Cypress EZ-USB® FX3™ SDK

Getting Started with FX3 SDK

Version 1.3.4

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 Universal Serial Bus

The universal serial bus (USB) has gained wide acceptance as the connection method of choice for PC peripherals. Equally successful in the Windows and Macintosh worlds, USB has delivered on its promises of easy attachment, an end to configuration hassles, and true plug-and-play operation.

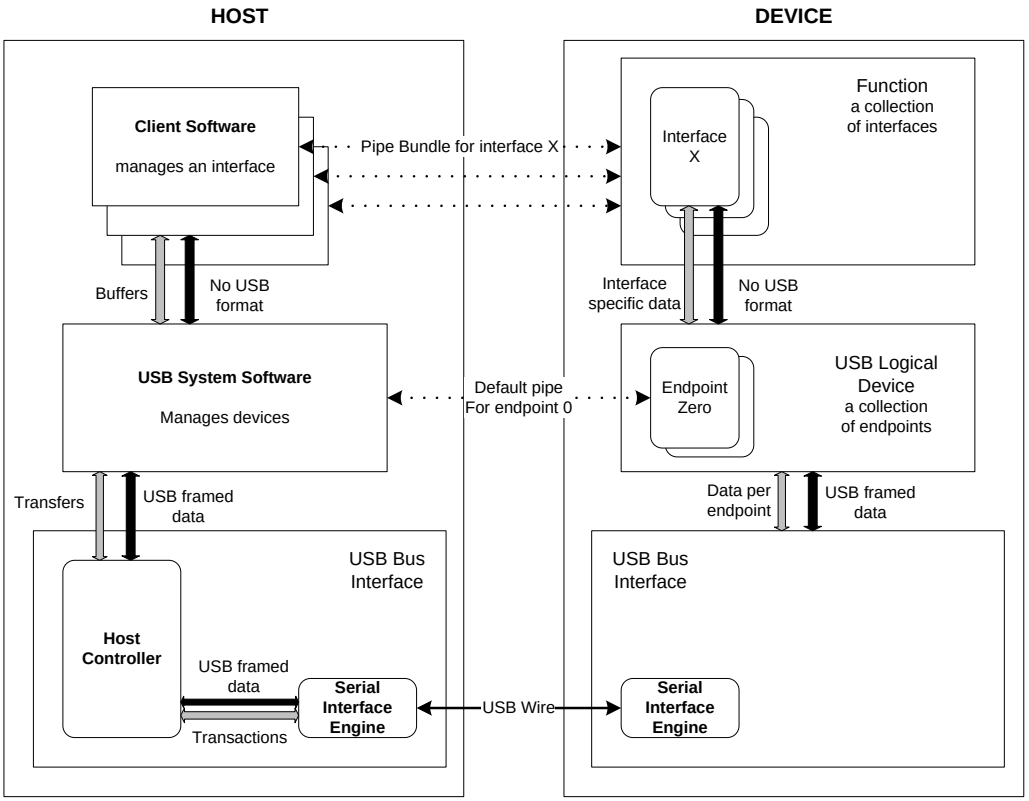


Figure 1-1: Communication layers in a USB system

The USB system has a single master: the host computer. The host connects to and controls a maximum of 127 devices (or peripherals) connected in a tiered topology through one or more hubs.

The USB specification defines the generic transport protocols for point to point communication between the host and a peripheral. The USB Implementers Forum, Inc. (USB-IF) has defined a number of functional protocols which various classes of USB devices need to implement. These protocols (or class specifications) include the USB Video Class (UVC) specification, the USB Mass Storage Class (MSC) specification, the Human Interface Device Class (HID) specification etc.

The following sub-sections provide a brief introduction to some of the USB concepts. More details can be found in Chapter 2 of the FX3 Programmer's Manual or in the USB Specifications.

1.1.1 Endpoints (EP)

The data communication between the USB host and device is achieved through a set of pipes called as endpoints. Each USB device needs to implement at least one bi-directional pipe called as Endpoint Zero or the default endpoint. The device can support up to 15 additional pipes for host to device (OUT) communication and 15 additional pipes for device to host (IN) communication. The data transferred on each pipe is typically independent, though some class specifications may define a relation between the data transferred on different pipes.

USB Endpoints can be classified into four types based on their functionality and capabilities:

1. Control Endpoint: This is a type that is reserved for the default endpoint (endpoint 0). This endpoint is intended for device discovery and other control functions, and is not designed for high performance data transfer. The USB specification defines a three stage (Request, Data, Handshake) protocol for transfers on the control endpoint.

2. Bulk Endpoint: This is a type that is defined for variable latency, reliable, high performance data transfers. A handshake mechanism is used to check for transfer readiness as well as successful completion.

3. Interrupt Endpoint: This is a type that is defined for periodic (fixed latency), reliable, low performance data transfers. A handshake mechanism is used to check for transfer readiness as well as successful completion.

4. Isochronous Endpoint: This is a type that is defined for periodic (fixed latency), lossy, high performance data transfers. Handshakes are not used in this case and any corrupted data packets are dropped by the receiver.

1.1.2 Functions

A service such as UVC or MSC provided by a USB device to the host is called as a function. A function can be divided into one or more interfaces, where each interface defines an independent communication protocol. For example, a UVC

function typically has two interfaces: a Video Control (VC) interface and a Video Streaming (VS) interface.

Each USB interface can have zero or more endpoints. An interface with no endpoints is still meaningful because it has access to the default endpoint (Endpoint zero). Endpoints other than endpoint zero cannot be shared across multiple interfaces.

Each USB device can implement one or more functions, and the capabilities of the device are reported to the host through a set of data blocks called descriptors. The format for the descriptors are defined by the USB specification.

1.1.3 Device Enumeration

When a USB device is connected to the host (directly or through a hub), its presence is detected through electrical signaling: a pull-up on the USB 2.0 D+ pin or the presence of a valid termination resistor on the USB 3.0 SSRX pins. A notification of the device connection is relayed upwards from the nearest hub to the host. Once the host is notified that a new device has been connected, it tries to identify the device by reading the descriptors that specify the device's capabilities. These descriptors are read through a standard set of requests and data that are exchanged using Endpoint zero. If the host is able to identify the software drivers that can talk to the device (based on the descriptors received), it enables the device functions by sending a SET_CONFIGURATION request. If the host is unable to locate a driver that can talk to the device, it leaves the device in an un-configured state.

This process or device identification and driver selection is called as USB Device Enumeration. If the enumeration is successful, the software driver starts talking to the USB device and accessing the functions provided by it. For example, the storage class driver starts reading the data from a Mass Storage (MSC) function and makes the data available to the user through a disk drive (or mount point).

1.2 FX3 Device Overview

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

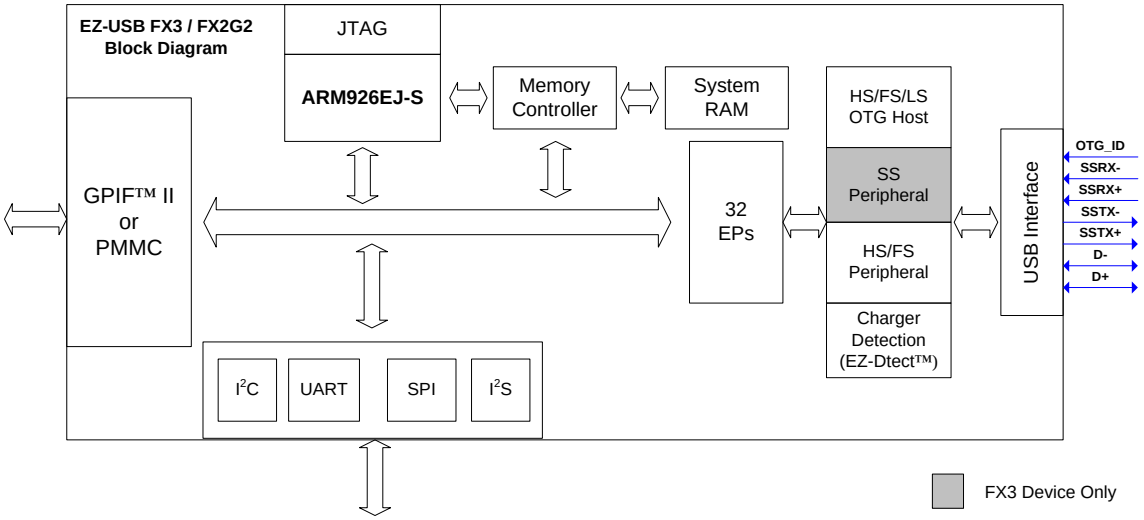


Figure 1-2: EZ USB FX3 System Diagram

1.2.1 USB Interface

The FX3 device supports operation as a USB device under the following USB connection modes:

1. USB 3.0 (SuperSpeed) interface at 5 Gbps
2. USB 2.0 (Hi-Speed) interface at 480 Mbps
3. USB 1.1 (Full Speed) interface at 12 Mbps

The FX3 device also supports operation as a USB 2.0 embedded host that supports a single USB device operating at:

1. USB 2.0 (Hi-Speed) interface at 480 Mbps
2. USB 1.1 (Full Speed) interface at 12 Mbps
3. USB 1.0 (Low Speed) interface at 1 Mbps

The FX3 device also supports the USB On-The-Go (OTG) specification which allows the device to switch roles between host and peripheral based on the device connected to it.

The device is also compliant with the USB Battery Charging Specification v1.1.

1.2.2 Processor Core

The FX3 device has an embedded high-performance ARM926EJ-S core with up to 512 KB of SRAM (SYSMEM) connected to it. The SYSMEM is a unified memory block which is used for code and data storage. The FX3 device also provides Instruction and Data caches of 8 KB each. Access to frequently accessed

instructions such as Interrupt Service Routines (ISRs) and frequently accessed data such as runtime stacks, are speeded up by locating them in zero latency memory areas called as Tightly Coupled Memories (TCMs). The FX3 device has a dedicated 16 KB Instruction TCM (I-TCM) for code and 8 KB Data TCM (D-TCM) for data storage.

The ARM core is connected to Vectored Interrupt Controller (VIC) that manages interrupts in the system. The VIC can handle up to 32 input interrupt requests including internal interrupts from various hardware blocks and external interrupts triggered through GPIO pins.

The ARM core also supports the standard ARM JTAG debug port which allows runtime debugging of firmware applications using any standard JTAG debug probe. The CYUSB3KIT-003 kit provides an on-board JTAG interface implemented using the Cypress CY7C65215 USB to Serial bridge part.

1.2.3 General Programmable Interface (GPIF-II)

The device supports a configurable General Programmable Interface (GPIF-II) interface which makes it possible to interface the device with microprocessors, ASICs and FPGAs with a variety of interfaces. The GPIF-II block can be configured to have a variable set of signals (data, address and control pins) with user defined interface timing.

The General Programmable Interface GPIF II is an enhanced version of the GPIF in FX2LP, Cypress's flagship USB 2.0 product. The GPIF II Controller drives the behavior and timing of the GPIF II Interface based on one or more sets of user defined configurations (GPIF State Machines). It provides an easy and glue-less 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 GPIF State Machine corresponding to the desired interface behavior can be generated using the GPIF II Designer tool which is part of the FX3 SDK.

The device also supports a MMC Slave (PMMC) interface which allows connection to the SD/MMC flash interface which is commonly available on microprocessors. As the PMMC interface makes use of the same pins as GPIF-II, the user has to choose between the GPIF-II and PMMC interfaces while designing the system.

1.2.4 DMA Support

The most common applications of FX3 require high performance data transfers between the USB and GPIF-II blocks of the FX3 device. The FX3 device has an in-built DMA controller which allows data to be transferred at high rates across these interfaces.

All data transferred through the FX3 device needs to go through the SYSMEM on the device. The data is buffered into the SYSMEM by the receiving block (PRODUCER) and then read and sent out by the transmitting block (CONSUMER). The distributed DMA logic allows the PRODUCER and CONSUMER to communicate directly to signal data readiness, without having to rely on firmware intervention.

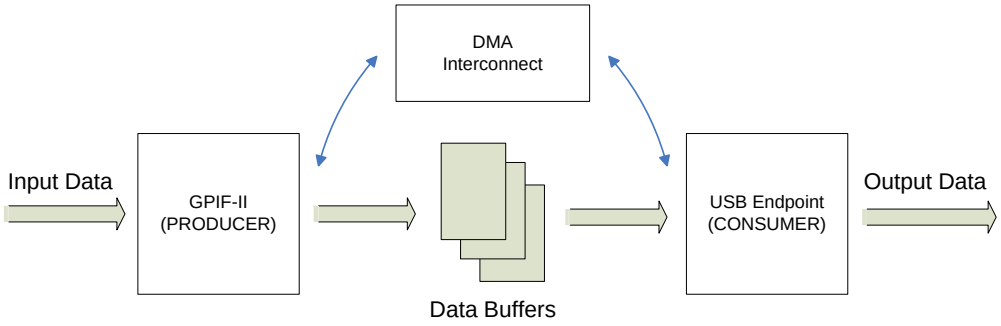


Figure 1-3: DMA data flow through FX3

1.2.5 Serial Interfaces

The FX3 also supports serial peripheral interfaces: namely UART, SPI, I2C, and I2S for communicating to on board peripherals (for example, the I2C interface is typically connected to an image sensor or an EEPROM).

Pins on the FX3 device can also be configured to function as GPIOs. The GPIOs can support simple input/output functions, interrupt generation and more specialized functions such as Pulse Width Modulated output (PWMs) and input pulse measurement.

1.2.6 FX3 Device Family

Cypress also offers a set of other USB controllers which are based on the FX3 device architecture.

The EZ-USB FX2G2 controller is a USB 2.0 version of the FX3 controller. It supports all of the features of the FX3 device except the USB 3.0 interface.

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. Up to two storage (SD/MMC) ports are supported, and only one of the UART, SPI or I2S interfaces can be used along with these ports. The device architecture allows the storage peripherals to be accessed at high speeds through the USB as well as the GPIF-II or PMMC interfaces.

The Benicia device offers the same features as the FX3S device, but is offered in a smaller Wafer Level Chip Scale Package (WLCSP). The Bay device is similar to the Benicia device, but does not support the USB 3.0 interface.

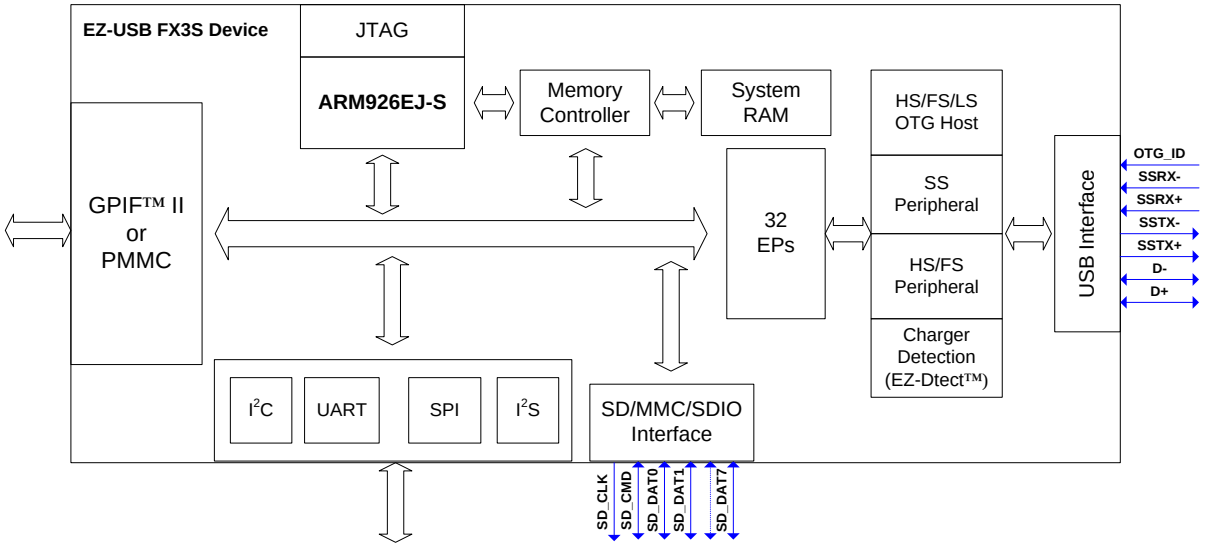


Figure 1-4: EZ-USB FX3S System Diagram

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. The I2C interface on the CX3 is used to configure the MIPI CSI-2 interface in addition to controlling any externally connected slave devices.

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.

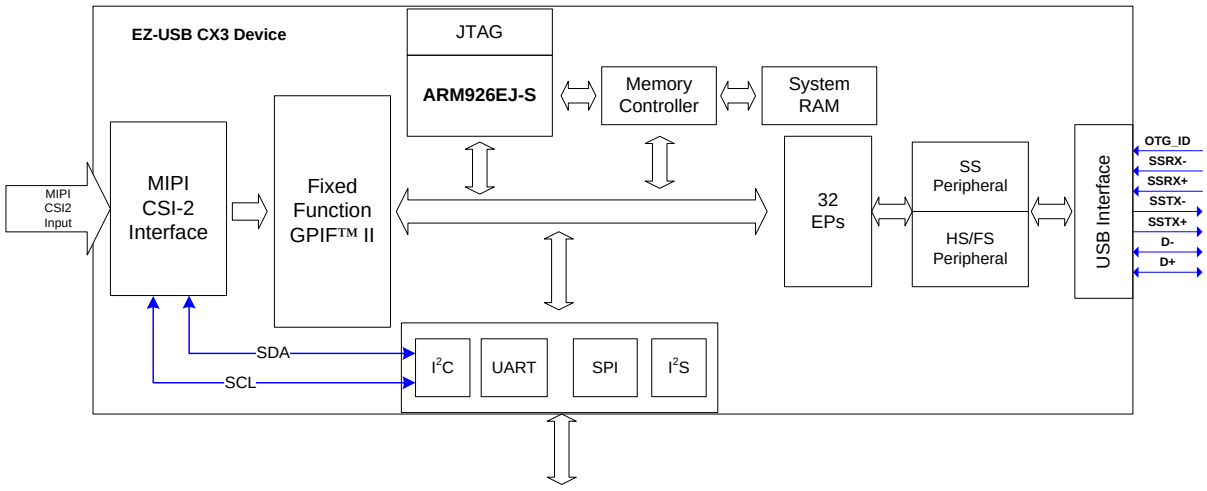


Figure 1-5: CX3 System Diagram

The SD3 controller is a USB 3.0 to SD/eMMC/SDIO bridge device, and does not support other processor interfaces like GPIF-II or MMC slave. The SD2 controller is similar to SD3, but does not support the USB 3.0 interface.

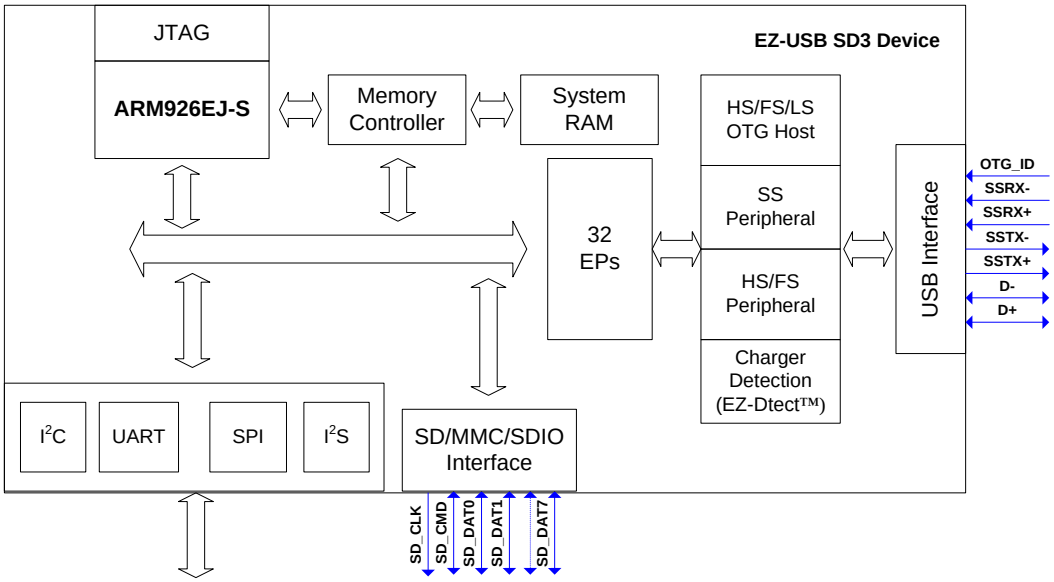


Figure 1-6: SD3 System Diagram

Table 1-1 shows the various USB controllers in the FX3 family.

Note: Most of these parts have multiple variants with different operating ranges and packaging. Only one of the parts is listed in this table.

Device Type	Part Number(s)	Feature List
EZ-USB FX3	CYUSB3014-BZXC	• USB 3.0 Device • USB 2.0 Device, Host, OTG • GPIF-II up to 32 bits • MMC-Slave interface • SPI, I2C, I2S, UART • 512 KB System RAM
	CYUSB3013-BZXC	• USB 3.0 Device • USB 2.0 Device, Host, OTG • GPIF-II up to 16 bits • MMC-Slave interface • SPI, I2C, I2S, UART • 512 KB System RAM
	CYUSB3012-BZXC	• USB 3.0 Device • USB 2.0 Device, Host, OTG • GPIF-II up to 32 bits

		• MMC-Slave interface • SPI, I2C, I2S, UART • 256 KB System RAM
	CYUSB3011-BZXC	• USB 3.0 Device • USB 2.0 Device, Host, OTG • GPIF-II up to 16 bits • MMC-Slave interface • SPI, I2C, I2S, UART • 256 KB System RAM
FX2G2	CYUSB2014-BZXC	• USB 2.0 Device, Host, OTG • GPIF-II up to 32 bits • MMC-Slave interface • SPI, I2C, I2S, UART • 512 KB System RAM
FX3S	CYUSB3035-BZXC	• USB 3.0 Device • USB 2.0 Device, Host, OTG • GPIF-II up to 16 bits • MMC-Slave interface • Two SD/MMC/SDIO ports • SPI, I2C, I2S, UART • 512 KB System RAM
CX3	CYUSB3065-BZXC	• USB 3.0 Device • USB 2.0 Device • MIPI CSI-2 Interface • SPI, I2C, I2S, UART • 512 KB System RAM
SD3	CYUSB3025-BZXI	• USB 3.0 Device • USB 2.0 Device, Host, OTG • Two SD/MMC/SDIO ports • SPI, I2C, I2S, UART • 512 KB System RAM
SD2	CYUSB2025-BZXI	• USB 2.0 Device, Host, OTG • Two SD/MMC/SDIO ports • SPI, I2C, I2S, UART • 512 KB System RAM
Benicia	CYWB0263	• USB 3.0 Device • USB 2.0 Device, Host, OTG • GPIF-II up to 16 bits • MMC-Slave interface • Two SD/MMC/SDIO ports • SPI, I2C, I2S, UART • 512 KB System RAM

Bay	CYWB0163BB-FBXIT	• USB 2.0 Device, Host, OTG • GPIF-II up to 16 bits • MMC-Slave interface • Two SD/MMC/SDIO ports • SPI, I2C, I2S, UART • 512 KB System RAM

Table 1-1: Parts in the FX3 family

1.3 FX3 SDK Overview

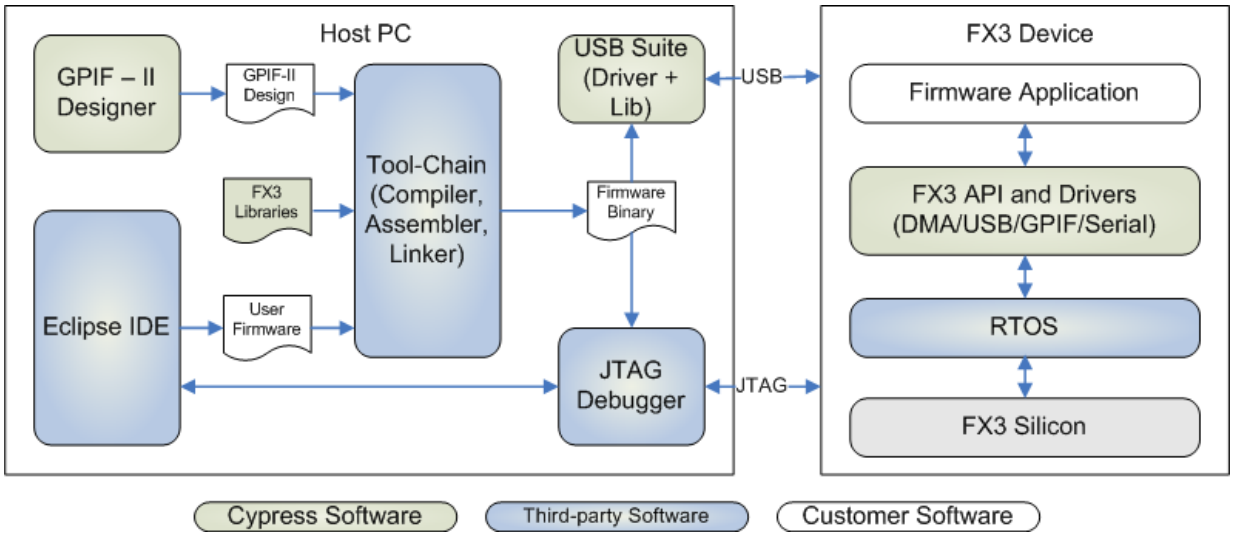


Figure 1-7: FX3/FX3S Software Solution (SDK)

Cypress delivers a complete software and firmware stack for FX3 and related devices, in order to easily integrate all USB applications in an embedded system environment.

The software development kit comes with industry standard development tools (Eclipse based IDE and GNU Tool-chain) and provides a number of application examples which help accelerate application development.

All of the devices shown in **Table 1-1** are supported using a common SDK environment. It is expected that users of a device will only incorporate the supported features in the applications that they build using the SDK. When the compiled firmware binary is loaded on the device, it will check whether the requested features are supported and report errors if a mismatch is found.

The components of the FX3 software development kit are shown in **Figure 1-7**.

1.3.1 Device Firmware

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 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.

1.3.2 Development Tools

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.

1.3.3 Host Side Software

The SDK also provides a set of software tools that run on standard USB host computers and communicate with the FX3 device. These tools are provided for Windows, Linux and Mac platforms.

1.3.3.1 CyUsb for Windows

The Windows version of the USB host side stack for the FX3 contains:

- Cypress generic USB 3.0 driver (CyUsb3.sys) certified for Windows 10, Windows 8.1, Windows 8, Windows 7, Windows Vista and Windows XP (32-bit). The driver will bind to the EZ-USB FX3 (and related devices) in the USB boot-loader mode as well as when running the firmware examples provided as part of the SDK. The driver binding is done based on the VID/PID pairs.

- 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. This utility also allows programming of firmware binaries onto the FX3 device RAM and flashing the firmware binary onto EEPROM or SPI FLASH devices connected to the FX3.

- Bulkloop application to perform data loopback on the Bulk endpoints. This is a test application that allows the user to check basic device and firmware functionality.

- Streamer application to perform data streaming over Isochronous or Bulk endpoints. This is a test application that can be used to measure the data transfer performance that can be achieved by the FX3 device as well as the host computer.

1.3.3.2 CyUsb for Linux

The Linux version of the CyUsb host side tools provides:

- A libcyusb wrapper on top of the general purpose libusb user space library. The libcyusb wrapper library provides a set of functions that can be used by C/C++ applications to exchange data with USB devices. The libcyusb wrapper is not specific to Cypress USB devices and can be used with other USB controllers as well.

- A cyusb_linux GUI tool which provides functionality similar to the USB control center Windows application.

- A set of console application examples using the libcyusb library that support FX3 device programming, data transfers and throughput measurement.

1.3.3.3 CyUsb for Mac

The Mac version of the CyUsb host side tools provides:

- A libcyusb wrapper on top of the general purpose libusb user space library. The libcyusb wrapper library provides a set of functions that can be used by C/C++ applications to exchange data with USB devices. The libcyusb wrapper is not specific to Cypress USB devices and can be used with other USB controllers as well.

- A set of console application examples using the libcyusb library that support FX3 device programming, data transfers and throughput measurement.

1.4 FX3 Development Platform Overview

Cypress provides two types of Development Kits for the EZ-USB FX3 device:

1. The SuperSpeed Explorer is an easy-to-use and inexpensive development platform which is built to facilitate evaluation and development of solutions using the FX3 device. Cypress also provides a set of interconnect boards which can be used to connect the SuperSpeed Explorer Kit to Image sensors and FPGAs. Please see http://www.cypress.com/?rID=99916 for more details and documentation about the SuperSpeed Explorer Kit.

2. The FX3 Development Kit (CYUSB3KIT-001) is a kit that supports USB 2.0 OTG operation, self powered device operation and third-party JTAG debugger usage in addition to the features provided by the SuperSpeed Explorer Kit. Please see http://www.cypress.com/?rID=58321 for more details and documentation about the FX3 Development Kit.

The firmware examples provided with the FX3 SDK can be used on both the SuperSpeed Explorer and FX3 DVK boards. The only constraint is that the USB 2.0 OTG/Host features are not supported by the SuperSpeed Explorer Kit.

Please see the respective kit packages for the corresponding used guides.

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 (x86)\Cypress\EZ-USB FX3 SDK\<version>, where 1.3 is the version for this release of the SDK.

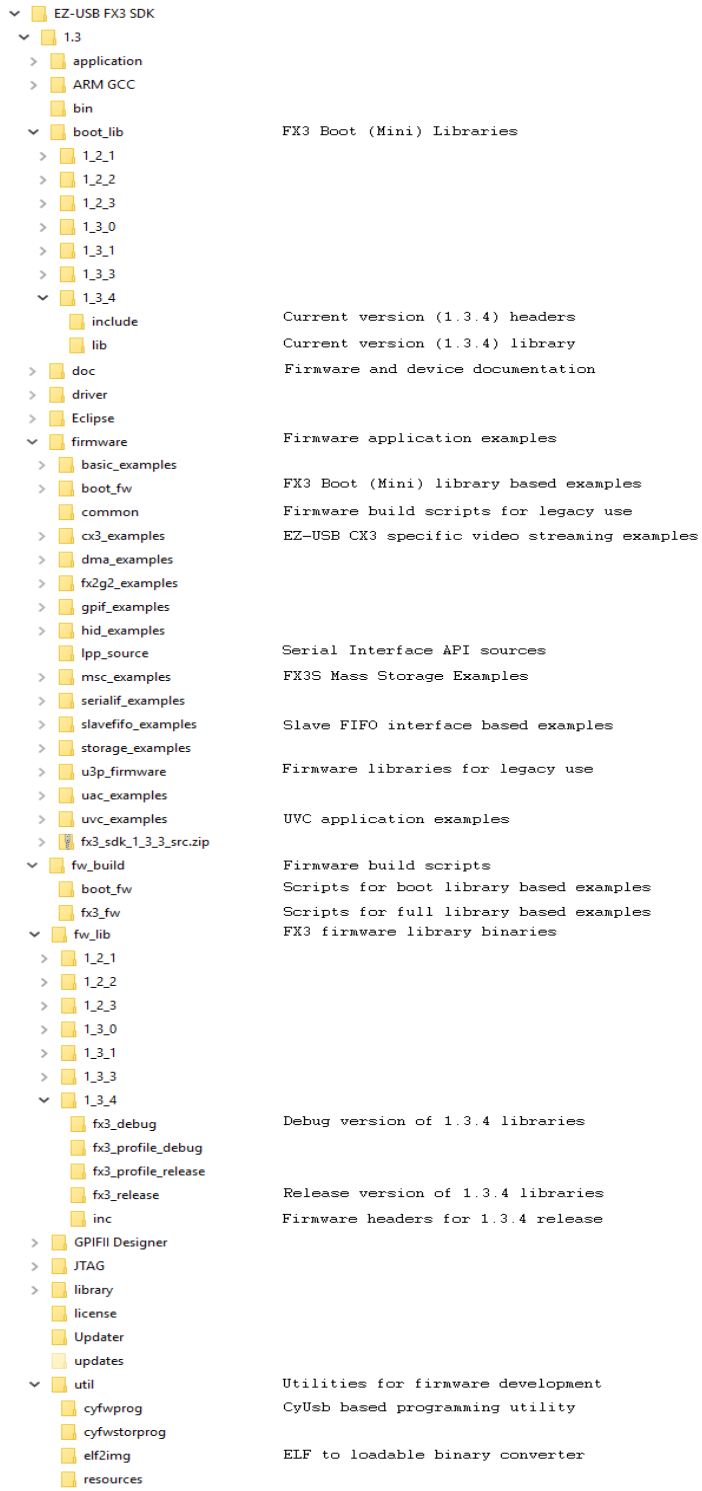
The FX3 SDK also includes the following documentation:

1. FX3 Release Notes
2. FX3 Programmer’s Manual
3. FX3 API Guide
4. FX3 SDK Trouble Shooting Guide
5. EzUSBSuite User Guide
6. FX3 Technical Reference Manual
7. CX3 Technical Reference Manual

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

2.2 Installed Directory Structure

Figure 2-1 shows the firmware related files in the FX3 SDK installation.



1

Figure 2-1: FX3 SDK: Structure of firmware files

Figure 2-2 shows the structure of the USB Suite (Windows driver and library) files in the FX3 SDK installation.

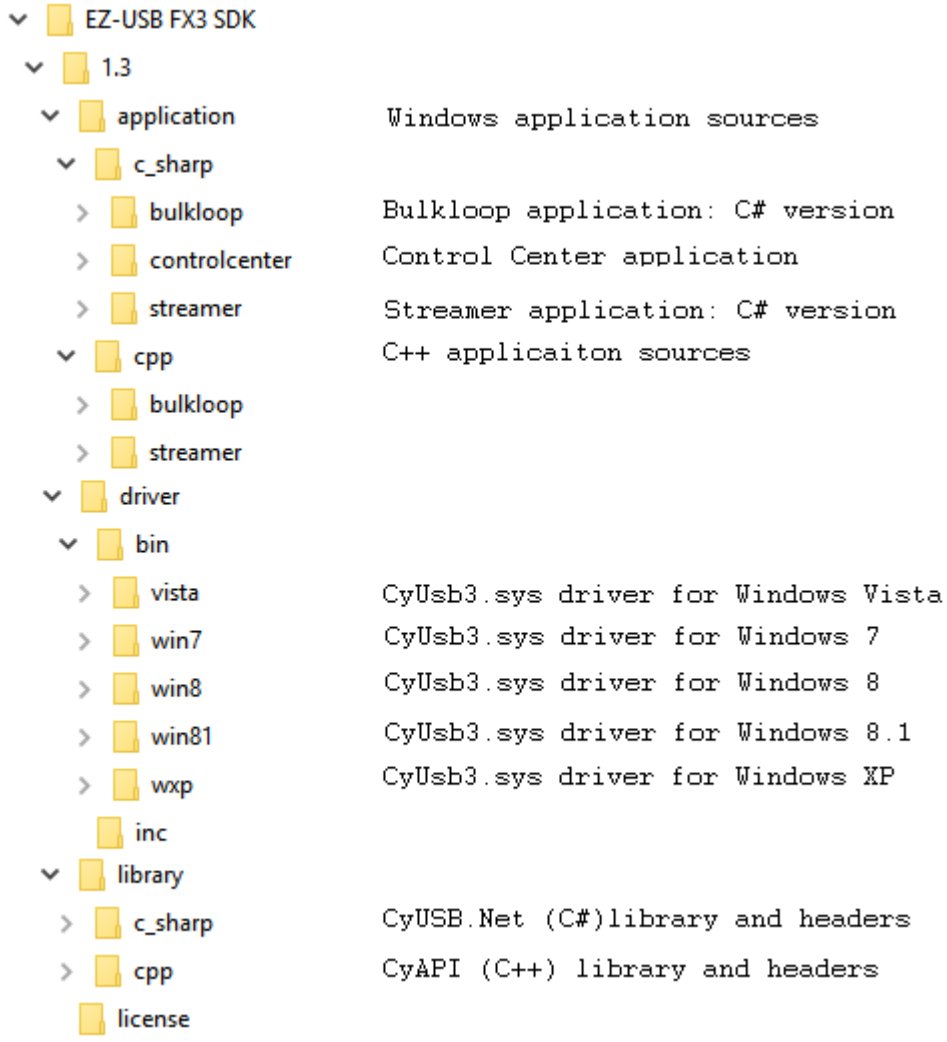


Figure 2-2: FX3 SDK: Structure of USBSuite files

3 Working with the FX3 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. Launch the IDE, import the example projects and build them
3. Connect the FX3 board to the host machine and power the board up
4. Bind the driver for the FX3 device.
5. Launch the CyControl utility
6. Download the boot image to the FX3 device RAM using the Control Center “Download Firmware” option
7. The FX3 device will start running the newly loaded 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.

It is assumed that the FX3 DVK or SuperSpeed Explorer 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.

3.1.1 Installing the SDK

The SDK can be installed by starting up the FX3SDKSetup.exe installer, and then following the on-screen installation instructions. If you have a previous version of the SDK installed, the Cypress Update Manager software can be used to select and install the new version.

3.1.2 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 the EzUsbSuite User Guide. This document details how to import the projects, configure settings for the projects, debug and run the example using the GNU debugger.

1. Open the EZ USB Suite IDE and create a new workspace.
2. Use the File -> Import option to bring up a window for selection of projects to be imported. Select the “General -> Existing Projects into Workspace” option and click on Next.

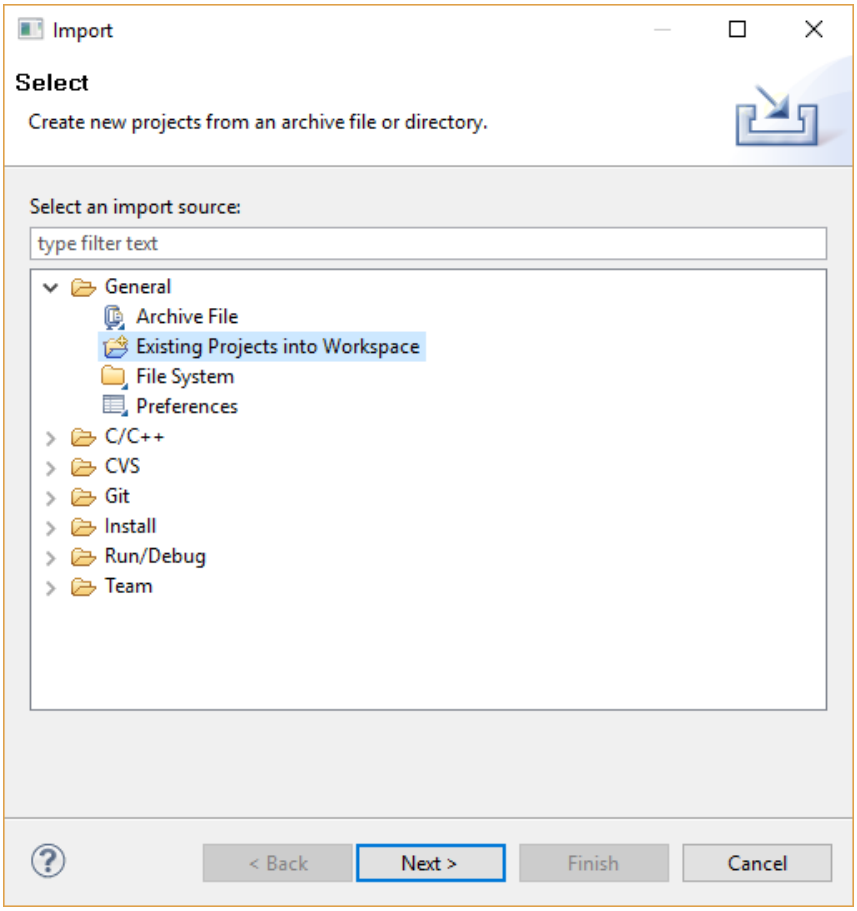


Figure 3-1: Eclipse Project Import Option

3. In the “Import” window that pops up, use the browse option next to “Select root directory” and point to the <FX3 SDK Install Location>/1.3/firmware folder. Eclipse locates and brings up a list of all firmware example projects under this folder. Select the projects of interest using the checkboxes and the “Select All” and “Deselect All” buttons. Make sure that the “Copy projects

into workspace” option is checked and click on “Finish” to start importing the projects into the workspace.

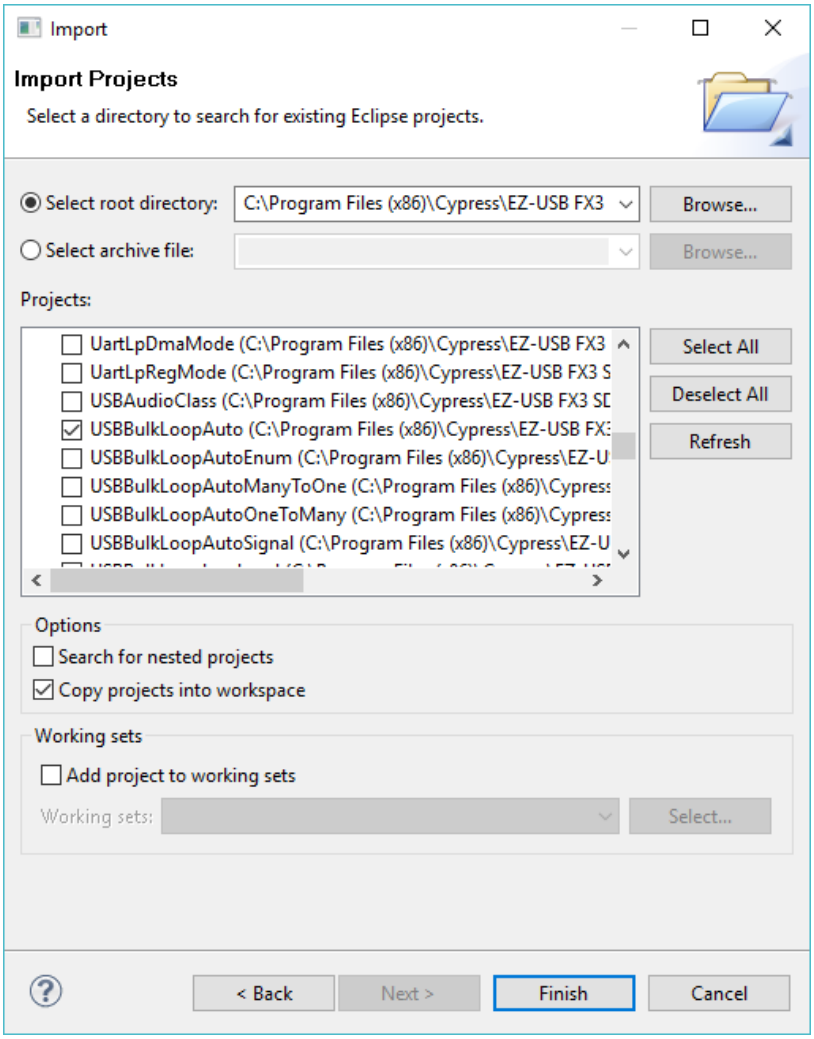


Figure 3-2: Selecting projects to be imported

4. Select the USBBulkLoopAuto project in the IDE, right-click and select the “Build Project” option. The application sources will be compiled into an ELF binary and then converted into a loadable image (.img) format as part of the build process.

3.1.3 Setting up the FX3 DVK Board

The FX3 device supports multiple boot modes using a ROMed Boot-loader program. The selection of the boot mode is done through a set of PMODE pin straps. Please refer to the AN76405 - EZ-USB® FX3™ Boot Options Application Note for more information on the boot options and their selection.

The USB boot mode is the easiest way to get started with the FX3 device, as it allows the firmware to be changed after each device reset. The following sections assume that the USB boot mode is used for testing the applications.

The SuperSpeed Explorer kit only supports two of the device boot modes (USB boot or boot from I2C EEPROM) and the selection between the two is done through a single jumper.

The FX3 DVK supports all boot modes, and the selection is done through a set of switches and jumpers.

Please refer to the appropriate kit user guide for the settings to select USB boot mode.

3.2 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. When connected to the host in USB boot mode, 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.

3.3 Firmware Download

The Control 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 Control Center (Start->Programs->Cypress->EZ-USB FX3 SDK->Cypress USBSuite->Control Center). The FX3 device shall be listed as “Cypress USB BootLoader” device in the main window, and you can look through the USB descriptors reported by the device.

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

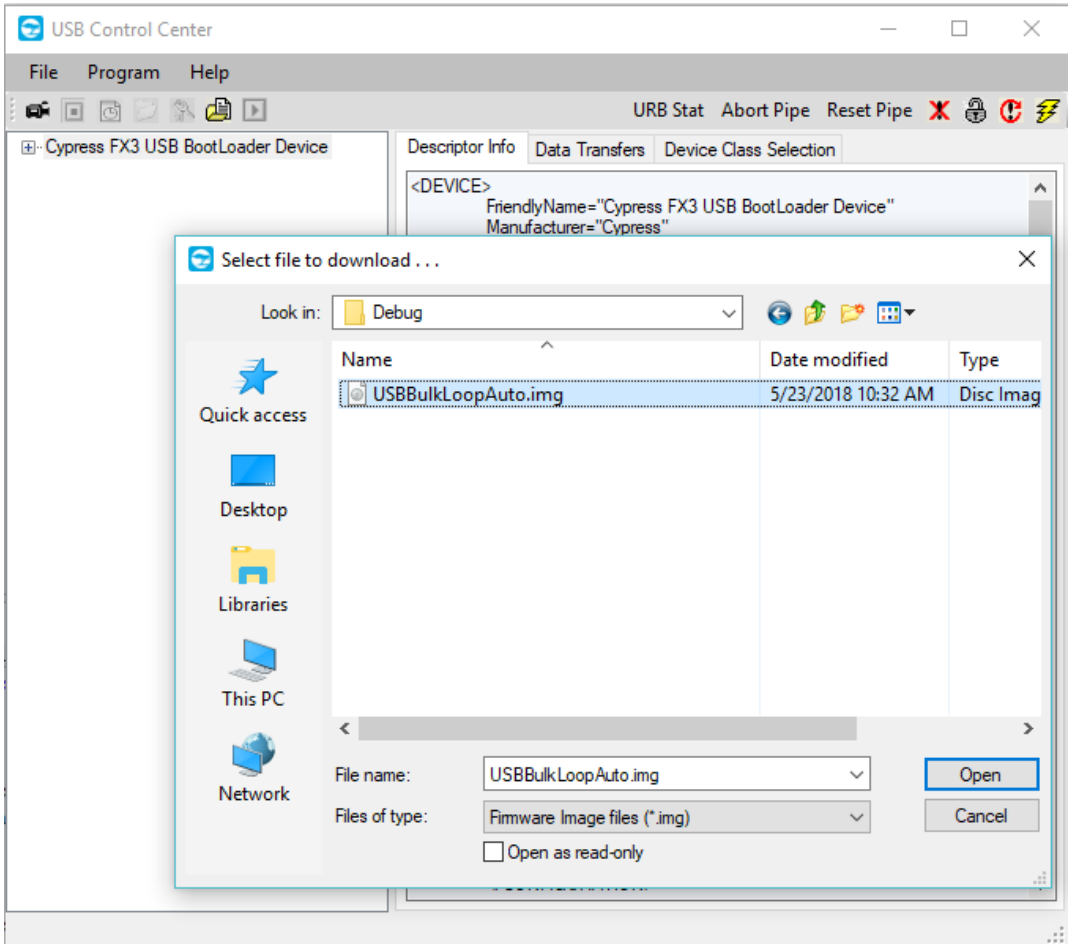


Figure 3-3: Firmware download to FX3 RAM

3.4 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.

3.4.1 Using Control Center

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

Expand the device name and configuration repeatedly until the endpoints on the device are listed. Select the OUT endpoint and move to the “Data Transfers” tab on the right-hand half of the UI. Enter the data to be sent in the “Text to send:” or “Data to send (Hex):” text box, and click on the “Transfer Data-OUT” button. The data transfer will be completed and the status and actual data transferred will be displayed in the lower half of the tool window.

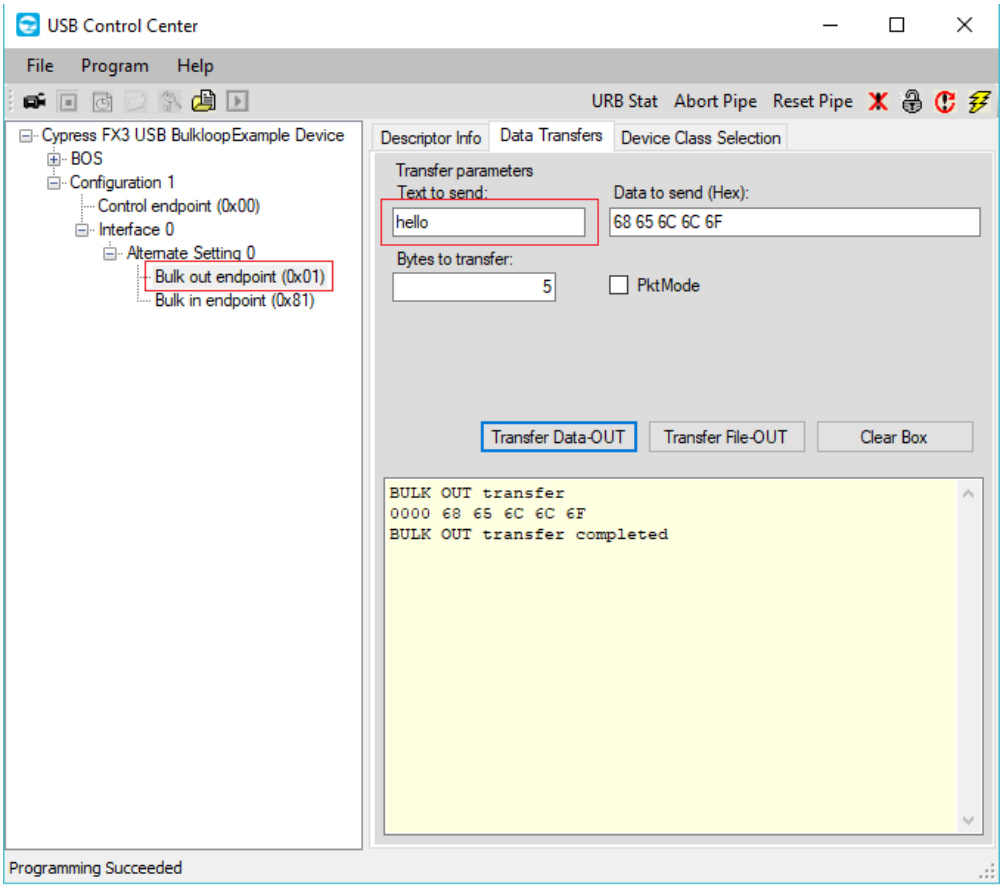


Figure 3-4: OUT data transfer using Control Center

Now select the IN endpoint on the left-hand side of the UI, and click on the “Transfer Data-IN” button to read data back from the device. Since the FX3 firmware is implementing a loop-back function, the same data that was sent out will be received when reading back.

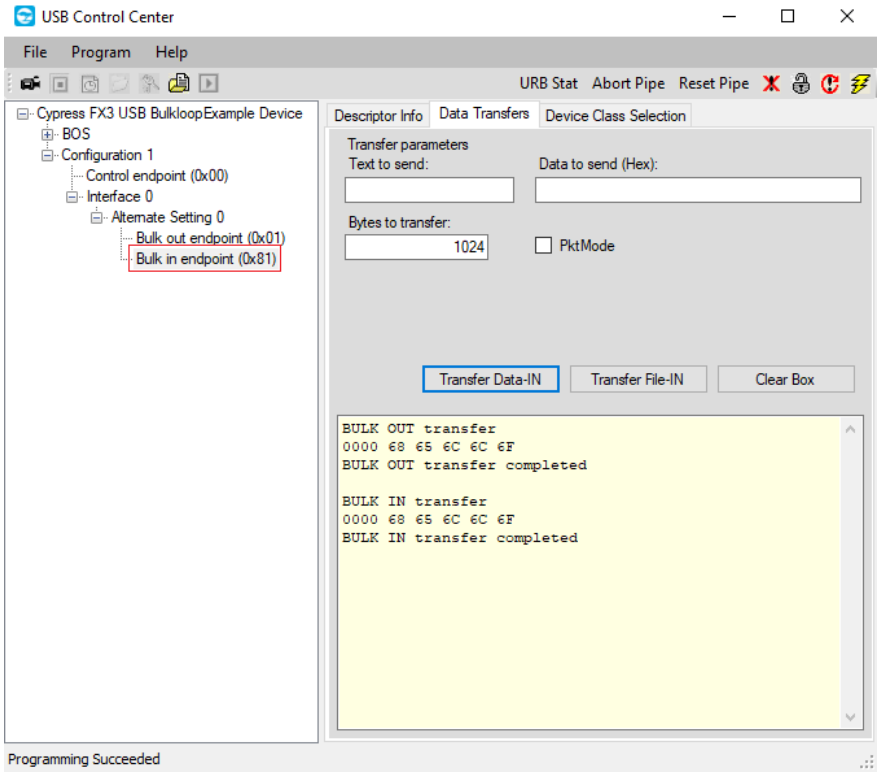


Figure 3-5: IN data transfer using Control Center

3.4.2 Using the BulkLoop Application

The BulkLoop application is a dedicated Windows application that is intended for testing data transfers with devices that implement a loop-back function. To launch the tool, select "Start→Programs→Cypress→EZ-USB FX3 SDK→Cypress USBSuite→Bulk Loop" from the start menu.

The Cypress USB device will already be identified by the utility. If the device only provides a single OUT and IN endpoint, these endpoints would also be selected automatically. If multiple endpoints are present, the target endpoints can be selected from the drop-down list.

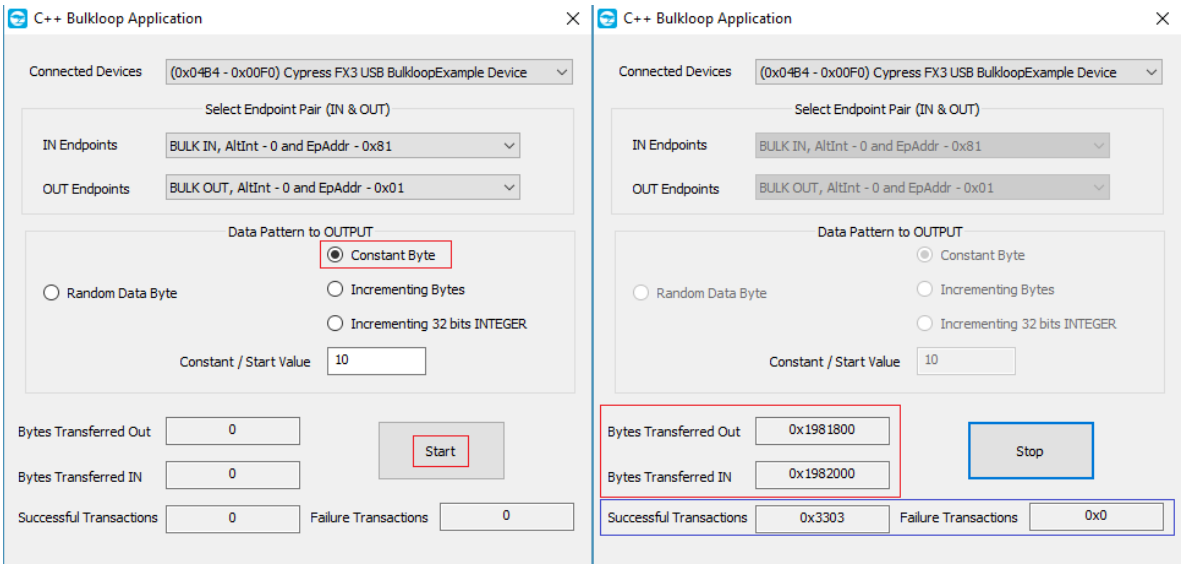


Figure 3-6: Bulk Loop: Setup and transfer results

The utility provides four options to define the data pattern that should be sent to the device through the OUT endpoint. Choose the desired option using the radio buttons, and click on “Start” to start data transfer.

This application repeatedly sends out data packets based on the selected patterns, reads the data back from the IN endpoint and verifies that the data is returned correctly. The amount of data transferred in each direction is displayed in the text boxes at the bottom of the UI. The UI also displays the number of successful transfers and failed transfers (including cases where the data read from the IN endpoint does not match the data sent on the OUT endpoint).

The transfers are performed continuously until the “Stop” button is clicked.

4 Firmware Overview

The FX3 SDK provides the sources for the FX3 firmware libraries as well as a number of examples that demonstrate the usage of these libraries.

4.1 Firmware Sources

The SDK package includes sources for most of the FX3 firmware libraries. The only exceptions are:

1. The ThreadX RTOS source. The ThreadX license terms only allow Cypress to provide the RTOS in a pre-compiled library form; and not in source form.
2. Sources for a small set of functions related to device initialization and identification. These functions deal with Cypress' proprietary information and are provided in pre-compiled library form only. This does not include any functionality that comes into play once the device has been initialized.
3. Sources for controlling the Aptina and OmniVision sensors that are used in the CX3 firmware examples. This is not provided because an NDA with the sensor vendor is required to access this information. Please contact the sensor vendor to complete the NDA process and then request Cypress for the corresponding firmware sources.

Please refer to the FX3FwSourceManual.pdf document for information on how to make use of the firmware sources, and to compile the FX3 firmware libraries from them.

4.1.1 Serial Peripheral Firmware Sources

As in prior FX3 SDK releases, the sources for the Serial Peripheral (GPIO, UART, I2C, I2S and SPI) drivers and APIs are also provided in the form of an Eclipse project. These sources are provided separately as it is more likely that these sources need to be customized based on the user's requirements.

These sources are also part of the source tree that includes the other driver and API sources.

4.2 Firmware Examples

This section lists the various firmware examples provided as part of the FX3 SDK. Each of these examples is provided in the form of an Eclipse project that can be imported into the EzUsbSuite IDE for modification and compilation.

These examples work at USB 3.0 and USB 2.0 speeds and are generally supported on all of the device variants such as FX3, FX3S, CX3 and FX2G2. Examples that make use of specific device features such as the MIPI CSI interface or the SD/MMC interface can only work on the corresponding device types.

Please note that all examples will function in USB 2.0 mode only when used with the FX2G2 device.

The term FX3 is used below in general to refer to all device variants. If an example requires a specific device type, the restriction is mentioned.

4.2.1 GPIO examples

These are simple starter applications that demonstrate how to initialize the FX3 device and the IOs on the device.

1. GpioApp: This example shows how to control output pins and sample the state of input pins on the FX3 device. This application uses a generic application structure that initializes other device blocks, and therefore has a large memory foot print.

   Location: firmware/serial_examples/cyfxgpioapp folder.
2. GpioComplexApp: This example shows how to use the more advanced GPIO features on the FX3 device. The application demonstrates:
   a) Driving an output pin with a Pulse Width Modulated (PWM) signal.
   b) Measuring the pulse duration on an input using an internal timer.
   c) Counting the number of edges on an input signal using a counter.

   Location: firmware/serial_examples/cyfxgpiocomplexapp folder.

4.2.2 USB Bulk data loopback examples

These examples illustrate a loopback mechanism between two/three USB Bulk Endpoints. The examples enumerate as Vendor Specific USB devices and are bound to the CyUsb3.sys driver based on the USB VID/PID pairs. These examples make use of a pair of BULK-OUT and BULK-IN endpoints to demonstrate how FX3 can be programmed to handle incoming and outgoing data on the USB interface. The DMA multi-channel examples use three endpoints for the loopback.

Several variants of this example are provided to demonstrate the different options that FX3 provides for handling data flows through the device. These variants follow a common design, with the only change being related to the way data transfers are handled.

Please refer to the DMA Engine section of the FX3 Programmers' Manual for a description of the various DMA flow types used in these examples.

The following are the different types of Bulk data loopback examples provided:

1. USBBulkLoopAuto: This example makes use of the DMA AUTO Channel for the loopback between the endpoints. This shows how FX3 can be programmed to handle data flows with zero firmware intervention in the actual data transfer.

   Location: firmware/dma_examples/cyfxbulklpauto

2. USBBulkLoopAutoSignal: This example makes use of the DMA AUTO Channel with Signaling for the loopback between the endpoints. As in the above case, the data transfer happens with zero firmware intervention; but the firmware is notified about the progress of data transfer.

   Location: firmware/dma_examples/cyfxbulklpautosig

3. USBBulkLoopManual: This example makes use of the DMA MANUAL Channel for the loopback between the endpoints. In this case, each data packet is individually received and forwarded by the firmware application.

   Location: firmware/dma_examples/cyfxbulklpmanual

4. USBBulkLoopManualInOut: This example makes use of the DMA MANUAL IN + DMA MANUAL OUT Channel for the loopback between the endpoints. In this case, the data has to be explicitly copied from the OUT endpoint to the IN endpoint by the firmware.

   Location: firmware/dma_examples/cyfxbulklpmaninout

5. USBBulkLoopAutoManyToOne: This example makes use of the Multi-channel DMA AUTO MANY TO ONE for the loopback between endpoints. This example makes use of two OUT endpoints and commits the received data in an interleaved fashion onto a single IN endpoint. The interleaving of data is done automatically by the FX3 hardware without any firmware intervention.

   Location: firmware/dma_examples/cyfxbulklpautomanytoone

6. USBBulkLoopAutoOneToMany: This example makes use of the Multi-channel DMA AUTO ONE TO MANY for the loopback between endpoints. This example has a single OUT endpoint and the data packets coming in are split across a pair of IN endpoints. The data splitting and forwarding is done automatically by the FX3 hardware without any firmware intervention.

   Location: firmware/dma_examples/cyfxbulklpautoonetomany

7. USBBulkLoopManualOneToMany: This example makes use of the Multi-channel DMA MANUAL ONE TO MANY for the loopback between endpoints. This is similar to the USBBulkLoopAutoManyToOne example, with the packet-by-packet data transfer being handled by firmware.

   Location: firmware/dma_examples/cyfxbulklpmanmanytoone

8. USBBulkLoopManualManyToOne: This example makes use of the Multi-channel DMA MANUAL MANY TO ONE for the loopback between endpoints. This is similar to the USBBulkLoopAutoOneToMany example, with the packet-by-packet data transfer being handled by firmware.

   Location: firmware/dma_examples/cyfxbulklpmanonetomany

9. USBBulkLoopMulticast: This example makes use of the Multi-channel DMA MULTICAST for the loopback between endpoints. In this case, data coming in on an OUT endpoint is sent out on two different IN endpoints. MULTICAST channels only support a manual mode of operation, where the forwarding of each packet has to be done by firmware.

   Location: firmware/dma_examples/cyfxbulklpmulticast

10. USBBulkLoopManualAdd: This example demonstrates the use of DMA MANUAL channels to modify data flowing through FX3 by adding fixed sized headers and footers. The headers and footers can be added with minimal processing overheads, as the bulk of the data does not need to be copied in memory.

    Location: firmware/dma_examples/cyfxbulklpman_addition

11. USBBulkLoopManualRem: 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 on the OUT endpoint is looped back to the IN endpoint after removing the header and footer. The removal of header and footer does not require the copy of data.

    Location: firmware/dma_examples/cyfxbulklpman_removal

12. USBBulkLoopLowLevel: 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 bulk loop-back example.

    Location: firmware/dma_examples/cyfxbulklplowlevel

4.2.3 USB Isochronous data loopback examples

These examples illustrate a loopback mechanism between two USB Isochronous Endpoints. These are similar to the Bulk loopback examples except that Isochronous endpoints are used.

Following are the different types of Isochronous data loopback examples provided.

1. USBIsochLoopAuto: This example makes use of an DMA AUTO channel for the loopback between the endpoints.

   Location: firmware/basic_examples/cyfxisolpauto

2. USBIsochLoopManualInOut: This example makes use of a pair of DMA MANUAL IN and DMA MANUAL OUT channels for the loopback between the endpoints.

   Location: firmware/basic_examples/cyfxisolpmaninout

4.2.4 USB debug example

This is a special USB example that demonstrates how a vendor specific USB interface can be used to obtain debug data from a FX3 device.

1. USBDebug: This example demonstrates the use of an USB interrupt endpoint to log the debug data from the FX3 device. The default debug logging in all other

examples is done through the UART port. This example shows how the debug data can be pushed out through any outgoing interface.

Location: firmware/basic_examples/cyfxusbdebug

4.2.5 FX3S Storage Examples

The following are examples that demonstrate the usage of FX3 APIs to access SD/eMMC storage devices connected to the FX3S device. These examples will only work on the FX3S, Benicia or Bay controllers, and fail to start up properly if attempted on the other device types.

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.

   Location: firmware/msc_examples/cyfx3s_msc
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.

   Location: firmware/gpif_examples/cyfxgpiftostorage
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.

   Location: firmware/storage_examples/cyfx3s_fatfs
4. FX3SSdioUart: This example implements a USB Virtual COM port that uses an 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.

   Location: firmware/storage_examples/cyfx3s_sdiouart
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.

   Location: firmware/msc_examples/cyfx3s_raid0

4.2.6 USB Video Class example

Video streaming is the most common application for devices in the FX3 family. The USB Video Class (UVC) specification from USB-IF defines a standard for streaming video from sensors to a USB host. The UVC specification requires the corresponding device firmware to support a number of class specific requests, as well as to provide the data in a specific format.

The SDK provides a pair of examples that demonstrate how the UVC data packaging and class specific request handling can be performed. These examples

do not make use of an external image sensor, but repeatedly send out pre-defined image data which is stored as part of the firmware image.

1. USBVideoClass: This example implements a USB Video Class “Camera” device which streams MJPEG video data using an ISOCHRONOUS IN endpoint. The FX3 device enumerates as a USB Video Class device on the USB host and makes use of a DMA Manual Out channel to send the data. The video frames are stored in FX3 device memory and sent to the host in a cyclical fashion. The streaming of these frames continuously from the device results in a video like appearance on the USB host.

   This is a low throughput application because the UVC class drivers on Windows 7 machines do not support Burst enabled ISOCHRONOUS endpoints. This means that a maximum of 3 KB can be transferred from the device in a 125 us micro-frame (Service interval).

   Location: firmware/uvc_examples/cyfxuvcinmem
2. USBVideoClassBulk: This example is similar to the USBVideoClass example, but makes use of BULK endpoints to stream the video data. As Burst enabled BULK endpoints are supported on Windows 7, this example can achieve much higher data rates that the ISOCHRONOUS example.

   Location: firmware/uvc_examples/cyfxuvcinmem_bulk

4.2.7 Slave FIFO Application examples

The Slave FIFO application examples demonstrate data loopback between the USB Host and an external FPGA/Controller. These examples make use of an FPGA that connects to the GPIF-II port of the FX3 device, and performs data transfers using the Cypress defined Slave FIFO protocol (Synchronous or Asynchronous version).

Please refer to the AN65974 - Designing with the EZ-USB® FX3™ Slave FIFO Interface application note for details about how to implement the Slave FIFO interface.

1. SlaveFifoAsync: Asynchronous mode Slave FIFO example using 2 bit FIFO address. As a 2-bit address is used, a maximum of 4 pipes on FX3 can be accessed by the FIFO master.

   Location: firmware/slavefifo_examples/slfifoasync
2. SlaveFifoAsync5Bit: Asynchronous mode Slave FIFO example using 5 bit FIFO address. In this case, additional signals are used on the Slave FIFO interface to allow the master to address a maximum of 32 different pipes on the FX3 side.

   Location: firmware/slavefifo_examples/slfifoasync5bit
3. SlaveFifoSync: Synchronous mode Slave FIFO example using 2 bit FIFO address. As a 2-bit address is used, a maximum of 4 pipes on FX3 can be accessed by the FIFO master.

   Location: firmware/slavefifo_examples/slfifosync

4. SlaveFifoSync5Bit: Synchronous mode Slave FIFO example using 5 bit FIFO address. In this case, additional signals are used on the Slave FIFO interface to allow the master to address a maximum of 32 different pipes on the FX3 side.

   Location: firmware/slavefifo_examples/slfifosync5bit

4.2.8 Serial Interface examples

The serial interface examples demonstrate data accesses to the Serial IO interfaces: I2C, I2S, SPI and UART.

4.2.8.1 UART examples

The UART block on FX3 can be configured for discrete data transfers using a register interface to the transfer FIFOs, or for block based data transfers using a DMA connection to the transfer FIFOs. These examples demonstrate the two different modes of operation for the UART block on the FX3 device.

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.

1. UartLpDmaMode: This example makes use of a MANUAL DMA channel to loop any data received through the UART receiver back to the UART transmitter.

   Location: firmware/serial_examples/cyfxuartlpdmamode

2. UartLpRegMode: This example implements the firmware logic to receive data from the UART receiver block and to loop it back to the UART transmitter side. Hardware interrupts are used detect incoming data on the UART side, and then a firmware task is enabled to take the received data and send it out.

   Location: firmware/serial_examples/cyfxuartlpregmode

3. UsbUart: This example implements a CDC-ACM compliant USB to UART bridge device using the UART port on the FX3 device.

   Location: firmware/serial_examples/cyfxusbuart

4.2.8.2 I2C examples

The I2C block on the FX3 device is a master-only interface. The I2C examples demonstrate how the I2C interface can be used to transfer data from/to an I2C EEPROM device in the two modes of operation: namely, register mode and DMA mode. The examples enumerate as a vendor specific USB device bound to the CyUsb3 driver, and implement a set of vendor specific control (EP0) requests to perform transfers from/to the I2C EEPROM device.

1. UsbI2cDmaMode: This example shows how to use the I2C block in DMA mode to read/write data from/to the I2C EEPROM.

   Location: firmware/serial_examples/cyfxusbi2cdmamode

2. UsbI2cRegMode: This example shows how to use the I2C block in register (firmware programmed) mode to read/write data from/to the I2C EEPROM.

   Location: firmware/serial_examples/cyfxusbi2cregmode

4.2.8.3 SPI examples

As in the I2C case, FX3 supports a master-only SPI interface block. The SPI examples in the SDK demonstrate how the FX3 can be used to read/write data on a SPI flash memory device. The examples enumerate as a vendor specific USB device and provide a set of vendor commands to read, write and erase the SPI flash memory blocks.

1. UsbSpiDmaMode: This example uses DMA channels to read/write data on the SPI flash memory.

   Location: firmware/serial_examples/cyfxusbspidmamode
2. UsbSpiRegMode: This example uses the FX3 SPI block's register interface to read/write data on the SPI flash memory.

   Location: firmware/serial_examples/cyfxusbspiregmode
3. UsbSpiGpioMode: The SPI block on the FX3 device is not available for use when the GPIF-II interface is used with a 32-bit wide data bus. In such a case, a lower performance SPI interface can be implemented by bit-banging on the FX3 device IOs. This example shows how a set of 4 GPIOs on the FX3 device can be used as the SPI Clock, MOSI, MISO and SSN# pins.

   Location: firmware/serial_examples/cyfxusbspigpiomode

4.2.8.4 I2S example

The FX3 provides a master-only I2C interface which can be used for audio output. The SDK provides a single I2C usage example which shows how the I2C interface can be used to send data out.

1. UsbI2sDmaMode: This example demonstrates the use of I2S APIs. The example enumerates as a vendor specific USB device and receives the data from the left and right I2C channels through two different BULK OUT endpoints. AUTO DMA channels are used to forward the received data to the I2S device from the FX3.

   Location: firmware/serial_examples/cyfxusbi2sdmamode

4.2.9 USB Bulk/Isochronous data source sink examples

These examples illustrate independent control of USB OUT and IN data paths using a pair of SOURCE and SINK endpoints. The examples enumerate as vendor specific USB devices and allow the USB host to perform data transfers to the available endpoints.

The examples are configured to serve as perfect data sinks that just keep dropping all received data and as perfect data sources that keep pushing full data packets on the IN endpoint.

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

   Location: firmware/basic_examples/cyfxbulksrcsink

2. USBIsoSourceSink: This is similar to the USBBulkSourceSink example, but makes use of ISOCHRONOUS endpoints. Multiple alternate interface settings are provided in this application, to control the data rates.

   Location: firmware/basic_examples/cyfxisosrcsink

3. USBIsoSource: 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.

   Location: firmware/basic_examples/cyfxisosrc

4. GpifToUsb: 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 is no firmware bottleneck involved. As the example configures the GPIF-II with a 32-bit wide bus, this cannot work with parts such as FX3S and CX3 which do not support 32-bit wide GPIF-II data.

   Location: firmware/basic_examples/cyfxgpiftousb

4.2.10 USB enumeration example

In most applications, the device's USB enumeration is handled by the Cypress provided firmware drivers using descriptors provided by the application. The firmware framework also allows user to handle the device enumeration by directly responding to USB control requests. This mode of operation is demonstrated using a specific firmware example.

1. USBBulkLoopAutoEnum: 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 using an AUTO DMA channel.

   Location: firmware/basic_examples/cyfxbulklpautoenum

4.2.11 Flash Programmer example

The FX3 device supports boot modes where it boots itself by reading firmware binaries from I2C based EEPROM devices or SPI based flash devices.

In order to have the FX3 boot itself through I2C/SPI, the firmware binaries generated by compiling the target application needs to be programmed on the memory devices. This programming can be achieved using the Flash Programmer firmware example. This example enumerates as a vendor specific USB device with no endpoints, and provides a set of vendor commands through the I2C EEPROM and SPI flash devices can be programmed.

1. USBFlashProg: This example 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. The binary produced by compiling this application is used by

the Cypress Control Center tool to support program to I2C EEPROM and program to SPI Flash operations.

Location: firmware/basic_examples/cyfxflashprog

4.2.12 Mass Storage Class example

1. USBMassStorageDemo: This example 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 a command/response protocol such as the USB Mass Storage Protocol can be implemented in FX3 applications.

   Location: firmware/msc_examples/cyfxmscdemo

4.2.13 USB Audio Class Example

1. USBAudioClass: 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.

   Location: firmware/uac_examples/cyfxuac

4.2.14 USB host and OTG examples

These examples demonstrate the host mode and OTG mode operation of the FX3 USB port. These examples can only work on FX3 device variants that support the OTG host mode.

1. USBHost: Demonstrates how FX3 can be used as USB 2.0 host controller. The example detects the device connection, identifies the device type and performs data transfers if the device connected is either a USB mouse (HID) or a Mass Storage Class (Thumb drive) device. Full-fledged class drivers for mouse and mass storage are not provided, and only limited command support is provided.

   Location: firmware/basic_examples/cyfxusbhost

2. USBOtg: Demonstrates FX3 functioning 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 bulk loop-back using a 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.

   Location: firmware/basic_examples/cyfxusbotg

3. USBBulkLoopOtg: This example demonstrates the full OTG capabilities such as Session Request Protocol (SRP) for VBus control and Host Negotiation Protocol (HNP) for role change negotiation. This example requires a pair of FX3 devices to be connected to each other using an OTG cable. Any one of the devices can function as a bulk loop-back device, and the other device will function as the host.

   Location: firmware/basic_examples/cyfxbulklpotg

4.2.15 CX3 Examples

These examples demonstrate the implementation of various USB video streaming modes using the CX3 device and MIPI CSI-2 compliant image sensors. These examples will fail to start up properly if run on devices other than CX3.

1. Cx3Rgb16AS0260: 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.

   Location: firmware/cx3_examples/cycx3_rgb16_as0260

2. Cx3Rgb24AS0260: 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.

   Location: firmware/cx3_examples/cycx3_rgb24_as0260

3. Cx3UvcAS0260: 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.

   Location: firmware/cx3_examples/cycx3_uvc_as0260

4. Cx3UvcOV5640: 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.

   Location: firmware/cx3_examples/cycx3_uvv_ov5640

4.2.16 FX2G2 Firmware Example

This is a dedicated firmware example for the USB 2.0 only FX2G2 device.

1. Fx2g2UvcDemo: This example demonstrates YUY2 uncompressed video streaming under USB 2.0 using the FX2G2 device. The example generates the video data patterns within the device memory instead of sourcing from an image sensor, so that the example can work directly on the development kits. The example supports UVC streaming over Bulk and Isochronous endpoints.

   Location: firmware/fx2g2_examples/cyfx2_uvcdemo

4.2.17 Co-processor Mode Example

The FX3 device supports a MMC Slave (PMMC) interface which can be used instead of the GPIF-II interface to connect it to an external processor. In such a

configuration, FX3 can serve as a co-processor that provides peripheral functions such as USB, SD/eMMC, UART, I2C, SPI etc. to the processor.

1. PibSlaveDemo: This example implements a slave mode firmware application which allows a master processor to control its operation through a MMC slave interface. The firmware allows the master to access SD/eMMC storage devices, configure the USB device characteristics and to connect the SD/eMMC storage to the USB host. As the example supports SD/eMMC interfaces, it will only work with FX3S devices.

   Location: firmware/storage_examples/cyfx3_pmmc

4.2.18 GPIF-II Master Examples

The Slave FIFO and other examples above make use of the GPIF-II interface in slave mode. The GPIF-II block on FX3 is also capable of operating as the bus master.

1. SRAMMaster: This example demonstrates how the FX3 device can be used as a master accessing an Asynchronous SRAM device. The CY7C1062DV33-10BGXI SRAM device on the CYUSB3KIT-003 is used for this demonstration. Only 1 KB of the SRAM can be addressed by FX3 on this board, and the example provides a means to repeatedly write/read the content of this memory segment using a pair of USB bulk endpoints.

   Location: firmware/gpif_examples/cyfxsrammaster

4.2.19 Two Stage Booter Examples

The full featured FX3 firmware libraries are based on the ThreadX RTOS. This provides a very powerful and flexible framework for firmware development. However, the embedded RTOS and associated code means that the memory requirement for applications is greater than 70 KB. This can be seen using the simple GPIO example applications.

In rare cases, such a large footprint may not be desirable; and the user is willing to make use of a simpler framework. The FX3 SDK facilitates such a usage model using a separate firmware library which is called the “Boot Firmware Library”.

This name is used because this library is mainly used to create custom boot-loader applications. However, this is a misnomer because the library supports almost all device features; and can be used for building full fledged applications as well.

A set of applications that demonstrate the capabilities of this firmware library are also provided with the SDK.

1. BootLedBlink: This is the simplest possible FX3 firmware application. It demonstrates how to control an FX3 output pin using an input pin. On the SuperSpeed Explorer board, this allows an LED to be controlled using an on-board DIP switch.

   Location: firmware/boot_fw/ledblink

2. FX3BootAppGcc: This example shows how the boot APIs can be used to implement a custom boot-loader application. The boot modes supported include SPI boot and USB boot. When USB boot is selected, this application supports loading and running a full firmware application; without going through a USB device re-enumeration.

   Location: firmware/boot_fw/src

3. BootGpifDemo: This example shows how the boot APIs can be used to implement a GPIF-II to USB data path. As the boot library does not support DMA channel level operation, this application shows how to create and manage AUTO and MANUAL DMA channels using low level constructs. The scope of the application is similar to the cyfxgpiftousb application, and the example requires the use of devices that support 32-bit wide GPIF-II data.

   Location: firmware/boot_fw/gpiftousb

5 FX3 Programming Guidelines

This section provides a few basic guidelines for developing applications using the FX3 SDK. These sections refer to APIs provided by the FX3 firmware libraries. Please refer to the FX3 API Guide (FX3APIGuide.pdf or FX3APIGuide.chm) document for a description of these APIs, their parameters and return types.

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 half of the CPU frequency or 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 and FX3 is clocked

using a 19.2 MHz crystal, 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 should be 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.

The storage ports are supported only on the FX3S and SD3 devices, and the s0Mode and s1Mode parameters are ignored when configuring other devices. Similarly, the isDQ32Bit parameter is ignored when configuring devices that do not support 32-bit GPIF data operation.

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. If the isDmaHandleDCache parameter passed to the CyU3PDeviceCacheControl() API is set to CyTrue, the DMA APIs take care of these cache operations.

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 Operating Systems with similar feature sets.

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

Two additional build configurations called as fx3_profile_debug and fx3_profile_release are provided in this version of the SDK. These configurations enable call profiling support within the ThreadX RTOS. They also allow the user to

map in FX3 GPIOs to reflect activity on various RTOS objects like threads, mutexes, semaphores and event flag groups.

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. Setting the timeSlice parameter passed to the CyU3PThreadCreate() API to CYU3P_NO_TIME_SLICE, selects co-operative scheduling. Further, each thread implementation needs to ensure that it does not keep the CPU locked for periods longer than a milli-second.

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 Driver	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
Processor Interface Block (PIB) Driver	PIB/GPIF driver that handles GPIF and MMC Slave related interrupts.	4	1 KB
Low Performance Peripherals Block (LPP) Driver	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 Driver	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 Handler	Debug log handler. Takes care of sending CyU3PDebugPrint and CyU3PDebugLog messages out through the selected debug port.	6	512 bytes
Storage Interface Block (SIB) Driver	Storage driver that handles SD/MMC interface related interrupts and processes device hotplug events. This thread is always idle if the storage module is not initialized.	4	2 KB

Table 5-1: List of threads created by FX3 firmware framework

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 when events of interest occur.

Callback Function	Registration API	Caller Context
GPIO interrupt	CyU3PGpioInit, CyU3PRegisterGpioCallBack	Interrupt handler
GPIF state machine interrupt	CyU3PGpifRegisterSMIntrCallback	Interrupt handler
OS timer callback	CyU3PTimerCreate	OS timer thread
GPIF event	CyU3PGpifRegisterCallback	PIB driver thread
PIB interrupt	CyU3PPibRegisterCallback	PIB driver thread
PMMC interrupt	CyU3PPmmcRegisterCallback	PIB driver thread
Mailbox interrupt	CyU3PMboxInit	PIB driver thread
I2C interrupt	CyU3PI2cSetConfig	LPP driver thread
I2S interrupt	CyU3PI2sSetConfig	LPP driver thread
SPI interrupt	CyU3PSpiSetConfig	LPP driver thread
UART interrupt	CyU3PuartSetConfig	LPP driver thread
DMA callback	CyU3PDmaChannelCreate CyU3PDmaMultiChannelCreate	DMA driver thread

USB event	CyU3PusbRegisterEventCallback	USB driver thread (SOF/ITP event is sent from interrupt handler)
USB control request	CyU3PUsbRegisterSetupCallback	USB driver thread
USB power mode change	CyU3PUsbRegisterLPMRequestCallback	USB driver thread
USB endpoint interrupt	CyU3PUsbRegisterEpEvtCallback	USB driver thread
USB host mode interrupt	CyU3PUsbHostStart	USB driver thread

Table 5-2: List of callbacks provided by FX3 firmware framework

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 non-zero initialized data variables in the FX3 application. The space required for this region is application specifix.
- BSS: This region contains all uninitialized and 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. Figure 5-1 shows the default memory map used by the firmware examples in the FX3 SDK.

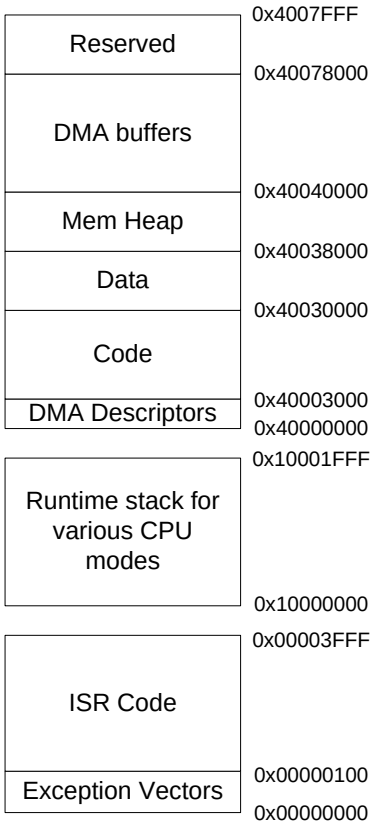


Figure 5-1: Default memory map used by FX3 firmware examples

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 USB SET_CONFIGURATION control 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. Doing this simplifies the application code and allows the configuration to be completed faster.

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.

The CyU3PUsbEpSetPacketsPerBuffer() API the number of packets that can be stored in one OUT endpoint buffer for isochronous endpoints. This API can be used in USB 2.0 mode when using isochronous endpoints with odd Max Packet Size settings.

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. Four power modes (U0, U1, U2 and U3) are defined for USB 3.0 links and two power modes (L0 and L1) are defined for USB 2.0 links. Data transfers can only be performed while the link is the U0 or L0 mode, and transitions between the power modes can be initiated by either host or device.

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 to ensure that the link does not get stuck in a low power mode.

It is recommended to disable the low power mode transitions by calling CyU3PUsbLPMDisable() API when the data transfers are active because there may be cases where the host performs very fast U1 Entry/Exit (Entry to Exit duration < 5 µs) and the firmware is not capable of responding to such fast low power mode transitions.

When the data flow pattern in the target FX3 application is known, it is possible to handle LPM states in a more flexible manner. The following sub-section describe how to handle LPM states in common USB applications.

5.4.4.1 LPM Handling in UVC Video Applications

In the case of a UVC video streaming solution, we need to ensure that low power modes do not delay the actual streaming of data. At the same time, we can use the low power modes to reduce power consumption in between video frames.

The above scheme can be implemented as follows:

1. Disable LPM entry and change link state to U0 when the first data packet in a frame has been received by FX3 (Use CyU3PusbSetLinkPowerState and CyU3PUsbLPMDisable).
2. Re-enable LPM entry (CyU3PUsbLPMEnable) once the consume event for the last packet in a video frame has been received.

This solution is implemented in all of the CX3 examples in the FX3 SDK. The same methods are applicable in UVC applications using the FX3 part with a parallel image sensor as well.

5.4.4.2 LPM Handling in Command-Response based Applications

Many USB applications such as USB Mass Storage follow a Command-Response protocol. In such cases, FX3 firmware can disable LPM entry while handling a command; and re-enable LPM entry once the active command handling is completed.

This solution is implemented in the FX3SmassStorage firmware example, and can be adapted to any application that follows a Command – Response protocol.

5.4.4.3 *USB 2.1 LPM-L1 Handling*

All SuperSpeed capable USB devices need to implement the LPM-L1 power mode required by an ECN to the USB 2.0 specification.

The L1 power mode allows a USB 2.0 link to move into between an active and inactive state with substantially lower latencies than a regular USB suspend / resume sequence.

LPM-L1 entry can only be initiated by the USB host. While the specification allows USB devices to accept or reject LPM-L1 entry requests, the FX3 device is only capable of accepting these requests. Therefore, the CyU3PUsbLPMDisable API handles LPM-L1 transitions by first accepting the LPM-L1 request and then initiating a remote wake sequence.

The CyU3PUsbLPMDisable() API can be used to handle LPM-L1 mode in the same way as it is used to handle U1/U2 modes in a USB 3.0 connection.

The library also provides an API called CyU3PUsb2Resume() which can be used to explicitly initiate remote wakeup when LPM is not disabled.

5.5 Support for different FX3 parts

The EZ-USB FX3 product family includes multiple parts as shown in **Table 1-1**.

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. Similarly, the MIPI CSI APIs for the CX3 device are packaged as a separate library (cyu3mipicsi.a).

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. This can be done by adjusting the buffer size and count parameters passed to the various DMA channel create calls.

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 Firmware Application from one SDK release to another

While new releases of the FX3 SDK provide new features and defect fixes, we try not to make API interface changes that require changes in the firmware application source code. In general, porting a firmware application to a newer SDK version is simple as installing the new SDK and then doing a clean + build of the Eclipse project.

However, some of the defect fixes or features make small interface changes necessary. In such cases, the user applications or their build settings need to be updated. The following sub-sections provide a list of changes to be made when porting a firmware application from one SDK version to the next. The changes need to be applied cumulatively when moving across multiple SDK versions.

5.6.1 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.

5.6.2 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.

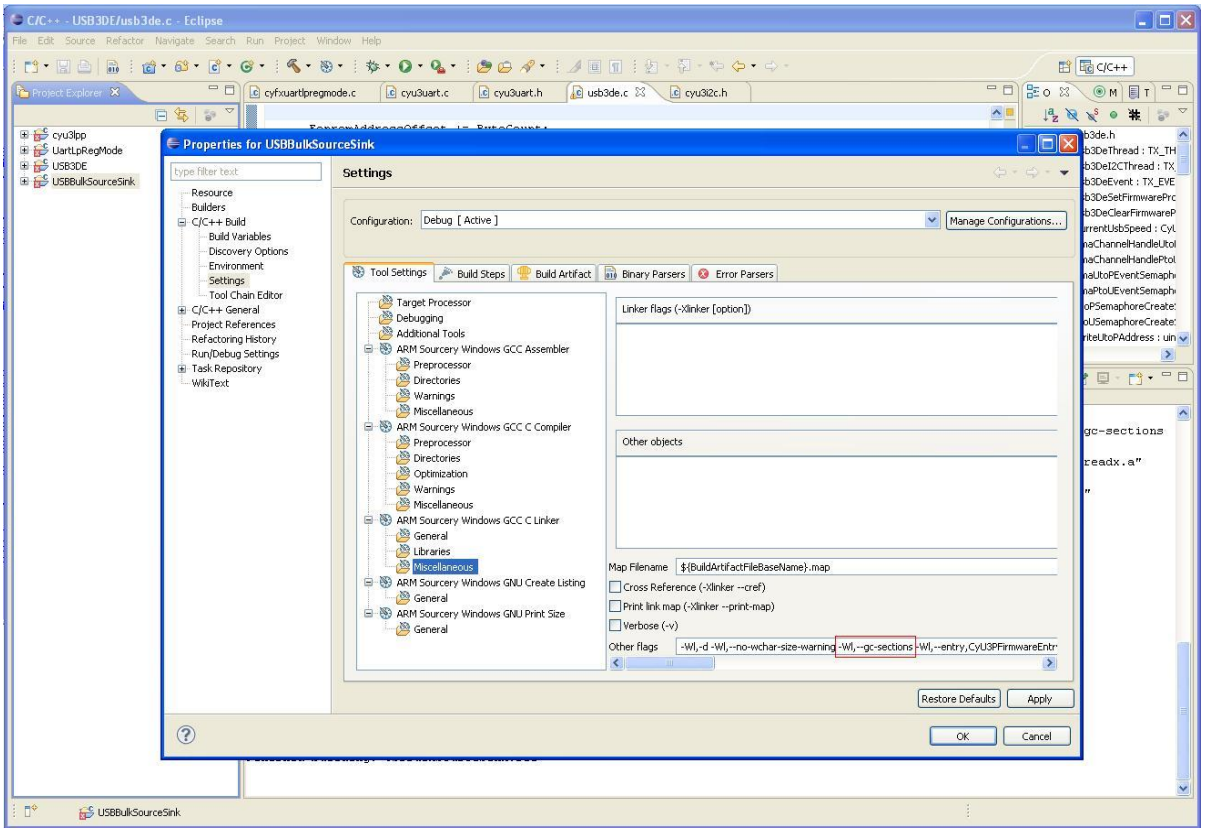


Figure 5-2: Eclipse Linker Flag Settings

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.

5.6.3 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.

5.6.4 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 will 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.

5.6.5 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.

5.6.6 Porting Applications from SDK 1.3.1 to SDK 1.3.3

The following aspects need to be taken care of when porting FX3 applications built on SDK 1.3.1 to SDK 1.3.3.

1. The firmware library now implements an internal work-around to prevent the USB data corruption due to concurrent BULK-IN and Control-IN transfers when operating at Hi-Speed. The BULK DMA channels are now suspended and resumed automatically from the CyU3PUsbSendEp0Data() function. Any code to do the same operations needs to be removed from the user application code.
2. The firmware library now implements a work-around to prevent USB data corruption caused due to intermittent protocol level CRC errors and retries on a USB 3.0 link. Part of the work-around makes use of a complete Endpoint Memory reset which can cause loss of data on an endpoint that has in-flight transfers.

   The firmware library provides an endpoint specific event notification (see CyU3PUsbRegisterEpEvtCallback and CYU3P_USBEP_SS_RESET_EVT) to notify the application when this happens. It is recommended that the application registers for this event, and uses the notification to stop and re-start data transfers on the endpoint. The easy way of doing this is to STALL the endpoint and then let the DMA channel be restarted when the CLEAR_FEATURE(EP_HALT) request is received from the host.
3. The memory allocation functions in the cyfxtx.c file have been enhanced with new features such as memory leak and corruption detection. These changes are optional and there is no strong requirement to use the changes. However, it is suggested that the user replaces the cyfxtx.c file in their application with the corresponding file from the firmware/common folder.
4. The firmware libraries and headers have been moved to a different directory hierarchy under <INSTALL_ROOT>/fw_lib and <INSTALL_ROOT>/boot_lib. This change is done so as to separate the libraries from the example sources and also in order to provide access to libraries from previous SDK versions. The original directory structure is still retained in the SDK to maintain compatibility with existing user projects. However, it is recommended that all projects be updated to use the new project templates used in the SDK examples.
5. The linker script used for building applications based on the boot firmware library has been moved from the firmware/boot_fw/src folder to the firmware/boot_fw/build folder. Corresponding changes will need to be made to the build settings of projects that use this library.

5.6.7 Porting Applications from SDK 1.3.3 to SDK 1.3.4

The following aspects need to be taken care of when porting FX3 applications built on SDK 1.3.3 to SDK 1.3.4.

1. A new API, CyU3PPibDllConfigure() has been added to the firmware library to configure the PIB (GPIF-II) DLL when the FX3 is acting as a GPIF Master. Any code to do the same operations can be replaced with this API in the user application code
2. The CX3 GPIF state machine has been enhanced in all the cx3 code examples in the SDK. This enhancement requires to use the CyU3PGpifControlSWInput() API in place of the CyU3PGpifSMSwitch() API in the DMA_CB_CONS_EVENT. Please refer to the cx3 examples in the SDK for more details on implementation.
3. New USB events have been added to the CyU3PUsbEventType_t structure in the cyu3usb.h header file. The USB event callback function may need to be updated to not execute any default actions when these events are received.
4. A new API, CyU3PSibEraseBlocks() has been added to the firmware library to support erase operation in boot partitions of eMMC devices. Low level SD protocol level APIs, if used in the application firmware for erasing the boot partitions can be replaced with this new API.
5. CyU3PSibVendorAccess() API has been enhanced to provide the response received from the SD card without any data-padding. Necessary changes in the user application code should be done.
6. A new API, CyU3PDmaSocketClearInterrupts() has been added to clear the interrupt on a particular DMA socket. The current SDK implementation directs to call the CyU3PDmaSocketGetConfig() API and then call the CyU3PDmaSocketSetConfig() API to clear the interrupt bit. Instead, this new API can be called which improves the low-level DMA handling.