AN76348

Differences in Implementation of EZ-USB® FX2LP™ and EZ-USB FX3 Applications

Author: Rama Sai Krishna V
Associated Project: No
Associated Part Family: CYUSB3014
Software Version: None
Related Application Notes: For a complete list of the application notes, click here.

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

AN76348 describes the differences between the implementation of applications based on EZ-USB FX2LP™ and EZ-USB FX3™. Through the use of several example applications, you learn about the differences between FX3 and FX2LP at the architectural, hardware level, and firmware framework levels.

Contents

Introduction

Cypress EZ-USB FX3 is the USB 3.0 peripheral controller, with highly integrated and flexible features that allow you to add USB 3.0 functionality to any system.

FX3 has a fully configurable, parallel, general programmable interface called GPIF II, which can connect to an external processor, ASIC, or FPGA. The GPIF II is an enhanced version of the GPIF in FX2LP, Cypress's flagship USB 2.0 product. GPIF II has easy and glueless connectivity to popular interfaces, such as asynchronous SRAM, asynchronous and synchronous address data multiplexed interface, and many others. FX3 has many enhancements over FX2LP; later sections give more details of these.

EZ-USB FX2LP based design cannot be used as it is with the FX3, because these are totally different devices. However, you can modify the application to work with FX3. A simple example, bulkloop is used to explain the differences in the firmware frameworks; and to provide guidelines on how the application can be modified to work on FX3. Bulkloop example shows how data can be loopback over the Bulk endpoints.

Note This application note is targeted to the customers who have experience working with FX2LP device.

These application notes are useful to get start working with FX2LP or FX3.

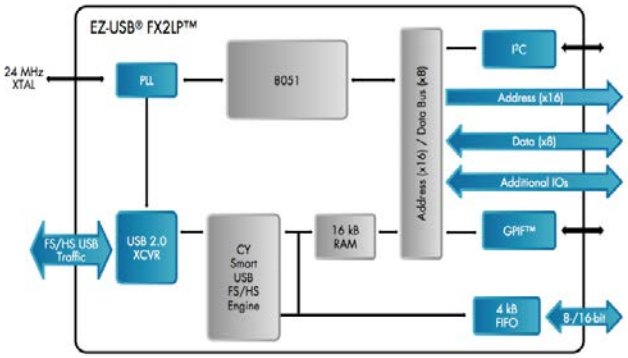
AN65209 - Getting Started with FX2LP™ gives you background information about USB 2.0 along with the hardware, firmware, and software aspects of working with the FX2LP.

AN75705 - Getting Started with EZ-USB® FX3™. background information about USB 3.0 and comparisons to USB 2.0. It details hardware, firmware, and, the software aspects of working with the FX3.

Architectural Differences

This section lists the differences between FX2LP and FX3 with the help of block diagrams. The FX2LP block diagram is in Figure 1.

Figure 1. FX2LP block diagram



FX3 block diagram is shown in Figure 2.

Figure 2. FX3 block diagram

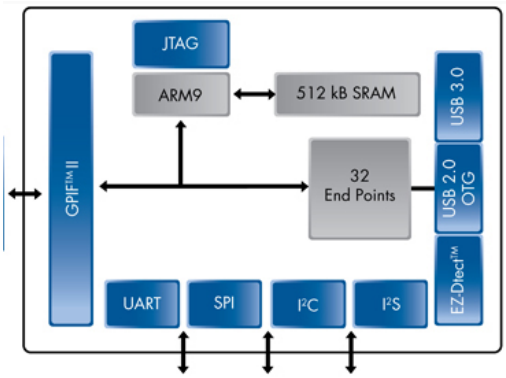


Table 1. Feature difference between FX2LP and FX3

Feature	FX2LP	FX3
Core	8051	ARM926EJ-S
CPU speed	48 MHz	200 MHz
RAM	16 KB	512 KB
Endpoints	7	32
Serial interfaces supported	I^2C, UART	I^2C, UART, I^2S, SPI
Flexible programmable interfaces	GPIF, 48 MHz, 8/16 – bit interface	GPIF II, 100 MHz, 8/16/32 – bit interface
USB	USB 2.0 device	USB 3.0 device (includes USB 2.0 device support), USB 2.0 OTG
Speeds supported	Hi-Speed, Full-Speed	Super-Speed, Hi-Speed, Full-Speed
GPIOs	Up to 40	Up to 60
JTAG debugger	Not available	Supported.
Support for battery charging spec 1.1	No	Yes
Package options	56-pin QFN (8 x 8 mm) 100-pin TQFP (14 x 20 x 1.4 mm) 128-pin TQFP (14 x 20 x 1.4 mm) 56-pin VFBGA (5 x 5 x 1.0 mm) 56-pin SSOP	121-ball FBGA (10 x 10 x 1.2 mm)

Serial Interfaces

Details on the serial interfaces supported by FX2LP and FX3 are listed in Table 2.

Table 2. Serial interfaces supported by FX2LP and FX3

Serial Interface	FX2LP	FX3
I^2C	master only at 100 and 400 kHz	master only at 100 kHz, 400 kHz and 1 MHz
UART	supports only 115.2 K baud and 230.4 K baud	range of baud rates from 300 bps to 4608 Kbps
I^2S	not supported	I^2S Master as transmitter only; sampling frequencies supported by the I^2S interface are 32 kHz, 44.1 kHz, and 48 kHz
SPI	not supported	SPI Master; maximum frequency of operation is 33 MHz

GPIF versus GPIF II

FX3 offers a high-performance general programmable interface, GPIF II. This interface enables functionality similar to, but more advanced than the FX2LP's GPIF and Slave FIFO interfaces.

The GPIF II is a programmable state machine that enables a flexible interface that functions either as a master or slave in either industry standard or proprietary interfaces. Both parallel and serial interfaces can be implemented with GPIF II. But the GPIF of FX2LP is used for parallel interfaces only.

The features of the GPIF II are:

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

Table 3 lists the main difference between FX2LP's GPIF interface and the FX3's GPIF II interface.

Table 3. Differences between GPIF and GPIF II

Feature	GPIF	GPIF II
Interface clock	48MHz	100MHz
Data bus width	8-bit and 16-bit	8-bit, 16-bit, 24-bit and 32-bit
Address lines	9	32 when data bus is 8-bit
Control/status lines	CTL[5:0] and RDY[5:0]	14 when 32 bit data bus is used 16 when 16 or 8 bit data bus is used
number of states	8 (including IDLE)	256
Software tool	GPIF designer	GPIF II designer

Hardware Differences

These application notes help you gain a better idea of hardware design guidelines for FX2LP and FX3.
AN15456 - Guide to Successful EZ-USB FX2LP and EZ-USB FX1 Hardware Design and Debug.
AN70707 - EZ-USB® FX3™/FX3S™ Hardware Design Guidelines and Schematic Checklist

Booting Options

FX2LP boots from USB or from an I^2C EEPROM only. But FX3 can load boot images from various sources, selected by the configuration of the PMODE pins. The boot options for FX3 are:

- Boot from USB
- Boot from I^2C (ATMEL and Microchip EEPROMs are supported)
- Boot from SPI (SPI devices supported are M25P16 (16 Mbit), M25P80 (8 Mbit), M25P40 (4 Mbit) and their equivalents
- Boot from GPIF II Async ADMUX mode
- Boot from GPIF II Sync ADMUX mode
- Boot from GPIF II Async SRAM mode

See AN50963 for more details on FX2LP boot options and AN76405 for more details on FX3 boot options.

Table 4 shows the levels of the PMODE[2:0] signals required for the different booting options.

Table 4. PMODE Signals Setting

PMODE[2:0]	Boot From
F00	Sync ADMUX (16-bit)
F01	Async ADMUX (16-bit)
F11	USB boot
F0F	Async SRAM (16-bit)
F1F	I^2C, On Failure, USB boot
1FF	I^2C only
0F1	SPI, on Failure, USB boot

F = Float. The PMODE pin can be made to float by leaving it unconnected.

If an external EEPROM is used on the I^2C bus for firmware image booting, place 1 kΩ pull-up resistors on the SCL and SDA lines for up to 1 MHz EEPROM communication.

We recommend adding pull-up and pull-down options on the PMODE [2:0] signals and load the combination needed for preferred booting option. Adding the options

gives the flexibility to debug the system during early development.

Crystal/Clock

FX2LP supports only crystal input, while FX3 supports an external clock input along with the crystal support. FX2LP has a CLKOUT pin that can supply 12, 24, or 48 MHz clock. But FX3 does not have this ability to provide a system clock to the external world. This system clock is different from the interface clock provided by GPIF or GPIF II.

Table 5 lists the details of the clock or crystal inputs that these two devices accept.

Table 5. Clock/crystal supported by FX2LP and FX3

	FX2LP	FX3
External clock	Not supported	19.2, 26, 38.4, and 52 MHz
Crystal	24MHz	19.2 MHz
CLKOUT	Available	Not available

For FX3, based on the clocking option that is used, the frequency select, FSLC[2:0] lines can be tied to power, through a weak pull-up resistor, or to ground. Table 6 shows the values of FSLC[2:0] for different clocking options.

Table 6. Frequency select configuration

FSLC[2]	FSLC[1]	FSLC[0]	Crystal/Clock
0	0	0	19.2 MHz crystal
1	0	0	19.2 MHz input clock
1	0	1	26 MHz input clock
1	1	0	38.4 MHz input clock
1	1	1	52 MHz input clock

Power Supply Configurations and Decoupling Capacitance

The supply voltage required for FX2LP is 3.3 V. The FX3 requires multiple power supplies to power each of its internal blocks. Table 7 shows the different power domains and the voltage settings on each of these domains for FX3.

Table 7. FX3 power domains

Parameter	Description	Min (V)	Max (V)	Notes
V_{DD}	Core voltage supply	1.15	1.25	1.2 V typical
A_{VDD}	Analog voltage supply	1.15	1.25	1.2 V typical
V_{IO1}	GPIF II I/O power domain	1.7	3.6	1.8, 2.5 and 3.3 V typical
V_{IO2}	IO2 power domain	1.7	3.6	1.8, 2.5 and 3.3 V typical
V_{IO3}	IO3 power domain	1.7	3.6	1.8, 2.5 and 3.3 V typical
V_{IO4}	UART/SPI/I^2S power domain	1.7	3.6	1.8, 2.5 and 3.3 V typical
V_{IO5}	I^2C and JTAG supply domain	1.15	3.6	1.2, 1.8, 2.5 and 3.3 V typical
V_{BATT}	USB voltage supply	3.2	6	3.7 V typical
V_{BUS}	USB voltage supply	4.1	6	5 V typical
C_{VDDQ}	Clock voltage supply	1.7	3.6	1.8, 3.3 V typical
$U3TX_{VDDQ}$	USB3.0 1.2 V supply	1.15	1.25	1.2 V typical
$U3RX_{VDDQ}$	USB3.0 1.2 V supply	1.15	1.25	1.2 V typical

Table 8 shows the I/O voltage comparison of FX2LP and FX3.

Table 8. I/O voltage comparison

Parameter	Description	Min (V)		Max (V)		Conditions	
		FX2LP	FX3	FX2LP	FX3	FX2LP	FX3
V_{IH}	Input HIGH voltage	2	0.625 x V_{CC}	5.25	V_{CC} +0.3	-	2.0V≤ V_{CC} ≤ 3.6 V*
			V_{CC} - 0.4		V_{CC} +0.3		1.7 V ≤V_{CC} ≤2.0 V*

Par	Descript	Min (V)		Max (V)		Conditions	
V_{IL}	Input LOW voltage	-0.5	-0.3	0.8	0.25 x V_{CC}	-	-
V_{OH}	Output HIGH voltage	2.4	0.9 x V_{CC}	-	-	I_{OUT} = 4mA	I_{OH} (max)= –100µA
V_{OL}	Output LOW voltage	-	-	0.4	0.1 x V_{CC}	I_{OUT} = -0.4mA	I_{OH} (min)= 100µA

*except the USB port; V_{CC} is the corresponding I/O voltage supply.

See the FX2LP, and FX3 datasheets for more details on I/O voltages.

In the case of FX2LP, we need to provide 0.1-µF ceramic capacitors to decouple device power input pins. The specific recommendation for the ceramic capacitor nearest to each FX3 power pin is given in Table 9.

Table 9. Power domain decoupling requirements

Cap Value (µF)	Number of Caps	Pin Name
0.01, 0.1, 22	4 of 0.01µF, 3 of 0.1µF, one 22µF	VDD
0.1, 2.2	1 of each	AVDD
0.1, 22	1 of each	U3TXVDDQ
0.1, 22	1 of each	U3RXVDDQ
0.1, 0.01	1 of each	CVDDQ
0.1, 0.01	1 of each per supply	VIO1-5
0.1	1	VBUS

Software Differences

Development Tools

The FX2LP firmware framework shown in the FX2LP DVK uses the Keil uVision2 IDE.

In the case of FX3, a set of development tools, which includes the GPIF II Designer and the third party ARM software development tools is provided with the SDK. The software development tools include an integrated development environment (Eclipse IDE) with compiler, linker, assembler, and JTAG debugger. You can download the FX3 SDK here:

http://www.cypress.com/?id=3521&rtID=119

USB Host Side Applications

Control Center: Cypress has a new Control Center application that comes along with the FX3 SDK. Use this application to program both FX2LP and FX3. Using this application download the code to RAM or program the EEPROM connected to the FX2LP or FX3 device.

Streamer: The streamer application for FX3, similar to the one in FX2LP. Using this application you measure the throughput numbers for ISO and BULK streams.

BulkLoop: The FX3 SDK includes a bulkloop application to test the bulkloop example. The application gives you the option to send different types of data to run this bulkloop test.

All these applications are found in path once you install the FX3 SDK in the default location:

C:\Program Files\Cypress\EZ-USB FX3 SDK\1.2\application\c_sharp

Table 10 lists the differences in software resources available for FX2LP and FX3.

Table 10.Software resources available for FX2LP and FX3

Software tools	FX2LP	FX3
Compiler	Keil C51 C Compiler	ARM GCC C Compiler
Assembler	Keil A51 Assembler	ARM GCC Assembler
IDE	Keil	Eclipse based IDE
Driver*	CyUSB.sys	CyUSB3.sys
Applications*	Control Center	Control Center
Tool to develop GPIF interface	GPIF Designer	GPIF II Designer**

*You can also use FX3 Driver and Applications to work with FX2LP.

**You cannot use the FX3 GPIF II Designer for FX2LP.

Programmer's View of FX3

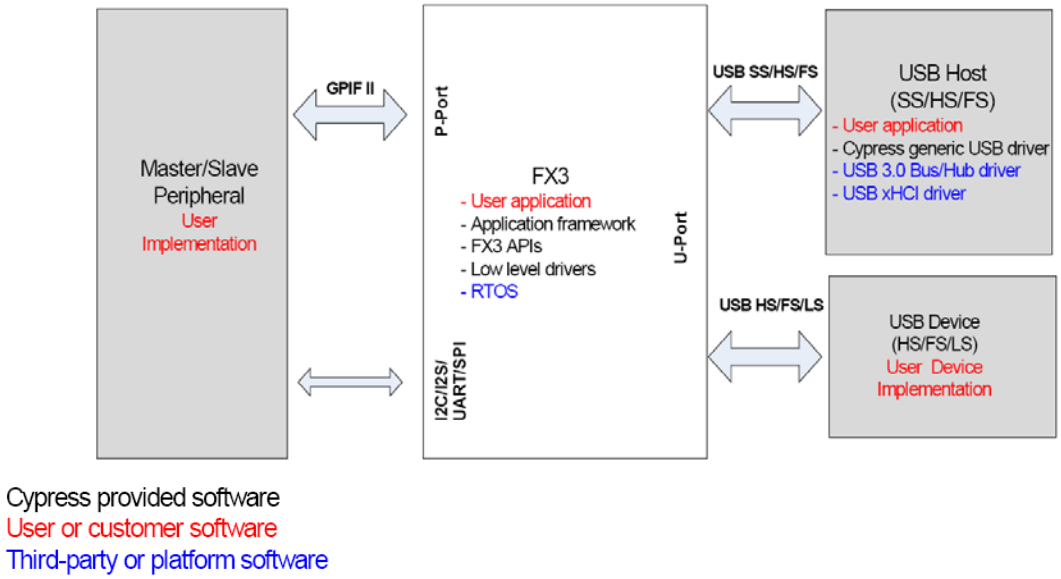
The FX3 comes with the easy-to-use EZ-USB tools giving you a complete solution for fast application development. Use the FX3 device to:

- Configure and manage USB functionality, such as charger detection, USB device/host detection, and endpoint configuration
- Connect to different master and slave peripherals on the GPIF interface
- Connect to serial peripherals (UART, SPI, GPIO, I^2C, I^2S)
- Set up, control, and monitor data flows between the peripherals (USB, GPIF, and serial peripherals)
- Perform necessary operations, such as data inspection, data modification, addition/deletion of packet header and footer information.

The two other important entities that are external to the FX3 are:

- **USB host/device**
  - When the FX3 is connected to a USB host, it functions as a USB device. The FX3 enumerates as a super-speed, high-speed, or full-speed USB peripheral corresponding to the host type.
  - When a USB device is connected, the FX3 plays the role of the corresponding high-speed, full-speed, or low-speed USB host.
- GPIF II master/slave
  - GPIF II is a fully configurable interface and can use any application specific protocol. Any processor, ASIC, DSP, or FPGA can be connected to the FX3. FX3 bootloader or firmware configures GPIF II to support the corresponding interface.

Figure 3. Programming view of FX3



FX2LP and FX3 Firmware Framework Differences

FX2LP Firmware Framework

All Cypress firmware examples are framework based, so that you can start your development by using one of the examples as a reference. These examples are available when you install the FX2LP DVK.

Differences in the firmware framework for FX2LP and FX3 are explained here using the bulkloop example.

BulkLoop Example on FX2LP

The bulkloop example is in this directory (Cypress\USB\Examples\FX2LP\Bulkloop) after installing the FX2LP DVK.

If you look at the FX2LP's firmware framework you see these files:

- ***fw.c:*** This is the main frameworks source file. It contains main(), the task dispatcher, and the SETUP command handler. For most firmware projects, there is no need to modify this file. There are a total of four dispatcher functions called in the main(). They are TD_Init(), TD_Poll(), TD_Suspend(), and TD_Resume(). TD_Init() is called once during the initialization of the frameworks. TD_Poll() is called repeatedly during device operation. The function should contain a state machine that implements the user's peripheral function.

  bulkloop.c: This source file contains initialization and task dispatch function definitions that are called from fw.c. This is where you customize the frameworks for your specific device. In this case, for bulkloop trans*fers.*

- ***dscr.a51:*** Assembly file that contains your device's custom descriptors.

- ***USBJmpTb.OBJ:*** Object code that contains the ISR jump table for USB and GPIF interrupts.

FX2LP Firmware API Library

EZUSB.LIB: The EZ-USB library is an 8051 *.LIB* file that implements functions that are common to many firmware projects. Typically, there is no reason to modify these functions so they are provided in library form. However, the kit includes the library source code in the event that you need to modify a function or if you just want to know how something is done.

The bulkloop application code is implemented in the TD_Poll() function in *bulkloop.c*. Bulkloop is a simple application that can be tested with the help of CyConsole/Control Center and FX2LP DVK. You can see EP2 and 4 are configured as OUT EPs, EP6 and 8 are configured as IN EPs after downloading the bulkloop firmware into FX2LP DVK using CyConsole. Now you can test this example by sending some bytes of data into EP2 and you can read the same data back from EP6. Similarly you can send some bytes to EP4 and read it back from EP8.

FX3 Firmware Framework

Powerful and flexible applications can be rapidly built using FX3 firmware examples and FX3 API libraries. These FX3 firmware examples are available on installing SDK. The FX3 firmware framework is built around a RTOS, and allows the creation/manipulation of multiple application threads. The firmware (or application) framework has all the startup and initialization code. The firmware also contains the individual drivers for the USB, GPIF, and serial interface blocks. The framework:

- Defines the program entry point
- Performs the stack setup
- Performs kernel initialization
- Provides placeholders for application thread startup code

Firmware API Library

The FX3 API library provides a comprehensive set of APIs to control and communicate with the FX3 hardware. These APIs provide a complete programmatic view of the FX3 hardware.

cyfxapi.a, cyu3lpp.a, cyu3threadx.a

A full-fledged API library is in the FX3 SDK. This API library is similar to EZUSB.LIB in case of FX2LP. You need to manually link these libraries to your project.

The API library and the corresponding header files provide all the APIs required for programming the different blocks of the FX3. The APIs provide for:

- Programming each of the individual blocks of the FX3 device - GPIF, USB, and serial interfaces
- Programming the DMA engine and setting up of data flows between these blocks
- The overall framework for application development, including system boot and init, OS entry, and application init
- ThreadX OS calls as required by the application
- Power management features
- Logging capability for debug support.

Embedded Real Time OS

The FX3 firmware framework makes use of an Embedded Real-Time Operating System. The drivers for various peripheral blocks in the platform are typically implemented as separate threads. Standard OS services such as Semaphores, Message Queues, Mutexes, and Timers are used for inter-thread communication and task synchronization; and are available through the library.

The framework gives hooks for the application logic to configure the device behavior and to perform data transfers through it. The application logic itself can be implemented across multiple threads and make use of all the OS services that are used by the Cypress provided drivers.

The ThreadX operating system from Express Logic is used as the embedded RTOS in the FX3 device. All the functionality supported by the ThreadX OS is made available for use by the application logic. There are some constraints on their use are to ensure the smooth functioning of all the drivers.

The ThreadX services are not directly exposed by the firmware framework. This is to ensure that the application logic is independent of the OS used and need not be changed to accommodate a future changes in the embedded OS. The OS services are made available through a set of platform specific wrappers that are placed around them.

BulkLoop Example on FX3

Even though the FX3 is able to implement complex applications, we show you the steps of bulkloop firmware development so that you understand the FX3 firmware framework easily.

Bulkloop example is in this directory (Cypress\EZ-USB FX3 SDK\1.2\firmware\dma_examples\cyfxbulklpauto) after installing the FX3 SDK.

The bulkloop example consists of:

- *cyfx_gcc_startup.S*: FX3 startup code. Explained in later sections.
- *cyfxbulklpauto.h*: This file contains the defines used in cyfxbulklpdscr.c
- *cyfxbulklpdscr.c*: This file contains the USB descriptors. This file is similar to *dscr.a51* in case of FX2LP.
- *cyfxtx.c*: This file defines the porting required for the ThreadX RTOS. This file is provided in source form and must be compiled with the application source code
- *cyfxbulklpauto.c*: This file contains the main application logic of the bulkloop example.

The entry point for the FX3 firmware is the `CyU3PFirmwareEntry()` function. The function is defined in the FX3 API library and is not visible to the user.

The firmware entry function performs these actions:

- Invalidates the caches (which were used by the bootloader)

- Initialize the memory management unit (MMU) and the caches
- Initializes the SYS, FIQ, IRQ, and SVC stack modes
- The execution is then transferred to the Tool chain initialization (`CyU3PToolChainInit()`) function.

Tool Chain Initialization

The next step in the initialization sequence is the tool chain initialization that is defined by the specific Toolchain used and provides a method to initialize the stacks and the C library.

Since all required stack initialization is performed by the firmware entry function, the Toolchain initialization is over ridden, that is, the stacks are not reinitialized.

The tool chain initialization function written for the GNU GCC compiler for ARM processors is presented as an example below. You can find this part of code in *cyfx_gcc_startup.S*. There is no need to modify this file.

```
global CyU3PToolChainInit
CyU3PToolChainInit:

# clear the BSS area
__main:
       mov    R0, #0
       ldr    R1, =_bss_start
       ldr    R2, =_bss_end
1:     cmp    R1, R2
       strlo  R0, [R1], #4
       blo    1b

       b      main
```

In this function, only two actions are performed:

- The BSS area is cleared
- The control is transferred to the `main()`.

Device Initialization

The function `main()` is the C programming language entry for the FX3 firmware. Three main actions are performed in this function.

1. Device initialization: This is the first step in the `main()`.

   ```
   status = CyU3PDeviceInit (NULL);
   if (status != CY_U3P_SUCCESS)
   {
   goto handle_fatal_error;
   }
   ```

   As part of the device initialization:

   a. The CPU clock is setup. A NULL is passed as an argument for `CyU3PDeviceInit()` that selects the default clock configuration.

   b. The VIC is initialized

   c. The GCTL and the PLLs are configured.

2. Device cache configuration: The second step is to configure the device caches. The device has an 8 KB data cache and an 8 KB instruction cache. The instruction cache is enabled as the data cache that is useful only when there is a large amount of CPU based memory accesses. When used in simple cases, the CPU can decrease performance due to large a number of cache flushes, and then cleans itself and it adds complexity to the code.

   ```
   status = CyU3PDeviceCacheControl
   (CyTrue, CyFalse, CyFalse);
   {
   goto handle_fatal_error;
   }
   ```

3. I/O matrix configuration: The third step is the configuration of the required I/Os. This includes the GPIF and the serial interfaces (SPI, I^2C, I^2S, GPIO, and UART).

   ```
   io_cfg.isDQ32Bit = CyFalse;
   io_cfg.useUart   = CyTrue;
   io_cfg.useI2C    = CyFalse;
   io_cfg.useI2S    = CyFalse;
   io_cfg.useSpi    = CyFalse;
   io_cfg.lppMode =
   CY_U3P_IO_MATRIX_LPP_UART_ONLY;
   /* No GPIOs are enabled. */
   io_cfg.gpioSimpleEn[0]  = 0;
   io_cfg.gpioSimpleEn[1]  = 0;
   io_cfg.gpioComplexEn[0] = 0;
   io_cfg.gpioComplexEn[1] = 0;
   status = CyU3PDeviceConfigureIOMatrix
   (&io_cfg);
   if (status != CY_U3P_SUCCESS)
   {
   goto handle_fatal_error;
   }
   ```

   In this bulkloop example:

   a. 16 bit data bus

   b. GPIO, I^2C, I^2S, and SPI are not used

   c. UART is used

   The I/O matrix configuration data structure is initialized and the `CyU3PDeviceConfigureIOMatrix` function (in the library) is invoked.

4. The final step in the `main()` function is an invocation of the OS scheduler. The invocation is done by issuing a call to the `CyU3PKernelEntry()` function. This function is defined in the library and is a non-returning call. It is a wrapper to the actual ThreadX OS entry call. This function:

   a. Initializes the OS

b. Sets up the OS timer used for scheduling

Application Definition

The function `CyFxApplicationDefine()` is called by the FX3 library after the OS is invoked. In this function application specific threads are created. This function is similar to the `TD_Init()` in FX2LP firmware, since this function is called only once.

In the bulkloop example, only one thread is created in the application define function. This is:

```
/* Allocate the memory for the threads */
    ptr = CyU3PMemAlloc
(CY_FX_BULKLP_THREAD_STACK);

/* Create the thread for the application */
    retThrdCreate = CyU3PThreadCreate
(&BulkLpAppThread,                    /*
Bulk loop App Thread structure */

"21:Bulk_loop_AUTO",
/* Thread ID and Thread name */

BulkLpAppThread_Entry,
/* Bulk loop App Thread Entry function */
                          0,
/* No input parameter to thread */
                          ptr,
/* Pointer to the allocated thread stack */

CY_FX_BULKLP_THREAD_STACK,
/* Bulk loop App Thread stack size */

CY_FX_BULKLP_THREAD_PRIORITY,
/* Bulk loop App Thread priority */

CY_FX_BULKLP_THREAD_PRIORITY,
/* Bulk loop App Thread priority */
                          CYU3P_NO_TIME_SLICE,
/* No time slice for the application thread */
CYU3P_AUTO_START
/* Start the Thread immediately */
                          );
```

Note that more threads (as required by the user application) can be created in the application define function. All other FX3 specific programming must be done only in the user threads.

Application Code

In the bulkloop example, 1 Auto DMA channel is created after setting up the Producer (OUT) and Consumer (IN) endpoint. This DMA channel connects the two sockets of the USB port. Two endpoints 1 IN and 1 OUT are configured as bulk endpoints. The endpoint maxPacketSize is updated based on the speed.

```
CyU3PUSBSpeed_t usbSpeed =
CyU3PUsbGetSpeed();
/* First identify the usb speed. Once that
is identified,
 * create a DMA channel and start the
transfer on this. */

/* Based on the Bus Speed configure the
endpoint packet size */
    switch (usbSpeed)
    {
        case CY_U3P_FULL_SPEED:
        size = 64;
        break;

        case CY_U3P_HIGH_SPEED:
        size = 512;
        break;

        case CY_U3P_SUPER_SPEED:
        size = 1024;
        break;

        default:
        CyU3PDebugPrint (4, "Error! Invalid
USB speed.\n");
        CyFxAppErrorHandler
(CY_U3P_ERROR_FAILURE);
        break;
    }

    CyU3PMemSet ((uint8_t *)&epCfg, 0,
sizeof (epCfg));
       epCfg.enable = CyTrue;
    epCfg.epType = CY_U3P_USB_EP_BULK;
    epCfg.burstLen = 1;
    epCfg.streams = 0;
    epCfg.pcktSize = size;

    /* Producer endpoint configuration */
    apiRetStatus =
CyU3PSetEpConfig(CY_FX_EP_PRODUCER,
&epCfg);
    if (apiRetStatus != CY_U3P_SUCCESS)
    {
        CyU3PDebugPrint (4,
"CyU3PSetEpConfig failed, Error code =
%d\n", apiRetStatus);
        CyFxAppErrorHandler (apiRetStatus);
    }

    /* Consumer endpoint configuration */
    apiRetStatus =
CyU3PSetEpConfig(CY_FX_EP_CONSUMER,
&epCfg);
    if (apiRetStatus != CY_U3P_SUCCESS)
    {
        CyU3PDebugPrint (4,
"CyU3PSetEpConfig failed, Error code =
%d\n", apiRetStatus);
        CyFxAppErrorHandler (apiRetStatus);
    }
```

```
/* Create a DMA Auto Channel between two
sockets of the U port.
 * DMA size is set based on the USB speed.
*/
    dmaCfg.size = size;
    dmaCfg.count =
CY_FX_BULKLP_DMA_BUF_COUNT;
    dmaCfg.prodSckId =
CY_FX_EP_PRODUCER_SOCKET;
    dmaCfg.consSckId =
CY_FX_EP_CONSUMER_SOCKET;
    dmaCfg.dmaMode = CY_U3P_DMA_MODE_BYTE;
    dmaCfg.notification = 0;
    dmaCfg.cb = NULL;
/*In case if we are going to use DMA Manual
then we need assign a call back function to
dmaCfg.cb. (dmaCfg.cb =
CyFxBulkLpDmaCallback;)
    dmaCfg.prodHeader = 0;
    dmaCfg.prodFooter = 0;
    dmaCfg.consHeader = 0;
    dmaCfg.prodAvailCount = 0;

    apiRetStatus = CyU3PDmaChannelCreate
(&glChHandleBulkLp,
            CY_U3P_DMA_TYPE_AUTO, &dmaCfg);
    if (apiRetStatus != CY_U3P_SUCCESS)
    {
        CyU3PDebugPrint (4,
"CyU3PDmaChannelCreate failed, Error code =
%d\n", apiRetStatus);
        CyFxAppErrorHandler(apiRetStatus);
    }
```

Application Thread

The Application entry point for the bulkloop example is the `BulkLpAppThread_Entry ()` function. This function is similar to `TD_Poll ()` in FX2LP firmware, where we write the application logic.

```
/* Entry function for the BulkLpAppThread.
*/
void
BulkLpAppThread_Entry (uint32_t input)
{
    /* Initialize the debug module */
    CyFxBulkLpApplnDebugInit();

    /* Initialize the bulk loop application */
    CyFxBulkLpApplnInit();

    for (;;)
    {
        CyU3PThreadSleep (1000);
    }
}
```

The main actions performed in this thread are:

- Initializing the debug mechanism
- Initializing the main bulkloop application

Each of these steps is explained as follows.

Debug Initialization

The debug module uses the UART to output the debug messages. The UART must be configured before the debug mechanism is initialized. This is done by invoking the UART init function.

```
/* Initialize the UART for printing debug
messages */
apiRetStatus = CyU3PUartInit();
```

The next step is to configure the UART. The UART data structure is first filled in and this is passed to the UART SetConfig function.

```
/* Set UART Configuration */
uartConfig.baudRate =
CY_U3P_UART_BAUDRATE_115200;
uartConfig.stopBit =
CY_U3P_UART_ONE_STOP_BIT;
uartConfig.parity = CY_U3P_UART_NO_PARITY;
uartConfig.txEnable = CyTrue;
uartConfig.rxEnable = CyFalse;
uartConfig.flowCtrl = CyFalse;
uartConfig.isDma = CyTrue;
apiRetStatus      =      CyU3PUartSetConfig
(&uartConfig, NULL);
```

The UART transfer size is set next that is configured to be infinite in size. So that the total debug prints are not limited to any size.

```
/* Set the UART transfer */
apiRetStatus    =    CyU3PUartTxSetBlockXfer
(0xFFFFFFFF);
```

Finally the Debug module is initialized. The two main parameters are:

- The destination for the debug prints, which is the UART socket
- The verbosity of the debug that is set to level 8, so all debug prints that are below this level (0 to 7) will be printed.

```
/* Initialize the Debug application */
apiRetStatus = CyU3PDebugInit
(CY_U3P_LPP_SOCKET_UART_CONS, 8);
```

Application Initialization

The application initialization consists of these steps:

USB Initialization

- The USB stack in the FX3 library is first initialized. The initialization is done by invoking the USB Start function.

```
/* Start the USB functionality */
apiRetStatus = CyU3PUsbStart();
```

- The fast enumeration is the easiest way to setup a USB connection, where all enumeration phase is handled by the library. Only the class and vendor requests need to be handled by the application. In case of FX2LP, this enumeration part is handled in the function SetupCommand(void) in *fw.c*.

  The next step is to register for callbacks. In this example, callbacks are registered for USB Setup requests and USB Events.

```
CyU3PUsbRegisterSetupCallback(CyFxBulkLpApplnUSBSetupCB, CyTrue);

    /* Setup the callback to handle the USB events. */

CyU3PUsbRegisterEventCallback(CyFxBulkLpApplnUSBEventCB);
```

  The callback functions and the callback handling are described in later sections.

- The USB descriptors are set and this is done by invoking the USB set descriptor call for each descriptor.

```
/* Set the USB Enumeration descriptors */
/* Device Descriptor */
apiRetStatus = CyU3PUsbSetDesc(CY_U3P_USB_SET_HS_DEVICE_DESCR, NULL,
(uint8_t *)CyFxUSB20DeviceDscr);
```

  The previous code snippet is for setting the Device Descriptor. The other descriptors set in the example are Device Qualifier, Other Speed, Configuration, BOS (for Super Speed), and String Descriptors.

- The USB pins are connected. The FX3 USB device is visible to the host only after this action.

  Hence it is important that all setup is completed before the USB pins are connected.

```
/* Connect the USB Pins */
/* Enable Super Speed operation */
apiRetStatus = CyU3PConnectState(CyTrue, CyTrue);
```

USB Setup Callback

Since the fast enumeration model is used, only vendor and class specific requests are received by the application. Standard requests are handled by the firmware library. Since there is no vendor or class specific requests to be handled, the callback just returns CyFalse.

```
/* Callback to handle the USB setup requests. */
CyBool_t
CyFxBulkLpApplnUSBSetupCB (
        uint32_t setupdat0, /* SETUP Data 0 */
        uint32_t setupdat1  /* SETUP Data 1 */
    )
{
    /* Fast enumeration is used. Only class, vendor and unknown requests
     * are received by this function. These are not handled in this
     * application. Hence return CyFalse. */
    return CyFalse;
}
```

USB Event Callback

The USB events of interest are: Set Configuration, Reset, and Disconnect. The bulkloop is started on receiving a SETCONF event and is stopped on a USB reset or USB disconnect.

```
/* This is the callback function to handle the USB events. */
void
CyFxBulkLpApplnUSBEventCB (
    CyU3PUsbEventType_t evtype, /* Event type */
    uint16_t            evdata  /* Event data */
    )
{
    switch (evtype)
    {
        case CY_U3P_USB_EVENT_SETCONF:
        /* Stop the application before re-starting. */
        if (glIsApplnActive)
            {
                CyFxBulkLpApplnStop ();
            }
        /* Start the loop back function. */
        CyFxBulkLpApplnStart ();
        break;

        case CY_U3P_USB_EVENT_RESET:
        case CY_U3P_USB_EVENT_DISCONNECT:
        /* Stop the loop back function. */
        if (glIsApplnActive)
            {
                CyFxBulkLpApplnStop ();
            }
        break;

        default:
        break;
    }
}
```

DMA Setup

The DMA channel transfer is enabled:

```
/* Set DMA Channel transfer size */
    apiRetStatus = CyU3PDmaChannelSetXfer
(&glChHandleBulkLp,
CY_FX_BULKLP_DMA_TX_SIZE);
    if (apiRetStatus != CY_U3P_SUCCESS)
    {
        CyU3PDebugPrint (4,
"CyU3PDmaChannelSetXfer Failed, Error code
= %d\n", apiRetStatus);
        CyFxAppErrorHandler(apiRetStatus);
    }
```

There is no `CyFxBulkLpDmaCallback` function since we use the Auto commit mode. But if you plan to use the manual channel then you need to commit the buffer in `CyFxBulkLpDmaCallback` function using this code.

```
if (type == CY_U3P_DMA_CB_PROD_EVENT)
    {
     status = CyU3PDmaChannelCommitBuffer
(chHandle, input->buffer_p.count, 0);
        if (status != CY_U3P_SUCCESS)
        {
            CyU3PDebugPrint (4,
"CyU3PDmaChannelCommitBuffer failed, Error
code = %d\n", status);
        }
     }
```

You can download USBBulkLoopAuto.img (*.img* is the file that you get after the project is built) to FX3 using CyControl and test this bulkloop example using CyControl or *BulkLoop.exe*.

For more details on the FX3 software development kit, see the documents available in the path Cypress\EZ-USB FX3 SDK\1.2\doc. (1.2 in this path is version of the SDK, it may change in future).

Differences between FX2LP Slave FIFO interface and FX3 Slave FIFO interface

This section explains the differences in synchronous Slave FIFO interface of FX2LP and FX3.

The synchronous Slave FIFO interface is suitable for applications in which an external processor or device needs to perform data read/write accesses to FX2LP or FX3's internal FIFO buffers. Register accesses are not done over the Slave FIFO interface.

The following two application notes give you the more details of Slave FIFO interface of FX2LP and FX3.

AN61345 - Describes the Synchronous Slave FIFO interface of FX2LP. Also, a sample design is included to demonstrate how an FPGA can be interfaced to FX2LP using synchronous Slave FIFO.

AN65974 - Describes the Synchronous Slave FIFO interface of FX3. Also, a complete design example is included to demonstrate how an FPGA can be interfaced to FX3 using synchronous Slave FIFO.

Table 11 lists the differences in synchronous Slave FIFO interface signals available for FX2LP and FX3.

Table 11. Synchronous Slave FIFO interface signals

Signal Name		Signal Description
FX2LP	FX3	
SLCS#	SLCS#	This is the chip select signal for the Slave FIFO interface, which needs to be asserted to perform any access to the Slave FIFO interface.
SLWR#	SLWR#	This is the write strobe for the Slave FIFO interface. It must be asserted for performing write transfers to Slave FIFO.
SLRD#	SLRD#	This is the read strobe for the Slave FIFO interface. It must be asserted for performing read transfers from Slave FIFO.
SLOE#	SLOE#	This is the output enable signal. It causes the data bus of the Slave FIFO interface to be driven by FX2LP or FX3. It must be asserted for performing read transfers from Slave FIFO.
PKTEND#	PKTEND#	The PKTEND# signal is asserted in order to write a short packet or a zero length packet to Slave FIFO
FIFOADR[1:0]	A[1:0]	2-bit address lines to select one of the 4 (EP2, EP4, EP6, EP8) endpoints in FX2LP. In case of FX3, these two bit address lines are used to address 4 sockets.*
FD[15:0]	D[31:0]	Data bus of the Slave FIFO interface. Data bus width supported by FX2LP is 8-bit or 16-bit. Data bus width supported by FX3 is 8-bit or 16-bit or 32-bit.
IFCLK	PCLK	This is the Slave FIFO interface clock. The maximum frequency supported in case of FX2LP Slave FIFO interface is 48MHz. The maximum frequency supported in case of FX3 Slave FIFO interface is 100MHz.

*See AN65974 for more details on FX3 sockets.

Flag Usage

The FLAG signals are monitored by the external processor for flow control. There are four flags (Flag A, Flag B, Flag C, Flag D) to report the status of FX2LP's FIFOs. The FLAGA, FLAGB, and FLAGC pins can operate in either of two modes: Indexed or Fixed, as selected by the PINFLAGSAB and PINFLAGSCD registers. The FLAGD pin operates in Fixed mode only. FLAG-FLAGC pins can be configured independently; some pins can be in Fixed mode while others are in Indexed mode. Flag pins configured for Indexed mode report the status of the FIFO currently selected by the FIFOADR[1:0] pins. Refer to "Slave FIFOs" chapter of FX2LP Technical Reference Manual.

The FX3 Slave FIFO interface is more flexible than the FX2LP Slave FIFO interface Since the FX3 Slave FIFO interface is developed by configuring GPIF II to act as a slave and with the help of a state diagram. Cypress provides this state diagram needed for implementing Slave FIFO interface along with the GPIF II project templates. This can be customized by using GPIF II designer, if needed. In the standard implementation, two flags are configured to show empty/full/partial status for a dedicated thread or the current thread being addressed. Refer to AN65974 for more details. User can add more flags, if needed.

Differences between an FX2LP-based UVC Camera Design and an FX3-based UVC Camera Design

This section starts with the implementation of UVC camera with the help of FX2LP and then it explains how to implement the same design with FX3.

UVC is a USB standard class that allows a video streaming device to be connected to a USB host to stream video like a webcam using standard UVC driver.

Image sensor interface

The various signals that are associated with transferring an image are: (these unidirectional signals are from image sensor to FX3)

1. FV - Frame Valid (indicates the start and stop of a frame)

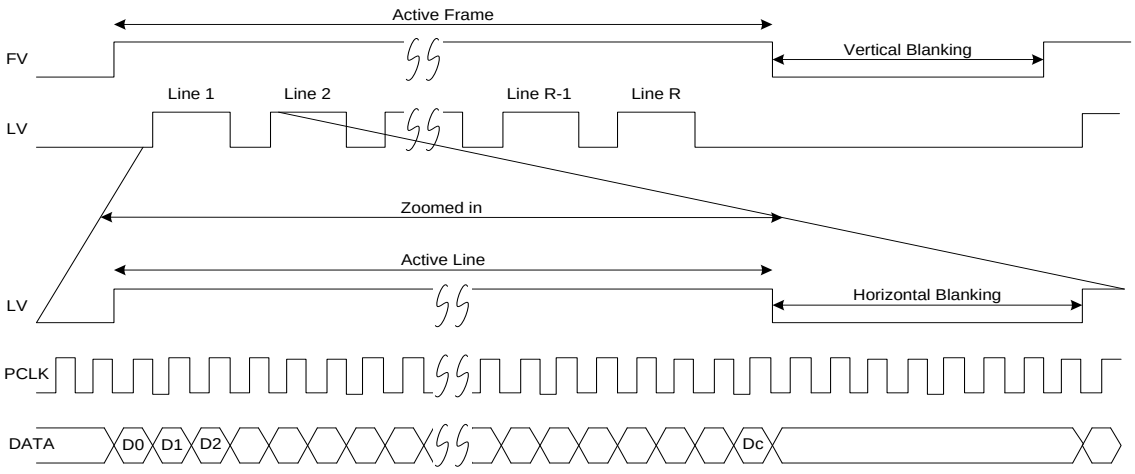
2. LV - Line Valid (indicates start and stop of a line)

3. PCLK – Pixel clock (the data output of the image sensor is synchronized with the pixel clock)

4. Data – 8 to 32 bit data lines for image data

Figure 4 shows the timing diagram of the FV, LV, PCLK and Data signals. The FV signal is asserted to indicate the start of a frame. Then, the image data is transferred line by line. The LV signal is asserted during each line transfer when the data is driven by the image sensor. This data can be 8bit to 32bit simultaneous transfer from the image sensor.

Figure 4. Image sensor interface timing diagram



System level diagram of USB camera is shown in Figure 5.

Figure 5. USB camera block diagram

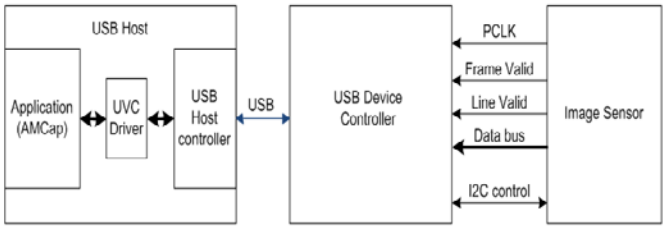


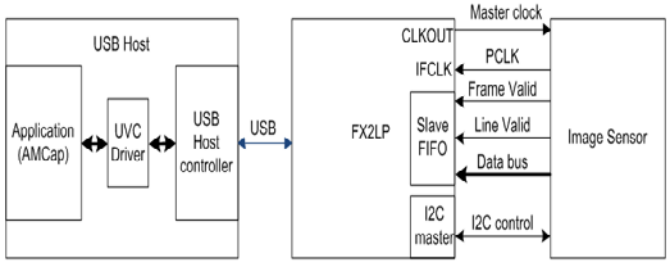
Image sensors typically use an I^2C interface to allow a controller to configure the image sensor registers. The I^2C block of FX2LP or FX3 can act as an I^2C master to configure the image sensor with the correct parameters.

Implementation with FX2LP

There are two ways to connect an image sensor to FX2LP, using Slave FIFO interface or using GPIF. FX2LP acts as slave in case of Slave FIFO interface and FX2LP acts as master in case if image sensor is connected to GPIF. The implementation of an image sensor interface becomes much simpler if it is connected to a Slave FIFO interface.

This block diagram shows how to connect an image sensor to FX2LP:

Figure 6. UVC camera design using FX2LP



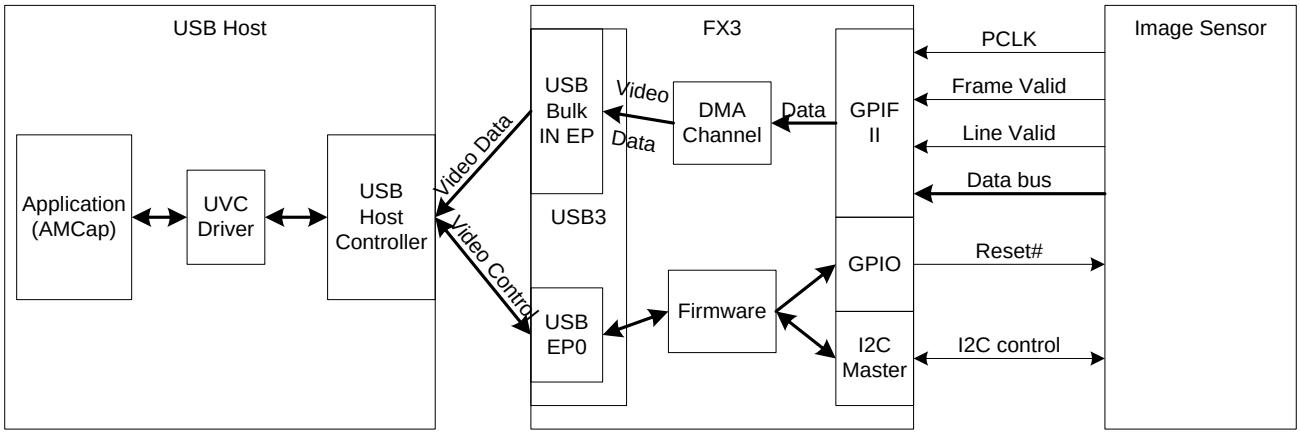
FX2LP provides a master clock (from CLKOUT pin) to sensor that eliminates the use of an extra crystal. But FX2LP can output only clock frequency of 12/24/48 MHz. If a different clock frequency is required for the image sensor, add a separate crystal in the design.

The FX2LP is configured in synchronous Slave FIFO mode. The FX2LP supplies a 12 MHz clock to the image sensor. The image sensor is configured using the FX2LP I2C module. Once the image sensor is configured, it gives the image data with Pixel Clock (PCLK) as 6 MHz. The Frame Valid is directly given to the SLCS# line of FX2LP and Line Valid is connected to SLWR# of Slave FIFO interface. UVC header is added at the start of each frame.

A host application like AMCap communicates through the UVC driver to configure the image sensor over a video control interface and receive video data over the video streaming interface.

Implementation with FX3

As described in earlier sections, GPIF and Slave FIFO interfaces are separate in FX2LP. But in FX3, the same GPIF II interface needs to be configured to act as master or save FIFO interface. the image sensor is interfaced to GPIF II of FX3 and GPIF II is configured to be compatible with the image sensor interface. So a state machine needs to be developed using GPIF II Designer tool to read data from the image sensor by monitoring the control signals named Frame Valid and Line Valid.



The image sensor is configured using the FX3 I2C module. The FX3 DMA channel streams data from the image sensor to internal buffers where a UVC header is added to the image data. This video data is then sent to the video streaming endpoint. More details on designing GPIF II state machine and application firmware are in AN75779. The same FX3 design can also be used in

USB2.0 mode. Here are the advantages of using this FX3 design in USB 2.0 compared to an FX2LP design:

- Image sensor interface can be operated in 24-bit or 32-bit. 16-bit is the maximum data bus width that you can use with FX2LP GPIF.
- UVC header can be added easily with the help of power full ARM processor in FX3.
- FX3 can also act as a SPI master if the image sensor needs to be configured over the SPI interface.
- The GPIF interface clock is limited to 48MHz where as GPIF II clock can go to 100MHz.
- The FX2LP maximum endpoint memory is 4 KB.

Use of an I²C module in FX2LP and FX3

Image sensor registers are configured through the I²C interface. FX2LP and FX3 both have the I²C module, which can act as master. Standard API functions are provided to perform read and write operations over the I²C interface.

EZUSB_WriteI2C() and EZUSB_ReadI2C() functions are used to write and read image sensor registers over FX2LP's I²C module. These two functions are part of EZUSB.LIB.

CyU3PI2cTransmitBytes() and CyU3PI2cReceiveBytes() functions are used to write and read image sensor registers over FX3's I²C module. These two functions are part of cyu3lpp.a library. Refer to the project attached to the application note AN75779 for more details.

How to debug firmware of FX2LP and FX3 using UART

Serial port debugging makes it possible to print debug messages and real-time values of variables to the HyperTerminal program. In this UVC camera application, register configuration can be verified by reading them back over the I²C interface and printing them on a HyperTerminal program.

AN58009 discusses the code to add to FX2LP firmware project to enable this debugging feature. Com port settings are: 38400 – 8 – None – 1- None. See AN58009 for more details.

In order to enable this debug feature on FX3, initialize and configure UART as described in the "**Debug Initialization"** section. Com port settings needed are: 115200 – 8 - None – 1 – None. See the project attached to the application note AN75779 for more details.

Available Collateral

FX2LP Development Kit

CY3684 EZ-USB FX2LP Development Kit

FX3 Development Kit

CYUSB3KIT-001 EZ-USB® FX3™ Development Kit

FX2LP Datasheet

CY7C68013A, CY7C68014A, CY7C68015A, CY7C68016A: EZ-USB® FX2LP™ USB Microcontroller High-Speed USB Peripheral Controller

FX3 Datasheet

CYUSB3014

FX3 SDK

EZ-USB FX3 Software Development Kit

FX2LP GPIF Designer

GPIF Designer

FX3 GPIF II Designer

GPIF™ II Designer

Application Notes

- AN75705 - Getting Started with FX3
- AN65209 - Getting Started with FX2LP™
- AN68829 - Slave FIFO Interface for EZ-USB® FX3™: 5-Bit Address Mode
- AN65974 - Designing with the EZ-USB® FX3 Slave FIFO Interface
- AN63787 - EZ-USB® FX2LP™ GPIF and Slave FIFO Configuration Examples Using 8-bit Asynchronous Interface
- AN70707 - EZ-USB® FX3 Hardware Design Guidelines and Schematic Checklist
- AN15456 - Guide to Successful EZ-USB FX2LP and EZ-USB FX1 Hardware Design and Debug.
- AN76405 - EZ-USB® FX3 Boot options
- AN50963 - EZ-USB® FX1™/FX2LP™ Boot Options
- AN75779 - Interfacing an Image Sensor to EZ-USB® FX3™ in a USB video class (UVC)
- Go to http://www.cypress.com to download the latest version of the product collateral

About the Author

Name:	Rama Sai Krishna V
Title:	Applications Engineer Sr.
Background:	Rama Sai Krishna holds an M.Tech in Systems and Control Engg. from IIT Bombay. He is currently working on Cypress USB peripherals.
Contact:	rskv@cypress.com

Document History

Document Title: Differences in Implementation of EZ-USB® FX2LP™ and EZ-USB FX3 Applications – AN76348

Document Number: 001-76348

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3565979	RSKV	03/30/2012	New application note
*A	3943464	RSKV	03/26/2013	Updated Document Title to read as “Differences in Implementation of EZ-USB® FX2LP™ and EZ-USB FX3 Applications - AN76348”. Updated Architectural Differences (Added Figure 1 and Figure 2, updated Table 1). Updated GPIF versus GPIF II (Added Table 3). Updated Hardware Differences (Added references to hardware guidelines application notes, added reference to FX3 and FX2LP boot options application notes, updated Crystal/Clock (Updated Table 5), updated Power Supply Configurations and Decoupling Capacitance (Added Table 8, updated Table 9)). Updated Software Differences (Updated USB Host Side Applications (Added Table 10). Added Differences between FX2LP Slave FIFO interface and FX3 Slave FIFO interface. Added Differences between an FX2LP-based UVC Camera Design and an FX3-based UVC Camera Design.

Worldwide Sales and Design Support

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

Products

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

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5LP

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

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

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