Cypress EZ-USB® FX3™ SDK

Quick Start Guide

Version 1.3.1

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intl): 408.943.2600
http://www.cypress.com

Contents

1 FX3 SDK

1.1 Device Overview

Cypress EZ-USB FX3 is the next generation USB3.0 peripheral controller which is highly integrated, flexible and enables system designers to add USB 3.0 capability to any system.

The EZ-USB FX3S device is an extension to the FX3 device, which adds the capability to control SD, eMMC and SDIO peripherals to the list of features supported.

The EZ-USB CX3 controller is an extension to the EZ-USB FX3 device which adds the ability to interface with and perform uncompressed video transfers from Image Sensors implementing the MIPI CSI-2 interface.

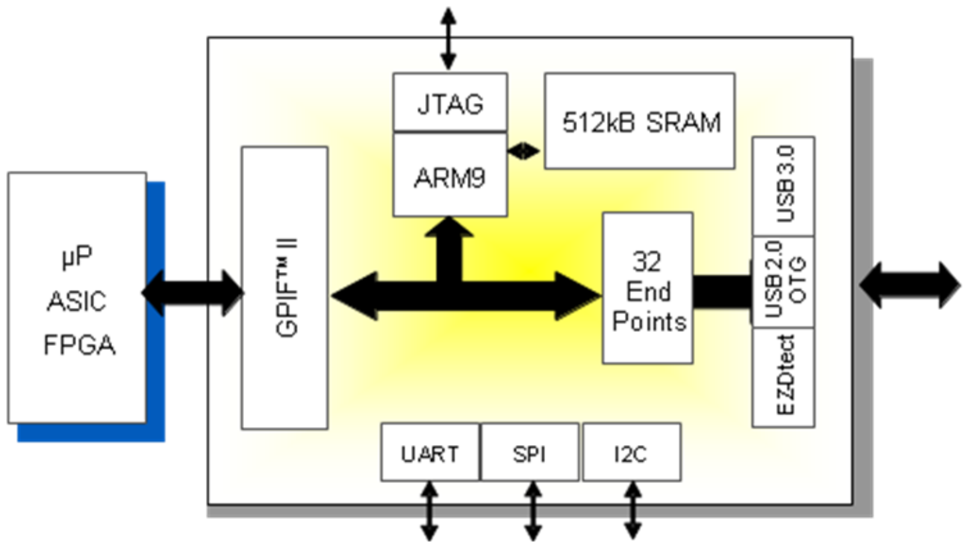


Figure 1-1: EZ USB FX3 System Diagram

The FX3 and FX3S devices have a fully configurable, parallel, General Programmable Interface called GPIF II, which can connect to any processor, ASIC, DSP, or FPGA.

The General Programmable Interface GPIF II is an enhanced version of the GPIF in FX2LP, Cypress's flagship USB2.0 product. The GPIF II Controller drives the behavior and timing of the GPIF II Interface based on one or more user configurable state machines. It provides an easy and glueless interface to popular interfaces such as asynchronous SRAM, synchronous SRAM, Address Data Multiplexed interface, parallel ATA, and can be programmed to implement most other parallel communication protocols.

The CX3 device makes use of a fixed GPIF II configuration to interface the USB portion of the controller with the MIPI CSI-2 interface. Therefore, the GPIF II interface is not available for other purposes in the CX3 device.

All of these devices have an integrated USB Phy and controller along with a 32-bit microcontroller (ARM926EJ-S) for powerful data processing and for building custom applications. The device has 512 KB (256 KB in some cases) of on-chip RAM that can be used for code and data storage. It has an inter-port DMA architecture which enables data transfers at speeds greater than 400 MBps.

These devices are fully compliant to the USB 3.0 v1.0 and USB 2.0 specifications. An integrated USB 2.0 OTG controller enables applications that need dual role usage scenarios. The device is also compliant with the USB Battery Charging Specification v1.1.

There are also serial peripherals such as UART, SPI, I2C, and I2S for communicating to on board peripherals (for example the I2C interface is typically connected to an EEPROM).

1.2 FX3 SDK Overview

Cypress delivers the complete software and firmware stack for FX3/FX3S/CX3 in order to easily integrate all USB applications in the embedded system environment. The software development kit comes with application examples which help accelerate application development.

The components of the FX3 software development kit are shown in Figure 1-2.

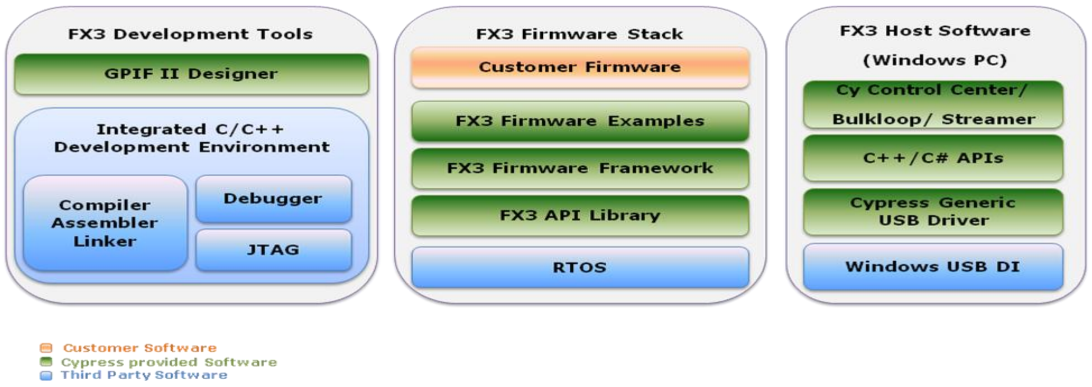


Figure 1-3: FX3/FX3S Software Solution (SDK)

On the device side, there is the FX3 firmware stack that consists of a full fledged API library and a comprehensive firmware framework. A well documented library of APIs enable users to access the hardware functions of the device. The SDK also includes application implementation examples in source form demonstrating different usage models. Users can also develop their own application using this firmware framework and modifying it as applicable. There is complete flexibility for users to develop custom applications using the firmware framework. An RTOS library is integrated with the firmware stack, allowing users to implement complex applications requiring multiple threads of firmware execution.

A set of development tools is provided with the SDK, which includes the GPIF II Designer and third party compiler tool-chain and IDE.

The firmware development environment will help the customer develop, build and debug firmware applications for FX3. The third party ARM® software development tool provides an integrated development environment with compiler, linker, assembler and JTAG debugger. A recent build of the free GNU tool-chain for ARM processors and an Eclipse based IDE is provided by Cypress.

The USB host side (Microsoft Windows) stack for the FX3 contains:

- Cypress generic USB 3.0 driver (WDF) on Windows 7 (32/64 bit) and Windows Vista (32/64 bit) and Windows XP (32 bit)
- Convenience APIs that expose generic USB driver APIs through C++ and C# interfaces
- USB control center, a Windows utility that provides interfaces to interact with the device at low levels such as configuring end points, data transfers etc
- Bulkloop application, a user application to perform data loopback on the Bulk endpoints.
- Streamer application, a user application to perform data streaming over Isochronous or Bulk endpoints.

1.3 FX3 DVK Board Overview

Cypress provides a FX3 DVK board that can be used for firmware and system development using the FX3 device. The DVK board provides the means to connect an external processor or device to the GPIF interface and to connect slave devices to the I2C, SPI and I2S interfaces.

The FX3 DVK Board can be used to execute the firmware examples in the SDK and also for firmware development based on the SDK. The hardware connection and programming sections below assume the use of the FX3 DVK board.

Please refer to the FX3 DVK User Guide for additional information regarding the DVK board.

2 SDK Installation

2.1 Components of the FX3 SDK

The FX3 SDK installation includes multiple components:

1. FX3 Firmware – This contains the FX3 Firmware libraries, Header files, Example code and firmware conversion utility
2. ARM GCC – This contains the GNU Toolchain for ARM processors
3. Eclipse – The eclipse IDE with the required plug-ins
4. USB Suite – The windows host driver, C++ & C# API libraries, and the Control center, Bulkloop & Streamer applications.
5. GPIF II Designer – Windows based tool for configuring the GPIF II port of the FX3 device.

The FX3 SDK installer installs all of these components when the “Typical” or “Complete” install options are chosen. They can also be selectively installed by using the “Custom” option on the installer.

The default installation path for the FX3 SDK is:

C:\Program Files\Cypress\EZ-USB FX3 SDK\<version>, where 1.3 is the version for this release of the SDK.

Note: On 64-bit Windows installations, the default installation path for the SDK is:

C:\Program Files (x86)\Cypress\EZ-USB FX3 SDK\<version>

The FX3 SDK also includes the following documentation:

1. FX3 Release Notes
2. FX3 Programmer’s Manual
3. FX3 API guide
4. FX3 SDK Trouble Shooting guide

After installation, these documents can be found in the <Installation folder>\doc\ folder.

2.2 Installed Directory Structure

The following figure shows the FX3 firmware related files in the FX3 SDK installation.

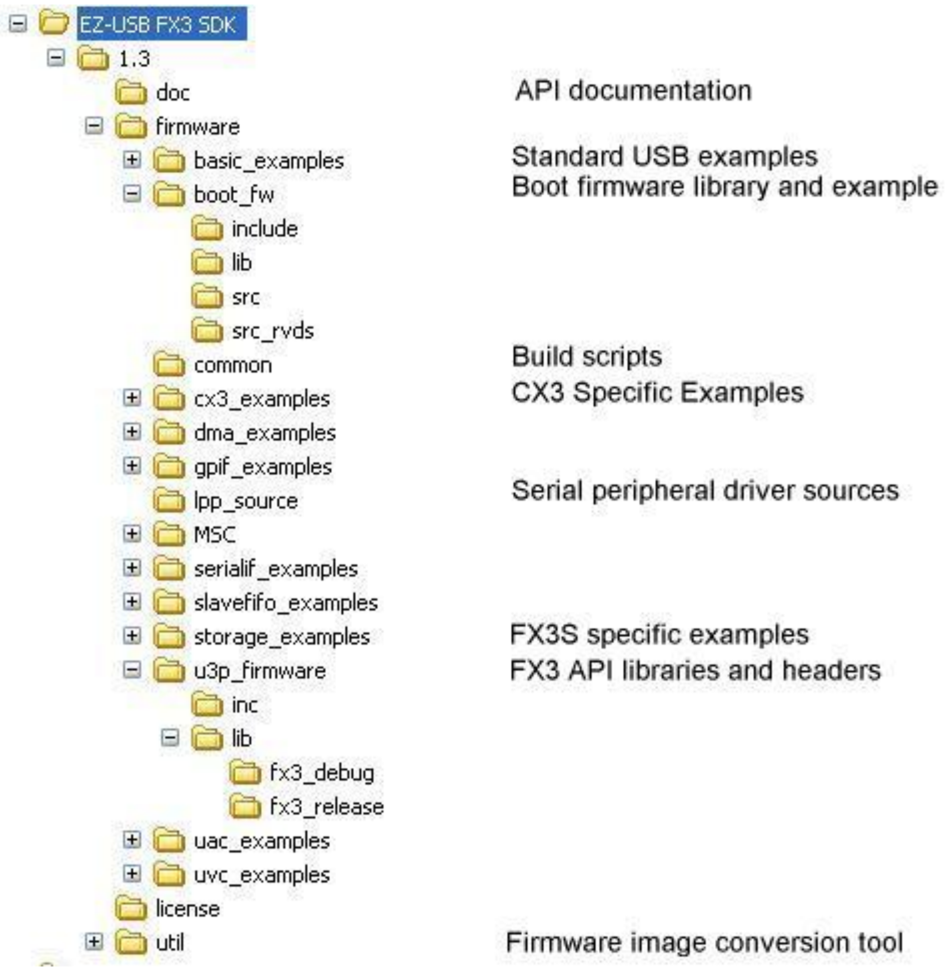
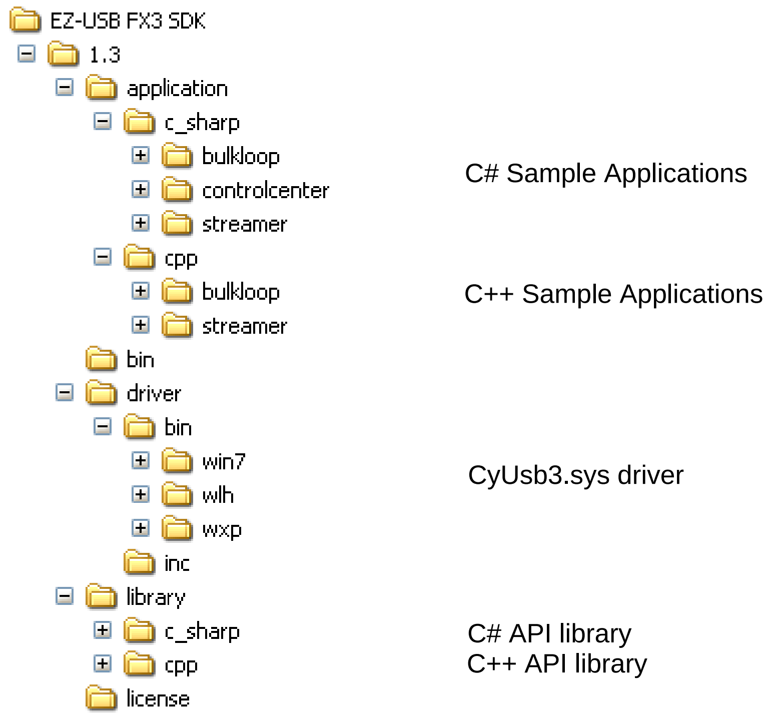


Figure 2-1: FX3 SDK Installed Directory Structure

The following figure shows the USBSuite related files in the FX3 SDK installation.



```
EZ-USB FX3 SDK
  1.3
    application
      c_sharp
        bulkloop
        controlcenter
        streamer
      cpp
        bulkloop
        streamer
    bin
    driver
      bin
        win7
        wlh
        wxp
      inc
    library
      c_sharp
      cpp
    license
```

Figure 2-2: USBSuite Installed Directory Structure

3 Working with the SDK

3.1 Programming the FX3 device

The FX3 examples can be built and run on the FX3 DVK. The following steps describe the setup and boot process:

1. Install the components of the SDK (Firmware, Toolchain, IDE and Host drivers) on the host machine.
2. Connect the FX3 board to the host machine and power the board up.
3. Bind the driver for the FX3 device.
4. Launch the IDE, import the example projects and build them
5. Launch the CyControl Center utility (available from USB Suite software)
6. Download the boot image to the FX3 device using the Control Center “Download Firmware” feature.
7. The FX3 device will reboot using this new firmware.

These steps are explained in more detail in the following sections.

The UsbBulkLoopAuto example is used here to explain the steps. This example implements a vendor specific USB 3.0 peripheral with two bulk endpoints. The firmware application loops back any data that is sent on the OUT endpoint on the IN endpoint.

Section 8 of the FX3 Programmers Manual describes the overall application structure.

It is assumed that the FX3 DVK board is used to run the firmware.

Note: The CyControl utility is a Windows based utility, and the above steps assume that a Windows Host is being used. Please refer to the CyUSB for Linux documentation for a description of how to perform these steps on a Linux host.

3.2 Building the firmware

Each of the firmware examples in the FX3 SDK has an Eclipse project associated with it, which can be used for building and debugging the application. The instructions for building and running the Firmware examples using the Eclipse IDE and GNU tool-chain are provided in detail in Section 11.2.2 of the FX3

Programmers Manual. The section details how to import the projects, configure settings for the projects, debug and run the example using GNU debugger.

Import all of the firmware examples in the SDK into the Eclipse IDE. The example applications are automatically built when they are imported, and generate ELF binaries. These binaries are converted by the Eclipse projects to binary image format using the elf2img utility. This binary image format can be programmed onto the FX3 part. The elf2img binary conversion utility is part of the FX3 SDK (<Installation folder>\util\elf2img). The eclipse project post build step uses the following command for performing the conversion.

```
elf2img.exe –i BulkLoopAuto.elf –o BulkLoopAuto.img
```

Refer to the readme.txt under (<Installation folder>\util\elf2img) for more information on the conversion tool and the command line options. Also refer to the application note AN76405 - EZ-USB FX3 Boot Options for information about the boot procedure and options.

3.2.1 Serial Peripheral Drivers and APIs

The drivers and APIs for the FX3 serial peripheral interfaces (I2C, I2S, SPI, UART and GPIO) are provided with the FX3 SDK in source form. Full register documentation for these blocks is part of the FX3 Programmer's Manual.

The drivers and APIs for the serial peripherals are built into a separate ARM EABI library (cyu3lpp.a). The source code as well as the Eclipse IDE projects for building this library can be found in the SDK under the <Installation Folder>\firmware\lpp_source folder.

All of the FX3 application examples link with this library in addition to the FX3 API library consisting of the drivers and APIs for the device, USB and GPIF blocks (cyfxapi.a).

3.3 Setting up the FX3 DVK Board

The FX3 device can be configured to boot from a USB host by enumerating as a custom device. The boot mode for the device is specified through a set of three PMODE pins that can be controlled through some jumpers and switches on the FX3 DVK board.

Please refer to the FX3 DVK User Guide for the jumper settings to control the PMODE pins.

3.4 Host driver binding

The steps for installation and binding of the host drivers for different Windows platforms are described in section 3 of the host driver help file, CyUSB.pdf, installed as part of the USB Suite.

Once the board has been connected to the host PC, the device will be seen enumerating and asking for driver selection. For booting, the FX3 enumerates as a USB 2.0 device with VID=0x04B4 and PID=0x00F3.

Select the “Install from a specific location” and point to the *<Installation Root>/driver/bin/<os>/<arch>/cyusb3.inf* file to bind the device with the Cypress CyUsb3.sys driver.

To bind with a new device or application with a different VID/PID pair; the corresponding VID/PID entries must be made in the cyusb3.inf file.

3.5 Firmware Download

The CyControl center utility, which can be used to communicate with the FX3 device, is described in CyControlCenter.pdf, installed as part of the USB Suite.

Once the driver binding has been completed, open the CyControl Center (Start->Programs->Cypress->Cypress USBSuite->Control Center). The Cypress FX3 Device should be visible through the tool and you can look through the USB descriptors reported by the device.

Use the Program->FX3 option on the tool to program the previously generated BulkLoopAuto.img binary file onto the device. This download is performed directly onto the RAM on the FX3 device.

3.6 Testing the application

Once the download is complete, a new USB device can be seen enumerating and asking for driver selection. Repeat the driver selection process performed during initial boot and associate the same CyUsb3.sys driver with the new device as well.

The data transfer feature of the CyControl Center can be used to verify the data loop-back functionality.

4 Firmware Example Overview

The firmware examples included in the FX3 SDK are listed below. These examples are provided as separate Eclipse projects.

4.1 USB Bulk data loopback examples

These examples illustrate a loopback mechanism between two/three USB Bulk Endpoints. The example comprises of Vendor Class USB enumeration descriptors with 2/3 Bulk Endpoints. The DMA multichannel examples use 3 endpoints for the loopback.

Following are the different types of Bulk data loopbacks provided. These examples are provided as Eclipse projects.

1. cyfxbulklpauto: This example makes use of the DMA AUTO Channel for the loopback between the endpoints.
2. cyfxbulklpautosig: This example makes use of the DMA AUTO Channel with Signaling for the loopback between the endpoints.
3. cyfxbulklpmanual: This example makes use of the DMA MANUAL Channel for the loopback between the endpoints.
4. cyfxbulklpmaninout: This example makes use of the DMA MANUAL IN + DMA MANUAL OUT Channel for the loopback between the endpoints.
5. cyfxbulklpautomanytoone: This example makes use of the Multichannel DMA AUTO MANY TO ONE for the loopback between endpoints.
6. cyfxbulklpautoonetomany: This example makes use of the Multichannel DMA AUTO ONE TO MANY for the loopback between endpoints.
7. cyfxbulklpmanonetomany: This example makes use of the Multichannel DMA MANUAL ONE TO MANY for the loopback between endpoints.
8. cyfxbulklpmanmanytoone: This example makes use of the Multichannel DMA MANUAL MANY TO ONE for the loopback between endpoints.
9. cyfxbulklpmulticast: This example makes use of the Multichannel DMA MULTICAST for the loopback between endpoints.
10. cyfxbulklpman_removal: This example demonstrates the use of DMA MANUAL channels where a header and footer get removed from the data before sending

out. The data received from EP1 OUT is looped back to EP1 IN after removing the header and footer. The removal of header and footer does not require the copy of data.

11. cyfxbulklplowlevel: The DMA channel is a helpful construct that allows for simple data transfer. The low level DMA descriptor and DMA socket APIs allow for finer constructs. This example uses these APIs to implement a simple bulkloop back example where a buffer of data received from EP1 OUT is looped back to EP1 IN.
12. cyfxbulklpmandcache: FX3 device has the data cache disabled by default. The data cache is useful when there is large amount of data modifications done by the CPU. But enabling D-cache adds additional constraints for managing the data cache. This example demonstrates how DMA transfers can be done with the data cache enabled.

All the above examples are implemented to work at USB 2.0 and USB 3.0 speeds.

4.2 USB Isochronous data loopback examples

These examples illustrate a loopback mechanism between two USB Isochronous Endpoints. The example comprises of Vendor Class USB enumeration descriptors with 2 Isochronous Endpoints.

Following are the different types of Isochronous data loopbacks provided. These examples are provided as Eclipse projects.

1. cyfxisolpauto: This example makes use of the DMA AUTO Channel for the loopback between the endpoints.
2. cyfxisolpmaninout: This example makes use of the DMA MANUAL IN + DMA MANUAL OUT Channel for the loopback between the endpoints.

The above examples are implemented to work at USB 2.0 and USB 3.0 speeds.

4.3 USB debug example

cyfxusbdebug: This example demonstrates the use of USB interrupt endpoint to log the debug data from the FX3 device. The default debug logging in all other examples are done through the UART. This example shows how any consumer socket can be used to log FX3 debug data.

4.4 FX3S Storage Examples

1. FX3SMassStorage: Implementation of a USB mass storage class device using the FX3S APIs backed with SD/eMMC storage devices. This example demonstrates the high data rates that are achievable on the USB -> Storage data path through the FX3S device.
2. GpifToStorage: Example that shows how to access SD/eMMC storage devices from an external processor that connects to the FX3S device through the GPIF port.

3. FX3SFileSystem: Example that demonstrates the use of the FX3S storage API to integrate a file system into the firmware application. The FatFs file system by Elm Chan is used here to manage FAT volumes stored on the cards connected to FX3S.
4. FX3SSdioUart: This example implements a USB Virtual COM port that uses a Arasan SDIO – UART bridge chip for the UART functionality. This example shows how the SDIO APIs can be used to transfer data between the USB host and a SDIO peripheral.
5. FX3SRaid0: This example implements a RAID-0 disk using a pair of SD cards or eMMC devices for storage. Access to the RAID-0 disk is provided through a USB mass storage class (MSC) interface. This implementation provided nearly twice the throughput of a MSC function using a single SD card or eMMC device.

4.5 USB Video Class example

1. cyfxuvcinmem: This example implements a USB Video Class Driver with the help of the appropriate USB enumeration descriptors. With these descriptors the FX3 device enumerates as a USB Video Class device on the USB host. The video streaming is accomplished with the help of a DMA Manual Out channel. Video frames are stored in contiguous memory locations as a constant array. The streaming of these frames continuously from the device results in a video like appearance on the USB host.
2. cyfxuvcinmem_bulk: This example demonstrates the USB video class device stack implementation for FX3. The example is similar to the UVC example, but uses Bulk endpoints instead of isochronous endpoints.

These examples are implemented to work at USB 2.0 and USB 3.0 speeds.

4.6 Slave FIFO Application examples

The Slave FIFO application example demonstrates data loopback between the USB Host and the PCI host. The example comprises of two data pipes between the USB Host and the PCI host. Data is looped back at the PCI host. The loopback is observed on the USB host. GPIF™ - II interface is configured to implement the Slave FIFO protocol for Sync and Async modes (16 and 32 bit configurations).

1. cyfxslaveasync: Asynchronous mode Slave FIFO example using 2 bit FIFO address.
2. cyfxslavesync: Synchronous mode Slave FIFO example using 2 bit FIFO address.
3. cyfxslaveasync5bit: Asynchronous mode Slave FIFO example using 5 bit FIFO address.
4. cyfxslavesync5bit: Synchronous mode Slave FIFO example using 5 bit FIFO address.

The above examples are implemented to work at USB 2.0 and USB 3.0 speeds.

4.7 Serial Interface examples

The serial interface examples demonstrate data accesses to the Serial IOs, I2C, SPI and UART.

4.7.1 UART examples

In the UART examples, the data is looped from the PC host back to the PC host through the FX3 device. The loopback can be observed on the PC host. The examples are provided for both DMA mode and Register mode of data transfers for the UART.

1. cyfxuartlpdmamode: This example makes use of the DMA MANUAL Channel for the loopback to the PC host over UART. The DMA buffer is 32 bytes.
2. cyfxuartlpregmode: This example loops back data one byte at a time.
3. cyfxusbuart: This example implements a CDC-ACM compliant USB to UART bridge device using the UART port on the FX3 device.

4.7.2 I2C examples

In the I2C examples, a USB I2C data loopback is achieved over 2 USB Isochronous endpoints through an I2C slave device. A DMA MANUAL IN and a DMA MANUAL OUT channels are used for the Isochronous endpoints. The data is looped back one I2C page at a time which is 64 bytes. The examples are implemented for USB 2.0 speeds.

1. cyfxusbi2cdmamode: This example makes use of the DMA MANUAL IN and a DMA MANUAL OUT channel for the I2C DMA access.
2. cyfxusbi2cregmode: This example reads/writes one page to the I2C device using the register mode accesses.

4.7.3 SPI examples

In the SPI examples, a USB SPI data loopback is achieved over 2 USB Isochronous endpoints through a SPI slave device. A DMA MANUAL IN and a DMA MANUAL OUT channels are used for the Isochronous endpoints. The data is looped back one page at a time which is 16 bytes. The examples are implemented for USB 2.0 speeds.

1. cyfxusbspidmamode: This example makes use of the DMA MANUAL IN and a DMA MANUAL OUT channel for the SPI DMA access.
2. cyfxusbspiregmode: This example reads/writes one page to the SPI device using the register mode accesses.
3. cyfxusbspigpiomode: This example demonstrates the use of GPIO to build an SPI master. The example read / writes data to an SPI Flash attached to the FX3 device using FX3 GPIOs.

4.7.4 I2S example

cyfxusbi2sdmamode: This example demonstrates the use of I2S APIs. The example sends the data received on EP1 OUT to the left channel and EP2 OUT to the right channel.

4.7.5 GPIO examples

This is a simple GPIO application example illustrating the usage of GPIO operations with the help of an Input and an Output GPIO.

4. cyfxgpioapp: This example implements simple Set operation on the Output GPIO and Get operation on the Input GPIO. It also implements GPIO interrupt for the Input GPIO.
5. cyfxgpiocomplexapp: This example implements the following features:
   a) A PWM output.
   b) Input line which measures low time of the input signal.
   c) A counter which increments on the negative edge of input signal.

4.8 USB Bulk/Isochronous data source sink examples

These examples illustrate data source and data sink mechanism with two USB Bulk/Isochronous Endpoints. The example comprises of Vendor Class USB enumeration descriptors with 2 Bulk/Isochrounous Endpoints. The examples are implemented to work for USB 2.0 and USB 3.0 speeds.

1. cyfxbulksrcsink: This example makes use of the DMA MANUAL IN channel for sinking the data received from the Bulk OUT endpoint and DMA MANUAL OUT Channel for the sourcing the data to the Bulk IN endpoint.
2. cyfxisosrcsink: This example makes use of the DMA MANUAL IN channel for sinking the data received from the Isochronous OUT endpoint and DMA MANUAL OUT Channel for the sourcing the data to the Isochronous IN endpoint.
3. cyfxisosrc: This example makes use of a DMA MANUAL OUT channel to implement a single Isochronous IN endpoint. Multiple bandwidth settings and vendor specific control transfers are also supported in this example.
4. cyfxgpiftousb: This example makes use of a DMA AUTO channel to continuously stream data to a BULK IN endpoint. The data is sourced by continuously latching data from the GPIF II data bus. As no control signals are used, there is no need for any actual device connectivity on the GPIF II interface when using this example. This helps identify the maximum USB throughput through the FX3 device, when there are no firmware bottlenecks involved.

4.9 USB Bulk Streams example

This example illustrates data source and data sink mechanism with two USB Bulk Endpoints using the Bulk Streams. A set of Bulk Streams are defined for each of the endpoints (OUT and IN). Data source and data sink happen through these streams. Streams are applicable to USB 3.0 speeds only. The example works similar to Bulk source sink example (cyfxbulksrcsink) when connected to USB 2.0.

1. cyfxbulkstreams: This example makes use of the DMA MANUAL IN channel for sinking the data received from the streams of the Bulk OUT endpoint and DMA MANUAL OUT Channel for the sourcing the data to the streams of the Bulk IN endpoint.

4.10 USB enumeration example

The enumeration example is an implementation of a USB bulk loop with normal mode enumeration in FX3.

1. cyfxbulklpautoenum: The example illustrates the normal mode enumeration mechanism provided for FX3 where all setup requests are handled by the application. The example implements a simple bulk loop.

4.11 Flash Programmer example

This example (cyfxflashprog) illustrates the programming of I2C EEPROMS and SPI flash devices from USB. The read / write operations are done using pre-defined vendor commands. The utility can be used to flash the boot images to these devices.

4.12 Mass Storage Class example

This example (cyfxmscdemo) illustrates the implementation of a USB mass storage class (Bulk Only Transport) device using a small section of the FX3 device RAM as the storage device. The example shows how mass storage commands can be parsed and handled in the FX3 firmware.

4.13 USB Audio Class Example

This example creates a USB Audio Class compliant microphone device, which streams PCM audio data stored on the SPI flash memory to the USB host. Since the audio class does not require high bandwidth, this example works only at USB 2.0 speeds.

4.14 Two Stage Booter Example

A simple set of APIs have been provided as a separate library to implement two stage booting. This example demonstrates the use of these APIs. Configuration files that can be used for Real View Tool chain are also provided.

4.15 USB host and OTG examples

These examples demonstrate the host mode and OTG mode operation of the FX3 USB port.

1. cyfxusbhost: Mouse and MSC driver for FX3 USB Host. This example demonstrates the use of FX3 as a USB 2.0 single port host. The example supports simple HID mouse class and simple MSC class devices.
2. cyfxusbotg: FX 3 as an OTG Device. This example demonstrates the use of FX3 as an OTG device which when connected to a USB host is capable of doing a bulkloop back using DMA AUTO channel. When connected to a USB mouse, it can detect and use the mouse to track the three button states, X, Y, and scroll changes.
3. cyfxbulklpotg: FX3 Connected to FX3 as OTG Device. This example demonstrates the full OTG capability of the FX3 device. When connected to a USB PC host, it acts a bulkloop device. When connected to another FX3 in device mode running the same the firmware, both can demonstrate session request protocol (SRP) and host negotiation protocol (HNP).

4.16 CX3 Examples

These examples demonstrate the implementation of various USB video streaming modes using the CX3 device and MIPI CSI-2 compliant image sensor.

1. cycx3_rgb16_as0260: This example implements a Bulk-only RGB-565 video streaming example over the UVC protocol which illustrates the usage of the CX3 APIs using an Aptina AS0260 sensor. It streams uncompressed 16-Bit RGB-565 video from the image sensor over the CX3 to the host PC.
2. cycx3_rgb24_as0260: This example implements a Bulk-only RGB-888 video streaming example over the UVC protocol which illustrates the usage of the CX3 APIs using an Aptina AS0260 sensor. It streams uncompressed 16-Bit RGB-565 video from the image sensor to the MIPI interface on the CX3, which up-converts the stream to 24-Bit RGB-888 and transmits to the host PC over USB.
3. cycx3_uvc_as0260: This example implements a Bulk-only UVC 1.1 compliant example which illustrates the use of the CX3 APIs using an Aptina AS0260 sensor. This example streams Uncompressed 16-Bit YUV video at data rates of from the image sensor over the CX3 to the host PC.
4. cycx3_uvc_ov5640: This example implements a Bulk-only UVC 1.1 example, which illustrates the use of the CX3 APIs using an Omnivision OV5640 sensor. This example streams Uncompressed 16-Bit YUV video from the image sensor over the CX3 to the host PC.

5 FX3 Programming Guidelines

This section provides a few basic guidelines for developing applications using the FX3 SDK.

5.1 Device Initialization

5.1.1 Clock Settings

The first step involved in initializing the FX3 device is setting up the frequencies for various internal clocks. There are two classes of clocks on the FX3 device:

1. Some of the clocks such as the clock for the ARM CPU, the memory mapped register access and system DMA are expected to run continuously at all times.
2. Other clocks such as those for the USB block, the GPIF block, and the serial peripherals are only enabled when required; i.e., when the corresponding block init function has been called.

All of the clocks on the FX3 device are derived from a master clock which runs at approximately 400 MHz. If the FX3 device is being clocked using a 19.2 MHz crystal or using a 38.4 MHz clock input; the master clock frequency is set to 384 MHz by default. If the FX3 device is being clocked using a 26 MHz or 52 MHz input clock, the master clock frequency is set to 416 MHz.

The clock frequency for the ARM CPU is set to one half of the master clock frequency. The clock frequency for the system DMA and register access is set to one-fourth of the master clock frequency. These frequencies can be reduced further using the CyU3PDeviceInit() API.

If the system design requires the FX3 device to receive data on a 32 bit wide GPIF interface running at 100 MHz (yielding a maximum data rate of 400 MBps on the GPIF interface); a master clock setting of 384 MHz is insufficient for the system DMA to handle the incoming data. In such a case, the master clock needs to be changed to a value greater than 400 MHz.

This change is done through the CyU3PDeviceInit() API call. The clkCfg->setSysClk400 parameter passed to this function needs to be set to CyTrue to enable this frequency change. If this parameter is set to CyTrue, the firmware causes the master clock frequency to be changed to 403.2 MHz.

Note: It has been noted that changing the master clock frequency causes a transient instability on the device interfaces, which may cause an ongoing JTAG debug session to break. To minimize the impact of this instability, this frequency change is performed by firmware before any of the external interfaces on the device are initialized.

5.1.2 Setting up the IO Matrix

The next step in the device initialization is setting up the functionality for various IO pins on the device. Almost all of the device IOs can serve multiple functions, and the actual function to be used is selected through the CyU3PDeviceConfigureIOMatrix API call.

Please note that some constraints apply to the possible IO configurations. For example, it is not possible to use the SPI interface on the FX3 device if the GPIF is running in a 32 bit wide configuration.

Any of the pins on the device can be overridden to function as a GPIO instead of serving as part of another interface such as GPIF, UART, I2C etc. The CyU3PDeviceConfigureIOMatrix() makes some sanity checks to ensure that a pin that is otherwise in use, cannot be overridden as a GPIO pin.

e.g., if the UART interface is enabled using the useUart parameter; the API does not allow any of the UART pins (TX, RX, RTS and CTS) to be used as GPIOs.

It is possible that the above checks in the API constrain the user. For example, if the user does not plan to use flow control for the UART, there is no reason to prevent the RTS and CTS pins from being used as GPIOs.

In such a case, the CyU3PDeviceGpioOverride() API can be used to forcibly override the pin functionality to serve as a GPIO.

Please note that the storage ports are only available of the FX3S (CYUSB3035) device, which does not support a 32-bit wide GPIF configuration. Therefore, the s0Mode and s1Mode values should be set to CY_U3P_SPORT_INACTIVE for all FX3 applications; and the isDQ32Bit value should be set to CyFalse for all FX3S applications.

5.1.3 Using the Caches

The FX3 device implements 8 KB of instruction and data caches that can be used to speed up instruction and data access from the ARM CPU. It is recommended that the instruction cache be kept enabled in all applications for optimal functioning of the firmware.

If the firmware application makes use of any CPU bound data copy actions, turning the data cache on will improve the performance significantly.

The instruction and data caches are enabled/disabled using the CyU3PDeviceCacheControl() API.

Whenever the data cache is turned on, care needs to be exercised to prevent data corruption due to unexpected cache line evictions. It is required that the isDmaHandleDCache parameter to the CyU3PDeviceCacheControl() API be set to CyTrue, whenever the isDCacheEnable parameter is set to CyTrue.

5.1.4 Initializing other interface blocks

This section applies to the initialization sequence for interface blocks like USB, GPIF, I2C, UART etc. All of these blocks will be held in reset at the time when firmware execution starts.

The procedure for turning on any of these blocks involves:

1. Turning on the clock for the block
2. Bringing the block out of reset.
3. All of the interface blocks on the FX3 device (except the GPIOs) have a set of DMA sockets associated with them in addition to the core interface logic. It is required that these sockets be reset and initialized with clean default values when the corresponding block is being turned on.

These steps are performed by the init function associated with these blocks (CyU3PUsbStart, CyU3PPibInit, CyU3PUartInit etc.).

Since the DMA sockets associated with a block are reset during the block initialization; it is expected that the blocks are initialized before any DMA channels using these sockets are created by the application.

e.g., the PIB (GPIF) block needs to be initialized before any DMA channels using the PIB sockets are created.

The user also needs to ensure that the block is not repeatedly initialized at a later stage, thereby affecting the DMA channel functionality.

5.2 Embedded Operating System

The FX3 firmware libraries include the ThreadX Operating System from Express Logic, Inc. All of the OS services that are provided in version 5.1 of the ThreadX operating system are included in the cyu3threadx.a library. A set of OS abstraction wrappers (macros and functions) are provided around the core ThreadX API, to allow porting of the firmware solution to other embedded Oses with similar feature sets.

The debug (fx3_debug) build of the OS library provides OS services with additional error checks built in. This checks result in greater memory footprint and slower execution, and are disabled in the release (fx3_release) build.

5.2.1 Execution Model

The ThreadX operating system supports the creation of multiple threads with different priority levels. There is no direct restriction on the number of threads.

However, each thread requires a set of resources (memory for context data structures and thread stack, as well as a CPU execution time); and there will be a practical limit which is application specific.

ThreadX assigns thread priorities ranging from 0 to 31, with 0 being the highest priority. It is expected that the highest thread priorities are reserved for the driver threads described in section 5.2.2. User level application threads are recommended to use priorities ranging from 8 and lower.

While the ThreadX scheduler supports Pre-emption and time slicing, it is recommended that a co-operative scheduling mechanism be used in FX3 firmware applications. The FX3 drivers and APIs have only been tested under co-operative scheduling conditions, and the use of time slices can cause errors.

5.2.2 Driver Threads

The FX3 firmware framework makes use of a set of driver threads, which are started up before the user hook for application definition (CyFxApplicationDefine) is called. The table below shows the threads that are started up by the FX3 framework.

Thread	Description	Priority	Stack Size
DMA	DMA driver that handles various DMA related interrupts from all hardware blocks in the FX3 device. This thread implements most of the data processing logic associated with manual DMA channels.	2	1 KB
PIB	PIB/GPIF driver that handles GPIF and MMC Slave related interrupts.	4	1 KB
LPP	Serial peripheral driver that handles interrupts raised by all serial peripheral interfaces (UART, I2C, SPI, I2S and GPIO) on the FX3 device.	4	1 KB
USB	USB driver thread that handles all USB related interrupts (USB 3.0, USB 2.0, and USB 2.0 host/OTG) on the FX3 device.	4	1 KB
Debug	Debug log handler. Takes care of sending CyU3PDebugPrint and CyU3PDebugLog messages out through the selected debug port.	6	512 bytes
SIB	Storage driver that handles SD/MMC interface related interrupts and processes device hotplug events.	4	2 KB

All of the driver thread stacks are created on the Memory Heap that is initialized through the CyU3PMemInit() function. The FX3 framework code requires about 10 KB of space in this memory heap.

When implementing a typical USB 3.0 device application, the USB and DMA driver threads are expected to be very active, with the other threads becoming active quite rarely.

5.2.3 Callback Functions

The FX3 APIs allow users to register a set of callback functions which will called by the FX3 driver modules s when events of interest occur.

Typically, these callback functions are invoked from the respective driver threads. For example, the DMA callback functions are called from the DMA driver thread; and the USB callback (setup as well as event callbacks) are called from the USB driver thread. As the callback functions are called from the driver threads, they have the same priorities as the driver threads. This means that DMA callback functions take higher priority than any other type of callback in the FX3 system. This is desirable because prompt handling of DMA events is required to achieve the best possible throughput for USB applications.

The GPIO interrupt callback is an exception to the above rule that callback functions are invoked from the driver thread. In this case, the callback function is called from the GPIO interrupt handler itself. Therefore, it is not possible to make any blocking API calls from a GPIO callback function.

It is possible to make blocking API calls from other callback functions like DMA or USB callbacks. However, such usage should be restricted because blocking within a callback will delay other operations to be performed by the driver thread. For example, blocking within a DMA callback will delay the processing of DMA interrupts on all channels in the system.

Note: The CyU3PDebugPrint() function is a blocking API call that waits until a DMA buffer is available for data transfer. Therefore, use of the CyU3PdebugPrint from within a DMA or USB callback function should be avoided.

5.3 Memory Usage

An FX3 firmware application requires the following memory regions:

- Code: This is where all the executable code for the application goes. Part of the code can be placed in the I-TCM of the FX3 device for faster execution. The space required for this regions is application specific, and will be lesser for release builds of the firmware.
- Data: This region contains all initialized data variables in the FX3 application. The space required for this region is application specifix.
- BSS: This region contains all zero initialized data variables in the application.
- Mem Heap: This is a heap region that is used for runtime allocation of thread resources like stacks and message queues. The FX3 firmware library requires 10 KB of memory from this heap. Additional memory will be

required based on the firmware implementation. The FX3 SDK provides an implementation of this heap using the ThreadX byte pool services.

- Buffer Heap: This is a heap region which is used for runtime allocation of buffers used for data transfers through the FX3 device. The SDK provides a home grown implementation of a buffer heap manager, which ensures that all allocated buffers are cache line aligned.
- Runtime Heap: This is an optional heap which is required only if the application makes use of standard memory allocation routines like malloc or new. The size requirement for this heap will depend on the application.

The code, data, bss and runtime heap region boundaries are set through the linker script. The mem heap and buffer heap regions are specified in code while initializing them.

Please refer to the firmware/common/fx3.ld file for a linker script that initializes the code, data and bss regions. Please refer to the firmware/common/fx3cpp.ld file for a linker script that initializes the runtime heap in addition to the above regions. Please refer to the firmware/common/cyfxtx.c source file for the placement and initialization of the mem heap and buffer heap regions.

All of these regions have to be placed in the memory available on the FX3 device, which includes 16 KB of I-TCM and 512 or 256 KB or System RAM.

5.4 USB Device Handling

5.4.1 USB Device Enumeration

The FX3 firmware API supports two models of handling USB device enumeration.

In the fast enumeration model, the user registers a set of USB descriptors with the API; and the USB driver handles the control requests from the USB host using this data. The advantage of this model is that the control request handling in the driver has been tested to pass all USB compliance test requirements; and the user does not need to implement all of the negative condition checks required to pass these tests. The driver will continue to forward control request callbacks for any unknown requests (vendor or class specific, as well as requests for unregistered strings) to the user application.

In the application enumeration model, all control requests will trigger a callback to the user application; and the application can handle the request as suitable. The advantage of this model is that allows the application to implement any number of configurations and provides greater flexibility.

The model of enumeration is selected using the CyU3PUsbRegisterSetupCallback() API.

5.4.2 Handling Control Requests

On receiving a USB control request through the setup callback, the user application can perform one of four actions:

1. Call CyU3PUsbAckSetup() to complete the status handshake part of a control request with no data phase.
2. Call CyU3PUsbStall(0, CyTrue, CyFalse) to stall endpoint 0 so as to fail the control request.
3. Call CyU3PUsbSendEP0Data() to send the data associated with a control request with an IN data phase.
4. Call CyU3PUsbGetEP0Data() to receive the data associated with a control request with an OUT data phase.

It is possible to make multiple SendEP0Data/GetEP0Data calls to complete the data phase of a transfer if the amount of data to be transferred is large. In such a case, the amount of data transferred using each API call should be an integral multiple of the maximum packet size for the control endpoint (64 bytes for USB 2.0, 512 bytes for USB 3.0).

Please note that the FX3 device architecture does not allow a control transfer to be stalled after the data phase has been completed. Therefore, the application should NOT stall the control endpoint if there is an error in handling a control transfer after the OUT data has been read using the CyU3PUsbGetEP0Data API.

e.g., Do not stall EP0 if the write to I2C slave fails after receiving data through the CyU3PUsbGetEP0Data() API call.

5.4.3 Endpoint Configuration

All of the FX3 application examples in the SDK configure the non-control endpoints while processing the SET_CONFIGURATION request. This allows the application to determine the USB connection speed, and then configure the endpoint accordingly.

It is also possible to configure the endpoints and DMA channels ahead of the USB device enumeration. As long as the endpoint and DMA channels are configured using the USB 3.0 parameters, they will continue to work fine at hi-speed and full speed as well.

There is one caveat that applies in this case.

If there is an OUT endpoint that has a maximum packet size of N bytes, and the DMA channel has been created with a buffer size of M * N bytes (where M is an integer greater than 1); the data received on the endpoint will be accessible to the consumer only after the buffer is filled up, or a short packet is received.

e.g., If the maximum packet size is 512 bytes and the DMA buffer size is 1024 bytes; the buffer can be read by the consumer only after the buffer is filled up (receives two full packets) or if a short packet is received. If only one full packet is received on the endpoint, the data will sit in the buffer and will not be accessible to the consumer.

If this behavior is undesirable, the endpoint behavior can be modified using the CyU3PSetEpPacketSize() or CyU3PUsbSetEpPktMode() APIs.

The CyU3PSetEpPacketSize() API allows the maximum packet size definition for the endpoint to be modified such that all incoming packets are treated as short packets, resulting in immediate buffer commit. For example, if this API is used to set the maximum packet size as 1024 bytes in the above case, all 512 byte packets will be treated as short packets; and made available to the consumer immediately.

The CyU3PUsbSetEpPktMode() API configures the endpoint in packet mode, where it commits each incoming data packet into one DMA buffer (independent of whether it is full or short). The downside of this approach is that it can limit data transfer performance in some cases. It is recommended that the packet mode not be used for any applications that use large DMA buffers for better performance.

5.4.4 USB Low Power Mode Handling

The USB 3.0 specification includes provision for link level low power modes which are used to save system power consumption when the link is idle. It has been noted that different host controllers (in some cases even the host controller drivers) use the low power modes differently. In some cases, the host is very aggressive and tends to push the link into U1/U2 modes whenever it is waiting for data from the device. In other cases, the host waits a little longer before trying to initiate a transition into a low power mode.

The FX3 device handles U1/U2 transitions passively in most cases, because the device cannot initiate a U0->Ux transition without firmware intervention. Due to this constraint, the normal power mode handling in FX3 involves the device accepting or rejecting low power mode requests from the host; and then initiating a transition back to the U0 mode when instructed by firmware.

By default, the device is placed in a state where it accepts or rejects low power mode requests automatically based on whether it has any data ready to send out or not. This behavior can be overridden to a state where it systematically rejects all attempts by the host to push the link into a low power mode. This change is requested through the CyU3PUsbLPMDisable () API.

The use of this API is recommended if the power mode transitions on the USB link are limiting the performance of the system to unacceptable levels.

The FX3 device does not automatically initiate a transition back to U0 (from U1/U2) when it has data ready to go out. In the case of control transfers, the USB driver in the FX3 firmware is aware that data is ready; and initiates a transition back to U0 at the appropriate time.

In the case of other endpoints, it is possible that the data transfer be blocked because the link is stuck in a low power mode. It is recommended that any applications that do not use the CyU3PUsbLPMDisable() API, should call the

CyU3PUsbSetLinkPowerState(CyU3PUsbLPM_U0) API periodically to ensure that the link does not get stuck in a low power mode.

5.5 Support for different FX3 parts

The EZ-USB FX3 product family includes multiple parts as shown in the table below.

Part Number	USB 3.0 Support	Host / OTG Support	GPIF Support	SD/MMC Support	SRAM size
CYUSB3014	Yes	Yes	Up to 32 bit	No	512 KB
CYUSB3013	Yes	Yes	Up to 16 bit	No	512 KB
CYUSB3012	Yes	Yes	Up to 32 bit	No	256 KB
CYUSB3011	Yes	Yes	Up to 16 bit	No	256 KB
CYUSB3035 (FX3S)	Yes	Yes	Up to 16 bit	Yes	512 KB
CYUSB3065 (CX3)	Yes	No	Fixed function	No	512 KB

The FX3 SDK supports firmware development for each of these parts. The storage driver and API are packaged as a separate library (cyu3sport.a) that needs to be linked with FX3S applications alone. FX3 applications do not need to link with this library.

The SDK identifies the part that is in use and returns appropriate error codes to the init APIs to indicate that specific functions are not supported by the part. For example, calling any of the Storage APIs when using the FX3 part will result in a CY_U3P_ERROR_NOT_SUPPORTED return code. These checks ensure that the application does not lock up the device by trying to access non-existent functionality.

When using the FX3 parts which have limited (256 KB) RAM available, the application memory map needs to be modified to fit all code, data and buffers within the available memory. The SDK provides separate linker scripts that are targeted for these parts. Please refer to the boot_fw/src/cyfx3_256k.ld and firmware/common/fx3_256k.ld files when using the Boot Firmware and the full firmware libraries respectively.

The following changes are required to update the firmware examples to work on the CYUSB3011 or CYUSB3013 parts:

1. If using the full firmware library:
    - i. Copy the contents of the fx3_256k.ld file into the fx3.ld file. The original linker settings will still be available in the fx3_512k.ld file.

ii. Add the CYMEM_256K symbol to the list of pre-processor definitions for the project. Another option is to edit the cyfxtx.c file and add "#define CYMEM_256K" to it.

iii. Make sure that the application is not using more than 64 KB of buffering across all DMA channels.

2. If using the boot firmware library:

   i. Copy the contents of the cyfx3_256k.ld file into the cyfx3.ld file.

   ii. Add the CYMEM_256K symbol to the list or pre-processor definitions.

5.6 Porting Applications from SDK 1.2 to SDK 1.2.1

There are no major changes to the FX3 API set between the 1.2 and 1.2.1 releases. However, the following points need to be kept in mind when migrating an existing SDK 1.2 based application to the SDK 1.2.1 version.

1. A new API called CyU3PUsbEnableEPPrefetch() has been added in the 1.2.1 SDK version. This API updates the DMA interface settings for the USB block to pre-fetch data from the sockets more aggressively. These settings were previously left enabled in all cases.

   If the firmware application makes use of multiple USB IN endpoints on a regular basis, this API should be called immediately after the CyU3PUsbStart() API. This is not required if the firmware application has only one IN endpoint which is accessed regularly.

2. The multicast DMA channel handlers have been kept disabled from regular application builds to reduce memory footprint. If the firmware application makes use of any multicast DMA channels, these handlers need to be enabled by calling CyU3PDmaEnableMulticast(). This API needs to be called before creating any multicast DMA channels.

3. A new USB event called CY_U3P_USB_EVENT_LNK_RECOVERY has been added to provide notification of cases where the device enters USB 3.0 link recovery. As this callback is provided in interrupt context, performing time consuming operations such as calling CyU3PDebugPrint is not allowed.

   If the USB event callback in the firmware application performs such actions in the default case (where the event type does not match other specified values), it will need to be updated to avoid these actions for this event type.

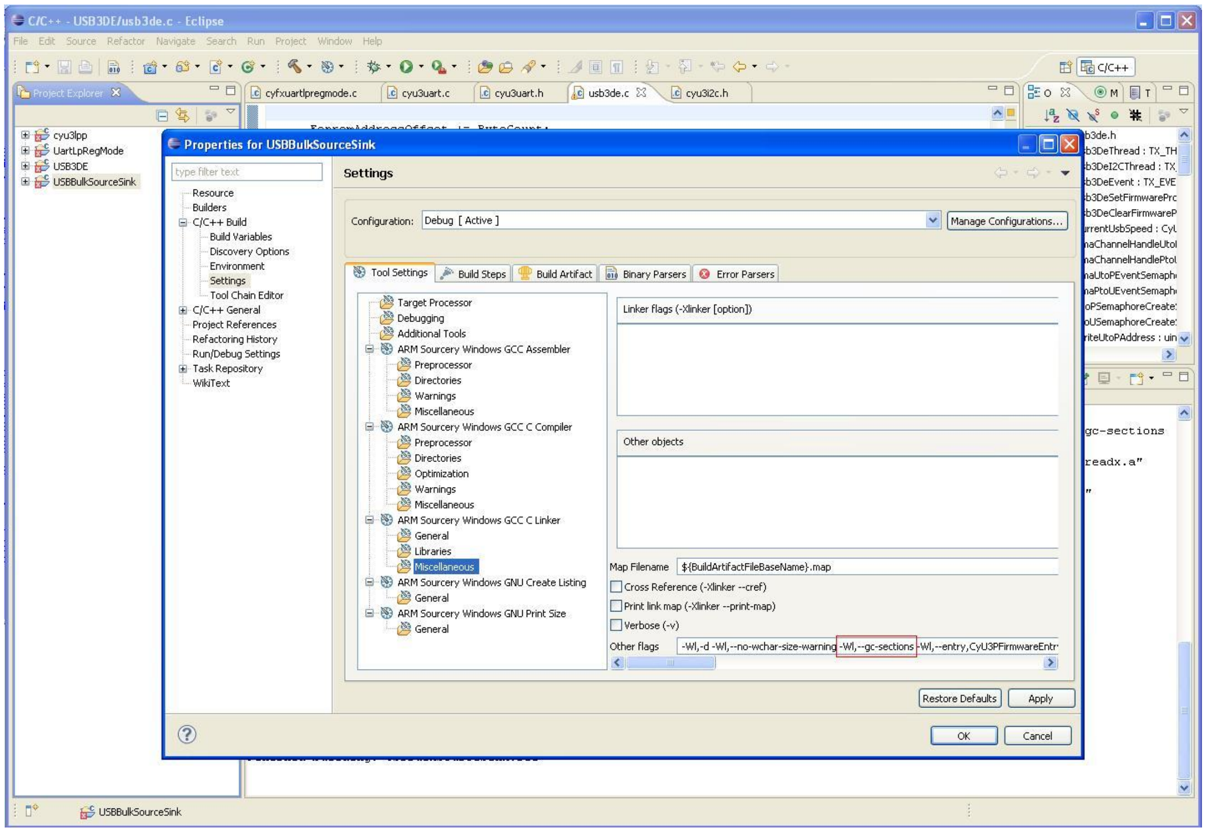
5.7 Porting Applications from SDK 1.2.1 to SDK 1.2.2

There are no major changes to the FX3 API set between the 1.2.1 and 1.2.2 releases. Please refer to the Release Notes document for details on the defect fixes and feature additions that have been included. The following points can be considered to make better use of the SDK changes in the application.

1. No code changes should be required in user applications while porting from SDK 1.2.1 to SDK 1.2.2. However, it is possible that the memory footprint of an application be increased when using the SDK 1.2.2. This happens because of some additional error checks that have been added to the ThreadX OS services used in the SDK.

   This increase in footprint can be offset through the use of the “-Wl,--gc-sections” linker command line flag. The FX3 libraries in SDK 1.2.2 have been updated to compile each function into a separate section; so that the use of the above option can give significant footprint reduction.

   All of the Eclipse projects in the SDK have been updated to include this linker flag. The following Eclipse UI screenshot shows the place where this linker flag can be updated.



Configuration: Debug [Active]

Linker flags (-Xlinker [option])

Other objects

Map Filename ${BuildArtifactFileBaseName}.map

☐ Cross Reference (-Xlinker --cref)

☐ Print link map (-Xlinker --print-map)

☐ Verbose (-v)

Other flags -Wl,-d -Wl,--no-wchar-size-warning -Wl,--gc-sections -Wl,--entry,CyU3PFirmwareEntr

Figure 5-1: Eclipse Linker Flag Settings

2. SDK 1.2.2 provides USB event notifications when it detects the connection or removal of a valid voltage on the USB Vbus pin. These events are provided even if the CyU3PConnectState() API has not been called.

   It is possible for a self-powered FX3 application to wait until the Vbus voltage is available before enabling the USB connection through the CyU3PConnectState() API.

5.8 Porting Applications from SDK 1.2.2 to SDK 1.2.3

1. The return type of the CyU3PdmaBufferFree function (defined in the cyfxtx.c file under the firmware applications) has been changed from void to int. This change was made to allow the implementation to flag memory free failures to the caller.

   The implementation of this function in all existing application code needs to be updated to change the return type. An updated implementation of this function is available in the cyfxtx.c file under all examples in SDK 1.2.3.

5.9 Porting Applications from SDK 1.2.3 to SDK 1.3

Some minor changes have been made to the core API of the FX3 SDK for adding support for the FX3S part. These changes will require the following changes to be made to all FX3 applications when porting from the 1.2.2 SDK to the 1.3 SDK.

1. Two new members called s0Mode and s1Mode have been added to the CyU3PIoMatrixConfig_t structure passed as parameter to the CyU3PDeviceConfigureIOMatrix API. Both of these should be set to the value CY_U3P_SPORT_INACTIVE for FX3 applications. The API is likely to return the CY_U3P_ERROR_BAD_ARGUMENT error if these fields are left un-initialized.
2. New API libraries (cyu3sport.a, cyu3mipicsi.a and cy_as0260.a) are available under the u3p_firmware/lib/fx3_debug and u3p_firmware/lib/fx3_release folders. These libraries needs to be added to the Miscellaneous linker settings when compiling FX3S or CX3 firmware applications. Please note that the these libraries need to be listed before the cyu3lpp.a firmware library; as they have dependencies on the GPIO and I2C functionalities provided by the cyu3lpp.a library.
3. The CyU3PusbControlVBusDetect() API has been modified to add a new parameter which specifies whether Vbatt should be used instead of Vbus for USB connection detection. Any calls to this API should be updated to pass in CyFalse for the useVbatt parameter. This parameter should be set as CyTrue only if the Vbatt supply is being derived from the Vbus input from the USB cable.
4. The runtime stacks in the boot firmware library (cyfx3boot.a) have been moved to the D-TCM region instead of being placed in the SYSMEM region. This change means that buffers placed in the runtime stack can no longer be used for DMA transfers. The CY_FX3_BOOT_ERROR_INVALID_DMA_ADDR error code will be returned by the CyFx3BootUsbDmaXferData, CyFx3BootSpiDmaXferData, CyFx3BootI2cDmaXferData and CyFx3BootUartDmaXferData APIs if a DTCM address is used for the DMA buffer.

Please ensure that buffers that are placed in the SYSMEM region are used for all DMA transfers. This can be achieved by using global buffers or by explicitly locating the buffer at a valid SYSMEM address.

5.10 Porting Applications from SDK 1.3 to SDK 1.3.1

No major changes to the FX3 API interfaces have been made between SDK 1.3 and 1.3.1. A minor change has been made to the FX3S control structures, which will require corresponding changes in FX3S application code.

1. The lvGpioState field has been added to the CyU3PSibIntfParams_t structure passed to the CyU3PSibSetIntfParams API. This field should be initialized with the polarity of the GPIO used for SD voltage control.

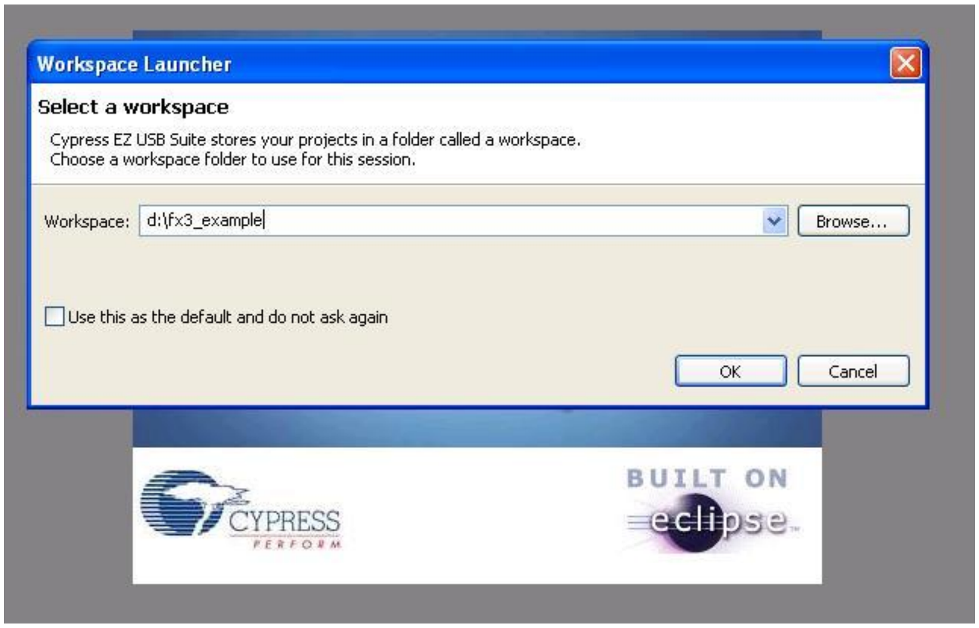
6 FX3 Application Example

This chapter outlines the steps involved in creating a FX3 firmware application, using the GpifToUsb project from the SDK as a reference.

6.1 Creating the Project

The easiest way to create a new FX3 firmware project under the Eclipse IDE, is to import an existing project; and make changes to it.

1. Start the EZ USB Suite IDE from the Cypress group in the Start Menu. When prompted for workspace selection, provide a path to an existing workspace, or type in the path where the new workspace is to be created.



Select a workspace

Cypress EZ USB Suite stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: d:\fx3_example

☐ Use this as the default and do not ask again

Figure 6-1: Workspace Selection Dialog

2. Choose the Import option under the File menu to bring up the Import options window. Choose “Existing Projects into Workspace” from the “General” group.

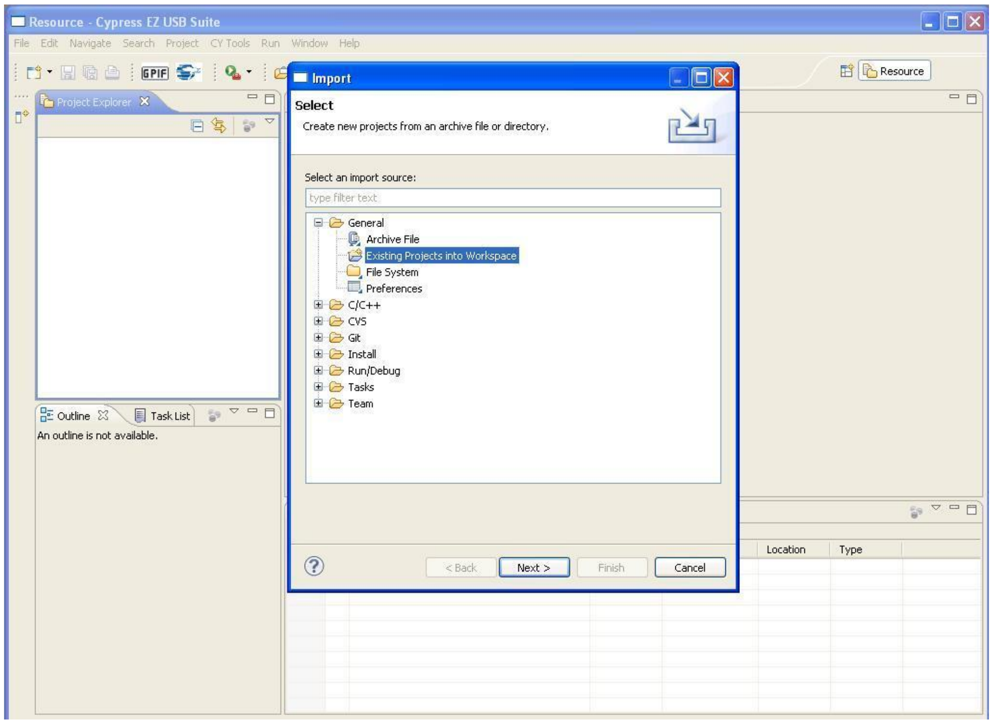


Figure 6-2: Project Import Dialog

3. In the “Browse for Folder” window that pops up, navigate into the firmware/basic_examples/cyfxgpiftousb folder in the FX3 SDK installation.

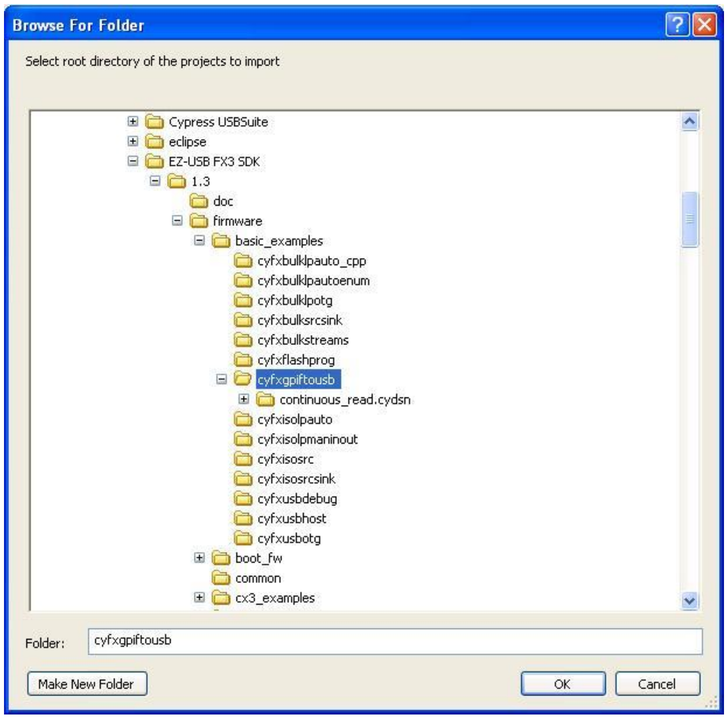
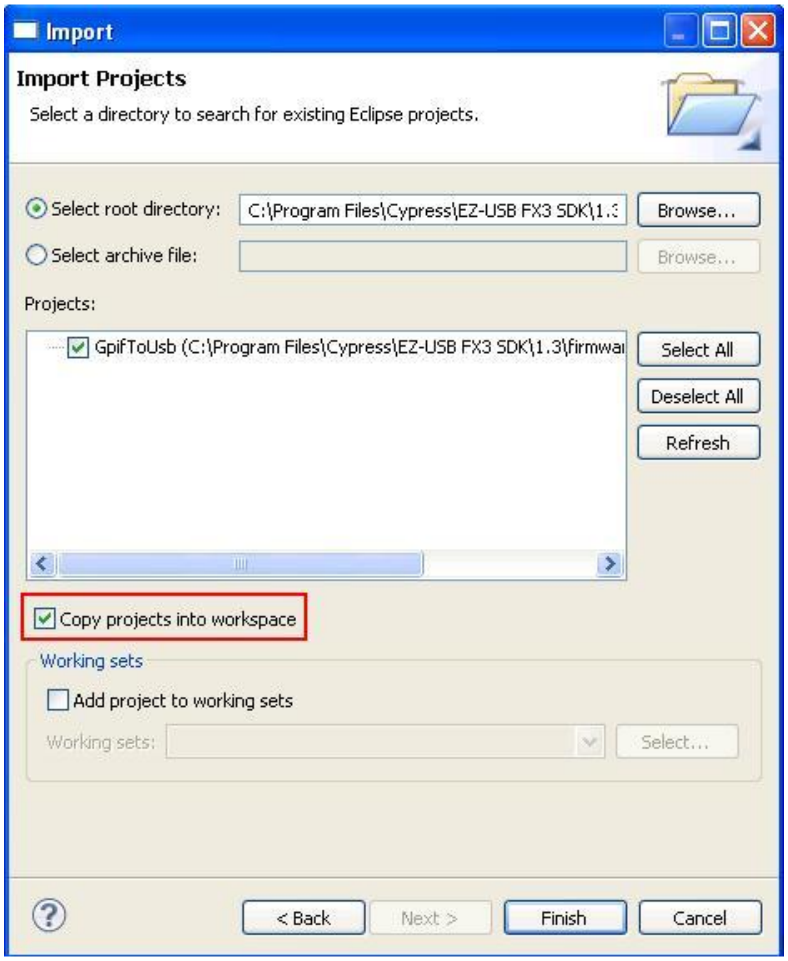


Figure 6-3: Folder Browse Dialog

4. Select the “Copy projects into workspace” option in the Import window, so that we get a new copy of the source files that can be modified.


Import

Import Projects

Select a directory to search for existing Eclipse projects.

(●) Select root directory: C:\Program Files\Cypress\EZ-USB FX3 SDK\1.3

( ) Select archive file:

Projects:

[✓] GpifToUsb (C:\Program Files\Cypress\EZ-USB FX3 SDK\1.3\firmwa

[✓] Copy projects into workspace

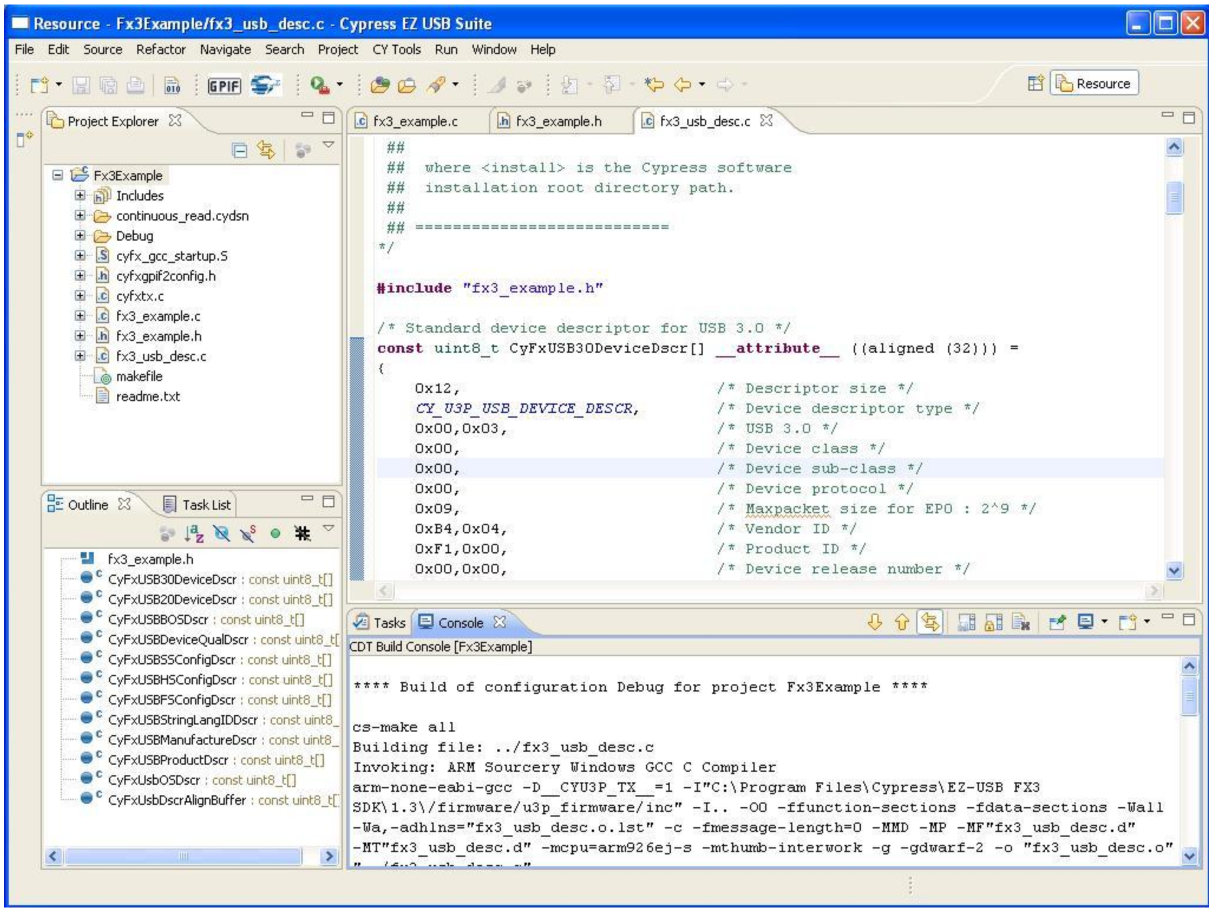
Working sets

[ ] Add project to working sets

Working sets:

Figure 6-4: Import Projects Dialog

5. If desired, right-click the project name and use the “Rename” option to rename the project. The respective source files in the project can also be renamed as desired. If the header files are renamed, references to the header file in the source files will need to be updated.


Resource - Fx3Example/fx3_usb_desc.c - Cypress EZ USB Suite
File Edit Source Refactor Navigate Search Project CY Tools Run Window Help
Resource
Project Explorer
Fx3Example
Includes
continuous_read.cydsn
Debug
cyfx_gcc_startup.S
cyfxgpif2config.h
cyfxtx.c
fx3_example.c
fx3_example.h
fx3_usb_desc.c
makefile
readme.txt
Outline
Task List
fx3_example.h
CyFxUSB30DeviceDscr : const uint8_t[]
CyFxUSB20DeviceDscr : const uint8_t[]
CyFxUSBBOSDscr : const uint8_t[]
CyFxUSBDeviceQualDscr : const uint8_t[
CyFxUSBSSConfigDscr : const uint8_t[]
CyFxUSBHSConfigDscr : const uint8_t[]
CyFxUSBFSConfigDscr : const uint8_t[]
CyFxUSBStringLangIDDscr : const uint8_
CyFxUSBManufactureDscr : const uint8_
CyFxUSBProductDscr : const uint8_t[]
CyFxUsbOSDscr : const uint8_t[]
CyFxUsbDscrAlignBuffer : const uint8_t[
fx3_example.c
fx3_example.h
fx3_usb_desc.c
Tasks
Console
CDT Build Console [Fx3Example]

```
 ##
 ##  where <install> is the Cypress software
 ##  installation root directory path.
 ##
 ## ===========================
*/

#include "fx3_example.h"

/* Standard device descriptor for USB 3.0 */
const uint8_t CyFxUSB30DeviceDscr[] __attribute__ ((aligned (32))) =
{
    0x12,                           /* Descriptor size */
    CY_U3P_USB_DEVICE_DESCR,        /* Device descriptor type */
    0x00,0x03,                      /* USB 3.0 */
    0x00,                           /* Device class */
    0x00,                           /* Device sub-class */
    0x00,                           /* Device protocol */
    0x09,                           /* Maxpacket size for EP0 : 2^9 */
    0xB4,0x04,                      /* Vendor ID */
    0xF1,0x00,                      /* Product ID */
    0x00,0x00,                      /* Device release number */
```

```
**** Build of configuration Debug for project Fx3Example ****

cs-make all
Building file: ../fx3_usb_desc.c
Invoking: ARM Sourcery Windows GCC C Compiler
arm-none-eabi-gcc -D__CYU3P_TX__=1 -I"C:\Program Files\Cypress\EZ-USB FX3
SDK\1.3\/firmware/u3p_firmware/inc" -I.. -O0 -ffunction-sections -fdata-sections -Wall
-Wa,-adhlns="fx3_usb_desc.o.lst" -c -fmessage-length=0 -MMD -MP -MF"fx3_usb_desc.d"
-MT"fx3_usb_desc.d" -mcpu=arm926ej-s -mthumb-interwork -g -gdwarf-2 -o "fx3_usb_desc.o"
```

Figure 6-5: EZ USB Suite Eclipse Based IDE

6. Add additional source files as required. The build settings for the project will already be functional. Right-Click on the project name and use the "Properties" menu item to view and modify the build settings.
7. Build the project to verify that the project import and renaming is successful.

6.2 Application Description

6.2.1 Application Outline

The GpifToUsb example demonstrates moving data from the GPIF-II port of the FX3 device to the USB port through a BULK-IN endpoint.

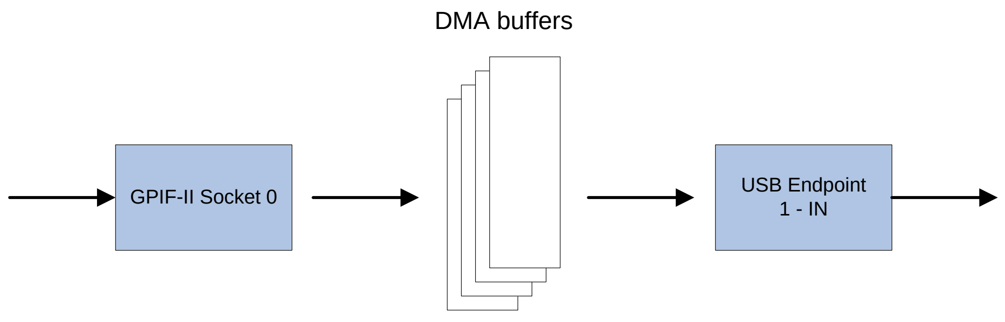


Figure 6-6: GPIF-to-USB Data Flow

A dummy GPIF-II configuration which continuously latches data from the GPIF-II data pins is used in this example. This example does not require any external device to be connected on the GPIF-II port. The data pins can be tied to various logic levels, and the respective changes can be viewed in the data received on the USB side.

The firmware only sets up the required data path, which then runs by itself without firmware intervention.

6.2.2 Project Files

The following source files are part of this firmware project.

1. cyfx_gcc_startup.S: Start-up code for the ARM-9 core on the FX3 device. This file can be taken as such from the firmware/common folder or any of the application folders in the FX3 SDK.
2. cyfxtx.c: ThreadX RTOS wrappers and utility functions used by the example. This can also be taken as such from the firmware/common folder in the SDK installation.
3. cyfxgpif2config.h: This is the GPIF-II designer generated configuration file which implements the simple state machine used in this example. The GPIF-II designer project itself is present under the continuous_read.cydsn folder.
4. cyfxgpiftousb.h: Configuration settings and constants used by the application.
5. cyfxbulkdscr.c: USB descriptors used by the application.
6. cyfxgpiftousb.c: Main source file associated with this firmware application.

6.2.3 Code Flow

As with all FX3 examples in the SDK, the CyU3PFirmwareEntry function is chosen as the entry point for this application. This function is part of the FX3 firmware library, and sets up the operating modes and initial runtime stacks for the ARM9 CPU. Once these functions are performed, the entry jumps back to the CyU3PToolChainInit function that is defined in the cyfx_gcc_startup.S file.

The CyU3PToolChainInit function zeroes out the whole of the BSS segment (ranging from _bss_start to _bss_end), and then jumps to the main() function that is defined in cyfxgpiftousb.c.

The code flow from this point on is described below. Please refer to the code in cyfxgpiftousb.c file to understand how these operations are performed in code.

1. The main function uses the CyU3PDeviceInit() to set up the clocks for the FX3 system.
2. The CyU3PDeviceCacheControl function is used to enable the required caches. This example enables only the Instruction cache. If the data cache is enabled, the cache handling in the DMA APIs also needs to be enabled.
3. Use the CyU3PDeviceConfigureIOMatrix function to configure the IO functionalities used in this application. The GPIF-II bus is configured as 32 bit so that the maximum data throughput is achieved on the GPIF-II port. UART is enabled for debugging, and no other serial interfaces are enabled.
4. Call the CyU3PKernelEntry() function to start off the ThreadX RTOS scheduler. This is a non-returning function call, and no further code is placed in the main() function.[1]
5. Once the RTOS has been started up, the scheduler calls the CyFxApplicationDefine(). The user can define all threads and other OS objects required by the application in this function. In this case, the CyFxApplicationDefine() function only creates a thread which implements the core application logic. The priority for this thread is set to 8 (the default priority for all example application threads) and the thread stack size is defined as 4 KB. Time slicing is not enabled, and the OS is instructed to start the thread as soon as possible using the CYU3P_AUTO_START parameter.
6. The remaining part of the code flow is application specific. When the device is reset, all of the external interface blocks (USB, GPIF-II, UART, I2C, SPI, I2S and Storage) are left disabled and powered off. Therefore, the corresponding driver threads will have no work to do and will remain idle. Device operation will only start when each of the required blocks is enabled by user code. This is done in the entry functions of the various threads created in the CyFxApplicationDefine() function.
7. The first operation performed by the CyFxAppThread_Entry() function is to enable the UART on the device for logging. This is done in the CyFxApplnDebugInit() function. The steps involved in doing this are:
    a. Calling CyU3PUartInit() to initialize the UART. This function will only succeed if the useUart flag has been set to CyTrue in the device IO configuration.

[1] Steps 1 through 4 are expected to be similar in all FX3 applications. Only the parameters passed to the functions will change based on the system design.

b. Configuring the UART interface parameters as required using the CyU3PUartSetConfig API. In this case, the UART is setup to transmit data from the consumer DMA channel with the interface set to 115200 8-N-1.

c. Calling CyU3PUartTxSetBlockXfer to setup an infinite transfer size, so that the UART can keep transmitting debug data as long as required.

d. Calling CyU3PDebugInit to enable the debug logger and to direct all log output to the UART consumer socket. The CyU3PDebugInit API creates a MANUAL_OUT DMA channel with the socket specified as the consumer. This channel is managed by the debug driver in the SDK, and no user intervention is required.

e. Calling CyU3PDebugPreamble to disable the output of a message header along with each output log. The message header is designed for automated parsing of the log output and is left turned ON by default. However, leaving this ON can make the log messages harder to read on a debug console.

8. The next step in the CyFxAppThread_Entry() function is to initialize the PIB (GPIF) and USB blocks on the FX3 device.

   a. The CyU3PPibInit API is called to initialize the PIB (Processor Interface Block) on the FX3 device. The PIB is a hardware block that implements the GPIF-II controller as well as a Pseudo MMC Slave (PMMC) implementation. The PIB can be configured in either of these modes, with the GPIF-II mode being the default. In this application, the clock for the GPIF-II interface is provided by FX3, and this clock frequency is set to the maximum possible value (1/4 of the System Clock, which is about 101.8 MHz).

   b. The CyU3PUsbStart API is called to initialize the USB block in device (peripheral) mode. This API internally creates a pair of DMA channels which are used to do the control endpoint data transfers.

   c. Callback functions for the notification of general USB events, USB control requests and LPM related events are registered next through the CyU3PUsbRegisterEventCallback, CyU3PUsbRegisterSetupCallback and CyU3PUsbRegisterLPMRequestCallback APIs. The fast enumeration mode is selected so that most of the USB control requests are handled by the USB driver in the SDK.

   d. The CyU3PUsbSetDesc API is used repeatedly to register all relevant USB descriptors with the USB driver. At a minimum, the USB 2.0 and 3.0 device descriptors; the USB-FS, HS and SS configuration descriptors; and the BOS descriptor should be registered using this API call.

e. The CyU3PConnectState API is used to enable USB device connection to a host. This API only enables the device to connect to the host, and does not necessarily result in turning the USB PHY blocks on. The USB driver will start monitoring the Vbus signal once this API is called, and will enable the PHY when Vbus becomes valid. The SuperSpeed enable parameter is set to true, so that FX3 can connect as a SuperSpeed, Hi-Speed or Full speed device based on the host's capabilities.

9. After the above initialization steps, the CyFxAppThread_Entry function waits for the USB device configuration to be completed. The actual GPIF-II and USB endpoint configuration is not performed during start-up of the application. This is only performed when the USB host sends down a SET_CONFIGURATION request to the FX3 device. This is done through the CyFxApplnStart function, once the CY_U3P_USB_EVENT_SETCONF event has been received in the USB event callback.

10. The CyFxApplnStart function completes the GPIF-II configuration and the USB endpoint configuration; and then sets up the data path for steering data from the GPIF-II interface to the USB endpoint.

    a. The CyU3PUsbGetSpeed() API is used to identify the USB connection speed. The packet size of the USB endpoint is determined based on this connection speed.

    b. The CyU3PSetEpConfig API is used to configure EP 1-IN as a BULK-IN endpoint with the desired maximum packet size. If FX3 is connected as a SuperSpeed device, the endpoint is configured to support CY_FX_EP_BURST_LENGTH packets in a single burst.

    c. Any stale data in the endpoint buffers is flushed using the CyU3PUsbFlushEp API.

    d. The CyU3PDmaChannelCreate API is used to create an Automatic (no firmware intervention) DMA data path from the GPIF-II socket to the USB endpoint. The DMA channel is configured to use CY_FX_DMA_BUF_COUNT buffers of CY_FX_DMA_BUF_SIZE bytes each. Even though CY_FX_DMA_BUF_SIZE is greater than the endpoint's maximum packet size, the data in the buffer will be automatically broken into appropriately sized packets by the FX3 hardware.

    e. The CyU3PDmaChannelSetXfer API is used to enable the data path for data transfers. A data count of zero is specified, so that the data transfer can continue without any limits.

    f. The CyU3PGpifLoad API is used to load the GPIF-II state machine configuration onto the device registers. The configuration data is taken from the cyfxgpif2config.h header file.

   g. The CyU3PGpifSMStart API is used to start executing the GPIF-II state machine.
   h. Once these steps are complete, FX3 starts latching the state of the GPIF-II data bus and storing it in the DMA buffers. This operation is performed continuously with automatic flow control kicking in, if the DMA buffers are not available.

11. The above steps complete all of the initialization required for the GpifToUsb data transfer application. The application continuously keeps transferring data to the USB-IN endpoint whenever a free DMA buffer is available for data transfer. The Streamer application can be run on the USB host side to read this data, and to calculate the data transfer performance.
12. The application identifies a USB disconnect event through the USB event callback. It then de-initializes the DMA channel, the GPIF state machine and the USB endpoints through the CyFxApplnStop function.

7 CX3 MIPI CSI-2 Interface

This section provides an introduction to MIPI CSI-2 interface on the CX3 device. It also provides a usage guide for the preliminary release of the CX3 Receiver Configuration tool available as part of the Cypress EZ USB Suite Eclipse based IDE. More details on the CX3 part are available on the Cypress website at http://www.cypress.com/cx3/

7.1 Introduction to the CX3 device

The CX3 is a variant of the FX3 device that features an integrated MIPI CSI-2 receiver mated to the GPIF as shown in the CX3 Logic block diagram. This device provides the ability to add USB 3.0 connectivity to Image Sensors implementing the MIPI CSI-2 interface.

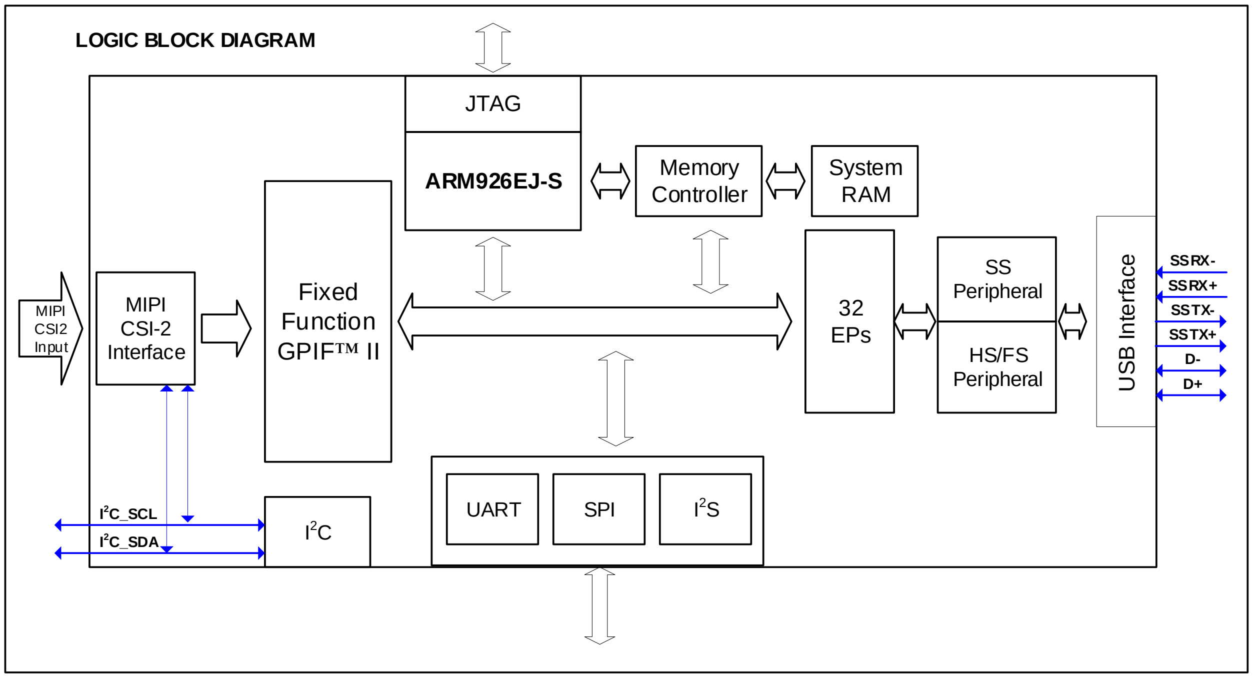


Figure 7-1: CX3 Device Block Diagram

The MIPI CSI-2 interface on the CX3 supports 1 to 4 CSI-2 data lanes and RAW8/10/12/14, YUV422, and RGB888/666/565 image formats. It reads image data from the sensor, de-packetizes it and sends it in to the parallel interface of the fixed-function GPIF II interface on the CX3.

Support for the MIPI CSI-2 interface is provided through a new firmware library added to the SDK (cyu3mipicsi.a). The APIs provided by the library and the data structures and enumerations used by this interface are provided through the cyu3mipicsi.h header file.

7.2 CX3 MIPI CSI-2 interface

7.2.1 Data Lanes

The MIPI CSI-2 interface on the CX3 supports 1 to 4 CSI-2 data lanes each capable of transfers of up to 1 Gbps. The number of data lanes to be selected for a CX3 application will depend upon the number of data lanes provided by the Image Sensor and the total required data transfer rate.

7.2.2 MIPI CSI-2 stream formats supported by the CX3

The MIPI CSI-2 interface on the CX3 supports the following stream formats and output modes:

Format and Mode	CX3 Enumeration Type	Description	CSI-2 Data Type	Output Stream
RAW8	CY_U3P_CSI_DF_RAW8	Bayer format 8-bits per pixel data stream.	0x2A	8-Bit Output: RAW[7:0]
RAW10	CY_U3P_CSI_DF_RAW10	Bayer format 10-bits per pixel data stream.	0x2B	16-Bit Output: 6’b0, RAW[9:0]
RAW12	CY_U3P_CSI_DF_RAW12	Bayer format 12-bits per pixel data stream.	0x2C	16-Bit Output: 4’b0,RAW[11:0]
RAW14	CY_U3P_CSI_DF_RAW14	Bayer format 14-bits per pixel data stream.	0x2D	16-Bit Output: 2'b0,RAW[13:0]
RGB888	CY_U3P_CSI_DF_RGB888	RGB 888 format 24- bits per pixel data stream.	0x24	24-Bit Output: R[7:0],G[7:0],B[7:0]
RGB666 Mode 0	CY_U3P_CSI_DF_RGB666_0	RGB 666 format 24- bits per pixel data stream.	0x23	24-Bit Output: 2’b0,R[5:0],2’b0,G[5:0], 2’b0,B[5:0]
RGB666 Mode 1	CY_U3P_CSI_DF_RGB666_1	RGB 666 format 24- bits per pixel data stream.	0x23	24 Bit Output: 6’b0,R[5:0],G[5:0], B[5:0]
RGB565 Mode 0	CY_U3P_CSI_DF_RGB565_0	RGB 565 format 24- bits per pixel data stream.	0x22	24 Bit Output: 2’b0,R[4:0],3’b0,G[5:0], 2’b0,B[4:0],1’b0

Format and Mode	CX3 Enumeration Type	Description	CSI-2 Data Type	Output Stream
RGB565 Mode 1	CY_U3P_CSI_DF_RGB565_1	RGB 565 format 24- bits per pixel data stream.	0x22	24 Bit Output: 3’b0,R[4:0],2’b0,G[5:0], 3’b0,B[4:0]
RGB565 Mode 2	CY_U3P_CSI_DF_RGB565_2	RGB 565 format 16-bits per pixel data stream.	0x22	16 Bit Output: R[4:0],G[5:0], B[4:0]
YUV422 8-Bit Mode 0	CY_U3P_CSI_DF_YUV422_8_0	YUV422 format 8-bits per pixel data stream.	0x1E	8 Bit Output: P[7:0] Data Order: U1,Y1,V1,Y2,U3,Y3,....
YUV422 8-Bit Mode 1	CY_U3P_CSI_DF_YUV422_8_1	YUV422 format 8-bits per pixel data stream.	0x1E	16 Bit Output: P[15:0] Data Order: {U1,Y1},{V1,Y2},{U3,Y3},{V3,Y4}...
YUV422 8-Bit Mode 2	CY_U3P_CSI_DF_YUV422_8_2	YUV422 format 8-bits per pixel data stream.	0x1E	16 Bit Output: P[15:0] Data Order: {Y1,U1},{Y2,V1},{Y3,U3},{Y4,V3}....
YUV422 10-Bit	CY_U3P_CSI_DF_YUV422_10	YUV422 format 10-bits per pixel data stream.	0x1F	16 Bit Output: 6'b0,P[9:0] Data Order: U1,Y1,V1,Y2,U3,Y3,V3,Y4.

If the GPIF interface used is larger than the width of the output stream (e.g. if a 24-bit GPIF II interface is used for the CY_U3P_CSI_DF_YUV422_8_1 type) the upper bits on the GPIF II interface will be padded with 0s.

7.2.3 MIPI CSI-2 interface clocks

The primary clocks on the MIPI CSI-2 interface of the CX3 are shown in Figure 7-2 below.

The interface takes a reference clock as its input and generates the required clocking using a PLL and multiple clock dividers on the PLL generated clock. Clock configuration parameters are a part of the *CyU3PMipicsiCfg_t* structure which passed to the *CyU3PMipicsiSetIntfParams()* API to configure the CSI-2 interface.

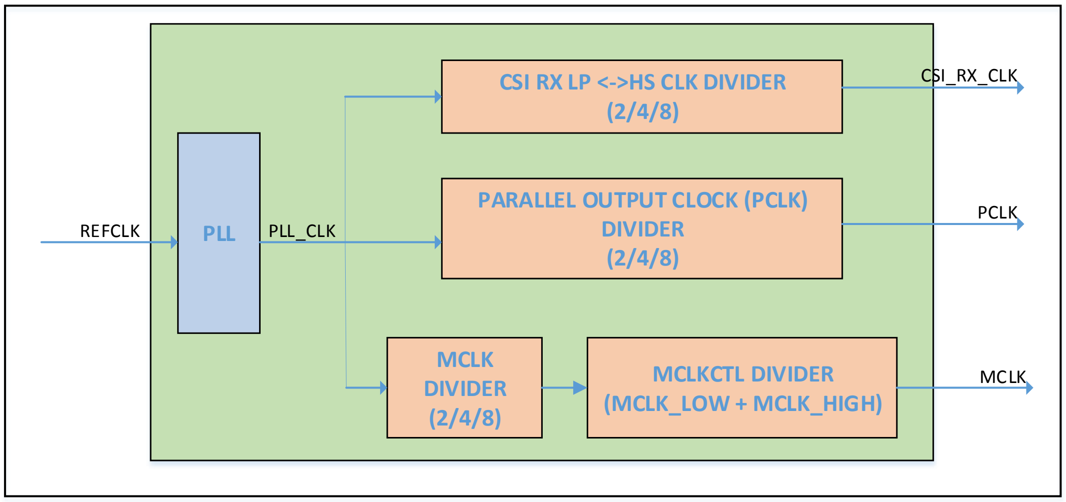

CSI RX LP <->HS CLK DIVIDER (2/4/8)
CSI_RX_CLK
PARALLEL OUTPUT CLOCK (PCLK) DIVIDER (2/4/8)
PCLK
REFCLK
PLL
PLL_CLK
MCLK DIVIDER (2/4/8)
MCLKCTL DIVIDER (MCLK_LOW + MCLK_HIGH)
MCLK

Figure 7-2: CX3 MIPI CSI-2 Interface Clocks

A brief description of each of the clocks is provided below:

1. Reference Clock (REFCLK)

This is the input reference clock provide to the MIPI-CSI interface. This input clock should be between 6 and 40 MHz.

2. PLL Clock (PLL_CLK)

The PLL_CLK is the primary clock on the MIPI CSI-2 interface. The minimum legal value for PLL clock is 62.5 MHz and maximum legal value for the PLL is 1 GHz.

All other internal/output clocks are derived from this clock.

The PLL clock frequency is generated from the Input Reference Clock using the following equation:

PLL_CLK = **REFCLK** * [(**pllFbd** + 1)/(**pllPrd** + 1)] /(2^**pllFRS**)

where

pllPrd is the input divider whose range is between 0 and 0x0F.

pllFbd is the feedback divider whose range is between 0 and 0x1FF.

pllFRS is the frequency range selection parameter which takes the following values:

0 if PLL Clock is between 500MHz- 1GHz.

1 if PLL Clock is between 250MHz- 500MHz.

2 if PLL Clock is between 125MHz- 250MHz.

3 if PLL Clock is between 62.5MHz- 125MHz.

E.g.:

If RefClk is 19.2 MHz, pllFbd is 69, pllPrd is 1, and PLL Clock range is in the 500MHz-1GHz range (i.e. pllFrs = 0),

PLL_Clock (MHz) = 19.2 * [(69+1)/(1+1)]/(2^0)

= 19.2 * [70/2]/1

= 672 MHz

For the same values, changing PLL frequency range to 125-250MHz (pllFrs = 2) will change the PLL Clock value to

PLL_Clock (MHz) = 19.2 * [(69+1)/(1+1)]/(2^2)

= 19.2 * [70/2]/4

= 168 MHz

3. CSI RX LP←→HS Transition Clock

This clock is used for detecting the CSI link LP<->HS transition. It is generated by dividing the PLL_Clock by a value of 2, 4 or 8.

The minimum value for this clock is 10Mhz and maximum value for this clock is 125MHz.

4. Output Parallel Clock (PCLK)

This clock is the PCLK output which drives the fixed-function GPIF interface on the CX3. It is generated by dividing the PLL_Clock by a value of 2, 4 or 8.

The maximum value for this clock is 100MHz.

5. MCLK

The MCLK is an optional clock output which can be used as the input reference clock for the image sensor. It is sourced from the PLL_Clk by first dividing down using the mClkRefDiv (2/4/8) and then dividing using the MCLKCTL divider. The MCLKCTL divider specifies the high time and low time counted by the divided down PLL_CLK.

The upper 8 bits define the high time count (1-255) and the lower 8 bits define the low time count (1-255).

MCLK is computed using the following equation:

MCLK = (**PLL_CLK**/**mClkRefDiv**)/[(HighByte(**mClkCtl**)+1)+(LowByte(**mclkCtl**)+1)]

Eg:

If PLL_CLOCK is 672 MHz as given above, to generate MCLK of 24 MHz, we set mClkRefDiv to 4, mClkCtl to 0x0203 to get

MCLK = (672/4) / ((2+1) + (3+1))

= 168/ 7 = 24 MHz

The MCLK is only output when both HighByte(mClkCtl) and LowByte(mClkCtl) are non-zero.

7.3 Configuring the CX3 MIPI CSI-2 Interface

The MIPI CSI-2 interface is configured using the *CyU3PMipicsiSetIntfParams* API. This API takes an object of type *CyU3PMipicsiCfg_t* which contains various parameters (Output data format, Clock dividers and configuration parameters, Data lanes to use, horizontal resolution etc.).

A preliminary release of a graphical tool (CX3 Receiver Configuration Tool) that can be used to generate the *CyU3PMipicsiCfg_t* structure object is provided as part of the EZ-USB Suite Eclipse based IDE provided by Cypress.

This section provides the steps required to create the configuration data from the graphical tool.

1. **Create a new CX3 Configuration File.**

   From the File menu of the Eclipse IDE, select **New -> Other**.

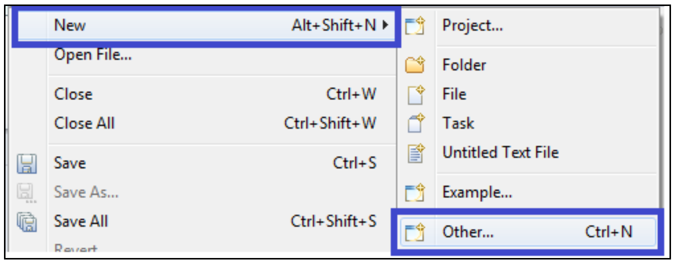


Figure 7-3: CX3 Configuration File Creation Menu

From the New file dialog box, select the **CX3 Configuration** option under the Cypress Folder and click on **Next.**

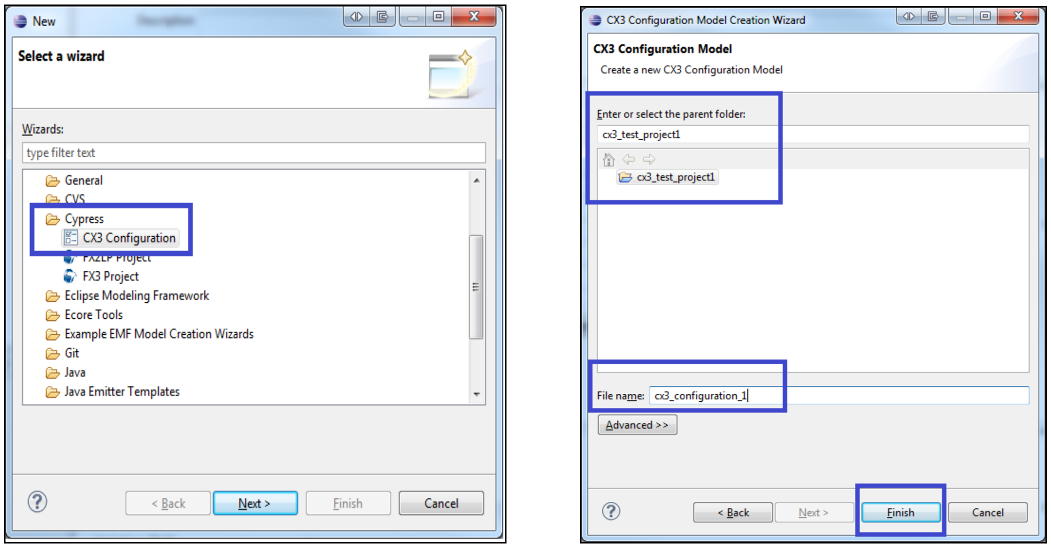


Figure 7-4: CX3 Configuration File Creation

In the *CX3 Configuration Model Creation Wizard* dialog box which comes up, select the parent folder for the project, enter a name for the CX3 configuration file, and click on Finish to generate the CX3 configuration file.

2. **CX3 Receiver Configuration Tool**

   A new CX3 Configuration file (with a .cycx extension) will get created under the selected project and the file will be opened up in the CX3 Receiver Configuration Tool as shown below.

   The file can also be opened by clicking on the cycx file in the project explorer.

   Any unsaved changes on the configuration file will be indicated by the ***** mark on the window title. The CX3 configuration file (with .cycx extension) can be saved by Clicking on the Save or Save All buttons.

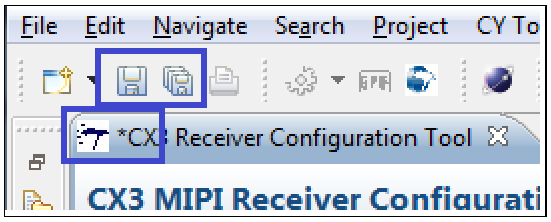
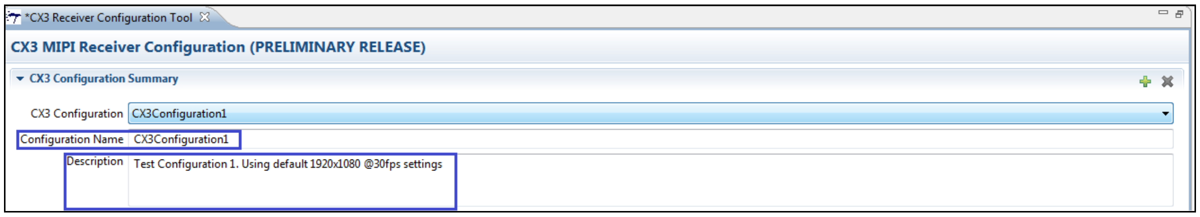

File
Edit
Navigate
Search
Project
CY To
*CX3 Receiver Configuration Tool
CX3 MIPI Receiver Configurati

Figure 7-5: Saving the CX3 Configuration File

3. **Creating a configuration**

   The first step in creating the configuration is to set the Configuration name and Description. The configuration name is used as the name of the structure object that is generated and should therefore follow the naming conventions for C/C++ variables. The description is provided as a comment for the structure object and therefore should not contain '/*' or '*/'.

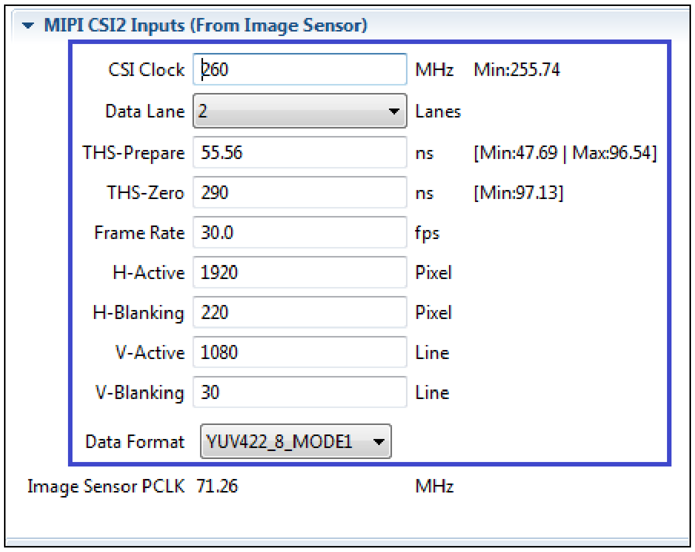

*CX3 Receiver Configuration Tool
CX3 MIPI Receiver Configuration (PRELIMINARY RELEASE)
CX3 Configuration Summary

CX3 Configuration: CX3Configuration1
Configuration Name: CX3Configuration1
Description: Test Configuration 1. Using default 1920x1080 @30fps settings

Figure 7-6: Configuration Name Selection

4. **Configure the MIPI CSI-2 Inputs**

   The MIPI CSI-2 inputs section should be configured to match the Image Sensor's output parameters.



MIPI CSI2 Inputs (From Image Sensor)

CSI Clock	260	MHz	Min:255.74
Data Lane	2	Lanes	
THS-Prepare	55.56	ns	[Min:47.69 | Max:96.54]
THS-Zero	290	ns	[Min:97.13]
Frame Rate	30.0	fps	
H-Active	1920	Pixel	
H-Blanking	220	Pixel	
V-Active	1080	Line	
V-Blanking	30	Line	
Data Format	YUV422_8_MODE1		
Image Sensor PCLK	71.26	MHz	

Figure 7-7: CSI-2 Input Parameters

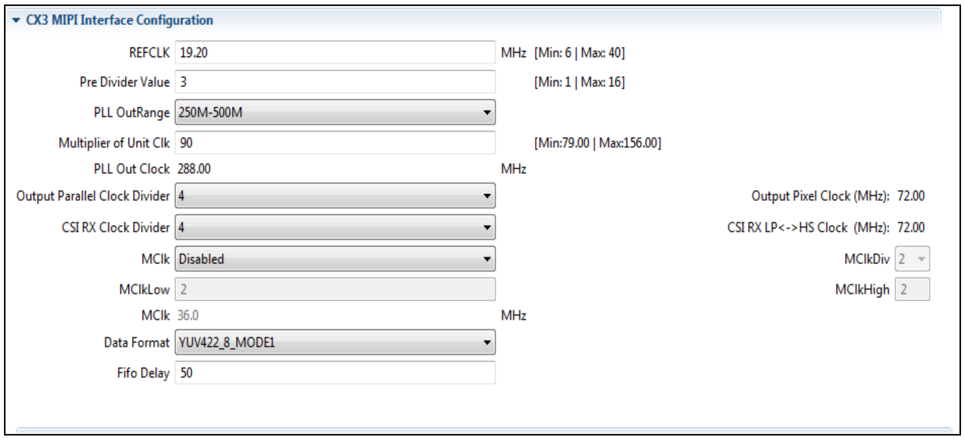
The Image Sensor PCLK value is computed based upon the H-Active, H-Blanking, V-Active, V-Blanking, Data format and Frame Rate fields.

The CSI Clock frequency and the Data Lane entries along with the frame properties are used to compute the total input data throughput on the CSI-2 interface which needs to be output on the parallel interface to the GPIFII.

In general, the Image Sensor PCLK value should not exceed 100 MHz.

5. Computing the CX3 MIPI CSI-2 Interface Configuration

The first field for the CX3 MIPI CSI-2 interface configuration is for the REFCLK frequency being provided to the block. This value along with the Pre Divider Value, PLL Out Range and Multiplier of Unit Clock are used to generate the pllPRD, pllFbd and pllFrs parameters which provide the PLL_CLK frequency using the equation listed in Section 7.2.3 above.

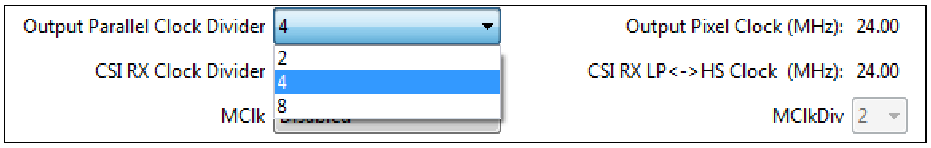


CX3 MIPI Interface Configuration

REFCLK	19.20	MHz	[Min: 6 | Max: 40]		
Pre Divider Value	3		[Min: 1 | Max: 16]		
PLL OutRange	250M-500M				
Multiplier of Unit Clk	90		[Min:79.00 | Max:156.00]		
PLL Out Clock	288.00	MHz			
Output Parallel Clock Divider	4			Output Pixel Clock (MHz):	72.00
CSI RX Clock Divider	4			CSI RX LP<->HS Clock (MHz):	72.00
MClk	Disabled			MClkDiv	2
MClkLow	2			MClkHigh	2
MClk	36.0	MHz			
Data Format	YUV422_8_MODE1				
Fifo Delay	50				

Figure 7-8: CSI-2 Interface Configuration Parameters

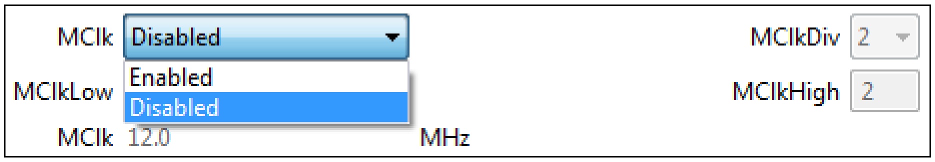
Once the PLL clock frequency has been generated, the Output Pixel Clock (PCLK) and CSI RX clock frequency can be obtained by selecting the appropriate divider value as shown in the image below.



Output Parallel Clock Divider: 4
CSI RX Clock Divider
MClk
Output Pixel Clock (MHz): 24.00
CSI RX LP<->HS Clock (MHz): 24.00
MClkDiv: 2

Figure 7-9: Clock Dividers

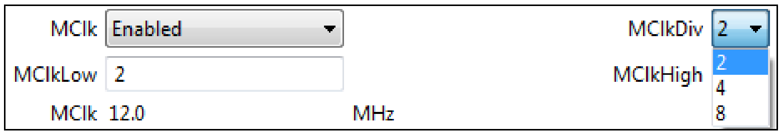
To configure the MCLK, select Enabled from the MClk dropdown box to enable configuration of the MCLK related dividers.



MClk: Disabled
MClkLow
MClk 12.0 MHz
MClkDiv: 2
MClkHigh: 2

Figure 7-10: MCLK Settings

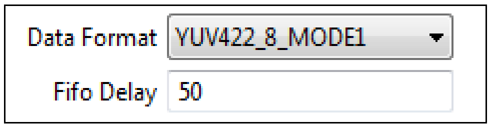
Select a MClkDiv value and then change MClkLow and MClkHigh values to achieve the desired MCLK frequency



MClk: Enabled
MClkLow: 2
MClk 12.0 MHz
MClkDiv: 2
MClkHigh

Figure 7-11: MCLK Dividers

The Data Format field is used to determine the pixel width and format as described in Section 7.2.2 above; and the Fifo Delay field is used to set a threshold of pixels to be met before data output on the parallel interface is started.

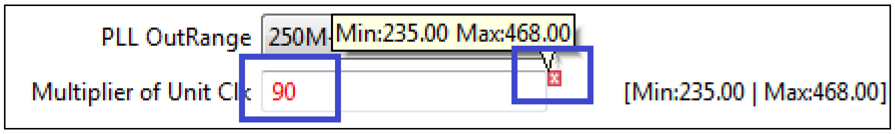


Data Format: YUV422_8_MODE1
Fifo Delay: 50

Figure 7-12: Output Data Format Selection

6. Managing Errors in the Configuration Data

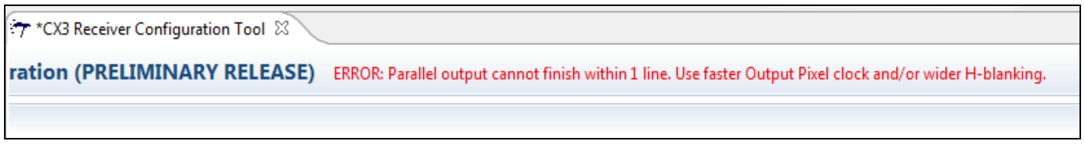
If an individual field violates permitted minimum or maximum values, it is highlighted in a RED color as shown in the following image and a small RED x mark is shown to the right of the input box. Hovering over the x will show the user the permitted values for the field.


PLL OutRange 250M Min:235.00 Max:468.00

Multiplier of Unit Clk 90 [Min:235.00 | Max:468.00]

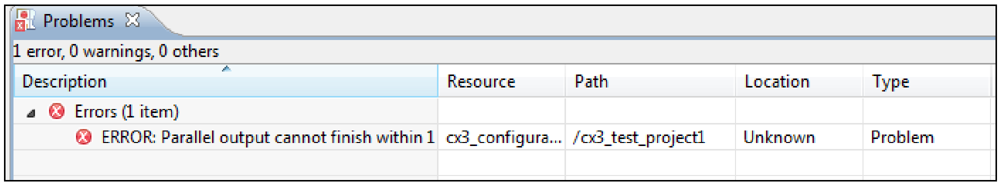
Figure 7-13: Errors in Input Data

Additionally, if any changes make data output not feasible across the interface, an error message listing the error and action to be taken to resolve the error is provided at the top of the screen and also in the Eclipse Problems view.


*CX3 Receiver Configuration Tool

ration (PRELIMINARY RELEASE) ERROR: Parallel output cannot finish within 1 line. Use faster Output Pixel clock and/or wider H-blanking.

Figure 7-14: Error Displayed on the Top Bar


Problems

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
ERROR: Parallel output cannot finish within 1	cx3_configura...	/cx3_test_project1	Unknown	Problem

Figure 7-15: Error Displayed in the Problems View

Please follow the instructions provided by the error message to resolve the issue.

7. Viewing/Using the source

Tabs allowing the user to switch between the configuration window and the source generated based on the configuration details are provided at the bottom left of the Configuration Tool window.

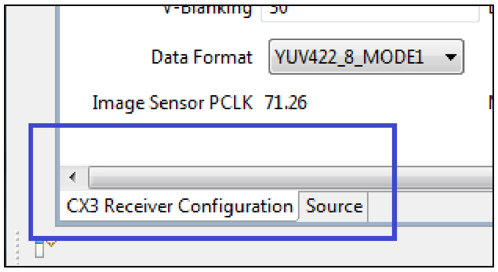
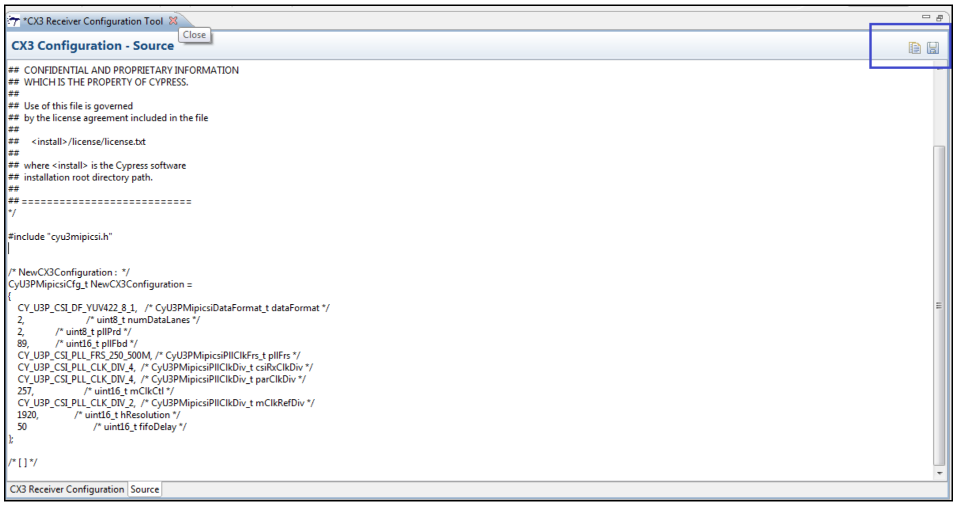


Figure 7-16: Source and Configuration Selection Tabs

Selecting Source, will take the user to the Source window from where the user can export the generated source by copying it to the clipboard, or by saving it to a C file using the provided export options.

The user can also copy just the structure and use in their source files.

The source file will always have CyU3PMipcsiCfg_t data based on the current values in the Configuration Tab even if the Configuration tab data has errors in it. The user must ensure that all errors on the configuration tab are resolved before using the code from the Source tab.



```
## CONFIDENTIAL AND PROPRIETARY INFORMATION
## WHICH IS THE PROPERTY OF CYPRESS.
##
## Use of this file is governed
## by the license agreement included in the file
##
##    <install>/license/license.txt
##
## where <install> is the Cypress software
## installation root directory path.
##
## ===========================
*/

#include "cyu3mipicsi.h"


/* NewCX3Configuration : */
CyU3PMipicsiCfg_t NewCX3Configuration =
{
   CY_U3P_CSI_DF_YUV422_8_1,   /* CyU3PMipicsiDataFormat_t dataFormat */
   2,                  /* uint8_t numDataLanes */
   2,          /* uint8_t pllPrd */
   89,         /* uint16_t pllFbd */
   CY_U3P_CSI_PLL_FRS_250_500M, /* CyU3PMipicsiPllClkFrs_t pllFrs */
   CY_U3P_CSI_PLL_CLK_DIV_4,  /* CyU3PMipicsiPllClkDiv_t csiRxClkDiv */
   CY_U3P_CSI_PLL_CLK_DIV_4,  /* CyU3PMipicsiPllClkDiv_t parClkDiv */
   257,                /* uint16_t mClkCtl */
   CY_U3P_CSI_PLL_CLK_DIV_2,  /* CyU3PMipicsiPllClkDiv_t mClkRefDiv */
   1920,           /* uint16_t hResolution */
   50                    /* uint16_t fifoDelay */
};

/* [ ] */
```

Figure 7-17: Source Screen

8. Creating Multiple Configurations

Multiple MIPI CSI-2 configurations can be created as part of the same cycx file by clicking on the green **+** icon on the top right of the configuration window. A configuration can be deleted by clicking on the **X** icon next to the green **+**.

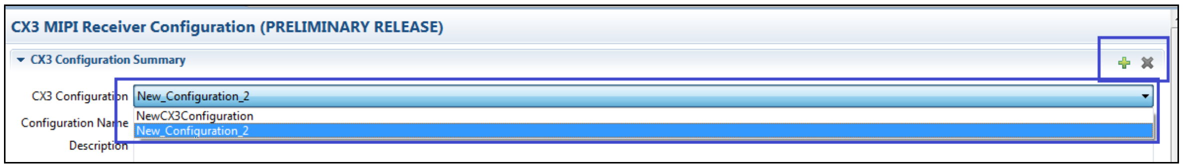


Figure 7-18: Creating Multiple Configurations

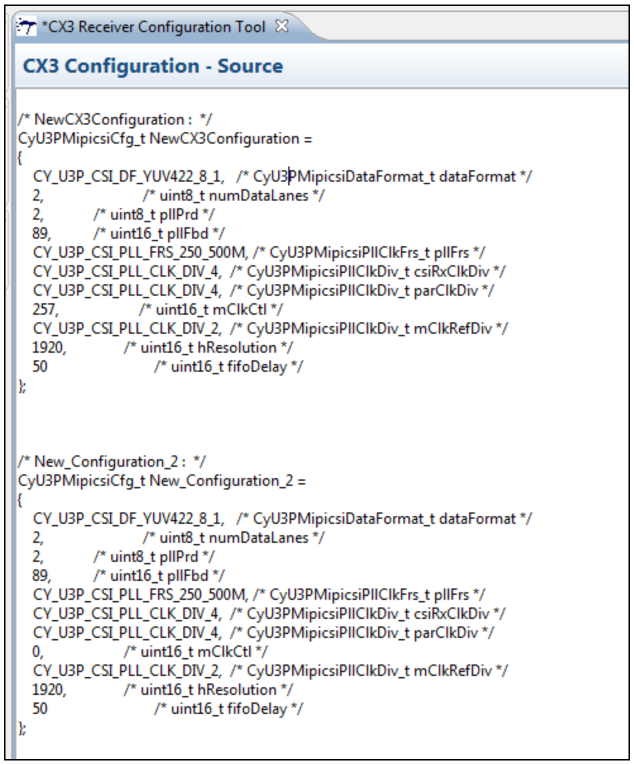
If using multiple configurations, the user should ensure that each configuration is differently named.

To configure options from a specific configuration, click on the CX3 Configuration dropdown box, and select the appropriate configuration.

When multiple configurations exist in a cycx file, one CyCx3MipicsiCfg_t object per configuration will be created in the source view as shown in the following image.

Please note that error and warning messages are only displayed for the active configuration that is displayed.

The user should ensure that **all** configurations are error free before using the code from the Source tab.



*CX3 Receiver Configuration Tool

CX3 Configuration - Source

```
/* NewCX3Configuration :  */
CyU3PMipicsiCfg_t NewCX3Configuration =
{
   CY_U3P_CSI_DF_YUV422_8_1,   /* CyU3PMipicsiDataFormat_t dataFormat */
   2,                    /* uint8_t numDataLanes */
   2,          /* uint8_t pllPrd */
   89,         /* uint16_t pllFbd */
   CY_U3P_CSI_PLL_FRS_250_500M, /* CyU3PMipicsiPllClkFrs_t pllFrs */
   CY_U3P_CSI_PLL_CLK_DIV_4,  /* CyU3PMipicsiPllClkDiv_t csiRxClkDiv */
   CY_U3P_CSI_PLL_CLK_DIV_4,  /* CyU3PMipicsiPllClkDiv_t parClkDiv */
   257,                 /* uint16_t mClkCtl */
   CY_U3P_CSI_PLL_CLK_DIV_2,  /* CyU3PMipicsiPllClkDiv_t mClkRefDiv */
   1920,            /* uint16_t hResolution */
   50                    /* uint16_t fifoDelay */
};


/* New_Configuration_2 :  */
CyU3PMipicsiCfg_t New_Configuration_2 =
{
   CY_U3P_CSI_DF_YUV422_8_1,   /* CyU3PMipicsiDataFormat_t dataFormat */
   2,                    /* uint8_t numDataLanes */
   2,          /* uint8_t pllPrd */
   89,         /* uint16_t pllFbd */
   CY_U3P_CSI_PLL_FRS_250_500M, /* CyU3PMipicsiPllClkFrs_t pllFrs */
   CY_U3P_CSI_PLL_CLK_DIV_4,  /* CyU3PMipicsiPllClkDiv_t csiRxClkDiv */
   CY_U3P_CSI_PLL_CLK_DIV_4,  /* CyU3PMipicsiPllClkDiv_t parClkDiv */
   0,               /* uint16_t mClkCtl */
   CY_U3P_CSI_PLL_CLK_DIV_2,  /* CyU3PMipicsiPllClkDiv_t mClkRefDiv */
   1920,            /* uint16_t hResolution */
   50                    /* uint16_t fifoDelay */
};
```

Figure 7-19: Source from Multiple Configurations

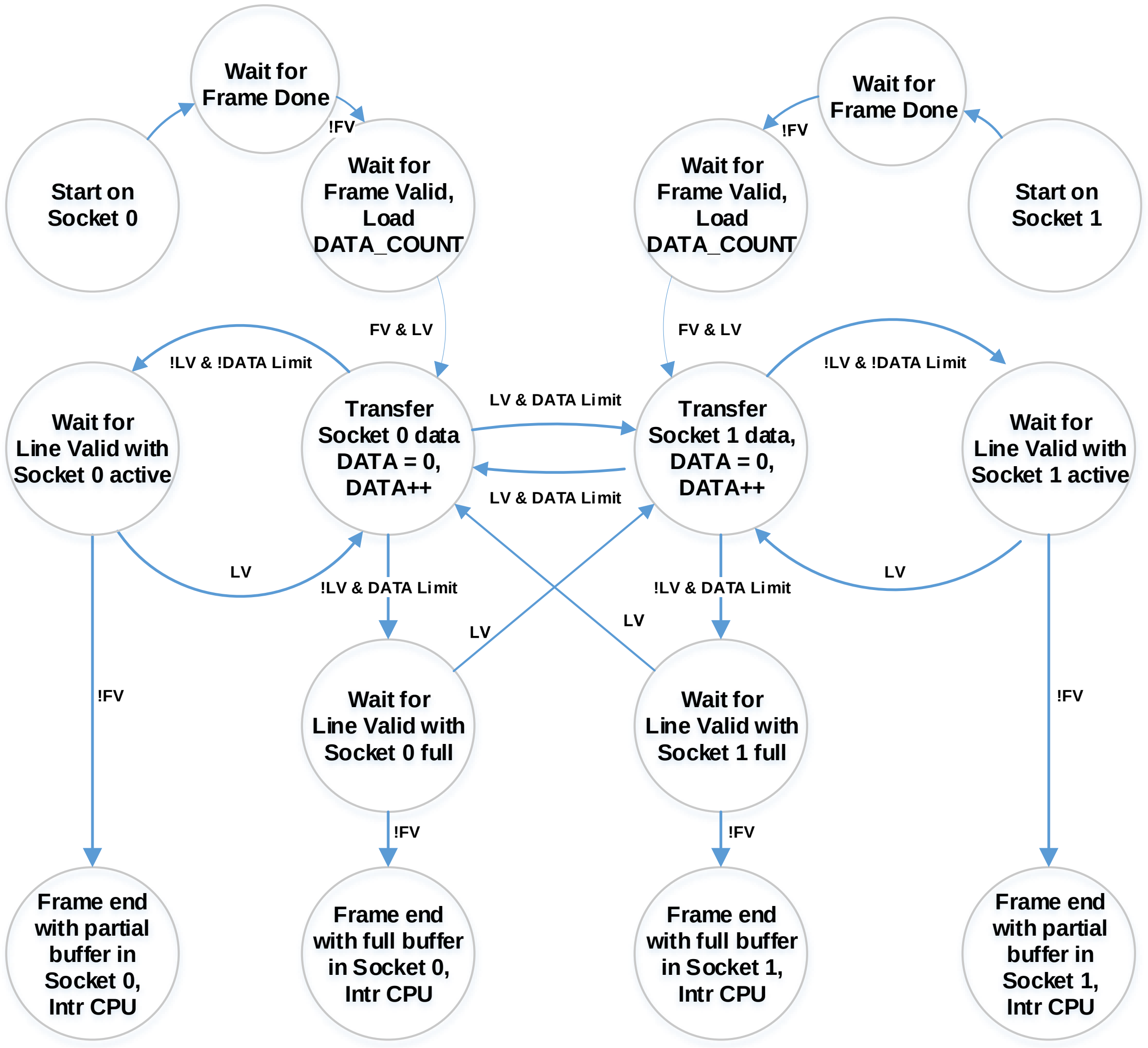
7.4 Fixed function GPIF-2 interface on the CX3

7.4.1 Fixed-Function GPIF-2 Waveform

The fixed-function GPIF II state machine on the CX3 implements the state machine shown in Figure 7-20. The state machine makes the parallel data provided by the MIPI CSI-2 available for transfer over two GPIF-II sockets which can be connected to a Many to One DMA channel.

The functionality of this state machine is similar to the GPIF II state machine described in Application Note AN75779 - How to Implement an Image Sensor Interface with EZ-USB® FX3™ in a USB Video Class (UVC) Framework.

Detailed description of this state machine along with instructions on creating CX3 based application projects will be provided via an Application Note on “How to interface a MIPI CSI-2 Image Sensor with EZ-USB® CX3” which will be made available from the Cypress website.



Wait for Frame Done
!FV
Start on Socket 0
Wait for Frame Valid, Load DATA_COUNT
Wait for Frame Done
!FV
Wait for Frame Valid, Load DATA_COUNT
Start on Socket 1
FV & LV
FV & LV
!LV & !DATA Limit
!LV & !DATA Limit
Wait for Line Valid with Socket 0 active
Transfer Socket 0 data DATA = 0, DATA++
LV & DATA Limit
LV & DATA Limit
Transfer Socket 1 data, DATA = 0, DATA++
Wait for Line Valid with Socket 1 active
LV
!LV & DATA Limit
!LV & DATA Limit
LV
LV
LV
!FV
!FV
Wait for Line Valid with Socket 0 full
Wait for Line Valid with Socket 1 full
!FV
!FV
Frame end with partial buffer in Socket 0, Intr CPU
Frame end with full buffer in Socket 0, Intr CPU
Frame end with full buffer in Socket 1, Intr CPU
Frame end with partial buffer in Socket 1, Intr CPU

Figure 7-20: CX3 Fixed Function GPIFII State Machine