              SYNCDELAY;
              EP2FIFOBUF[507] = 0x05;
              SYNCDELAY;
              EP2BCH = 0x02;
              SYNCDELAY;
              EP2BCL = 0x00;        // commit edited pkt. to interface fifo
              SYNCDELAY;
         }
```

8.4 Testing the Project (Manual Mode)

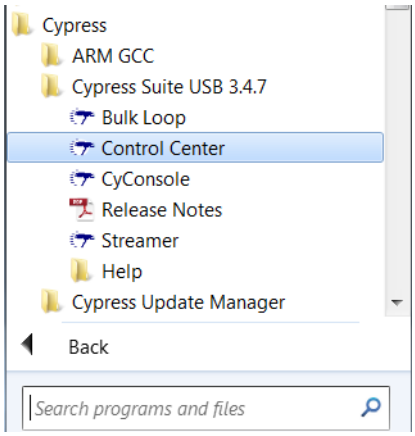
1. Download and install Cypress SuiteUSB 3.4. This installs a utility named Control Center.
2. Positions of the switches SW1 and SW2 of the DVKs: Keep these switches according to the states given in Table 3.

Table 3. DVK Switch Positions

State	Operation Being Performed	SW1	SW2
1	Programming the RAM	Don't Care	No EEPROM
2	Programming the large EEPROM	Large EEPROM	EEPROM

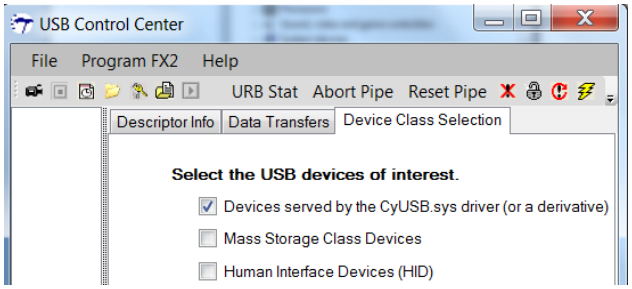
3. Connect the two FX2LP DVK boards to each other as shown in Figure 6. Set SW1 and SW2 on both the boards in state 1 as shown in Table 3, and connect each of them to the host PC using USB cables. They enumerate with the default internal descriptors. Use the *CyUSB.inf* file in the Drivers folder (*AN63787\FX2LP Back To Back\FX2LP Back To Back\Drivers*) to bind with the device. For help with binding the driver, see *MatchingDriverToUSBDevice.htm* in the *Drivers* folder.
4. Launch the USB Control Center (see Figure 13) from the path: **Start** ( ) > **All Programs** > **Cypress** > **Cypress Suite USB 3.4.7** > **Control Center**.

Figure 13. Opening the USB Control Center Application



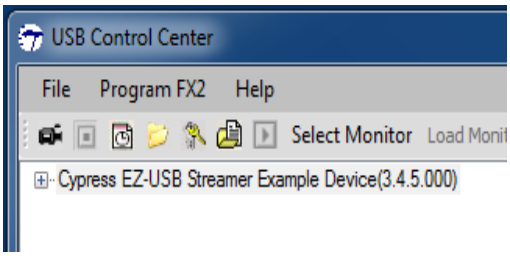
5. You should see the FX2LP board along with other connected USB devices listed in the left panel. To view only the FX2LP board, click the **Device Class Selection** tab in the right pane and uncheck everything but Device served by the *CyUSB.sys* driver (or a derivative), as shown in Figure 14.

Figure 14. Option to Select the Devices That Use *CyUSB.sys*



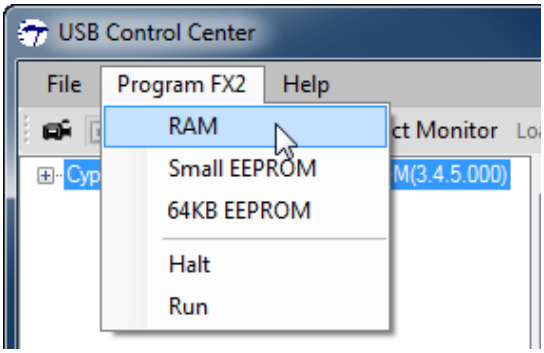
Then the left pane should appear as shown in Figure 15.

Figure 15. USB Control Center Finds the FX2LP Board



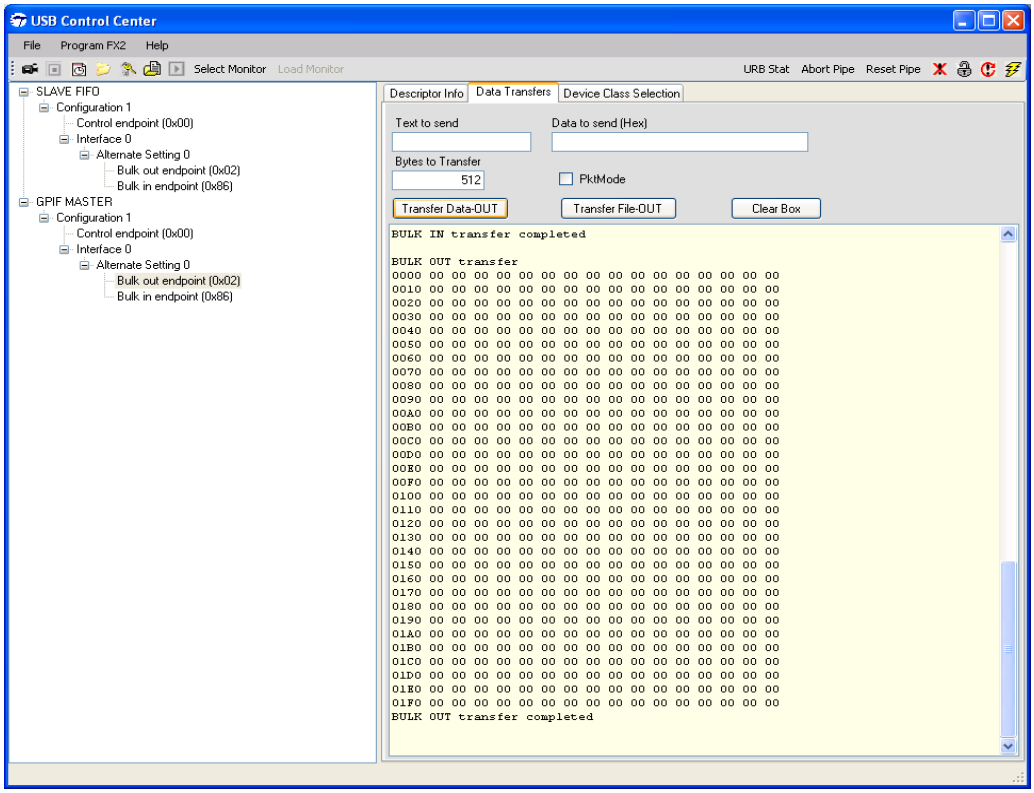
6. Now you can load the Keil-compiled *slave.hex* (for RAM) file available in *FX2LP Back To Back\Firmware\Manual Mode\slave* onto the RAM of the first (slave) FX2LP DVK. Highlight the EZ-USB entry and select **Program FX2** > **RAM** to download the code into the FX2LP RAM, as shown in Figure 16. When programming is successful, a Programming succeeded message appears at the bottom left corner of the Control Center window.

Figure 16. Programming the FX2LP DVK RAM



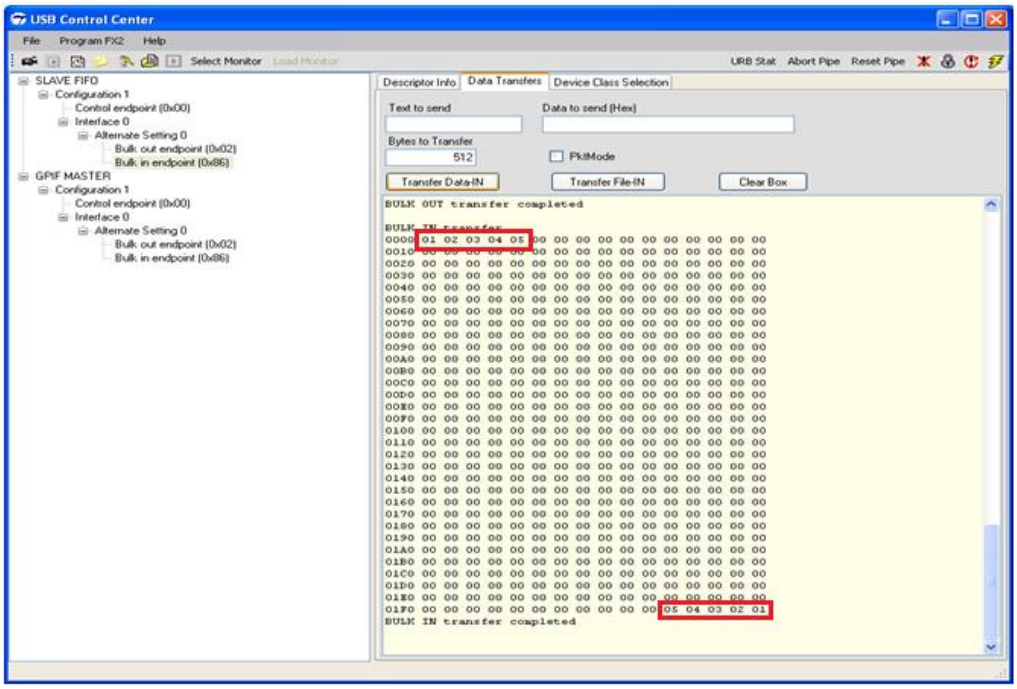
7. Pop-up windows appear asking to bind the driver. Use the correct *CyUSB.inf* file (according to the OS used) located in the *Drivers* folder to bind.
8. Now, download *master.hex* in the attachment (available in *FX2LP Back To Back\Firmware\Manual Mode\master*) to the RAM of the second (master) FX2LP using the Control Center utility. Select the master device from the left pane. Choose **Program** > **RAM** for programming the RAM.
9. Pop-up windows appear asking to bind the driver. Use the same *CyUSB.inf* file located in the *Drivers* folder to bind.
10. Expand the trees in the left pane completely until the endpoints.
11. Use the Control Center to send 512 bytes into EP2 of the master FX2LP. Click the **Data Transfers** tab in the right pane. Select the **Bulk out endpoint (0x02)** of the master device from the left pane. Check that the Bytes to Transfer field equals 512. Click **Transfer Data-OUT**. Observe the data transfer as shown in Figure 17.

Figure 17. GPIF Master OUT Data Transfer



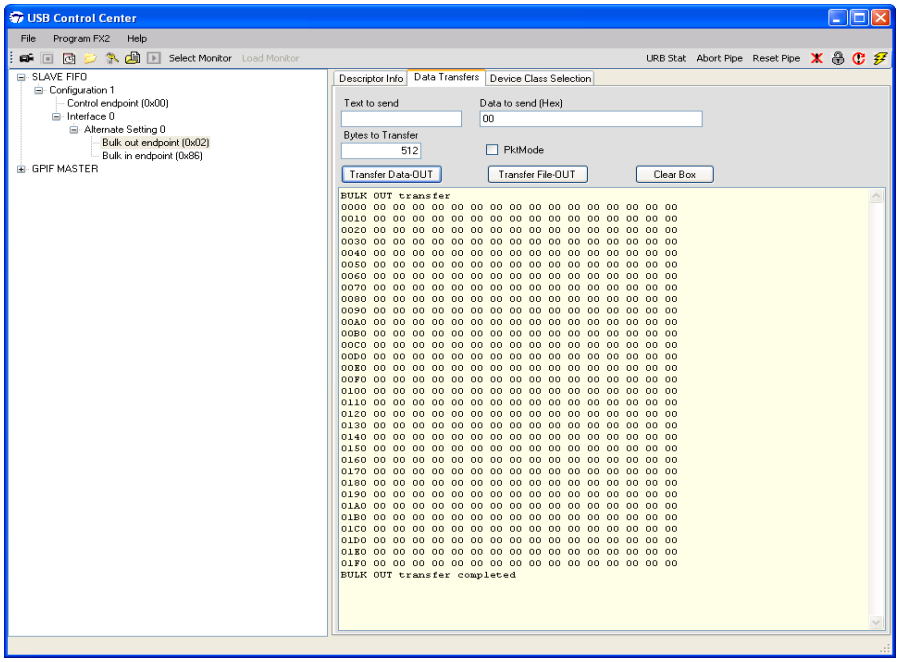
12. Issue a BULK IN transfer of 512 bytes from EP6 of the slave FX2LP. Select the **Bulk in Endpoint (0x86)** of the slave device from the left pane. Check that the Bytes to Transfer field equals 512. Click **Transfer Data-IN**. The data received should be the same as that sent to EP2 OUT of the master FX2LP, except the first and the last five bytes, which are modified. The master modifies the first five bytes, and the slave modifies the last five bytes. Observe the data transfer as shown in Figure 18.

Figure 18. Slave FIFO IN Transfer



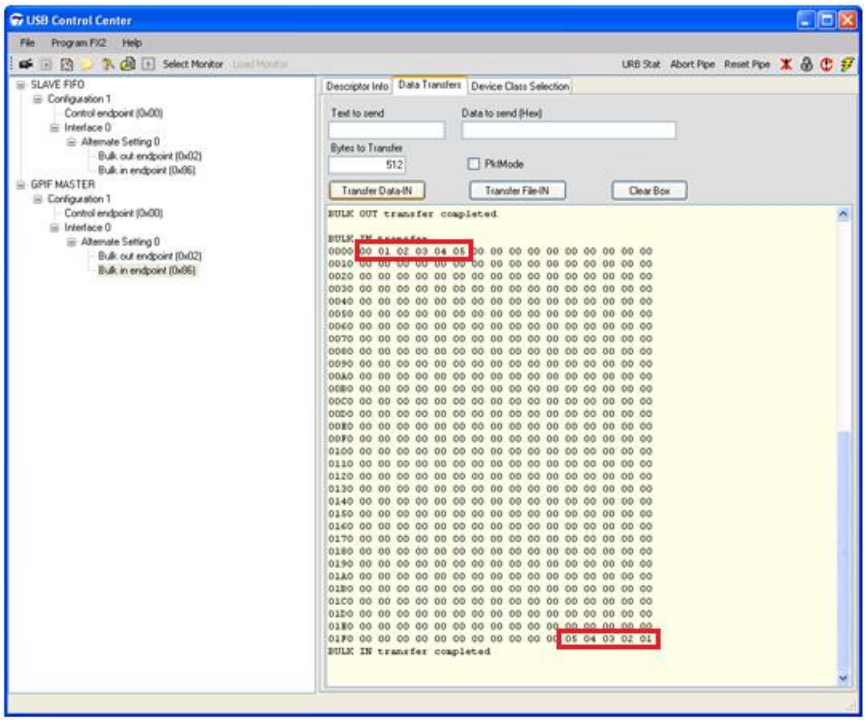
13. Using a procedure similar to that in step 12, send 512 bytes of data from EP2 of the slave FIFO. Observe the data transfer as shown in Figure 19.

Figure 19. Slave FIFO OUT Transfer



14. Using a procedure similar to that in step 13, issue a Bulk IN transfer of 512 bytes from EP6 of the master FX2LP. In the read data, the first five bytes and last five byes are modified. The master modifies the first five bytes, and the slave modifies the last five bytes. Observe the data transfer as shown in Figure 20.

Figure 20. GPIF Master IN Transfer



9 FX2LP Auto Mode Operation in GPIF and Slave FIFO Configuration

Two FX2LP chips are interfaced with each other, as shown in Figure 6, except the PC0 and PC1 connections, which are not required in auto mode. One of the chips functions in GPIF auto mode, and the other functions in slave FIFO auto mode.

9.1 Firmware

To view the code written for the master, open *master.uv2* from *FX2LP Back To Back\Firmware\Auto Mode\master*. To view the code for the slave FX2LP, open *slave.uv2* from *FX2LP Back To Back\Firmware\Auto Mode\slave*.

9.2 Initialization for FX2LP in GPIF Master Mode

The `TD_Init` function takes care of the entire initialization with the following steps:

1. Configure EP2 and EP6 as quad-buffered endpoints with a buffer size equal to 512.
2. Reset the FIFOs of endpoints EP2 and EP6.
3. Configure the endpoint EP2 and EP6 FIFOs to 8-bit auto mode.

9.2.1 GPIF Auto OUT Mode

The TD_Poll function of the GPIF master performs the data loopback from EP2 OUT of the GPIF master to EP6 IN of the slave FIFO. The data is transferred from the master FX2LP to the slave using the **FIFOWr** waveforms of GPIF.

```
if( GPIFTRIG & 0x80 )                // if GPIF interface IDLE
  {
      if ( ! ( EP24FIFOFLGS & EP2EMPTY ) )  // if there's a packet in the peripheral domain
                                            //for EP2
      {
        PERIPH_FIFOADR0 = 0;      // FIFOADR[1:0]=10 - point to peripheral EP6 of Slave FX2LP
         PERIPH_FIFOADR1 = 1;
         SYNCDELAY;               // used here as "delay", refer to Synchronization
         if ( SLAVENOTFULL )            // if the slave is not full
        {
            if(enum_high_speed)
            {
               SYNCDELAY;
               GPIFTCB1 = 0x02;             // setup transaction count 512
               SYNCDELAY;
              GPIFTCB0 = 0x00;
               SYNCDELAY;
            }
            else
            {
               SYNCDELAY;
               GPIFTCB1 = 0x00;             // setup transaction count 64
               SYNCDELAY;
               GPIFTCB0 = 0x40;
               SYNCDELAY;
            }
              SYNCDELAY;
              GPIFTRIG = GPIFTRIGWR | GPIF_EP2;  //launch GPIF WRITE Transaction from EP2
             SYNCDELAY;
             while( !( GPIFTRIG & 0x80 ) );      // poll GPIFTRIG.7 GPIF Done bit

             SYNCDELAY;
        }
      }
  }
```

As soon as the packet comes into EP2 OUT of the master FX2LP, it is auto-committed to the peripheral domain. Thus, inside the TD_Poll function, you should only trigger the GPIF Write transaction whenever there is any data in the EP2 OUT endpoint of the master FX2LP.

9.2.2 GPIF Auto IN Mode

```
if (SLAVEREADY)
{
if ( GPIFTRIG & 0x80 )                  // if GPIF interface IDLE
  {
      PERIPH_FIFOADR0 = 0;
      PERIPH_FIFOADR1 = 0;              // FIFOADR[1:0]=00 - point to peripheral EP2
      SYNCDELAY;
      if ( SLAVENOTEMPTY )                // if slave is not empty
      {
         if ( !( EP68FIFOFLGS & EP6FULL ) )     // if EP6 FIFO is not full
         {
            if(enum_high_speed)
           {
              SYNCDELAY;
               GPIFTCB1 = 0x02;                // setup transaction count 512
               SYNCDELAY;
               GPIFTCB0 = 0x00;
               SYNCDELAY;
           }
           else
           {
```

```
                SYNCDELAY;
                GPIFTCB1 = 0x00;                // setup transaction count 64
                 SYNCDELAY;
                GPIFTCB0 = 0x40;
                SYNCDELAY;
              }
               GPIFTRIG = GPIFTRIGRD | GPIF_EP6;  // launch GPIF READ Transaction to EP6

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

The EP6 IN endpoints of both FX2LPs are in auto mode, and the AUTO IN packet length is set to 512. Therefore, if the data transferred is an integral multiple of 512 (bytes), it is auto-committed from the peripheral to the host side and is available on the host. This firmware can transfer data only when it is in integral multiples of 512 (bytes).

9.2.3 Slave Auto OUT and IN Mode

There is no need for any code in the slave FIFO apart from the initialization, because the packets are auto-committed. The slave firmware in auto mode is responsible for LED control and telling the master through the PC2 (`SLAVEREADY`) flag that its firmware is up and running. Thus, there is no data-handling code inside the `TD_Poll` function for the slave FX2LP.

9.3 Testing the Project (Auto Mode)

Testing the project is similar to that of the manual mode. See Figure 6 for the connection diagram. There is no need for the two additional connections between PC0 and PC1 in auto mode; this is required only for handshaking in manual mode configuration.

The code to be downloaded to the master FX2LP and the slave FX2LP is:

- Master FX2LP: *master.hex* included in the attachment and available in *FX2LP Back To Back\Firmware\ Auto Mode\master*.
- Slave FX2LP: *slave.hex* included in the attachment along with this code example and available in *FX2LP Back To Back\Firmware\Auto Mode\slave*
- The only difference observed with auto mode compared to manual mode is that the data received from the EP6 IN endpoint of the slave (master) FX2LP will be the same as the data that was sent to EP2 OUT of the master (slave) FX2LP.

10 Debug LEDs

The FX2LP DVK has four LEDs (D2, D3, D4, and D5) that can be used for debugging purposes. This application uses three LEDs, namely D2, D4, and D5, while D3 is made to glow at all times. The firmware of master and slave in both auto and manual mode contains the `LED_Control` function, which controls the behavior of LEDs D2, D4, and D5.

The specific states of the device represented by each of these LEDs remain the same for both master and slave, regardless of mode of operation (auto or manual). Following is a description of the LED states:

- D2: This LED blinks continuously as long as the firmware is running on the device.
- D3: This LED is not assigned to any specific state and glows at all times.
- D4: This LED represents the “non-empty” state of endpoint 2 of the device. This means that the device’s endpoint 2 has data to transfer to the other device connected with it; for example, LED D4 glowing on the master (slave) FX2LP means that its EP2 has data that can be transferred to the slave (master) device. D4 turns off when EP2 of the device is empty. Arithmetically, this LED glows when the EP2 has at least one packet of data.
- D5: This LED represents the “not-full” state of endpoint 6 of the device. This means that the device’s EP6 is not completely full and it can accept more packets from the other connected device; for example, LED D5 glowing on the slave (master) device means that its endpoint 6 is not completely full and can accept more data packet(s) from the master (slave) device. D5 turns off when EP6 is completely filled, and no more data can be transferred to the device without the host first taking in some data from the device. Arithmetically, this LED glows when EP6 has less than four packets of data (remember use of quad buffering for FIFOs).

Code for `LED_Control()` Function in Master Device

The LED D2-D5 functionality can be turned off by commenting the `#define LED_Enable` macro in the same *fx2.h* file (*FX2LP Back To Back\Firmware\Auto(Manual)\inc*).

```
//This function controls the state of D4 and D5 LEDs on the Master FX2LP DVK based upon
the state of EP2 and EP6 FIFOs. Also it blinks LED D2 while the firmware on the device
is running
void LED_Control()
{
//For LED D4 and D5
if (!( EP24FIFOFLGS & EP2EMPTY ))       //LED D4 turns on whenever EP2 has got data to
                                                //transfer to Slave i.e. EP2 is not Empty
LED_On(bmBIT2);
else
LED_Off(bmBIT2);

if (!( EP68FIFOFLGS & EP6FULL ))        //LED D5 turns on whenever EP6 can accept data
from                                            //Slave i.e. EP6 is not Full
LED_On(bmBIT3);
else
LED_Off(bmBIT3);
//For LED D2, LED D2 blinks to indicate that firmware is running.
if (++LED_Count == Blink_Rate)           //Blink_rate=10000 for Seven_segment enabled
and                                             //30000 otherwise
  {
    if (LED_Status)
    {
      LED_Off (bmBIT0);
      LED_Status = 0;
    }
    else
    {
      LED_On (bmBIT0);
      LED_Status = 1;
    }
    LED_Count = 0;
  }}
```

11 Seven-Segment Display

The firmware uses the seven-segment display present on the FX2LP DVK to indicate the number of data packets that are present in the EP6 FIFO buffers on the device. Since this application has used quad buffering for EP6, the value on the seven segments can go from 0 (denoting empty EP6) to 4 (denoting full EP6).

The following steps describe how the seven-segment display functions:

1. When a data packet is transferred from host to master EP 2, and if the slave EP6 FIFO is not full, then that data packet is transferred to the slave, and hence the seven-segment display on the slave side gets incremented by one.
2. If the host takes in data from slave EP6 FIFO, then the count on the slave display gets reduced by one, provided the master EP2 does not have any data that it otherwise would transfer to slave EP6 if it sees any space there; that is, EP6 is not full.
3. Similar logic is used for the transfer that takes place from host > slave > master > host.

Note that if you do not want to use seven-segment functionality, then it can be done by commenting the `#define Seven_segment` macro present in the *fx2.h* file (*FX2LP Back To Back\Firmware\Auto(Manual)\inc*). You can also access this file directly inside the Keil IDE. If you comment this macro, then the project needs to be built again for both master and slave.

You can control the seven-segment display using the following code:

```
// 7-segment readout
#define LED_ADDR           0x21
BYTE xdata Digit[] = { 0xc0, 0xf9, 0xa4, 0xb0, 0x99, 0x92, 0x82, 0xf8, 0x80, 0x98, 0x88,
0x83, 0xc6, 0xa1, 0x86, 0x8e };

  #ifdef Seven_segment
  EZUSB_INITI2C();                      // initialize I2C for 7-seg readout
  #endif

// update 7-seg readout with number of IN packets in EP6 waiting for transfer to the
host
#ifdef Seven_segment
  waiting_inpkts = (EP6CS & 0xF0)>>4;                //right shift by 4 bits
  EZUSB_WriteI2C(LED_ADDR, 0x01, &(Digit[waiting_inpkts]));
  EZUSB_WaitForEEPROMWrite(LED_ADDR);
#endif
```

12 Summary

This application note described how to set up the GPIF to transfer data over an 8-bit asynchronous interface (to the slave FIFO of another EZ-USB FX2LP). It includes hardware setup, the creation of GPIF waveforms, and the writing of 8051 code, which arbitrarily handles both USB INs and OUTs.

The document centered on a specific back-to-back board setup with two EZ-USB FX2LP boards. However, many concepts and insights conveyed in this application note can be applied to and used as a basic framework for mainstream applications.

13 Related Documents

The following documents provide the additional information required to understand the FX2LP chip and to run the firmware projects attached to this application note:

- EZ-USB Development Kit User Guide: This document contains information on how to use the CY3684 kit for the first time and is available at *C:\Cypress\USB\CY3684_EZ-USB_FX2LP_DVK\1.1\Documentation* (after FX2LP DVK installation).
- AN65209 - Getting Started with FX2LP: This document serves as a starting point for the new user to become familiar with FX2LP.
- AN66806 - Getting Started with EZ-USB® FX2LP™ GPIF: This document introduces the GPIF unit and its graphical design tool called GPIF Designer and also includes an example demonstrating how to incorporate a USB connection into a GPIF design.
- AN57322 - Interfacing SRAM with FX2LP over GPIF: This document discusses how to connect Cypress SRAM CY7C1399B to FX2LP over the General Programmable Interface (GPIF). It describes how to create read and write waveforms using the GPIF Designer. This application note is also useful as a reference to connect FX2LP to other SRAMs.
- AN70983 - Designing a Bulk Transfer Host Application for EZ-USB® FX2LP™/FX3™: This document introduces the .NET class library and shows how to create a Windows example to send and retrieve data using a “bulkloop” firmware example running on an FX2LP or FX3 Development Kit (DVK).
- AN73609 - EZ-USB® FX2LP™/FX3™ Developing Bulk-Loop Example on Linux: This document describes how you can use libusb to develop a USB host application on a Linux-based operating system for Cypress EZ-USB FX2LP/FX3 products. This application note is provided only for reference. Cypress also provides Linux SDK which can be found at FX3 SDK web page.
- AN74505 - EZ-USB® FX2LP™ - Developing USB Application on MAC OS X using LIBUSB: This document describes how libusb-1.0 can be used to develop USB host applications (Cocoa Application) on MAC OS X 10.6/10.7 for Cypress EZUSB FX2LP products. This application note is provided only for reference. Cypress also provides MAC SDK which can be found at FX3 SDK web page.
- For complete list of USB Hi-Speed Code Examples, visit the web page.
- EZ-USB Technical Reference Manual: This document includes detailed information on the slave FIFOs and GPIF, including the register-level descriptions.

About the Author

Name: Gayathri Vasudevan.

Title: Applications Engineer Sr

Document History

Document Title: AN63787 - EZ-USB FX2LP GPIF and Slave FIFO Configuration Examples Using an 8-Bit Asynchronous Interface

Document Number: 001-63787

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3016455	CPPK	08/26/2010	New example project.
*A	3172695	CPPK	02/14/2011	Removed all references to FX2 and CY3682 DVK. Updated to new template.
*B	3383901	GAYA	09/28/2011	Updated Document Title. Added new section explaining annual mode operation.
*C	3520175	GAYA	02/14/2012	Converted code example to application note.
*D	3664694	GAYA	07/03/2012	Fixed the code comments. Fixed VID/PID issue.
*E	3720173	GAYA	08/22/2012	Updated Document Title.
*F	4199844	RSKV	11/22/2013	Merged AN6077. Added FX2LP architecture overview section. Enabled debug through LEDs and 7-segment display.
*G	4932954	GAYA	10/05/2015	Updated FX2LP Architecture Overview, GPIF Mode, FX2LP Firmware Project Files Structure, Hardware Connections, FX2LP Manual Mode Operation in GPIF and Slave FIFO Configuration, Testing the Project (Manual Mode) and Related Documents. Updated to new template.
*H	5687801	AESATMP7	04/19/2017	Updated Cypress Logo and Copyright.
*I	6384422	ANNR	11/14/2018	Sunset review Updated template Updated the FX2LP DVK installation path Minor formatting changes

Worldwide Sales and Design Support

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

Products

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

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

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