Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

FX3 Programmers Manual

Doc. # 001-64707 Rev. *K

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
http://www.cypress.com

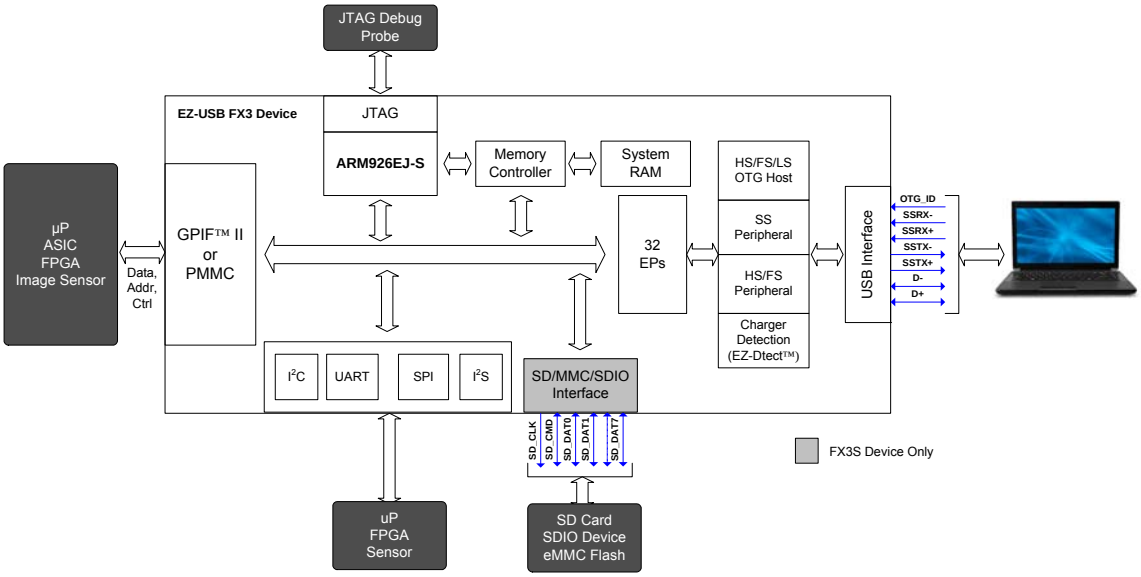
Copyrights

Contents

1. Introduction

Cypress EZ-USB® FX3™ is the next-generation USB 3.0 peripheral controller providing highly integrated and flexible features that enable developers to add USB 3.0 functionality to any system.

Figure 1-1. EZ USB FX3 System Diagram



EZ-USB FX3 has a fully configurable, parallel, general programmable interface called GPIF™ II, which can connect to any processor, ASIC, DSP, image sensor, or FPGA. It has an integrated PHY and controller along with a 32-bit microcontroller (ARM926EJ-S) for powerful data processing and for building custom applications. It has an interport DMA architecture that enables data transfers of greater than 400 MBps.

FX3 is a fully compliant USB 3.0 and USB 2.0 peripheral. An integrated USB 2.0 OTG controller enables applications that need dual role usage scenarios. It has 512 KB of on-chip SRAM for code and data. It supports serial peripherals such as UART, SPI, I2C, and I2S that enable communicating to on board peripherals; for example, the I2C interface is typically connected to an EEPROM.

GPIF II is an enhanced version of the GPIF in FX2LP™, Cypress's flagship USB 2.0 product. It provides easy and glueless connectivity to popular industry interfaces such as asynchronous and synchronous Slave FIFO, asynchronous SRAM, asynchronous and synchronous Address Data Multiplexed interface, parallel ATA, and so on. The GPIF II controller on the FX3 device supports a total of 256 states. It can be used to implement multiple disjointed state machines.

The FX3 also supports a Pseudo MultiMediaCard (PMMC) or MMC slave interface through which it can be connected to processors that support an SD or MMC memory interface. This interface uses the same pins as the GPIF-II and the user has to choose between the GPIF-II and MMC interfaces.

The FX2G2 device is a USB 2.0 controller, which supports all other features of the FX3 controller. The ARM9 core and DMA capabilities, along with the GPIF-II support, make this a high-performance USB 2.0 controller.

FX3 is fully compliant with USB 3.0 V1.0 Specification and is also backward compatible with USB 2.0. It is also complaint with the Battery Charging Specification V1.1 and USB 2.0 OTG Specification.

The FX3S device is an extension to the FX3 that supports a storage interface that can be connected to SD cards or eMMC devices. The FX3S device allows developers to add high performance persistent storage interfaces to their USB design, and supports the SD 3.0 specification and the MMC 4.41 specification.

The Benicia device is similar to the FX3S device but comes in a smaller wafer-level chip scale package (WLCSP). The Bay device is a USB 2.0 only version of the Benicia controller. The small chip footprint and high-performance flash memory support (SD/eMMC) make these devices a good fit for solutions such as mobile phones.

The SD3 device is a programmable USB 3.0 to SD/eMMC/SDIO bridge device based on the FX3 architecture. This device does not support the GPIF-II or PMMC interfaces.

The EZ-USB CX3 device is an extension of the EZ-USB FX3 device. It includes the ability to interface with and perform uncompressed video transfers from image sensors implementing the MIPI CSI-2 interface over a fixed-function GPIF interface.

The FX3 comes with the easy-to-use EZ-USB tools providing a complete solution for fast application development. The software development kit includes application examples to accelerate time to market.

The FX3 product family has multiple devices with a varied feature set. The FX3 SDK works with all of the FX3 and FX3S parts and is capable of identifying the type of device being used at runtime. Refer to Table 1-1 for details of the features supported by each of the FX3 and FX3S parts.

Table 1-1. Features Supported by FX3 and FX3S Parts

Part Number	USB 3.0 Support	Host/OTG Support	GPIF Support	SD/MMC Support	SRAM size
CYUSB3014	Yes	Yes	Up to 32 bit	No	512 KB
CYUSB3013	Yes	Yes	Up to 16 bit	No	512 KB
CYUSB3012	Yes	Yes	Up to 32 bit	No	256 KB
CYUSB3011	Yes	Yes	Up to 16 bit	No	256 KB
CYUSB3035 (FX3S)	Yes	Yes	Up to 16 bit	Yes	512 KB
CYUSB3025 (SD3)	Yes	Yes	No	Yes	512 KB
CYUSB2104 (FX2G2)	No	Yes	Up to 32 bit	No	512 KB
CYUSB3065 (CX3)	Yes	No	CSI-2 interface	No	512 KB
CYWB0263 (Benicia)	Yes	Yes	Up to 16 bit	Yes	512 KB
CYWB0163 (Bay)	No	Yes	Up to 16 bit	Yes	512 KB

1.1 Chapter Overview

The following chapters describe in greater detail each of the Programmers Manual components:

Introduction to USB on page 13 presents an overview of the USB standard.

FX3 Overview on page 25 presents a hardware overview of the FX3 system.

FX3 Software on page 53 provides an overview of the SDK that is provided with the FX3.

FX3 Firmware on page 57 provides a brief description of each programmable firmware block. This includes the system boot and initialization, USB, GPIF 2, serial interfaces, DMA, power management, and debug infrastructure.

FX3 APIs on page 83 provides the description of the APIs for USB, GPIF2, serial interfaces, DMA, RTOS, and debug.

FX3 Application Examples on page 85 presents code examples, which illustrate the API usage and the firmware framework.

FX3 Firmware Application Structure on page 99 describes the FX3 application framework and usage model for FX3 APIs.

FX3 Serial Peripheral Register Access chapter on page 113 describes the register based access from an application processor when FX3 device is configured for PP mode slave operation.

FX3 Boot Image Format chapter on page 129 describes the FX3 image (img) format as required by the FX3 boot-loader.

FX3 Development Tools on page 133 describes the available options for the firmware development environment, including JTAG based debugging.

FX3 Host Software on page 135 describes the Cypress generic USB 3.0 WDF driver, the convenience APIs, and the USB control center.

GPIF™ II Designer on page 137 provides a guide to the GPIF II Designer tool.

1.2 Document Revision History

Table 1-2. Revision History

Revision	PDF Creation Date	Origin of Change	Description of Change
**	05/10/2011	SHRS	New user guide
*A	07/14/2011	SHRS	FX3 Programmers Manual update for beta release.
*B	03/27/2012	SHRS	FX3 Programmers Manual update for FX3 SDK 1.1 release.
*C	08/10/2012	SHRS	FX3 Programmers Manual update for SDK 1.2 release.
*D	02/05/2013	KYS	Added sections Segger J-Link on page 156, and on page 133 Updated on page 133
*E	05/24/2013	KYS	Extensive updates throughout the document.
*F	09/13/2013	KYS	FX3 Programmers Manual update for SDK 1.3 release
*G	11/21/2013	VOM	Updates for SDK 1.3.1 release
*H	12/23/2014	KYS	Updates for SDK 1.3.3 release.
*I	06/23/2015	KYS	No technical updates. Completing Sunset Review.
*J	03/08/2016	SUKU	Updated for SDK 1.3.4 release.
*K	05/22/2018	KYS	Updated the template.

1.3 Documentation Conventions

Table 1-3. Document Conventions for Guides

Convention	Usage
`Courier New`	Displays file locations, user entered text, and source code: `C:\ ...cd\icc\`
Italics	Displays file names and reference documentation: Read about the *sourcefile.hex* file in the *PSoC Designer User Guide*.
[**Bracketed, Bold**]	Displays keyboard commands in procedures: [**Enter**] or [**Ctrl**] [**C**]
File > Open	Represents menu paths: File > Open > New Project
Bold	Displays commands, menu paths, and icon names in procedures: Click the **File** icon and then click **Open**.
Times New Roman	Displays an equation: 2 + 2 = 4
Text in gray boxes	Describes Cautions or unique functionality of the product.

2. Introduction to USB

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. The USB is the most successful PC peripheral interconnect ever. In 2006 alone, over 2 billion USB devices were shipped and there are over 6 billion USB products in the installed base today.

2.1 USB 2.0 System Basics

A USB system is an asynchronous serial communication 'host-centric' design, consisting of a single host and a myriad of devices and downstream hubs connected in a tiered-star topology. The USB 2.0 Specification supports the low-speed, full-speed, and high-speed data rates. It employs a half-duplex two-wire signaling featuring unidirectional data flow with negotiated directional bus transitions.

2.1.1 Host, Devices, and Hubs

The USB system has one master: the host computer. Devices implement specific functions and transfer data to and from the host (for example: mouse, keyboard, and thumb drives). The host owns the bus and is responsible for detecting a device as well as initiating and managing transfers between various devices. Hubs are devices that have one upstream port and multiple down stream ports and connect multiple devices to the host creating a tiered topology. Associated with a host is the host controller that manages the communication between the host and various devices. Every host controller has a root hub associated with it. A maximum of 127 devices may be connected to a host controller with not more than seven tiers (including root hubs). Because the host is always the bus master, the USB direction OUT refers to the direction from the host to the device, and IN refers to the device to host direction.

2.1.2 Signaling Rates

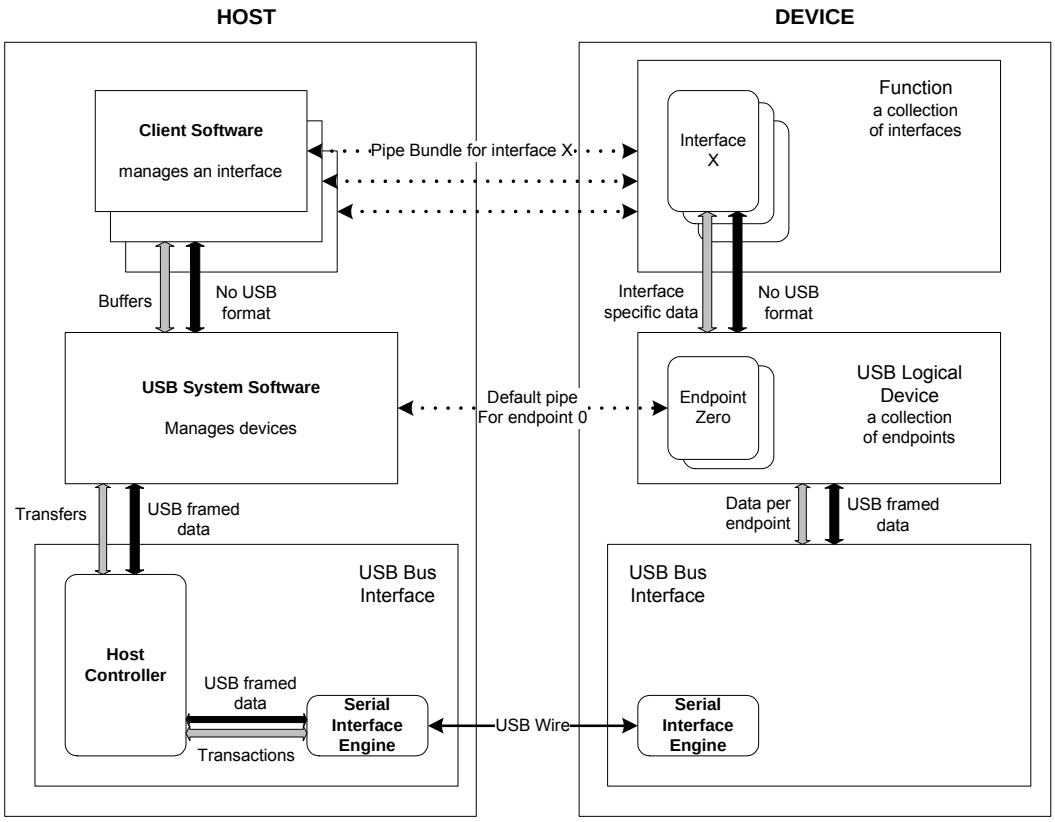
USB 2.0 supports following signaling rates:

- A low-speed rate of 1.5 Mbit/s is defined by USB 1.0.
- A full-speed rate of 12 Mbit/s is the basic USB data rate defined by USB 1.1. All USB hubs support full speed.
- A high-speed (USB 2.0) rate of 480 Mbit/s introduced in 2001. All high-speed devices are capable of falling back to full-speed operation if necessary; they are backward compatible.

2.1.3 Layers of Communication Flow

A layered communication model view is adopted to describe the USB system because of its complexity and generic nature. The components that make up the layers are presented here.

Figure 2-1. USB Layers from USB



2.1.3.1 *Pipes, Endpoints*

USB data transfer can occur between the host software and a logical entity on the device called an endpoint through a logical channel called pipe. A USB device can have up to 32 active pipes, 16 for data transfers to the host, and 16 from it. An interface is a collection of endpoints working together to implement a specific function.

2.1.3.2 *Descriptors*

USB devices describe themselves to the host using a chain of information (bytes) known as descriptors. Descriptors contain information such as the function the device implements, the manufacturer of the device, number of endpoints, and class specific information. The first two bytes of any descriptor specify the length and type of descriptor respectively.

All devices generally have the following descriptors.

- Device descriptors
- Configuration descriptors
- Interface descriptors
- Endpoint descriptors
- String descriptors

A device descriptor specifies the Product ID (PID) and Vendor ID (VID) of the device as well as the USB revision that the device complies with. Among other information listed are the number of configurations and the maximum packet size for endpoint 0. The host system loads looks at the VID

and PID to load the appropriate device drivers. A USB device can have only one device descriptor associated with it.

The configuration descriptor contains information such as the device's remote wake up feature, number of interfaces that can exist for the configuration, and the maximum power a particular configuration uses. Only one configuration of a device can be active at any time.

Each function of the device has an interface descriptor associated with it. An interface descriptor specifies the number of endpoints associated with that interface and other alternate settings. Functions that fall under a predefined category are indicated using the interface class code and sub class code fields. This enables the host to load standard device drivers associated with that function. More than one interface can be active at any time.

The endpoint descriptor specifies the type of transfer, direction, polling interval, and maximum packet size for each endpoint. Endpoint 0 is an exception; it does not have any descriptor and is always configured to be a control endpoint. String descriptors are human-readable strings that identify a USB device and the various interfaces supported by it.

2.1.3.3 *Transfer Types*

USB defines four transfer types through its pipes. These match the requirements of different data types that need to be delivered over the bus.

Bulk data is 'bursty,' traveling in packets of 8, 16, 32, or 64 bytes at full speed or 512 bytes at high speed. Bulk data has guaranteed accuracy, due to an automatic retry mechanism for erroneous data. The host schedules bulk packets when there is available bus time. Bulk transfers are typically used for printer, scanner, modem data, and storage devices. Bulk data has built-in flow control provided by handshake packets.

Interrupt data is similar to bulk data; it can have packet sizes of 1 through 64 bytes at full-speed or up to 1024 bytes at high-speed. Interrupt endpoints have an associated polling interval that ensures they are polled (receive an IN token) by the host on a regular basis.

Isochronous data is time-critical and used to stream data similar to audio and video. An isochronous packet may contain up to 1023 bytes at full-speed, or up to 1024 bytes at high-speed. Time of delivery is the most important requirement for isochronous data. In every USB frame, a certain amount of USB bandwidth is allocated to isochronous transfers. To lighten the overhead, isochronous transfers have no handshake and no retries; error detection is limited to a 16-bit CRC.

Control transfers configure and send commands to a device. Because they are so important, they employ the most extensive USB error checking. The host reserves a portion of each USB frame for control transfers.

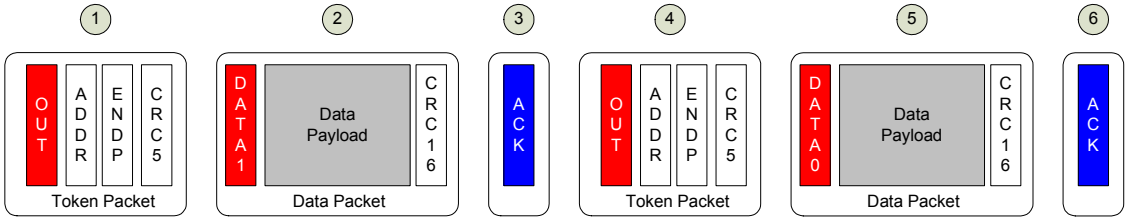
2.1.3.4 *Protocol Layer*

The function of the protocol layer is to understand the type of transfer, create the necessary packet IDs and headers, packet long data and generate CRCs, and pass them on to the link layer. Protocol level decisions similar to packet retry are also handled in this layer.

All communication over USB happen in the form of packets. Every USB packet, consist of a Packet ID (PID). These PIDs may fall into one of the four different categories and are listed here.

PID Type	PID Name
Token	IN, OUT, SOF, SETUP
Data	DATA0, DATA1, DATA2, MDATA
Handshake	ACK, NAK, STALL, NYET
Special	PRE, ERR, SPLIT, PING

Figure 2-2. USB Packets



A regular pay load data transfer requires at least three packets: Token, Data, and Ack. Figure 2-2 illustrates a USB OUT transfer. Tokens sent by the host are shown in red and tokens sent by the device are shown in blue. Packet 1 is an OUT token, indicated by the OUT PID. The OUT token signifies that data from the host is about to be transmitted over the bus. Packet 2 contains data, as indicated by the DATA1 PID. Packet 3 is a hand-shake packet, sent by the device using the ACK (acknowledge) PID to signify to the host that the device received the data error-free. Continuing with Figure 2-2, a second transaction begins with another OUT token 4, followed by more data 5, this time using the DATA0 PID. Finally, the device again indicates success by transmitting the ACK PID in a handshake packet 6.

SETUP tokens are unique to CONTROL transfers. They preface eight bytes of data from which the peripheral decodes host device requests. At full-speed, start of frame (SOF) tokens occur once per millisecond. At high speed, each frame contains eight SOF tokens, each denoting a 125-µs microframe.

Four handshake PIDs indicate the status of a USB transfer: ACK (Acknowledge) means 'success'; the data is received error-free. NAK (Negative Acknowledge) means 'busy, try again.' It is tempting to assume that NAK means 'error,' but it does not; a USB device indicates an error by not responding. STALL means that something is wrong (probably as a result of miscommunication or lack of cooperation between the host and device software). A device sends the STALL handshake to indicate that it does not understand a device request, that something went wrong on the peripheral end, or that the host tried to access a resource that was not there. It is similar to HALT, but better, because USB provides a way to recover from a stall. NYET (Not Yet) has the same meaning as ACK - the data was received error-free - but also indicates that the endpoint is not yet ready to receive another OUT transfer. NYET PIDs occur only in high-speed mode. A PRE (Preamble) PID precedes a low-speed (1.5 Mbps) USB transmission.

One notable feature of the USB 2.0 protocol is the data toggle mechanism. There are two DATA PIDs (DATA0 and DATA1) in Figure 2-2. As mentioned previously, the ACK handshake is an indication to the host that the peripheral received data with-out error (the CRC portion of the packet is used to detect errors). However, the handshake packet can get garbled during transmission. To detect this, each side (host and device) maintains a 'data toggle' bit, which is toggled between data packet transfers. The state of this internal toggle bit is compared with the PID that arrives with the data, either DATA0 or DATA1. When sending data, the host or device sends alternating DATA0-DATA1 PIDs. By comparing the received Data PID with the state of its own internal toggle bit, the receiver can detect a corrupted handshake packet.

The PING protocol was introduced in the USB 2.0 specification to avoid wasting bus bandwidth under certain circumstances. When operating at full speed, every OUT transfer sends the OUT data, even when the device is busy and cannot accept the data. Such unsuccessful repetitive bulk data transfers resulted in significant wastage of bus bandwidth. Realizing that this could get worse at high speed, this issue was remedied by using the new 'Ping' PID. The host first sends a short PING token to an OUT endpoint, asking if there is room for OUT data in the peripheral device. Only when the PING is answered by an ACK does the host send the OUT token and data.

The protocol for the interrupt, bulk, isochronous and control transfers are illustrated in the following figures.

Figure 2-3. Two Bulk Transfers, IN and OUT

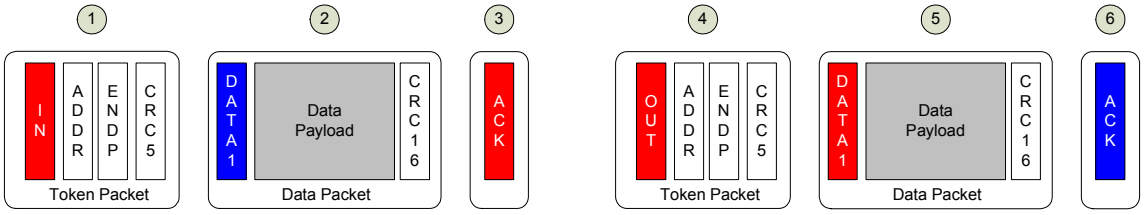


Figure 2-4. Interrupt Transfer

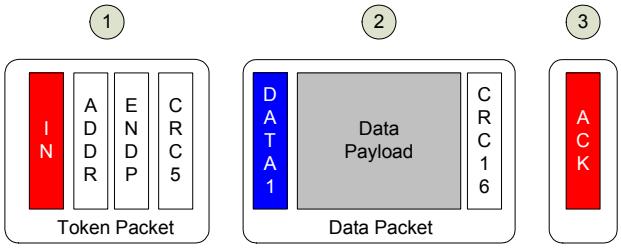


Figure 2-5. Isochronous Transfer

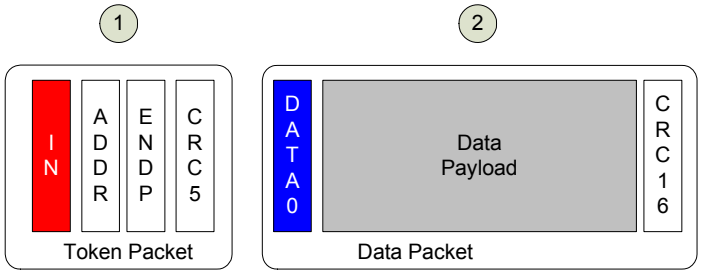
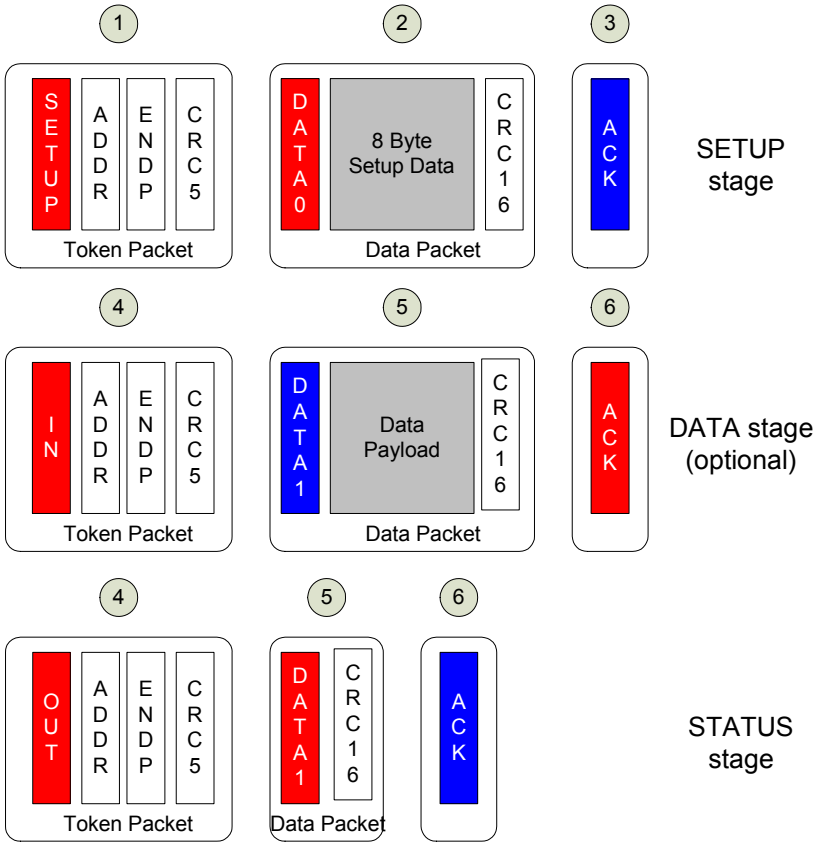


Figure 2-6. Control Transfer



2.1.3.5 Link/Physical Layer

The link layer performs additional tasks to increase the reliability of the data transfer. This includes byte ordering, line level framing, and so on.

More commonly known as the electrical interface of USB 2.0, this layer consists of circuits to serialize and de-serialize data, pre and post equalization circuits and circuits to drive and detect differential signals on the D+ and D– lines. All error handling is done at the protocol layer and there is no discernible low level link layer to manage errors.

2.1.4 Device Detection and Enumeration

One of the most important advantages of USB over other contemporary communication system is its plug-and-play capability. A change in termination at the USB port indicates that a USB device is connected.

When a USB device is first connected to a USB host, the host tries to learn about the device from its descriptors; this process is called enumeration. The host goes through the following sign on sequence

1. The host sends a Get Descriptor-Device request to address zero (all USB devices must respond to address zero when first attached).
2. The device responds to the request by sending ID data back to the host to identify itself.
3. The host sends a Set Address request, which assigns a unique address to the just-attached device so it may be distinguished from the other devices connected to the bus.

4. The host sends more Get Descriptor requests, asking for additional device information. From this, it learns every-thing else about the device such as number of endpoints, power requirements, required bus bandwidth, and what driver to load.

All high-speed devices begin the enumeration process in full-speed mode; devices switch to high-speed operation only after the host and device have agreed to operate at high speed. The high-speed negotiation process occurs during USB reset, via the 'Chirp' protocol.

Because the FX3 configuration is 'soft', a single chip can take on the identities of multiple distinct USB devices. When first plugged into USB, the FX3 enumerates automatically and downloads firmware and USB descriptor tables over the USB cable. A soft disconnect is triggered following which, the FX3 enumerates again, this time as a device defined by the downloaded information. This patented two-step process, called ReNumeration™, happens instantly when the device is plugged in, with no hint that the initial download step had occurred. ReNumeration avoids the need to physically disconnect and reconnect the device for the change in USB configuration to be visible to the host controller.

2.1.5 Power Distribution and Management

Power management refers to the part of the USB Specification that spell out how power is allocated to the devices connected downstream and how different communication layers operate to make best use of the available bus power under different circumstances.

USB 2.0 supports both self and bus powered devices. Devices indicate this through their descriptors. Devices, irrespective of their power requirements and capabilities are configured in their low power state unless the software instructs the host to configure the device in its high power state. Low power devices can draw up to 100 mA of current and high power devices can draw a maximum of 500 mA.

The USB host can 'suspend' a device to put it into a power-down mode. A 3 ms 'J' state (Differential '1' indicated by D+ high D– low) on the USB bus triggers the host to issue a suspend request and enter into a low power state. USB devices are required to enter a low power state in response to this request.

When necessary, the device or the host issues a Resume. A Resume signal is initiated by driving a 'K' state on the USB bus, requesting that the host or device be taken out of its low power 'suspended' mode. A USB device can only signal a resume if it has reported (through its Configuration Descriptor) that it is 'remote wakeup capable', and only if the host has enabled remote wakeup from that device.

This suspend-resume mechanism minimizes power consumed when activity on the USB bus is absent.

2.1.6 Device Classes

In an attempt to simplify the development of new devices, commonly used device functions were identified and nominal drivers were developed to support these devices. The host uses the information in the class code, subclass code, and protocol code of the device and interface descriptors to identify if built-in drivers can be loaded to communicate with the device attached. The human interface device (HID) class and mass storage class (MSC) are some of the commonly used device classes.

The HID class refers to interactive devices such as mouse, keyboards, and joy sticks. This interface use control and interrupt transfer types to transfer data because data transfer speeds are not critical. Data is sent or received using HID reports. Either the device or the interface descriptor contains the HID class code

The MSC class is primarily intended to transfer data to storage devices. This interface primarily uses bulk transfer type to transfer data. At least two bulk endpoints for each direction is necessary. The

MSC class uses the SCSI transparent command set to read or write sectors of data on the disk drive.

Details about other classes can be found at the Implementers forum website http://www.usb.org.

2.2 USB 3.0: Differences and Enhancements over USB 2.0

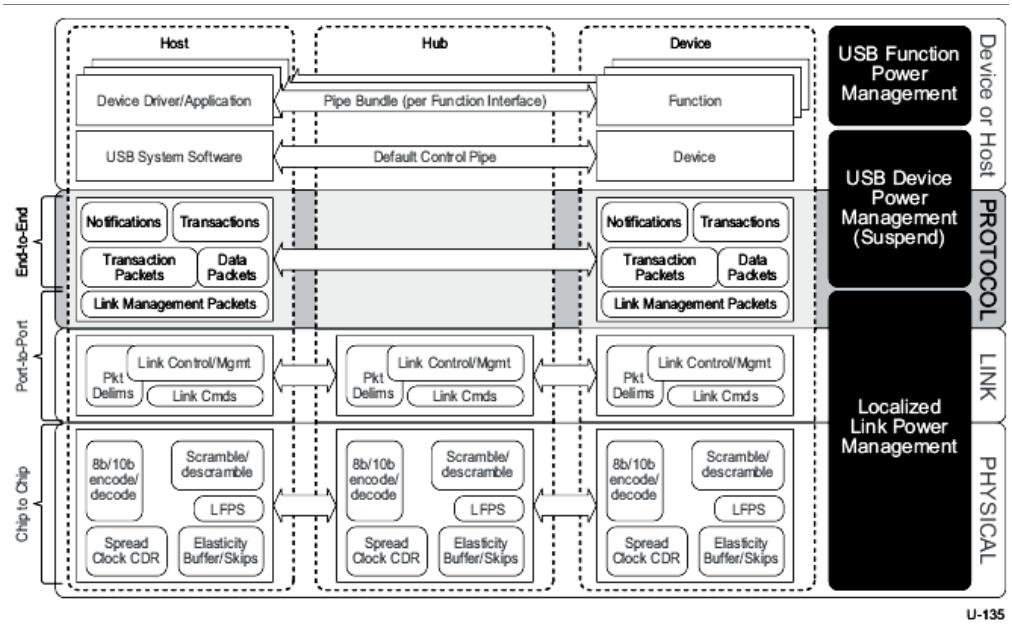
2.2.1 USB 3.0 Motivation

USB 3.0 is the next stage of USB technology. Its primary goal is to provide the same ease of use, flexibility, and hot-plug functionality but at a much higher data rate. Another major goal of USB 3.0 is power management. This is important for "Sync and Go" applications that need to trade off features for battery life.

The USB 3.0 interface consists of a physical SuperSpeed bus in addition to the physical USB 2.0 bus. The USB 3.0 standard defines a dual simplex signaling mechanism at a rate of 5 Gbits/s. Inspired by the PCI Express and the OSI 7-layer architecture, the USB 3.0 protocol is also abstracted into different layers as illustrated in the following sections.

In this document, USB 3.0 implicitly refers to the SuperSpeed portion of USB 3.0.

Figure 2-7. USB Protocol Layers

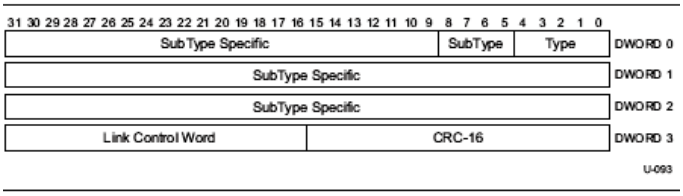


2.2.2 Protocol Layer

USB 3.0 SuperSpeed inherits the data transfer types from its predecessor retaining the model of pipes, endpoints and packets. Nonetheless, the type of packets used and some protocols associated with the bulk, control, isochronous, and control transfers have undergone some changes and enhancements. These are discussed in the sections to follow.

Link Management packets are sent between links to communicate link level issues such as link configuration and status and hence travel predominantly between the link layers of the host and the device. For example, U2 Inactivity Timeout LMP is used to define the timeout from the U1 state to the U2 state. The structure of a LMP is shown here.

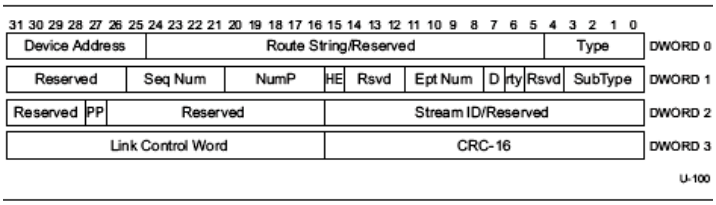
Figure 2-8. Link Management Packet Structure



Transaction packets reproduce the functionality provided by the Token and Handshake packets and travel between the host and endpoints in the device. They do not carry any data but form the core of the protocol.

For example, the ACK packet is used to acknowledge a packet received. The structure of a transaction packet is shown in Figure 2-9.

Figure 2-9. ACK Transaction Packet



Data packets actually carry data. These are made up of two parts: a data header and the actual data. The structure of a data packet is shown on the right.

Isochronous Time Stamp packets contain timestamps and are broadcast by the host to all active devices. Devices use timestamps to synchronize with the host. These do not have any routing information. The structure of an ITP is shown in Figure 2-11.

Figure 2-10. Example Data Packet

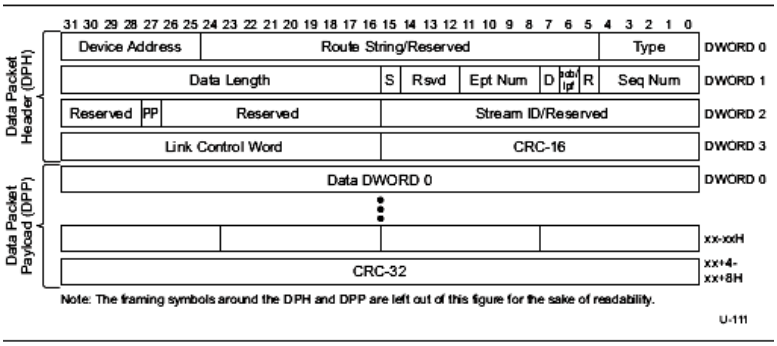
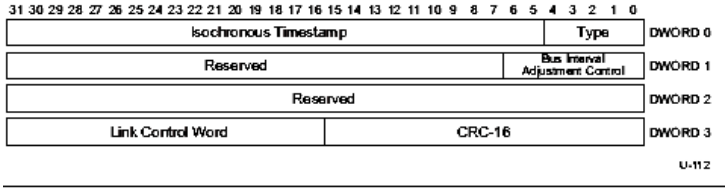


Figure 2-11. ITP Structure



OUT transfers are initiated by the host by sending a data packet on the downstream bus. The data packet contains the device routing address and the endpoint number. If the transaction is not an isochronous transaction, then, on receiving the data packet, the device launches an acknowledgement packet, which also contains the next packet number in the sequence. This process continues until all the packets are transmitted unless an endpoint responds with an error during the transaction. In transfers are initiated by the host by sending an acknowledge packet to the device containing the device, endpoint address and the number of packets that the host expects. The device then starts sending the data packets to the host. The response from the host acknowledges the previous transfer while initiating the next transfer from the device.

One important modification in the USB 3.0 specification is unicasting in place of broadcasting. Packets in USB 2.0 were broadcast to all devices. This necessitated every connected device to decode the packet address to check if the packet was targeted at it. Devices had to wake up to any USB activity regardless of its necessity in the transfer. This resulted in higher idle power. USB 3.0 packets (except ITP) are unicasted to the target. Necessary routing information for hubs is built into the packet.

Another significant modification introduced in USB 3.0 relates to interrupt transfers. In USB 2.0, Interrupt transfers were issued by the host every service interval regardless of whether or not the device was ready for transfers. However, SuperSpeed interrupt endpoints may send an ERDY/ NRDY in return for an interrupt transfer/request from the host. If the device returned an ERDY, the host continues to interrupt the device endpoint every service interval. If the device returned NRDY, the host stops interrupt request or transfers to the endpoint until the device asynchronously (not initiated by the host) notifies ERDY.

One of the biggest advantage the dual simplex bus architecture provides the USB 3.0 protocol with is the ability to launch multiple packets in one direction without waiting for an acknowledge packet from the other side which otherwise on a half duplex bus would cause bus contention. This ability is exploited to form a new protocol that dictates that packets be sent with a packet number, so that any missing or unfavorable acknowledges that comes after a long latency can be used to trigger the retransmission of the missed packet identified by the packet number. The number of burst packets that can be sent (without waiting for acknowledge) is communicated before the transfer.

Another notable feature of USB 3.0 is the stream protocol available for bulk transfers. Normal bulk (OUT) transfers transfer a single stream of data to an endpoint in the device. Typically, each stream of data is sourced from a buffer (FIFO) in the transmitter to another buffer (FIFO) in the receiver. The stream protocol allows the transmitter to associate a stream ID (1-65536) with the current stream transfer/request. The receiver of the stream or request sources or sinks the data to/from the appropriate FIFO. This multiplexing of the streams achieves mimicking a pipe which can dynamically shift its ends. Streams make it possible to realize an out-of-order execution model required for command queuing. The concept of streams enable more powerful mass storage protocols. A typical communication link consists of a command OUT pipe, an IN and OUT pipe (with multiple data streams), and a status pipe. The host can queue commands, that is, issue a new command without waiting for completion of a prior one, tagging each command with a Stream ID.

Because of the manner in which the USB 3.0 power management is defined, nonactive links (hubs, devices) may take longer to get activated on seeing bus activity. Isochronous transfers that activate the links take longer to reach the destination and may violate the service interval requirement. The Isochronous-PING protocol circumvents this issue. The host sends a PING transfer before an isochronous transaction. A PING RESPONSE indicates that all links in the path are active (or have been activated). The host can then send or request an isochronous data packet. USB 2.0 isochronous devices can not enter low power bus state in between service intervals.

2.2.3 Link Layer

The link layer maintains link connectivity and ensures data integrity between link partners by implementing error detection. The link layer ensure reliable data delivery by framing packet headers at the transmitting end and detecting link level errors at the receiving end. The link layer also implements protocols for flow control and participates in power management. The link layer provides an interface to the protocol layer for pass through of messages between the protocol layers. Link partners communicate using link commands.

2.2.4 Physical Layer

The two pairs of differential lines, one for OUT transfers and another for IN transfers define the physical connection between a USB 3.0 SuperSpeed host and the device. The physical layer accepts one byte at a time, scrambles the bits (a procedure that is known to reduce EMI emissions), converts it to 10 bits, serializes the bits, and transmits data differentially over a pair of wires. The clock data recovery circuit helps to recover data at the receiving end. The LFPS (Low frequency periodic signaling) block is used for initialization and power management when the bus is IDLE.

Detection of SuperSpeed devices is done by looking at the line terminations similar to USB 2.0 devices.

2.2.5 Power Management

USB 3.0 provides enhanced power management capabilities to address the needs of battery-powered portable applications. Two "Idle" modes (denoted as U1 and U2) are defined in addition to the "Suspend" mode (denoted as U3) of the USB 2.0 standard.

The U2 state provides higher power savings than U1 by allowing more analog circuitry (such as clock generation circuits) to be quiesced. This results in a longer transition time from U2 to active state. The Suspend state (U3) consumes the least power and again requires a longer time to wake up the system.

The Idle modes may be entered due to inactivity on a downstream port for a programmable period of time or may be initiated by the device, based on scheduling information received from the host. Such information is indicated by the host to the device using the flags Packet pending, End of burst, and Last packet. Based on these flags, the device may decide to enter an Idle mode without having to wait for inactivity on the bus. When a link is in one of these Idle states, communication may take place via low-frequency period signaling (LFPS), which consumes significantly lower power than SuperSpeed signaling. In fact, the Idle mode can be exited with an LFPS transmission from either the host or device.

The USB 3.0 standard also introduces the "Function Suspend" feature, which enables the power management of the individual functions of a composite device. This provides the flexibility of suspending certain functions of a composite device, while other functions remain active.

Additional power saving is achieved via a latency tolerance messaging (LTM) mechanism implemented by USB 3.0. A device may inform the host of the maximum delay it can tolerate from the time it reports an ERDY status to the time it receives a response. The host may factor in this latency tolerance to manage system power.

Thus, power efficiency is embedded into all levels of a USB 3.0 system, including the link layer, protocol layer, and PHY. A USB 3.0 system requires more power while active. But due to its higher data rate and various power-efficiency features, it remains active for shorter periods. A SuperSpeed data transfer can cost up to 50 percent less power than a hi-speed transfer. This is crucial to the battery life of mobile handset devices such as cellular phones.

2.3 Reference Documents

Some of this chapter's contents have been sourced from the following documents:

- Universal Serial Bus 3.0 Specification, Revision 1.0
- Universal Serial Bus Specification, Revision 2.0
- On-The-Go Supplement to the USB 2.0 Specification, Revision 1.3

3. FX3 Overview

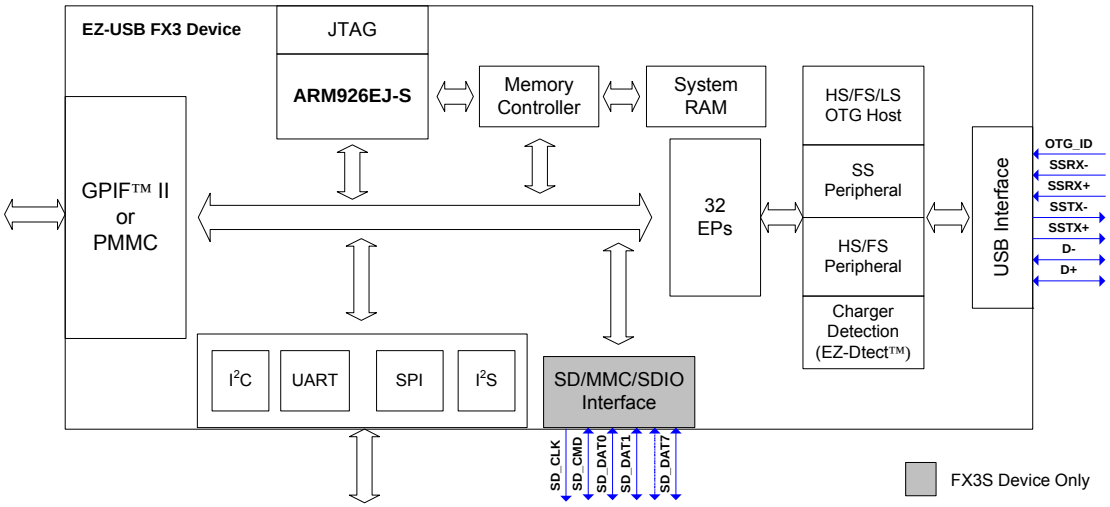
FX3 is a full-feature, general purpose integrated USB 3.0 Super-Speed controller with built-in flexible interface (GPIF II), which is designed to interface to any processor thus enabling customers to add USB 3.0 to any system.

The FX2G2 device is a variant of the FX3 device, which does not support the USB 3.0 interface. This part has the same ARM9 core and supports the USB 2.0 device and host mode functions as FX3.

FX3S is a variant of the FX3 device that features an integrated storage controller and can support up to 2 independent mass storage devices on its storage ports. It can support SD 3.0 and eMMC 4.41 memory cards. It can also support SDIO on these ports.

The SD3 is a variant of the FX3S device, which does not support the GPIF-II or MMC Slave interface. This device is a programmable USB 3.0 to SD/MMC/SDIO bridge and can be used for USB-based storage and networking applications.

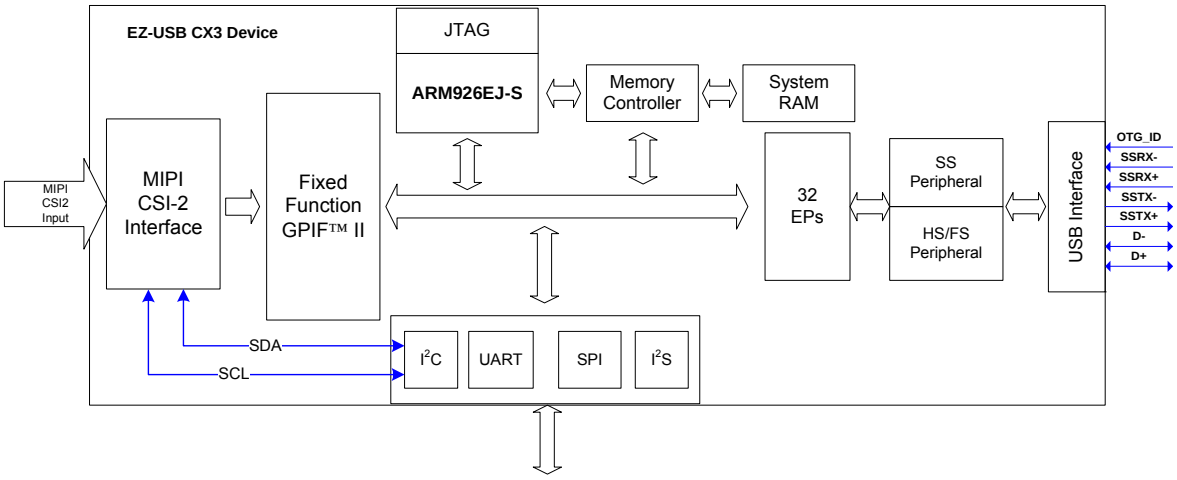
Figure 3-1. FX3 Logic Block Diagram



The logic block diagram shows the basic block diagram of FX3. The integrated USB 3.0 Phy and controller along with a 32-bit processor make FX3 powerful for data processing and building custom applications. An integrated USB 2.0 OTG controller enables applications that need dual role usage scenarios. A fully configurable, parallel, General Programmable Interface (GPIF II) provides connection to any processor, ASIC, DSP, or FPGA. There is 512 kB of on-chip SRAM for code and data. There are also low performance peripherals such as UART, SPI, I^2C, and I2S to communicate to onboard peripherals such as EEPROM. The CPU manages the data transfer between the USB, GPIF II, I2S, SPI, and UART interfaces through firmware and internal DMA interfaces.

The CX3 is a variant of the FX3 device, which features an integrated MIPI-CSI2 receiver mated to the GPIF, as shown in the CX3 Logic block diagram. This device provides the ability to add USB 3.0 connectivity to image sensors implementing the MIPI CSI-2 interface.

Figure 3-2. CX3 Logic Block Diagram



3.1 CPU

FX3 is powered by ARM926EJS, a 32-bit advanced processor core licensed from ARM that is capable of executing 220 MIPS [Wikipedia] at 200 MHz, the compute power needed to perform MP3 encoding, decryption, and header processing at USB 3.0 rates for the Universal Video Class

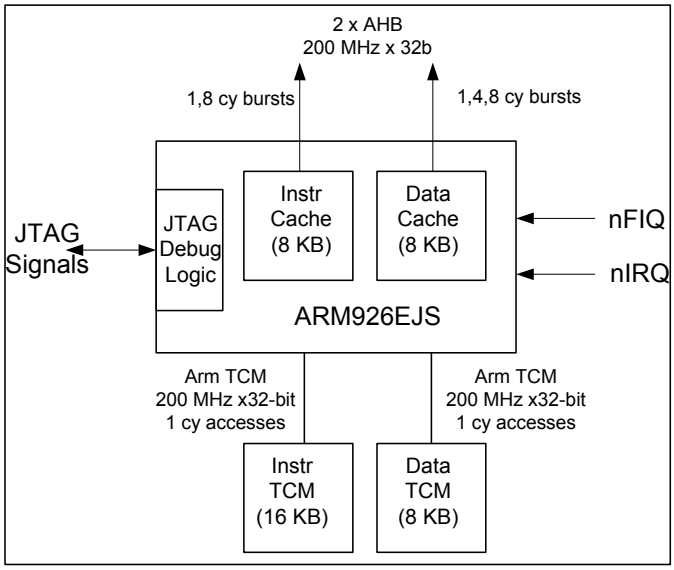
The 'Harvard Architecture' based processor accesses instruction and data memory through separate dedicated 32-bit industry standard AHB buses. Separate instruction and data caches are built into the core to facilitate low latency access to frequently used areas of code and data memory. In addition, the two tightly coupled memories (TCM) (one each for data and instruction) associated with the core provide a guaranteed low latency memory (without cache hit or miss uncertainties).

The ARM926 CPU contains a full Memory Management Unit (MMU) with virtual to physical address translation. FX3 contains 8 KB of data and instruction caches. ARM926-EJS has 4-way set associative caches and cache lines are 32 bytes wide. Each set therefore has 64 cache lines.

Interrupts vectored into one of the Fast Interrupt Request (FIQ) or Interrupt Request (IRQ) request lines provide a method to generate interrupt exceptions to the core.

The standard ARM JTAG TAP is integrated to allow debug support with any standard JTAG debugger.

Figure 3-3. Key CPU Features

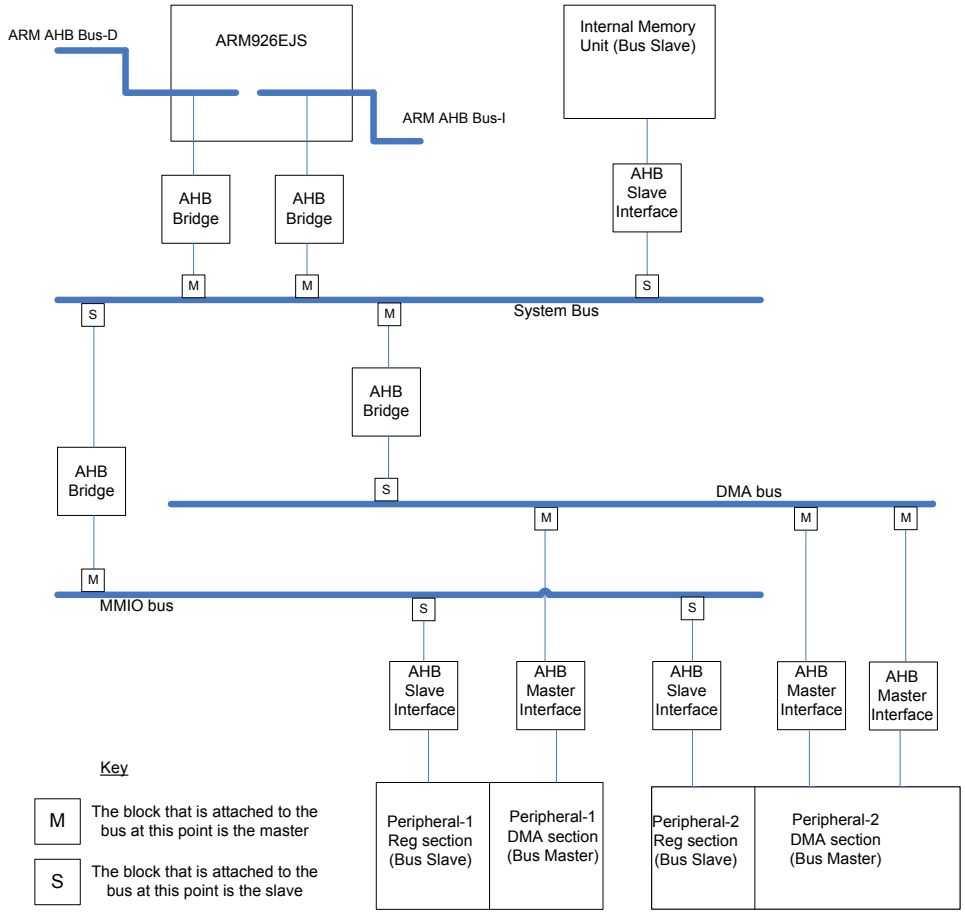


3.2 Interconnect Fabric

The Advanced Microcontroller Bus Architecture – Advanced High Performance Bus (AMBA AHB) interconnect forms the central nervous system of FX3. This fabric allows easy integration of processors, on-chip memories, and other peripherals using low power macro cell functions while providing a high-bandwidth communication link between elements that are involved in majority of the transfers. This multi-master high bandwidth interconnect has the following components:

- AHB bus master(s) that can initiate read and write operations by providing an address and control information. At any given instant, a bus can at most have one owner. When multiple masters demand bus ownership, the AHB arbiter block decides the winner.
- AHB bus slave(s) that respond to read or write operations within a given address-space range. The bus slave signals back to the active master the success, failure, or waiting of the data transfer. An AHB decoder is used to decode the address of each transfer and provide a select signal for the slave that is involved in the transfer.
- AHB bridges in the system to translate traffic of different frequency, bus width, and burst size. These blocks are essential in linking the buses
- AHB Slave/Master interfaces: These macro cells connect peripherals, memories, and other elements to the bus.

Figure 3-4. Interconnect Fabric

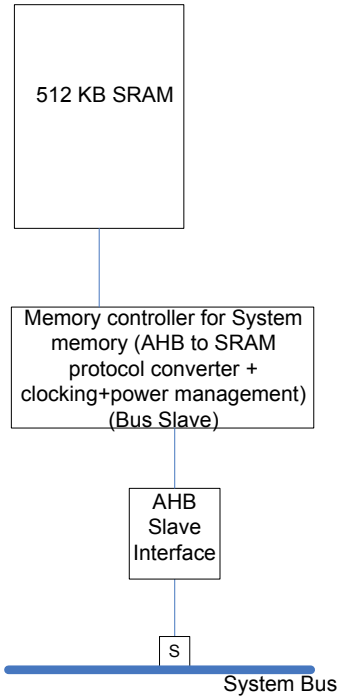


To allow implementation of an AHB system without the use of tristate drivers and to facilitate concurrent read/write operations, separate read and write data buses are required. The minimum data bus width is specified as 32 bits, but the bus width can be increased for realizing higher bandwidths.

3.3 Memory

In addition to the ARM core's tightly coupled instruction and data memories, a general purpose internal system memory block is available in FX3. The size of this memory is 512 KB or 256 KB depending on the device variant being used. The system SRAM is implemented using 64- or 128-bit wide SRAM banks, which run at full CPU clock frequency. Each bank may be built up from narrow SRAM instances for implementation specific reasons. A Cypress-proprietary high-performance memory controller translates a stream of AHB read and writes requests into SRAM accesses to the SRAM memory array. This controller also manages power and clock gating for the memory array. The memory controller is capable of achieving full 100% utilization of the SRAM array (meaning 1 read or 1 write at full width each cycle). CPU accesses are done 64 or 128 bit at a time to SRAM and then multiplexed/demultiplexed as 2/4 32-bit accesses on the CPU AHB bus. The controller does not support concurrent accesses to multiple different banks in memory.

Figure 3-5. Internal Memory Unit



The system memory is used for storing firmware code, data and DMA buffers. The first section of this area is used to store DMA instructions (also known as descriptors). The DMA hardware logic executes instructions from these locations. The remaining part of the memory can be used for code and data storage as well as for DMA buffers. If the MMU on the FX3 is being used to implement a virtual memory system, the last 16 K of the system memory can be used to store the page table. In the default configuration, virtual memory is not used and there is no need for a memory resident page table.

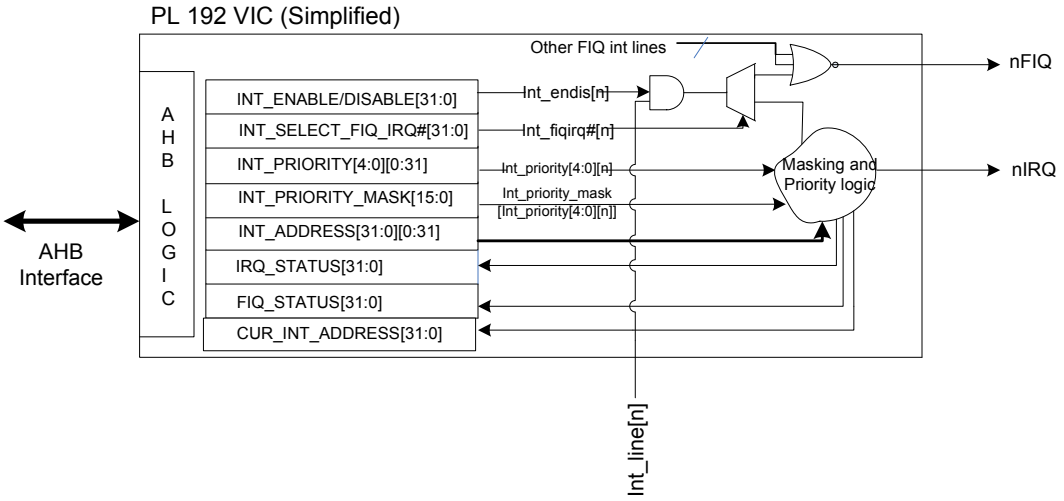
3.4 Interrupts

Interrupt exceptions are facilitated using the FIQ and IRQ lines of the ARM9 processor. The ISR branch instruction for each of these interrupts is provided in the 32 byte exception table located at the beginning of the ITCM.

The embedded PL192 vectored interrupt controller (VIC) licensed from ARM provides a hardware based interrupt management system that handles interrupt vectoring, priority, masking and timing, providing a real time interrupt status. The PL192 VIC supports 32 'active high' interrupt sources, the ISR location of which can be programmed into the VIC. Each interrupt can be assigned one of the 15 programmable priority levels; equal priority interrupts are further prioritized based on the interrupt number. While each interrupt pin has a corresponding mask and enable bits, interrupts with a particular priority level can all be masked out at the same time if desired. Each of the '32-interrupt' can be vectored to one of the active low FIQ or IRQ outputs of the VIC that are directly hooked to the corresponding lines of the ARM 9 CPU. PL192 also supports daisy chained interrupts, a feature that is not enabled in FX3.

Note Other exceptions include reset, software exception, data abort, and pre-fetch abort.

Figure 3-6. Vector Interrupt Controller



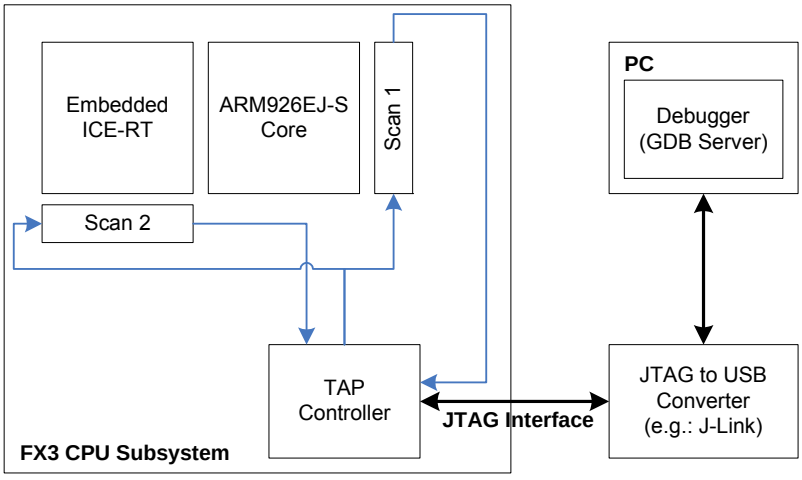
When both FIQ and IRQ interrupt inputs assert, the CPU jumps to the FIQ entry of the exception table. The FIQ handler is usually placed immediately after the table, saving a branch. The FIQ mode uses dedicated FIQ bank registers. When an IRQ line alone asserts, CPU jumps to the IRQ handler. The IRQ handler saves the workspace on stack, reads the address of the ISR from the VIC, and jumps to the actual ISR.

In general, high priority, low latency interrupts are vectored into FIQ while the IRQ line is reserved for general interrupts. Re-entrant interrupts can be supported with additional firmware.

3.5 JTAG Debugger Interface

Debug support is implemented by using the ARM9EJ-S core embedded within the ARM926EJ-S processor. The ARM9EJ-S core has hardware that eases debugging at the lowest level. The debug extensions allow to stall the core's program execution, examine the internal state of the core and the memory system, and further resume program execution.

Figure 3-7. ARM Debug Logic Blocks



The ARM debugging environment has three components: A debug-host resident program (Real View debugger), a debug communication channel (JTAG) and a target (Embedded ICE-RT). The two JTAG-style scan chains (Scan1 and Scan2) enable debugging and 'EmbeddedICE-RT-block' programming.

Scan Chain 1 is used to debug the ARM9EJ-S core when it has entered the debug state. The scan chain can be used to inject instructions into ARM pipeline and also read or write core registers without having to use the external data bus. Scan Chain 2 enables access to the EmbeddedICE registers. The boundary scan interface includes a state machine controller called the TAP controller that controls the action of scan chains using the JTAG serial protocol.

The ARM9EJ-S EmbeddedICE-RT logic provides integrated on-chip debug support for the ARM9EJ-S core. The EmbeddedICE-RT logic comprises two real time watchpoint units, two independent registers, the Debug Control Register and the Debug Status Register, and the debug communication channel. A watchpoint unit can either be configured to monitor data accesses (commonly called watchpoints) or monitor instruction fetches (commonly called breakpoints).

The EmbeddedICE-RT logic interacts with the external logic (logic outside the CPU subsystem) using the debug interface. In addition, it can be programmed (for example, setting a breakpoint) using the JTAG based TAP controller. The debug interface signals not only communicate the debug status of the core to the external logic but also provide a means to for the external logic to raise breakpoints if needed (disabled in FX3 by default).

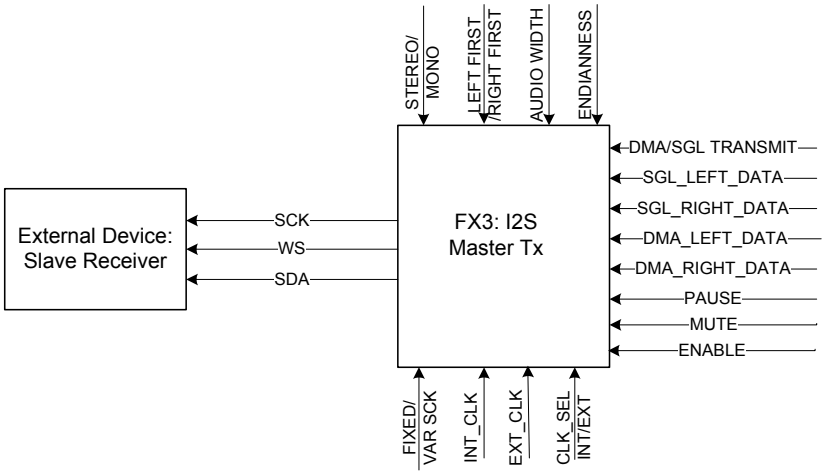
ARM9EJ-S supports two debug modes: Halt mode and Monitor mode. In halt mode debug, a watch-point or breakpoint request forces the core into debug state. The internal state of the core can then be examined and instructions inserted into its pipeline using the TAP controller without using the external bus thus leaving the rest of the system unaltered. The core can then be forced to resume normal operation. Alternately, the EmbeddedICE-RT logic can be configured in monitor mode, where watchpoints or breakpoints generate Data or Pre-fetch Aborts respectively. This enables the debug monitor system to debug the ARM while enabling critical fast interrupt requests to be serviced.

3.6 Peripherals

3.6.1 I2S

FX3 is capable of functioning as a master mode transmitter over its Integrated Inter-chip Sound (I2S) interface. When integrated with an audio device, the I2S bus only handles audio data, while the other signals, such as sub-coding and control, are transferred separately.

Figure 3-8. I2S Blocks



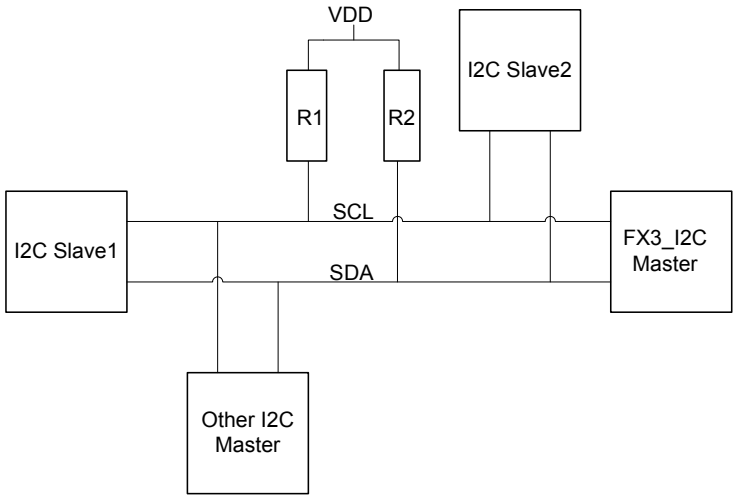
The I2S block can be configured to support different audio bus widths, endianess, number of channels, and data rate. By default, the interface is protocol standard big endian (most significant bit first); nevertheless, the block's endianess can be reversed. FX3 also supports the left justified and right justified variants of the protocol. When the block is enabled in left justified mode, the left channel audio sample is sent first on the SDA line.

In the mono mode, the 'left data' is sent to both channels on the receiver (WordSelect = Left and WordSelect = Right). Supported audio sample widths include 8, 16, 18, 24, and 32 bit. In the variable SCK (Serial Clock) mode, WS (WordSelect) toggles every Nth edge of SCK, where N is the bus width chosen. In fixed SCK mode, however, WS toggles every thirty-second SCK edge. In this mode, the audio sample is zero padded to 32 bit. FX3 supports word at a time (SGL_LEFT_DATA, SGL_RIGHT_DATA) I2S operations for small transfers and DMA based I2S operations for larger transfers. The Serial Clock can be derived from the internal clock architecture of FX3 or supplied from outside using a crystal oscillator. Typical frequencies for WS include 8, 16, 32, 44.1, 48, 96, and 192 kHz.

Two special modes of operation, Mute and Pause are supported. When Mute is held asserted, DMA data is ignored and zeros are transmitted instead. When paused, DMA data flow into the block is stopped and zeros are transmitted over the interface.

3.6.2 I²C

Figure 3-9. I2C Block Diagram



FX3 is capable of functioning as a master transceiver and supports 100 kHz, 400 kHz, and 1 MHz operation. The I²C block operates in big endian mode (most significant bit first) and supports both 7-bit and 10-bit slave addressing. Similar to I2S, this block supports both single and burst (DMA) data transfers.

Slow devices on its I²C bus can work with FX3's I²C using the clock stretching based flow control. FX3 can function in multi-master bus environments as it is capable of carrying out negotiations with other masters on the bus using SDA based arbitration. Additionally, FX3 supports the repeated start feature to communicate to multiple slave devices on the bus without losing ownership of the bus in between (see the stop last and start first feature in the following sections).

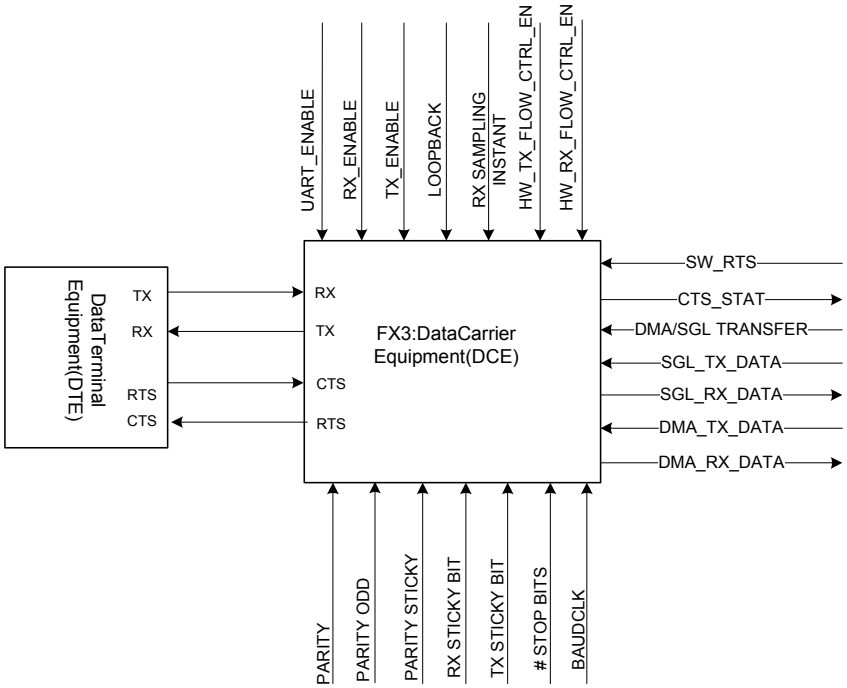
Combined format communication is supported, which allows the user to load multiple bytes of data (including slave chip address phases) into using special registers called preamble. The user can choose to place start (repeated) or stop bits between the bytes and can also define the master's behavior on receiving either a NAK or ACK for bytes in the preamble. In applications such as EEPROM reads, this greatly reduces firmware complexity and execution time by packing initial communication bytes into a transaction header with the ability to abort the header transaction on receiving NAK exceptions in the middle of an operation.

In addition, the preamble repeat feature available in FX3 simplifies firmware and saves time in situations - for instance, ACK monitoring from the EEPROM to check completion of a previously issued program operation. In this case, FX3's I²C can be programmed to repeat a single byte preamble containing the EEPROM's I²C address until the device responds with an ACK.

By programming the burst read count value for this block, burst reads from the slave (EEPROM for example), can be performed with no firmware intervention. In this case, the FX3 master receiver sends ACK response for all bytes received as long as the burst read counter does not expire. When the last byte of the burst is received, FX3's I²C block signals a NAK followed by a stop bit forcing the device to stop sending data.

3.6.3 UART

Figure 3-10. UART Block Diagram



FX3's UART block provides standard asynchronous full-duplex transfer using the TX and RX pins. Flow control mechanism, RTS (request to send) and CTS (clear to send) pins are supported. The UART block can operate at numerous baud rates ranging from 300 bps to 4608 Kbps and supports both one and two stop bits mode of operation. Similar to I2S and I^2C blocks, this block supports both single and burst (DMA) data transfers.

The transmitter and receiver components of the UART block can be individually enabled. When both components are enabled and the UART set in loopback mode, the external interface is disabled; data scheduled on the TX pin is internally looped back to the RX pin.

Both hardware and software flow control are supported. Flow control settings can individually be set on the transmitter and receiver components. When both (Tx and Rx) flows are controlled by hardware, the RTS and CTS signaling is completely managed by the UART block's hardware. When the flow control is completely handled by software, software can signal a request to send using the SW_RTS field of the block and monitor the block's CTS_STAT signal.

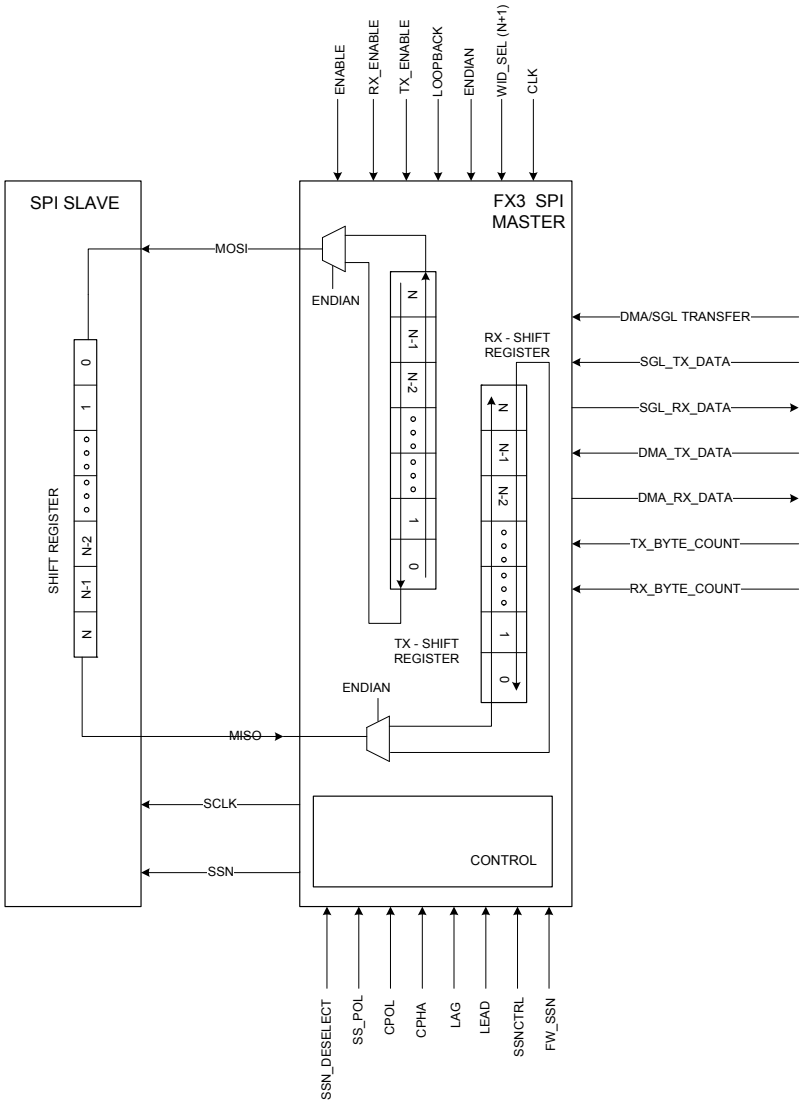
Transmission starts when a 0-level start bit is transmitted and ends when a 1-level stop bit is transmitted. This represents the default 10-bit transmission mode. A ninth parity bit can be appended to data in the 11-bit transmission mode. Both the even and odd parity variants are supported. Optionally, a fixed check bit (sticky parity) can be inserted in place of the parity bit. The value of this bit is configurable. Corresponding settings are also available for the Rx block.

The UART Rx line is internally oversampled by a factor of 8 and any three consecutive samples among the eight can be chosen for the majority vote technique to decide the received bit.

Separate interface devices can be used to convert the logic level signals of the UART to and from the external voltage signaling standards such as RS-232, RS-422, and RS-485.

3.6.4 SPI

Figure 3-11. SPI Block Diagram



FX3's SPI block operates in master mode and facilitates standard full-duplex synchronous transfers using the Master Out Slave In (MOSI), Master In Slave Out (MISO), Serial Clock (SCLK), and Slave Select (SS) pins. The maximum frequency of operation is 33 MHz. Similar to the I2S and I^2C and UART blocks, this block supports both single and burst (DMA) data transfers.

The transmit and receive blocks can be enabled independently using the TX_ENABLE/RX_ENABLE inputs. Independent shift registers are used for the transmit and receive paths. The width of the shift registers can be set to anywhere between 4 and 32 bits. By default, the Tx and Rx registers shift data to the left (big endian). This can be reversed, if necessary.

The SSPOL input sets the polarity of the Slave Select (SSN) signal.The CPOL input sets the polarity of the SCLK pin which is active high by default. The CPHA input sets the clock phases for data transmit and capture. If CPHA is set to ‘1’, the devices agree to transmit data on the asserting edge of the clock and capture at the de-asserting edge. However, if CPHA is set to 0, the devices agree to

capture data on the asserting edge of the clock and transmit at the de-asserting edge. When Idle, the SCLK pin remains de-asserted. Hence, the first toggle of SCLK in this mode (CPHA=0) will cause the receivers to latch data; placing a constraint on the transmitters to set up data before the first toggle of SCLK. To SSN LEAD setting is provided to facilitate the assertion of SS (and hence the first transmit data) a few cycles before the first SCLK toggle. The SSN LAG setting specifies the delay in SSN de-assertion after the last SCLK toggle for the transfer. This specific mode of operation (CPHA=0) also necessitate toggling the SSN signal between word transfers.

The SSN pin can be configured to either remain asserted always, deassert between transfers, handled by hardware (based on CPHA configuration) or managed using software. FX3's SPI block can share its MOSI, MISO, and SCLK pins with more than one slave connected on the bus. In this case, the SSN signal of the block cannot be used and the slave selects need to be managed using GPIOs.

3.6.5 GPIO/Pins

Several pins of FX3 can function as General Purpose I/O s. Each of these pins is multiplexed to support other functions/interfaces (such as UART and SPI). By default, pins are allocated in larger groups to functional blocks such as GPIF-II, I2C, and UART. In a typical application, not all blocks of FX3 are used. Also, not all pins assigned to a block may be getting used. For example, the UART_CTS and UART_RTS pins are mapped to the UART block but are rarely used. Unused pins in each block may be overridden as a simple or complex GPIO pin on a pin-by-pin basis.

Simple GPIO provides software controlled and observable input and output capability only. In addition, they can also raise interrupts. Complex GPIOs add 3 timer/counter registers for each and support a variety of time based functions. They either work off a slow or fast clock. Complex GPIOs can also be used as general purpose timers by firmware.

There are eight complex I/O pin groups, the elements of which are chosen in a modulo 8 fashion (complex I/O group 0 – GPIO 0, 8, 16., complex I/O group 1- GPIO 1,9,17., and so on). Each group can have different complex I/O functions (such as PWM one shot). However, only one pin from a group can use the complex I/O functions. The rest of the pins in the group are either used as block I/O or simple GPIO.

The following tables show the various functions that can be mapped to each GPIO pin on the FX3 device. The specific function for a pin is primarily selected on a block basis and can be over-ridden on an individual pin basis.

Table 3-1. Block I/O Selection on FX3 Devices

Blk I/O Selection Table	Choose I/O Pins from Blocks: GPIF	Choose I/O Pins from Blocks: GPIF, UART	Choose I/O Pins from Blocks: GPIF, SPI	Choose I/O Pins from Blocks: GPIF, I2S	Choose I/O Pins from Blocks: GPIF(DQ32/ extended), UART, I2S	Choose I/O Pins from Blocks: GPIF,UART, SPI, I2S
Blk IO[0]	GPIF IO[0]	GPIF IO[0]	GPIF IO[0]	GPIF IO[0]	GPIF IO[0]	GPIF IO[0]
Blk IO[1]	GPIF IO[1]	GPIF IO[1]	GPIF IO[1]	GPIF IO[1]	GPIF IO[1]	GPIF IO[1]
…	…	…	…	…	…	…
Blk IO[29]	GPIF IO[29]	GPIF IO[29]	GPIF IO[29]	GPIF IO[29]	GPIF IO[29]	GPIF IO[29]
Blk IO [30]	NC	NC	NC	NC	NC	NC
Blk IO [31]	NC	NC	NC	NC	NC	NC
Blk IO [32]	NC	NC	NC	NC	NC	NC
Blk IO [33]	GPIF IO [30]	GPIF IO [30]	GPIF IO [30]	GPIF IO [30]	GPIF IO [30]	GPIF IO [30]
…	…	…	…	…	…	…
Blk IO[44]	GPIF IO[41]	GPIF IO[41]	GPIF IO[41]	GPIF IO[41]	GPIF IO[41]	GPIF IO[41]
Blk IO[45]	NC	NC	NC	NC	NC	NC
Blk IO[46]	NC	NC	NC	NC	GPIF IO [42]	UART_RTS
Blk IO[47]	NC	NC	NC	NC	GPIF IO [43]	UART_CTS
Blk IO[48]	NC	NC	NC	NC	GPIF IO [44]	UART_TX
Blk IO[49]	NC	NC	NC	NC	GPIF IO [45]	UART_RX
Blk IO[50]	NC	NC	NC	NC	I2S_CLK	I2S_CLK
Blk IO[51]	NC	NC	NC	NC	I2S_SD	I2S_SD
Blk IO[52]	NC	NC	NC	NC	I2S_WS	I2S_WS
Blk IO[53]	NC	UART_RTS	SPI_SCK	NC	UART_RTS	SPI_SCK
Blk IO[54]	NC	UART_CTS	SPI_SSN	I2S_CLK	UART_CTS	SPI_SSN
Blk IO[55]	NC	UART_TX	SPI_MISO	I2S_SD	UART_TX	SPI_MISO
Blk IO[56]	NC	UART_RX	SPI_MOSI	I2S_WS	UART_RX	SPI_MOSI
Blk IO[57]	NC	NC	NC	I2S_MCLK	I2S_MCLK	I2S_MCLK
Blk IO[58]	I2C_SCL	I2C_SCL	I2C_SCL	I2C_SCL	I2C_SCL	I2C_SCL
Blk IO[59]	I2C_SDA	I2C_SDA	I2C_SDA	I2C_SDA	I2C_SDA	I2C_SDA
Blk O [60]	Charger det	Charger det	Charger det	Charger det	Charger det	Charger det

Note: Depending on the configuration, one of the columns of the table will be selected

- NC stands for Not Connected. Blk I/Os 30-32,45 are Not Connected
- Charger detect is output only.
- GPIF I/O are further configured into interface clock, control lines, address lines and data lines, car-kit UART lines depending on the desired total number of GPIF pins, number of data lines and address lines, whether the address and data lines need to be multiplexed and the car-kit mode. Depending on the GPIF configuration selected by the user, some of the GPIF I/O may remain unconnected and may only be used as GPIOs.

Table 3-2. Block I/O Selection on FX3S Device

Blk I/O Selection Table	Choose I/O Pins from blocks: GPIF	Choose I/O Pins from blocks: GPIF, S0	Choose I/O Pins from blocks: GPIF, S0, S1	Choose I/O Pins from blocks: GPIF, S0, S1, UART	Choose I/O Pins from blocks: GPIF, S0, S1, SPI	Choose I/O Pins from blocks: GPIF, S0, S1, I2S	Choose I/O Pins from blocks: GPIF, S0, UART, SPI, I2S
Blk IO[0]	GPIF IO[0]	GPIF IO[0]	GPIF IO[0]	GPIF IO[0]	GPIF IO[0]	GPIF IO[0]	GPIF IO[0]
Blk IO[1]	GPIF IO[1]	GPIF IO[1]	GPIF IO[1]	GPIF IO[1]	GPIF IO[1]	GPIF IO[1]	GPIF IO[1]
…	…	…	…	…	…	…	…
Blk IO[29]	GPIF IO[29]	GPIF IO[29]	GPIF IO[29]	GPIF IO[29]	GPIF IO[29]	GPIF IO[29]	GPIF IO[29]
Blk IO[30]	PMODE[0]	PMODE[0]	PMODE[0]	PMODE[0]	PMODE[0]	PMODE[0]	PMODE[0]
Blk IO[31]	PMODE[1]	PMODE[1]	PMODE[1]	PMODE[1]	PMODE[1]	PMODE[1]	PMODE[1]
Blk IO[32]	PMODE[2]	PMODE[2]	PMODE[2]	PMODE[2]	PMODE[2]	PMODE[2]	PMODE[2]
Blk IO[33]	NC	S0_DAT[0]	S0_DAT[0]	S0_DAT[0]	S0_DAT[0]	S0_DAT[0]	S0_DAT[0]
Blk IO[34]	NC	S0_DAT[1]	S0_DAT[1]	S0_DAT[1]	S0_DAT[1]	S0_DAT[1]	S0_DAT[1]
…	…	…	…	…	…	…	…
Blk IO[40]	NC	S0_DAT[7]	S0_DAT[7]	S0_DAT[7]	S0_DAT[7]	S0_DAT[7]	S0_DAT[7]
Blk IO[41]	NC	S0_CMD	S0_CMD	S0_CMD	S0_CMD	S0_CMD	S0_CMD
Blk IO[42]	NC	S0_CLK	S0_CLK	S0_CLK	S0_CLK	S0_CLK	S0_CLK
Blk IO[43]	NC	S0_WP	S0_WP	S0_WP	S0_WP	S0_WP	S0_WP
Blk IO[44]	NC	S0_DETECT	S0_DETECT	S0_DETECT	S0_DETECT	S0_DETECT	S0_DETECT
Blk IO[45]	NC	S0_MMCRESET	S0_MMCRESET	S0_MMCRESET	S0_MMCRESET	S0_MMCRESET	S0_MMCRESET
Blk IO[46]	NC	NC	S1_DAT[0]	S1_DAT[0]	S1_DAT[0]	S1_DAT[0]	UART_RTS
Blk IO[47]	NC	NC	S1_DAT[1]	S1_DAT[1]	S1_DAT[1]	S1_DAT[1]	UART_CTS
Blk IO[48]	NC	NC	S1_DAT[2]	S1_DAT[2]	S1_DAT[2]	S1_DAT[2]	UART_TX
Blk IO[49]	NC	NC	S1_DAT[3]	S1_DAT[3]	S1_DAT[3]	S1_DAT[3]	UART_RX
Blk IO[50]	NC	NC	S1_CMD	S1_CMD	S1_CMD	S1_CMD	I2S_CLK
Blk IO[51]	NC	NC	S1_CLK	S1_CLK	S1_CLK	S1_CLK	I2S_SD
Blk IO[52]	NC	NC	S1_WP	S1_WP	S1_WP	S1_WP	I2S_WS
Blk IO[53]	NC	NC	S1_DAT[4]	UART_RTS	SPI_SCK	NC	SPI_SCK
Blk IO[54]	NC	NC	S1_DAT[5]	UART_CTS	SPI_SSN	I2S_CLK	SPI_SSN
Blk IO[55]	NC	NC	S1_DAT[6]	UART_TX	SPI_MISO	I2S_SD	SPI_MISO
Blk IO[56]	NC	NC	S1_DAT[7]	UART_RX	SPI_MOSI	I2S_WS	SPI_MOSI
Blk IO[57]	NC	NC	S1_MMCRESET	NC	NC	I2S_MCLK	I2S_MCLK
Blk IO[58]	I2C_SCL	I2C_SCL	I2C_SCL	I2C_SCL	I2C_SCL	I2C_SCL	I2C_SCL
Blk IO[59]	I2C_SDA	I2C_SDA	I2C_SDA	I2C_SDA	I2C_SDA	I2C_SDA	I2C_SDA
Blk IO[60]	Charger Det	Charger Det	Charger Det	Charger Det	Charger Det	Charger Det	Charger Det

Table 3-3 shows how the functionality of each FX3 GPIO can be selected between block I/O (block-specific function, such as GPIF-II, UART, and I2C), simple GPIO, and complex GPIO. Any of the pins can be selected as a simple GPIO and controlled independently. As the device only supports eight complex GPIOs, the mapping of pins to complex GPIOs is done on a modulo-8 basis. Each complex GPIO function can only be made active on one of the eight pins mapped to it.

Table 3-3. Block I/O Override as Simple/Complex GPIO

FX3 Pin I/O Selection n = 0 to 60	**Override block IO as simple GPIO [pin n] = False Override block IO as complex GPIO [pin n] = False**	**Override block IO as simple GPIO [pin n] = True Override block IO as complex GPIO [pin n] = False**	**Override block IO as complex GPIO [pin n] = True**
IO Pin [0]	Blk IO [0]	Simple GPIO [0]	Complex GPIO [0]
IO Pin [1]	Blk IO [1]	Simple GPIO [1]	Complex GPIO [1]
…			
IO Pin [8]	Blk IO [8]	Simple GPIO [8]	Complex GPIO [0]
IO Pin [9]	Blk IO [9]	Simple GPIO [9]	Complex GPIO [1]
…			
IO Pin [59]	Blk IO [59]	Simple GPIO [59]	Complex GPIO [3]
O Pin [60]	Blk O [60]	Simple GPO [60]	Complex GPO [4]

Note: For each pin ‘n’ in column 1, one of column 2, 3 or 4 of the corresponding row will be selected based on the simple/override values for the corresponding pin

- Pins [30-32] are used as PMODE [0-2] inputs during boot. After boot, these are available as GPIOs.

The GPIF-II block on FX3 can be configured to have a 8-, 16-, 24-, or 32-bit wide data bus. Further, the device can have no address bus, a variable width address bus up to 16 bits wide, or an address bus multiplexed with the data bus. The number and function of control lines can also be configured in different ways. The actual function assigned to a GPIO in the GPIF-II block will depend on the configuration selected by the user. Table 3-4 shows the functions assigned to these GPIOs under various GPIF-II configurations.

Table 3-4. GPIF I/O Block Pin Connection for Different Configurations

Overall GPIF Pin Count	47-pin	43-pin	35-pin	31-pin	31-pin	47-pin	43-pin	35-pin	31-pin	31-pin
Data Bus Width	32b	24b	16b	16b	8b	32b	24b	16b	16b	8b
Address Data lines Muxed?	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
GPIF IO [16]	CLK	CLK	CLK	CLK	CLK	CLK	CLK	CLK	CLK	CLK
GPIF IO[0]	D0/A0	D0/A0	D0/A0	D0/A0	D0/A0	D0	D0	D0	D0	D0
INT#	INT	INT	INT	INT	INT	INT	INT	INT	INT	INT
GPIF IO[1]	D1/A1	D1 /A1	D1/A1	D1/A1	D1/A1	D1	D1	D1	D1	D1
GPIF IO[2]	D2/A2	D2/A2	D2/A2	D2/A2	D2/A2	D2	D2	D2	D2	D2
GPIF IO[3]	D3/A3	D3/A3	D3/A3	D3/A3	D3/A3	D3	D3	D3	D3	D3
GPIF IO[4]	D4/A4	D4/A4	D4/A4	D4/A4	D4/A4	D4	D4	D4	D4	D4
GPIF IO[5]	D5/A5	D5/A5	D5/A5	D5/A5	D5/A5	D5	D5	D5	D5	D5
GPIF IO[6]	D6/A6	D6/A6	D6/A6	D6/A6	D6/A6	D6	D6	D6	D6	D6

Table 3-4. GPIF I/O Block Pin Connection for Different Configurations

GPIF IO[7]	D7/A7	D7/A7	D7/A7	D7/A7	D7/A7	D7	D7	D7	D7	D7
GPIF IO[8]	D8/A8	D8/A8	D8/A8	D8/A8		D8	D8	D8	D8	A0
GPIF IO[9]	D9/A9	D9/A9	D9/A9	D9/A9		D9	D9	D9	D9	A1
GPIF IO[10]	D10/A10	D10/A10	D10/A10	D10/A10		D10	D10	D10	D10	A2
GPIF IO[11]	D11/A11	D11/A11	D11/A11	D11/A11		D11	D11	D11	D11	A3
GPIF IO[12]	D12/A12	D12/A12	D12/A12	D12/A12		D12	D12	D12	D12	A4
GPIF IO[13]	D13/A13	D13/A13	D13/A13	D13/A13	C15	D13	D13	D13	D13	A5
GPIF IO[14]	D14/A14	D14/A14	D14/A14	D14/A14	C14	D14	D14	D14	D14	A6
GPIF IO[15]	D15/A15	D15/A15	D15/A15	D15/A15	C13	D15	D15	D15	D15	A7
GPIF IO[30]	D16/A16	D16/A16				D16	D16			
GPIF IO[31]	D17/A17	D17/A17				D17	D17			
GPIF IO[32]	D18/A18	D18/A18				D18	D18			
GPIF IO[33]	D19/A19	D19/A19				D19	D19			
GPIF IO[34]	D20/A20	D20/A20				D20	D20			
GPIF IO[35]	D21/A21	D21/A21				D21	D21			
GPIF IO[36]	D22/A22	D22/A22				D22	D22			
GPIF IO[37]	D23/A23	D23/A23				D23	D23			
GPIF IO[38]	D24/A24	n/u				D24	A0			
GPIF IO[39]	D25/A25	C15				D25	C15/A1			
GPIF IO[40]	D26/A26	C14				D26	C14/A2			
GPIF IO[41]	D27/A27	C13				D27	C13/A3			
GPIF IO[42]	D28/A28					D28		A0		
GPIF IO[43]	D29/A29		C15			D29		C15/A1		
GPIF IO[44]	D30/A30		C14			D30		C14/A2		
GPIF IO[45]	D31/A31		C13			D31		C13/A3		
GPIF IO[29]	C12	C12	C12	C12	C12	C12/A0	C12/A4	C12/A4	C12/A0	C12/A8
GPIF IO[28]	C11	C11	C11	C11	C11	C11/A1	C11/A5	C11/A5	C11/A1	C11/A9
GPIF IO[27]	C10	C10	C10	C10	C10	C10/A2	C10/A6	C10/A6	C10/A2	C10/ A10
GPIF IO[26]	C9	C9	C9	C9	C9	C9/A3	C9/A7	C9/A7	C9/A3	C9/A11
GPIF IO[25]	C8	C8	C8	C8	C8	C8/A4	C8/A8	C8/A8	C8/A4	C8/A12
GPIF IO[24]	C7	C7	C7	C7	C7	C7/A5	C7/A9	C7/A9	C7/A5	C7/A13
GPIF IO[23]	C6	C6	C6	C6	C6	C6/A6	C6/A10	C6/A10	C6/A6	C6/A14
GPIF IO[22]	C5	C5	C5	C5	C5	C5/A7	C5/A11	C5/A11	C5/A7	C5/A15
GPIF IO[21]	C4	C4	C4	C4	C4	C4	C4	C4	C4	C4
GPIF IO[20]	C3	C3	C3	C3	C3	C3	C3	C3	C3	C3
GPIF IO[19]	C2	C2	C2	C2	C2	C2	C2	C2	C2	C2
GPIF IO[18]	C1	C1	C1	C1	C1	C1	C1	C1	C1	C1
GPIF IO[17]	C0	C0	C0	C0	C0	C0	C0	C0	C0	C0

Note:
(1) Depending on the configuration, one of the columns of the table will be selected

(2) Empty cells imply no connection
(3) Cx - Control line # x
(4) Ay - Address line #y
(5) Dz - Data line #z

Certain pins, such as the USB lines have specific electrical characteristics and are connected directly to the USB I/O-System. They do not have GPIO capability.

Pins belonging to the same block share the same group setting for drive strengths. Pins of a block overridden as GPIO share a different group setting for their drive strength. The drive strength of a pin cannot be controlled independent of the drive strength of other pins in the same group. FX3 provides software controlled pull up (50 kΩ) or pull down (10 kΩ) resistors internally on all digital I/O pins.

3.6.6 GPIF II

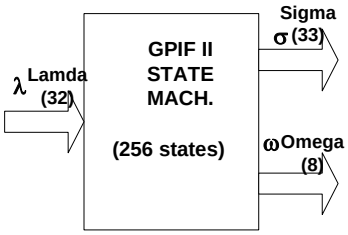
The FX3 device includes a GPIF II interface that provides an easy and glueless interface to popular interfaces such as asynchronous SRAM, synchronous SRAM, Address Data Multiplexed interface, parallel ATA, and so on. The interface supports up to 100 MHz. and has 45 programmable pins that are programmed by GPIF Waveform Descriptor as data, address, control, or GPIO function. For example, the GPIF II can be configured to a 16-bit ADM (Address/Data Multiplex), seven control signals, and seven GPIO.

The GPIF II interface features the following:

- The ability to play both the master and slave role, eliminating the need for a separate slave FIFO interface.
- A large state space (256 states) to enable more complex pipelined signaling protocols.
- A wider data path supporting 32-bit mode in addition to 16- and 8-bit.
- A deeper pipeline designed for substantially higher speed (more than 200 MHz internal clock frequency - “Edge Placement Frequency")
- High frequency I/Os with DLL timing correction (shared with other interface modes), enabling interface frequencies of up to 100 MHz.
- 45 programmable pins

The heart of the GPIF II interface is a programmable state machine.

Figure 3-12. State Machine



This state machine

- Supports up to 256 different states
- Monitors up to 32 input signals (lamda) to transition between states
- Provides for transition to two different states from the current state
- Can drive/control up to eight external pins of the device (omega)
- Can generate up to 33 different internal control signals (sigma)

The GPIF II is not connected directly to the USB endpoint buffers. Instead it is connected to FX3's internal DMA network. This enables the FX3's high-performance CPU to have more control over and

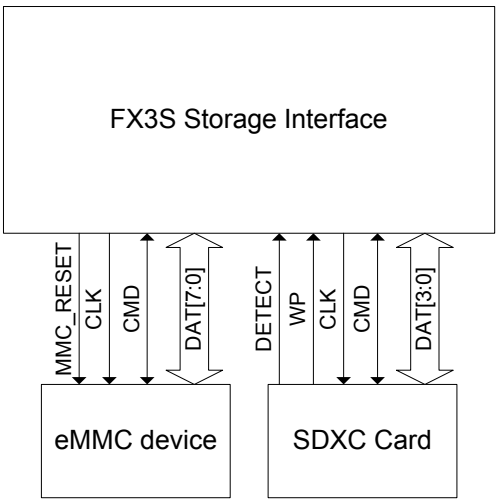
access to the data flows in the application thus enabling a wider range of applications, including ones that process, rather than just route, the actual data flows.

3.6.7 Storage Interface

The FX3S, SD3, Benicia, and Bay devices have an integrated mass storage controller that can connect to SD cards, eMMC devices or SDIO devices. The FX3S device provides two independent storage ports (S0 and S1), each of which can support the following peripheral types:

- SD cards up to SD specification version 3.0
- MMC cards or eMMC devices up to MMC System Specification version 4.41
- SDIO devices up to SDIO specification version 3.0

Figure 3-13. Storage Interface Connectivity on FX3S



The storage ports can be configured with an 8-bit or 4-bit wide data bus, and will support interface clock frequencies up to 104 MHz SDR or 52 MHz DDR. The storage ports do not support functioning in the legacy SPI mode.

The storage interface signals can be driven at 3.3 V or 1.8 V (low voltage operation) based on the voltage provided on the S0VDDQ or S1VDDQ input. When connecting to an SD 3.0 card, the interface needs to start operating at 3.3 V, and then switch to 1.8 V during the initialization sequence. Please refer to the SD Specifications Part 1, Version 3.00 for details of the initialization procedure.

The storage controller block on the FX3S device supports sending of any command with custom parameters, and receiving arbitrary lengths of device response. The data interface is connected to the DMA fabric, and all data transfers need to be performed through DMA sockets. The storage controller provides a maximum of 8 DMA sockets, each of which can be used with any of the storage ports.

The storage controller can be configured to transfer one or more blocks of data with arbitrary block lengths. However, it is expected that the data block length will be a multiple of 8 bytes.

The interface clock frequency is divided down from the master system clock (384 MHz or 416 MHz) through a set of dividers. The block supports clock frequencies ranging from 400 kHz to 104 MHz SDR (or 52 MHz DDR).

The storage controller supports stopping the interface clock to reduce power consumption at times when the interface is not in use. An Auto stop clock feature is supported to prevent data overflows

during read operations. The clock will automatically be stopped when the internal buffer is full, and will be restarted when a buffer is made available.

The storage controller supports detection of card insertion or removal through one of two mechanisms:

- Voltage change on the DAT[3] pin. This method requires a pull-down on the DAT[3] to be present on the board. This method is supported on both storage ports of the FX3S device.
- Voltage change on a dedicated GPIO pin connected to a micro-switch on the card socket. A single pin is provided for this purpose, and hence this method can only be used for one of the storage ports.

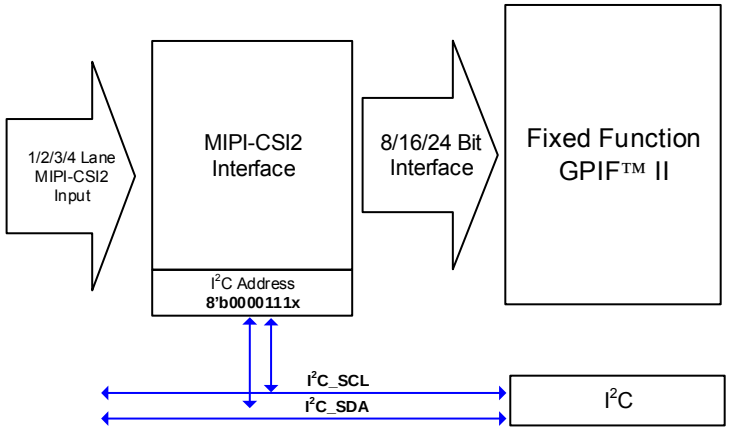
Dedicated GPIO pins are provided for resetting the eMMC devices and for checking the write protect status of the storage devices. These pins are operated under firmware control, and do not directly affect the storage port operation.

The storage controller supports SDIO specific features such as SDIO Interrupt detection and the SDIO read-wait and suspend-resume features, as specified in the SDIO specification Version 2.00.

3.6.8 MIPI-CSI2 Interface

The CX3 device has an integrated MIPI-CSI2 receiver, which is mated to the GPIF and can connect to an image sensor supporting MIPI-CSI2. The interface allows for up to four MIPI-CSI2 data lanes, capable of speeds up to 1 Gbps per lane. The MIPI-CSI2 receiver is connected to a fixed-function GPIF controller via 8/16/24 bit bus, which can be clocked at up to 100 MHz.

Figure 3-14. CSI-2 Interface on CX3 Device

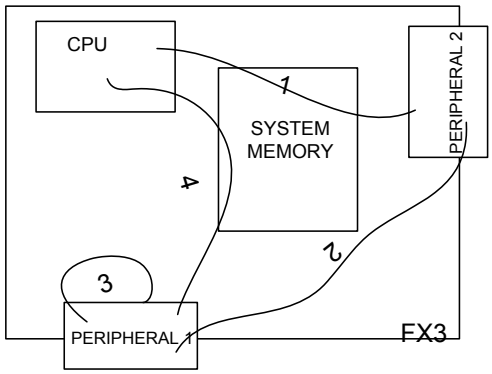


The MIPI-CSI2 receiver supports a wide variety of image formats including RAW8/10/12/14, YUV422, and RGB888/666/565.

Configuration of the MIPI-CSI2 receiver interface is done over the I2C bus. The MIPI-CSI2 receiver is available at the 7-bit I2C slave address 8'b0000111X (Read Address 0X0F; Write Address 0x0E). APIs to configure the MIPI-CSI2 interface and for selecting the GPIF interface width are provided as part of the FX3 SDK.

3.7 DMA Mechanism

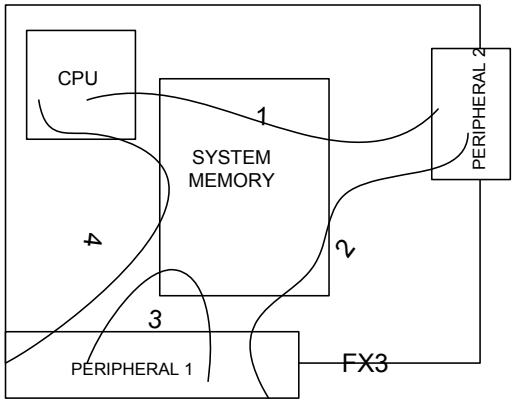
Figure 3-15. DMA Mechanism

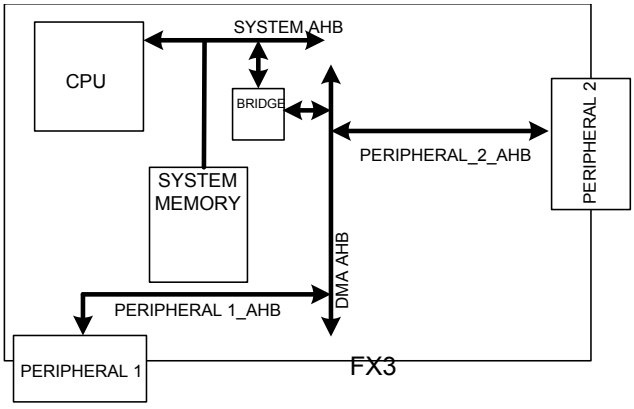


Non-CPU intervened data chunk transfers between a peripheral and CPU or system memory, between two different peripherals or between two different gateways of the same peripheral, loop back between USB end points, are collectively referred to as DMA in FX3.

Figure 3-15 shows a logical paths of data flow; however, in practice, all DMA data flows through the System memory, as shown in the following figure.

Figure 3-16. System Memory

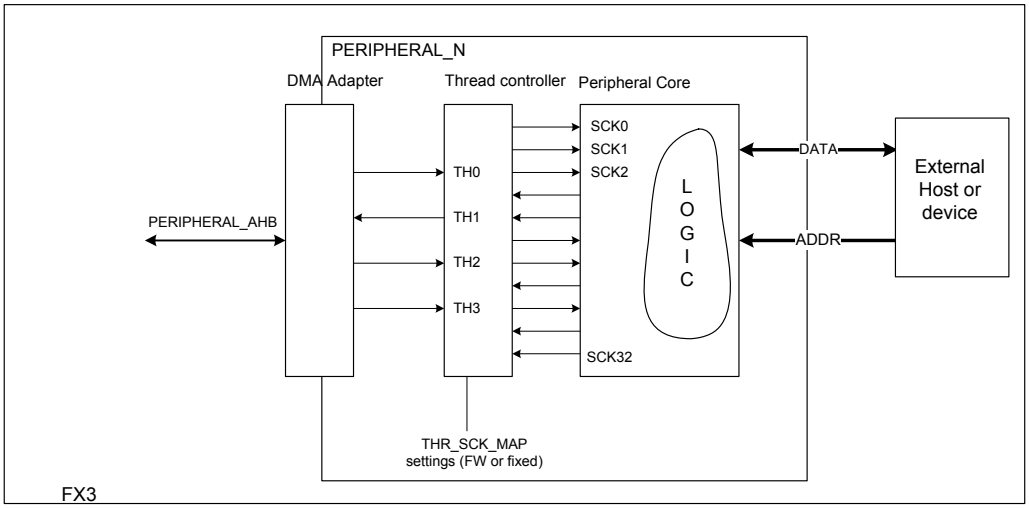




As explained earlier, the CPU accesses the System Memory (Sysmem) using the System AHB and the DMA paths of the peripherals are all hooked up to the DMA AHB. Bridge(s) between the System bus and the DMA bus are essential in routing the DMA traffic through the Sysmem.

The following figure illustrates a typical arrangement of the major DMA components of any DMA capable peripheral of FX3.

Figure 3-17. DMA Components

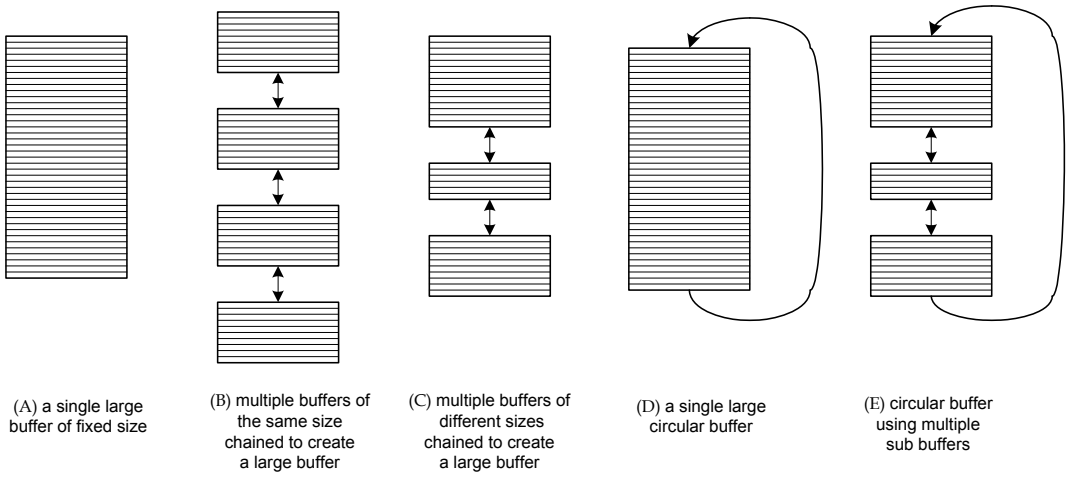


The figure shows a partition of any FX3 peripheral from a DMA view point. Any FX3 peripheral can be split into three main components - DMA adapter, DMA thread controller, and peripheral core. The peripheral attaches to the DMA fabric using AHB, width of which determines the throughput of the peripheral. The peripheral core implements the actual logic of the peripheral (I^2C, GPIF, and USB). Data transfers between the peripheral core and the external world is shown to happen over two buses - address and data. These buses symbolize that the peripherals of FX3 do not present themselves as a single large source or sink for data; rather the external host (or device) gets to index the data to/from different addresses.

In practice though, physical address and data buses may not exist for all peripherals. Each peripheral has its own interface to communicate addresses and data. For example, all information exchanges in the USB block happen over the D+ and D– lines. The core logic extracts the address from the token packet and data from the data packet. The I^2C interface exchanges all information over the SDA lines synchronized to a clock on the SCL line. The GPIF interface on the other hand can be configured to interface using a separate physical address and data bus.

The address specified by the external host (or device) is used to index data from/into one of the many entities called 'Sockets' which logically present themselves, at the interface, as a chain of buffers. The buffer chains themselves can be constructed in one of a several possible ways depending on the application.

Figure 3-18. DMA Configurations



The buffers do not reside in the peripherals; they are created (memory allocated) in the Sysmem area and a pointer is made available to the associated socket. Therefore any DMA access to the Sysmem needs to go through the DMA and the System AHB, the widths of which directly determine the DMA bandwidth.

Certain peripherals may contain tens of sockets, each associated with different buffer chains. To achieve reasonably high bandwidth on DMA transfers over sockets while maintaining reasonable AHB bus width, Socket requests for buffer reads/writes are not directly time multiplexed; rather the sockets are grouped into threads, the requests of which are time multiplexed. Only one socket mapped to a thread can actively execute at any instant of time. The thread handling the socket group needs to be reconfigured every time a different socket needs to execute. The thread-socket relations are handled by the thread controller. The DMA adapter block converts read/write queries on the threads to AHB requests and launches them on to the AHB. A single thread controller and DMA adapter block may be shared by more than one peripheral.

A socket of a peripheral can be configured to either write to or read from buffers, not both. As a convention, sockets used to write into system buffers are called 'Producers' and the direction of data flow in this case is referred to as 'Ingress'. Conversely, sockets used to read out system buffers are called 'Consumers' and the direction of data flow in this case is referred to as 'Egress'. Every socket has a set of registers, which indicate the status of the socket such as number of bytes transferred over the socket, current sub buffer being filled or emptied, location of the current buffer in memory, and no buffer available.

A DMA transfer in FX3 can be envisioned as a flow of data from a producer socket on one peripheral to a consumer socket on the other through buffers in the Sysmem. Specific DMA instructions called 'Descriptors' enable synchronizing between the sockets. Every buffer created in the Sysmem has a descriptor associated with it that contains essential buffer information such as its address, empty/full status and the next buffer/descriptor in chain. Descriptors are placed at specific locations in the Sysmem which are regularly monitored and updated by the peripherals' sockets.

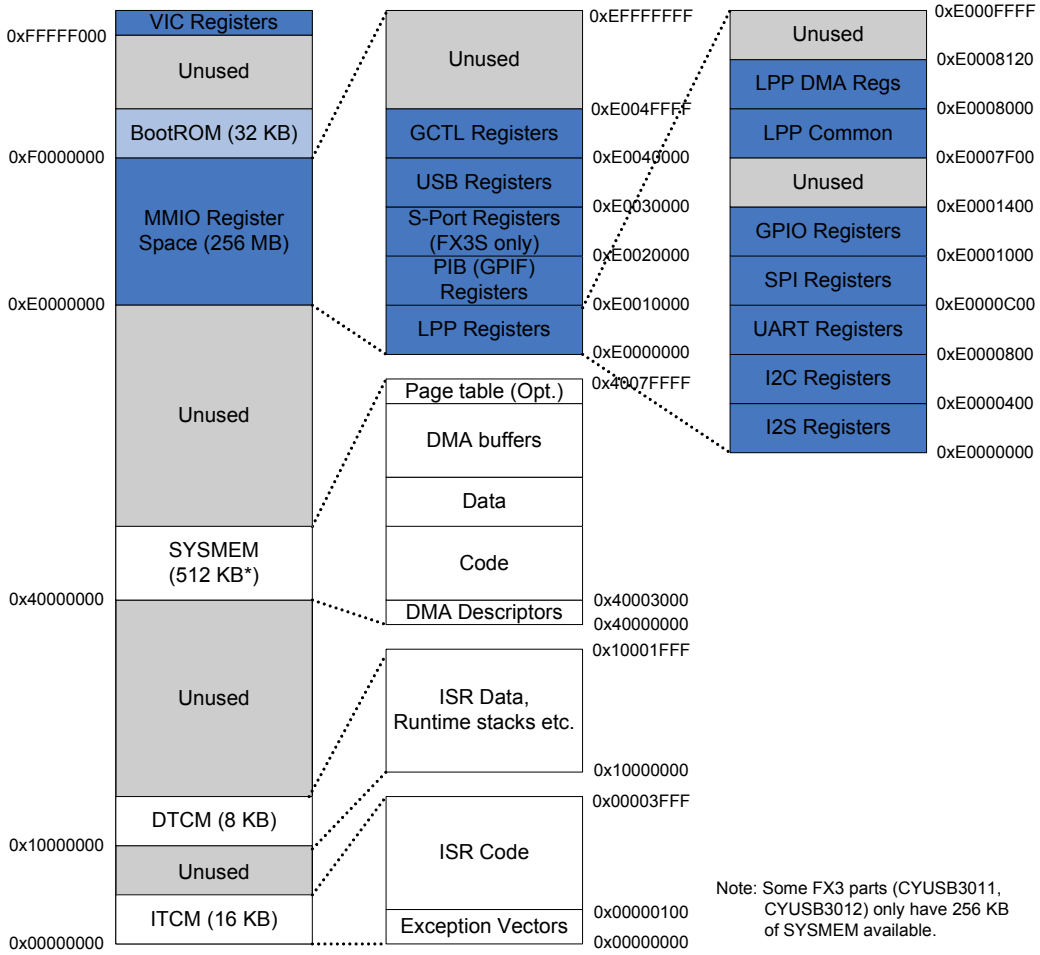
In a typical DMA transaction, the data source-peripheral sends data over the producing socket, which fills the current buffer of the socket. Upon the last byte of the producer's current buffer being written, the producer socket updates the descriptor of its current buffer and loads the next buffer in the chain. This process goes on until either the buffer chain ends or the next buffer in the chain is not free (empty to produce into). The corresponding consumer socket waits until the buffer it points to becomes available (gets filled to consume from). When available, the data destination-peripheral is notified to start reading the buffer over the consumer socket. When the last byte of the current buffer

is read, the consumer socket updates the descriptor of its current buffer and loads the next buffer in the chain. This process goes on until either the buffer chain ends or the next buffer in the chain is not available. The producer and consumer sockets only share the buffer/descriptor chain. It is not necessary for them to be executing the same descriptor in the chain at any given instant.

In non-conventional DMA transactions, the producer and consumer sockets can be two different sockets on the same peripheral. In some cases (for example, reading F/W over I^2C), there is no destination peripheral or consumer socket. In this case, the CPU is assumed to be the consumer. Conversely, the CPU can act as a producer to a consumer socket on the destination peripheral (for example, printing debug messages over UART).

3.8 Memory Map and Registers

Figure 3-19. Memory Map



The ARM PL192 VIC is located outside the regular MMIO space, on top of the BootROM at location FFFFF000. This conforms to the PL192 TRM, and facilitates single instruction jump to ISR from the vector table at location 00000000 for IRQ and FIQ. For more details see ARM PL192 TRM.

A peripheral of FX3 typically contains the following registers

- ID register
- Clock, power config register

- Peripheral specific config registers
- Interrupt enable, mask an status registers
- Error registers
- Socket registers for sockets associated with the peripheral

3.9 Reset, Booting, and Renum

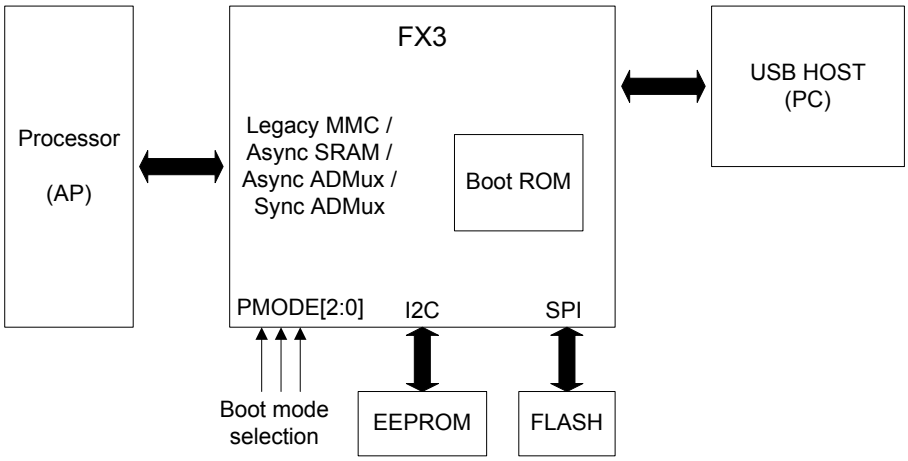
Resets in FX3 are classified into two: hard reset and soft reset.

A power on reset (POR) or a Reset# pin assertion initiates a hard reset. Soft Reset, on the other hand, is initiated by the firmware writing to the control registers.

Soft Resets are of two types - CPU or Warm Reset and Device Reset

CPU or warm reset involves resetting the CPU Program Counter without affecting any of the hardware blocks such as USB or GPIF-II. Selected blocks may be reset by the firmware as required. Firmware does not need to be reloaded following a CPU Reset. Device Reset is identical to Hard Reset. The firmware must be reloaded following a Device Reset.

Figure 3-20. Reset



FX3's flexible firmware loading mechanism allows code to be loaded from a device connected on the I2C bus (an EEPROM for example), an SPI-based flash device, or from a PC acting as a USB host or from an application processor (AP) connected on the flexible GPIF.

Every reset causes the device to start executing the bootloader code stored in the internal BootROM. On a full device reset, the BootROM identifies the desired boot source by checking the state of the PMODE pins. It then initializes the appropriate interface block (GPIF, I2C, SPI, or USB), and then tries to obtain the user firmware image. In I2C and SPI cases, the bootloader tries to read address 0 on the peripheral device, and checks whether a valid firmware header is received. If a valid header is detected, the bootloader continues the boot process and transfers control to the newly loaded application. In the case of GPIF or USB boot, the bootloader waits for the user to initiate a boot operation.

In some cases, an intelligent processor connected to the GPIF of FX3 may also be used to download the firmware for FX3. The processor can write the firmware bytes using the Sync ADMux, Async ADMux, or the Async SRAM protocol. In addition, the AP can also use the MMC initialization and write commands (PMMC legacy) to download firmware over its interface with FX3.

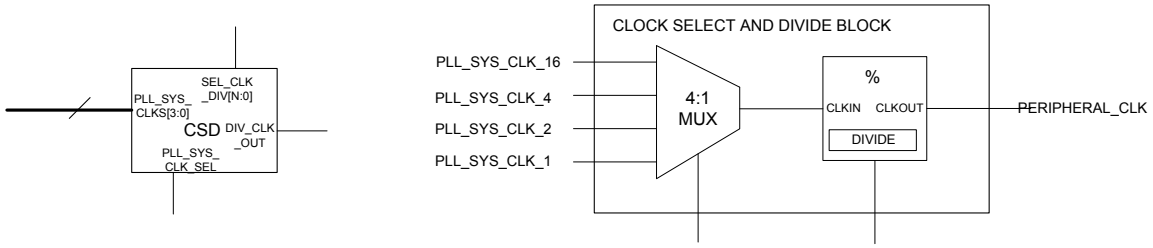
Alternatively, the user may simply wish to download the code from a PC using a USB cable. When FX3 is configured for USB boot, the boot loader first enables FX3's USB block to accept vendor commands over a bulk end point. When plugged into the host, the device enumerates with a Cypress default VID and PID. The actual firmware is then downloaded using Cypress drivers. If required, the firmware can then reconfigure the USB block (such as change the IDs and enable more end points) and simulate a disconnect-event (soft disconnect) of the FX3 device from the USB bus. On soft reconnect, FX3 enumerates with the new USB configuration. This two-step patented process is called renumeration.

The boot loader is also responsible for restoring the state of FX3 when the device transitions back from a low power mode to the normal active power mode. The process of system restoration is called 'Warm boot'.

3.10 Clocking

Clocks in FX3 are generated from a single 19.2 MHz (±100 ppm) crystal oscillator. It is used as the source clock for the PLL, which generates a master clock at frequencies up to up to 500 MHz. Four system clocks are obtained by dividing the master clock by 1, 2, 4, and 16. The system clocks are then used to generate clocks for most peripherals in the device through respective clock select and divide (CSD) block A CSD block is used to select one of the four system clocks and then divide it using the specified divider value. The depth and accuracy of the divider is different for different peripherals,

Figure 3-21. CSD



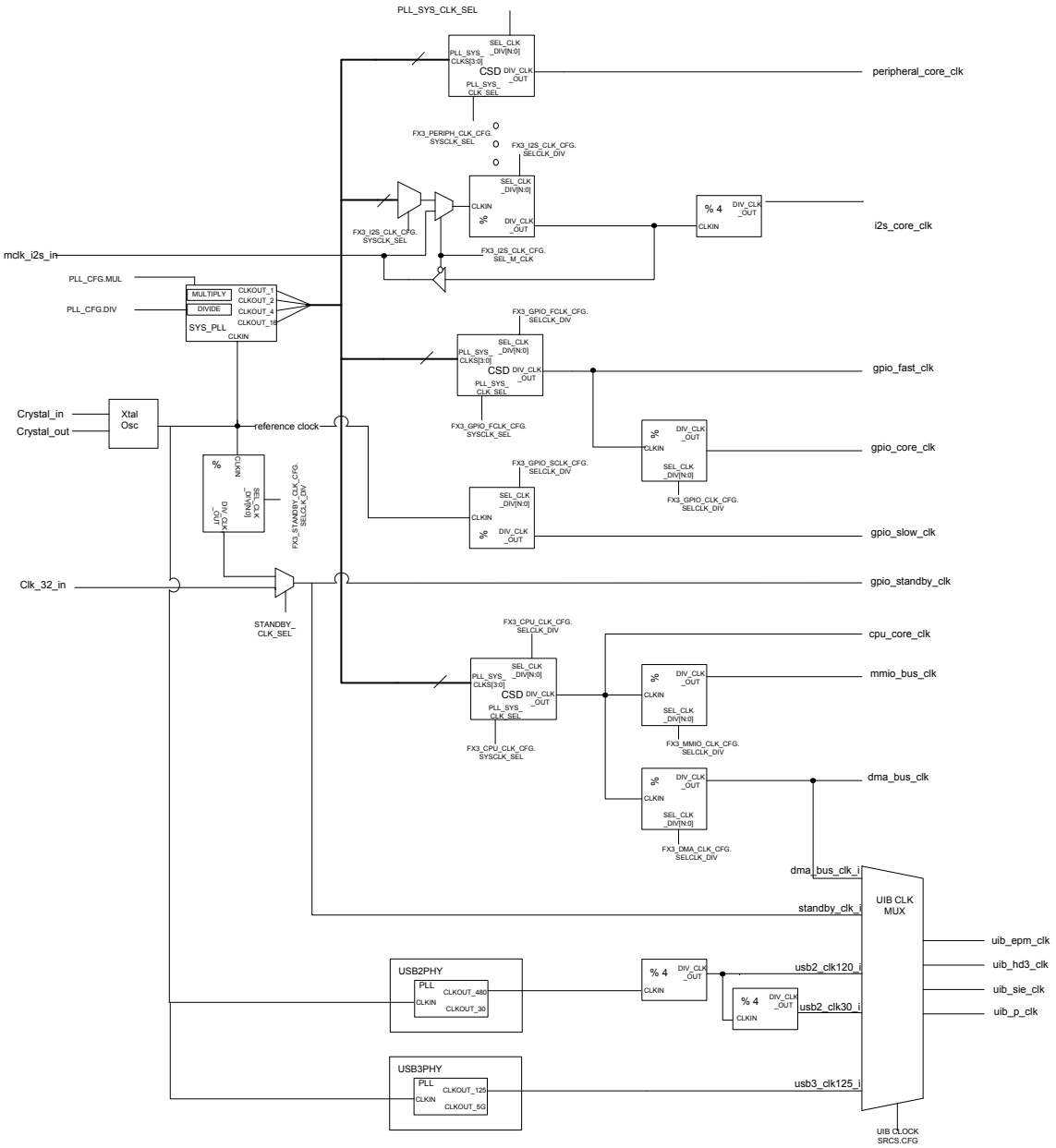
The CPU clock is derived by selecting and dividing one of the four system clocks by an integer factor anywhere between 1 and 16. The bus clocks are derived from the CPU clock. Independent 4-bit dividers are provided for both the DMA and MMIO bus clocks. The frequency of the MMIO clock, however, must be an integer divide of the DMA clock frequency.

A 32 kHz external clock source is used for low-power operation during standby. In the absence of a 32 kHz input clock source, the application can derive this from the reference clock produced by the oscillator.

Certain peripherals deviate from the general clock derivation strategy. The fast clock source of GPIO is derived from the system clocks using a CSD. The core clock is a fixed division of the fast clock. The slow clock source of GPIO is obtained directly from the reference clock using a programmable divider. The standby clock is used to implement 'Wake-Up' on GPIO. The I2S block can be run off an internal clock derived from the system clocks or from an external clock sourced through the I2S_MCLK pin of the device.

Exceptions to the general clock derivation strategy are blocks that contain their own PLL, because they include a PHY that provides its clock reference. For example, the UIB block derives its 'epm_clk', 'sie_clk', 'pclk', and 'hd3_clk' using the standby clock, dma-bus clock, 30/120-MHz clock from the USB2 PHY, and a 125-MHz clock from the USB3PHY. The sie_clk runs the USB 2.0 logic blocks while USB 3.0 logic blocks are run using the pclk. The hd3_clk runs certain host and USB 3.0 logic blocks while the epm_clk runs the end point manager.

Figure 3-22. Clock Generation Structure

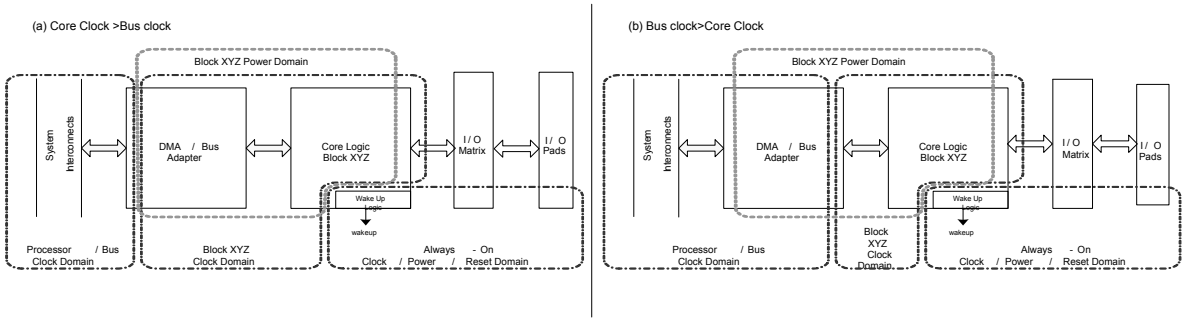


Note……
UART baud rate is 1/8th of uart_core_clk……..
I2C operating frequency is 1/10 th of I2C core clock
SPI operating frequency is ½ of spi core clock…….

The CPU, DMA, and MMIO clock domains are synchronous to each other. However, every peripheral assumes its core clock to be fully asynchronous from other peripheral core clocks, the computing clock or the wakeup clock.

If the core (peripheral) clock is faster than the bus clock, the DMA adapter for the block runs in the core clock domain. The DMA adapter takes care of the clock crossing on it's interconnect side. If the core clock is slower than the bus clock, the DMA adapter for that block runs in the bus clock domain. The DMA adapter takes care of the clock crossing on its core IP side.

Figure 3-23. DMA Adaptor



3.11 Power

3.11.1 Power Domains

Power supply domains in FX3 can be mainly classified in four - Core power domain, Memory power domain, I/O power domain, and Always On power domain. The I/O power domain is further divided into five power domains.

Core power domain encompasses a large section of the device including the CPU, peripheral logic, and the interconnect fabric. The system SRAM memory resides in the Memory power domain. I/O logic dwell in their respective peripheral I/O power domain (I^2C I/O power domain, I2S-UART I/O-SPI I/O-GPIO power domain, Clock I/O power domain, USB I/O power domain, and Processor Port I/O power domain). The Always On power domain hosts the power management controller, different wake-up sources and their associated logic.

Wake-up sources force a system in suspend or standby state to switch to the normal power operation mode. These are distributed across peripherals and configured in the 'Always On global configuration block'. Some of them include level match on level-sensitive wake-up I/Os, toggle on edge sensitive wake-up I/Os, activity on the USB 2.0 data lines, OTG ID change, LFPS detection on USB 3.0 RX lines, USB connect event, and watchdog timer – timeout event.

The 'Always On global configuration block' runs off the standby clock and will be turned off only if the device is powered off.

3.11.2 Power Management

At any instant, FX3 is in one of the four power modes - normal, suspend, standby, or core power down. In a typical scenario, when FX3 is actively executing its tasks, the system is in normal mode. The usual clock gating techniques in peripherals minimize the overall power consumption.

On detecting prolonged periods of inactivity, the chip can be forced to enter the suspend mode. All ongoing port (peripheral) activities are wrapped up, ports disabled, and wake up sources are set before entering the suspend state. In applications involving USB 3.0, the USB3 PHY is forced into the U3 state. USB2PHY, if used, is forced into suspend. The System RAM transitions to a low power stand by state; read and write to RAM cannot be performed. The CPU is forced into the halt state. The ARM core will retain its state, including the Program Counter inside the CPU. All clocks except the 32-kHz standby are turned off by disabling the System PLL is through the global configuration block. In the absence of clocks, the I/O pins can retain their state as long as the I/O power domain is not turned off The INT# pin can be configured to indicate FX3's presence in low power mode.

Further reduction in power is achieved by forcing FX3 into stand-by state where, in addition to disabling clocks, the core power domain is turned off. As in the case of suspend, I/O states of powered peripheral I/O domains are frozen and ports disabled. Essential configuration registers of

logic blocks are first saved to the System RAM. Following this, the System RAM itself is forced into low power memory retention only mode. Warm boot setting is enabled in the global configuration block. Finally the core is powered down. When FX3 comes out of standby, the CPU goes through a reset; the boot-loader senses the warm boot mode and restores the system to its original state after loading back the configuration values (including the firmware resume point) from the System RAM.

Optionally, FX3 can be powered down (core power down) from its Standby mode. This involves an additional process of removing power from the VDD pins. Contents of System SRAM are lost and I/O pins go undefined. When power is re-applied to the VDD pins, FX3 will go through the normal power on reset sequence.

4. FX3 Software

Cypress's EZ-USB FX3 is the next generation USB 3.0 peripheral controller. This is a highly integrated and flexible chip which enables system designers to add USB 3.0 capability to any system. The FX3 comes with easy-to-use EZ-USB tools providing a complete solution for fast application development.

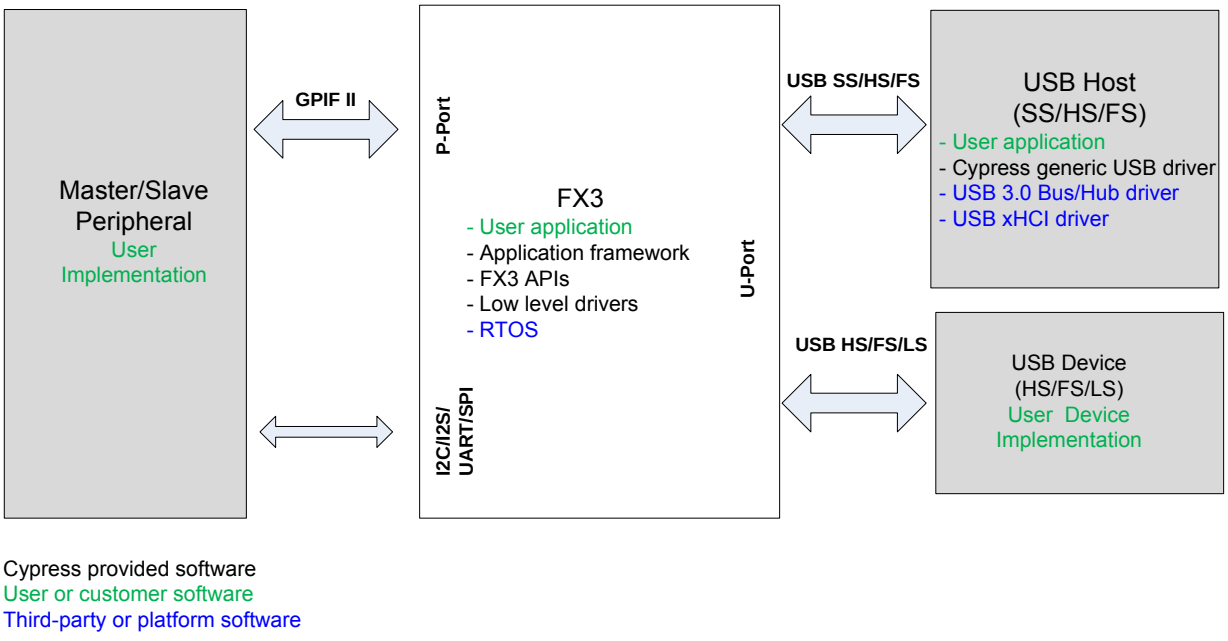
Cypress EZ-USB FX3 is a user-programmable device and is delivered with a complete software development kit.

4.1 System Overview

Figure 4-1illustrates the programmer's view of FX3. The main programmable block is the FX3 device. The FX3 device can be set up to:

- Configure and manage USB functionality such as charger detection, USB device/host detection, and endpoint configuration
- Interface to different master/slave peripherals on the GPIF interface
- Connect to serial peripherals (UART/SPI/GPIO/I^2C/I2S)
- Set up, control, and monitor data flows between the peripherals (USB, GPIF, and serial peripherals)
- Perform necessary operations such as data inspection, data modification, header/footer information addition/deletion

Figure 4-1. Programming View of FX3



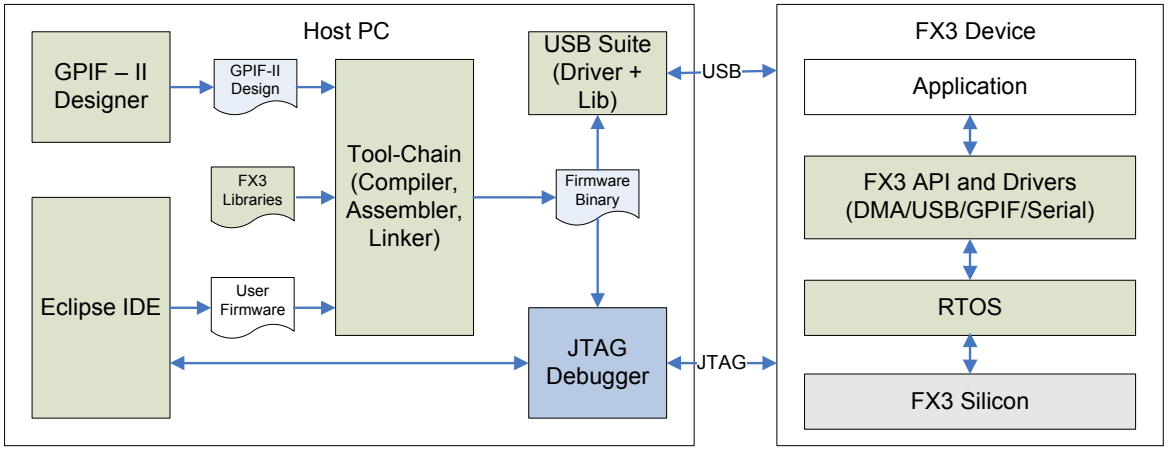
The two other important entities that are external to the FX3 are

- USB host/device
  - When the FX3 is connected to a USB host, it functions as a USB device. The FX3 enumerates as a super-speed, high-speed, or full-speed USB peripheral corresponding to the host type.
  - When a USB device is connected, the FX3 plays the role of the corresponding high-speed, full-speed or low-speed USB host.
- GPIF II master/slave: GPIF II is a fully configurable interface and can realize any application specific protocol as described in GPIF™ II Designer on page 137. Any processor, ASIC, DSP, or FPGA can be interfaced to the FX3. FX3 bootloader or firmware configures GPIF II to support the corresponding interface.

4.2 FX3 Software Development Kit (SDK)

The FX3 comes with a complete software development solution as illustrated in the following figure.

Figure 4-2. FX3 SDK Components



4.3 FX3 Firmware Stack

Powerful and flexible applications can be rapidly built using the FX3 firmware framework and FX3 API libraries.

4.3.1 Firmware Framework

The firmware (or application) framework has all the startup and initialization code. It also contains the individual drivers for the USB, GPIF, and serial interface blocks. The framework:

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

4.3.2 Firmware API Library

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

4.3.3 FX3 Firmware Examples

Various firmware (application) examples are provided in the FX3 SDK. These examples are provided in source form. These examples illustrate the use of the APIs and the firmware framework, putting together a complete application. The examples illustrate the following:

- Initialization and application entry
- Creating and launching application threads
- Programming the peripheral blocks (USB, GPIF, serial interfaces)
- Programming the DMA engine and setting up data flows
- Registering callbacks and callback handling
- Error handling
- Initializing and using the debug messages

The examples include:

- USB loop examples (using both bulk and isochronous endpoints)
- UVC (USB video class implementation)
- USB source/sink examples (using both bulk and isochronous endpoints)
- USB Bulk streams example
- Serial Interface examples (UART/I^2C/SPI/GPIO)
- Slave FIFO (GPIF-II) examples
- USB mass storage examples

4.4 FX3 Host Software

A comprehensive host side (Microsoft Windows) stack is included in the FX3 SDK. This stack includes the Cypress generic USB 3.0 driver, APIs that expose the driver interfaces, and application examples. Each of these components are described in brief in this section. Detailed explanations are presented in FX3 Host Software chapter on page 135.

4.4.1 Cypress Generic USB 3.0 Driver

A generic kernel mode (WDF) driver is provided on Windows 7 (32/64-bit), Windows Vista (32/64-bit), and Windows XP (32 bit only). This driver interfaces to the underlying Windows bus driver or a third party driver and exposes a non-standard IOCTL interface to an application.

4.4.2 Convenience APIs

These APIs (in the user mode) expose generic USB driver interfaces through C++ and C# interfaces to the application. This allows the applications to be developed with more intuitive interfaces in an object oriented fashion.

4.4.3 USB Control Center

This is a Windows utility that provides interfaces to interact with the device at low levels such as selecting alternate interfaces and data transfers. This can also be used to download a firmware binary to the FX3 device RAM, and to program the binary on to an EEPROM or SPI flash device.

4.4.4 Bulkloop

This is a windows application to perform data loop back on Bulk endpoints.

4.4.5 Streamer

This is a windows application to perform data streaming over Isochronous, bulk on interrupt end-points. This application can be used to measure the transfer throughput capabilities of the FX3 device as well as the USB host controller.

4.5 FX3 Development Tools

FX3 is a device with open firmware framework and driver level APIs allowing the customer to develop firmware that matches the application. This approach requires ARM code development and debug environment.

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

4.5.1 Firmware Development Environment

The firmware development environment helps to develop, build, and debug firmware applications for FX3. The third party ARM software development tool provides an integrated development environment (IDE) with compiler, linker, assembler, and JTAG debugger.

4.5.2 GPIF II Designer

GPIF II Designer is a Windows application provided to FX3 customers as part of the FX3 SDK. The tool provides a graphical user interface to allow customers to intuitively specify the necessary interface configuration appropriate for their target environment. The tool generates firmware code that eventually gets built into the firmware.

GPIF II Designer generates configurations and state machine descriptors for GPIF II interface module. The tool provides an user interface to express the users' design in the form of a state machine. In addition, the user can traverse through the state machine and observe the timing diagram for all of the external GPIF-II signals.

5. FX3 Firmware

The chapter presents the programmers overview of the FX3 device. The boot and the initialization sequence of the FX3 device is described. This sequence is handled by the firmware framework. A high level overview of the API library is also presented, with a description of each programmable block.

5.1 Initialization

The system initialization sequence sets up the CPU sub-system, initializes the firmware framework, and sets up other modules of the firmware framework. It is the initialization point for the RTOS.

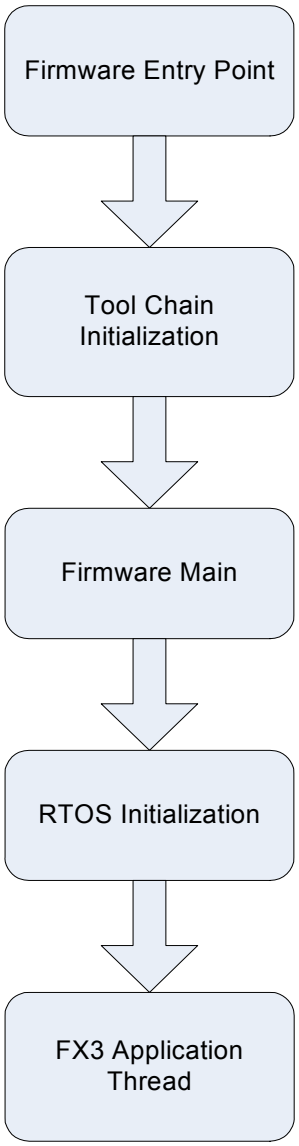
The following high level activities are handled as part of the initialization sequence.

- **Device configuration**: The FX3 boot mode and GPIF startup I/O interface configuration is determined by the PMODE pins. The I/O ports (USB, GPIF, and serial interfaces) are set up according to the device type and the internal I/O matrix is configured accordingly.
- **Clock setup**: The firmware framework sets the CPU clock at startup.
- **MMU and cache management**: The FX3 device does not support virtual memory. The FX3 device memory is a one to one mapping from virtual to physical addresses. This is configured in the MMU. The device MMU is enabled to allow the use of the caches in the system. By default, the caches are disabled and invalidated on initializing the MMU.
- **Stack initialization**: The stacks needed for all modes of operation for the ARM CPU (System, Supervisor, FIQ, IRQ) are set up by the system module.

  For all user threads, the required stack space must be allocated prior to thread creation. Separate APIs are provided to create a runtime heap and to allocate space from the heap.
- **Interrupt management**: The FX3 device has a vectored interrupt controller. Exception vectors and VIC are both initialized by this module. The exception vectors are in the I-TCM and are located from address 0x0 (refer to memory map).

The actual initialization sequence is shown in the following figure:

Figure 5-1. Initialization Sequence



1. The execution starts from the firmware image entry point. This is defined at the compile time for a given FX3 firmware image. This function initializes the MMU, VIC, and stacks.
2. The second step in the initialization sequence is the Tool Chain init. This is defined by the tool chain used to compile the FX3 firmware. Because the stack setup is complete, this function is only expected to initialize any application data.
3. The main() function, which is the C programming language entry for the firmware, is invoked next. The FX3 device is initialized in this function.
4. The RTOS kernel is invoked next from the main(). This is a non-returning call and sets up the Threadx kernel.
5. At the end of the RTOS initialization, all the driver threads are created.
6. In the final step, FX3 user application entry is invoked. This function is provided to create all user threads.

5.1.1 Device Boot

The boot operation of the device is handled by the boot-loader in the boot ROM. On CPU reset, the control is transferred to boot-ROM at address 0xFFFF0000.

For cold boot, download the firmware image from various available boot modes of FX3. The bootloader identifies the boot source from the PMODE pins and loads the firmware image into the system memory (SYS_MEM). The firmware entry location is read by the bootloader from the boot image and is stored at address 0x40000000 by the boot-loader at the time of cold boot.

The boot options available for the FX3 device are:

- USB boot
- I^2C boot. SPI Boot
- GPIF boot (where the GPIF is configured to be Async SRAM, Sync/Async ADMUX)

In case of warm boot or wake-up from standby, the boot-loader simply reads the firmware entry location (stored at the time of cold boot) and transfers control to the firmware image that is already present in the system memory.

5.1.2 FX3 Memory Organization

The FX3 device has the following RAM areas:

1. 512 KB of system memory (SYS_MEM) [0x40000000: 80000] – This is the general memory available for code, data and DMA buffers. The first 12KB is reserved for boot/DMA usage. This area should never be used.
2. 16KB of I-TCM [0x00000000: 4000] – This is instruction tightly coupled memory which gives single cycle access. This area is recommended for interrupt handlers and exception vectors for reducing interrupt latencies. The first 256 bytes are reserved for ARM exception vectors and this can never be used.
3. 8KB of D-TCM [0x10000000: 2000] – This is the data tightly coupled memory which gives single cycle data accesses. This area is recommended for RTOS stack. Data cannot be placed here during load time

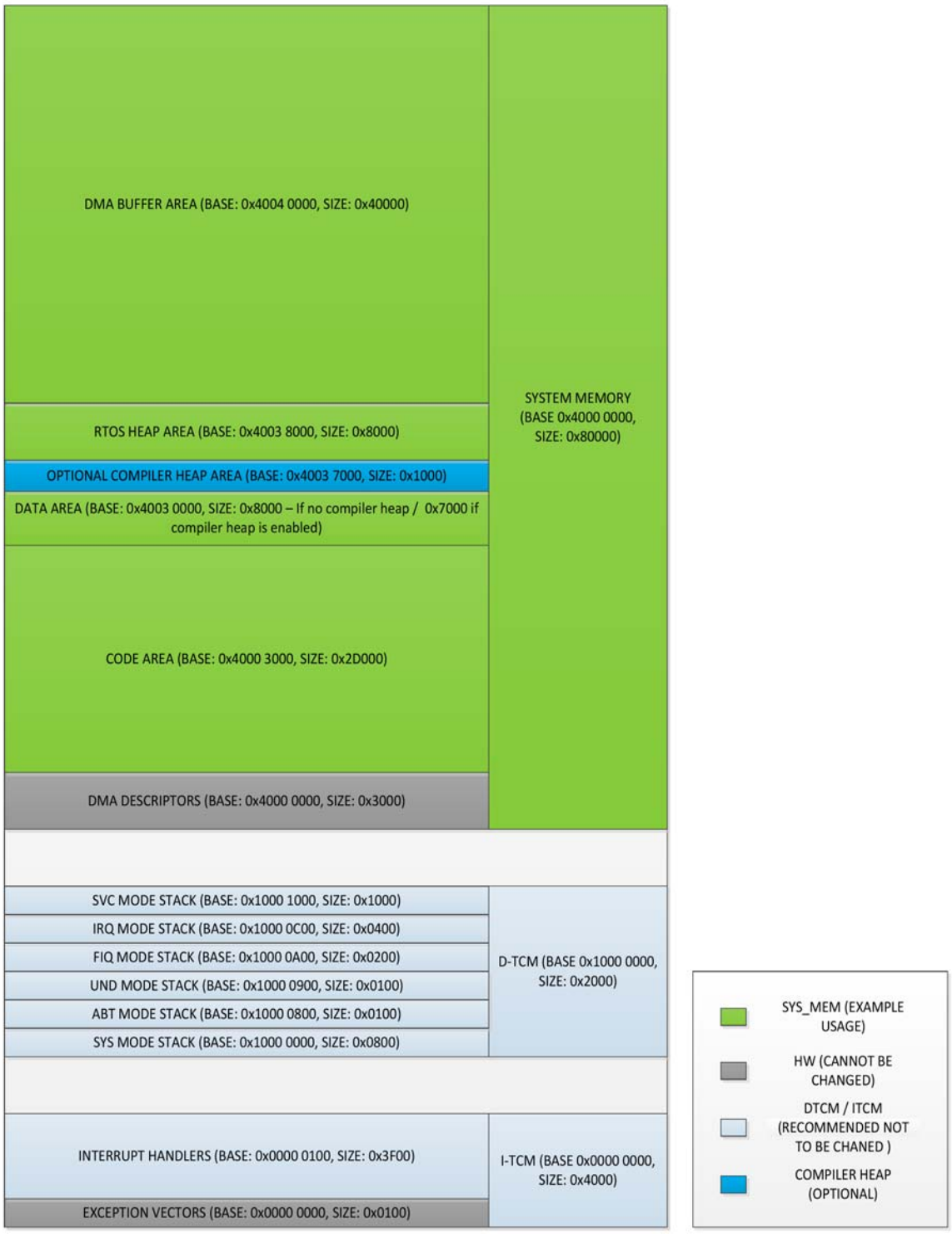
The memory requirements for the system can be classified as the following:

1. CODE AREA: All the instructions including the RTOS.
2. DATA AREA: All uninitialized and zero-initialized global data/constant memory. This does not include dynamically allocated memory.
3. STACK AREA: There are multiple stacks maintained: Kernel stacks as well as individual thread stacks. It is recommended to place all kernel stacks in D-TCM. The thread stacks can be allocated from RTOS heap area.
4. RTOS/LIBRARY HEAP AREA: All memory used by the RTOS provided heap allocator. This is used by CyU3PMemInit(), CyU3PMemAlloc(), CyU3PMemFree().
5. DMA BUFFER AREA: All memory used for DMA accesses. All memory used for DMA has to be 16 byte multiple. If the data cache is enabled, then all DMA buffers have to be 32 byte aligned and a multiple of 32 byte so that no data is lost or corrupted. This is used by the DMABuffer functions: CyU3PDmaBufferInit(), CyU3PDmaBufferAlloc(), CyU3PDmaBufferFree(), CyU3PDmaBufferDeInit().

5.1.3 FX3 Memory Map

The following figure shows a typical memory map for the FX3 device.

Figure 5-2. FX3 Memory Map



A linker script file is used to provide the memory map information to the GNU linker. The following example is from the linker script file distributed with the FX3 SDK (fx3.ld):

```
ENTRY(CyU3PFirmwareEntry);
```

```
MEMORY
{
     I-TCM: ORIGIN = 0x100LENGTH = 0x3F00
     SYS_MEM: ORIGIN = 0x40003000LENGTH = 0x2D000
     DATA: ORIGIN = 0x40030000LENGTH = 0x8000
}

SECTIONS
{
     .vectors :
     {
             *(CYU3P_ITCM_SECTION);
     } >I-TCM

     .text :
     {
             *(.text)
             *(.rodata*)
             *(.constdata)
             *(.emb_text)
             *(CYU3P_EXCEPTION_VECTORS);
              _etext = .;
     } > SYS_MEM

     .data :
     {
             _data = .;
             *(.data*)
             * (+RW, +ZI)
             _edata = .;
     } > DATA

     .bss :
     {
             _bss_start = .;
             *(.bss)
     } >DATA
     . = ALIGN(4);
     _bss_end = . ;

     .ARM.extab :
     {
             *(.ARM.extab* .gnu.linkonce.armextab.*)
     } > DATA

     __exidx_start = .;
     .ARM.exidx :
     {
             *(.ARM.exidx* .gnu.linkonce.armexidx.*)
     } > DATA
     __exidx_end = .;
}
```

The contents of each section of the memory map are explained below.

5.1.3.1 I-TCM

All instructions that are recommended to be placed under I-TCM are labeled under section CYU3P_ITCM_SECTION. This contains all the interrupt handlers for the system. If the application requires to place a different set of instructions it is possible. Only restriction is that first 256 bytes are reserved.

5.1.3.2 D-TCM

SVC, IRQ, FIQ and SYS stacks are recommended to be located in the D-TCM. This gives maximum performance from RTOS. These stacks are automatically initialized by the library inside the CyU3PFirmwareEntry location with the following allocation:

```
SYS_STACK       Base: 0x10000000 Size 2KB
ABT_STACK       Base: 0x10000800 Size 256B
UND_STACK       Base: 0x10000900 Size 256B
FIQ_STACK       Base: 0x10000A00 Size 512B
IRQ_STACK       Base: 0x10000C00 Size 1KB
SVC_STACK       Base: 0x10001000 Size 4KB
```

If for any reason the application needs to modify this, it can be done before invoking CyU3PDeviceInit() inside the main() function. Changing this is not recommended.

5.1.3.3 Code Area

The code can be placed in the SYS_MEM area. It is recommended to place the code area in the beginning and then the data/heap area. Code area starts after the reserved 12KB and here 180KB is allocated for code area in the linker script file. If the code space allocated is not sufficient, the linker will return an error when trying to generate the firmware binary. The *fx3.ld* file can then be modified to increase the allocated space, as required.

5.1.3.4 Data Area

The global data area and the uninitialized data area follow the code area. In the linker script file, 32 KB is allocated for this. As in the case of code space, the *fx3.ld* file can be modified to increase or decrease the space allocated for the data area.

5.1.3.5 RTOS managed heap area

This area is where the thread memory and also other dynamically allocated memory to be used by the application are placed. The memory allocated for this is done inside the RTOS port helper file `cyfxtx.c`.

To modify this memory size/location change the definition for:

```
#define CY_U3P_MEM_HEAP_BASE         ((uint8_t *)0x40038000)
#define CY_U3P_MEM_HEAP_SIZE         (0x8000)
```

32 KB is allocated for RTOS managed heap. The size of this area can be changed in `cyfxtx.c`.

Memory allocation for the CyU3PMemAlloc is implemented using the byte pool service provided by ThreadX. This scheme makes use of a list that tracks available memory blocks within the reserved heap area. These routines will ensure that all allocated blocks are aligned at 4-byte boundaries.

The thread stacks are allocated from the RTOS managed heap area using the `CyU3PMemAlloc()` function.

5.1.3.6 DMA buffer area

DMA buffer area is managed by helper functions provided as source in `cyfxtx.c` file. These are `CyU3PDmaBufferInit()`, `CyU3PDmaBufferAlloc()`, `CyU3PDmaBufferFree()` and `CyU3PDmaBufferDeInit()`. All available memory above the RTOS managed heap to the top of the SYS_MEM is allocated as the DMA buffer area.

The full implementation for these allocation routines is provided as part of the *cyfxtx.c* file. The allocator ensures that all blocks allocated are placed at 32-byte (cache-line) boundaries.

The allocator makes use of a bitmap to manage the free/used status of each 32-byte block within the area reserved for the buffer heap. The memory for the bitmap is obtained using the CyU3PMemAlloc function.

The CyU3PDmaBufferAlloc looks for the required number of free 32-byte blocks and then marks them occupied. The CyU3PDmaBufferFree function marks the corresponding memory blocks free.

The status bit for the last 32-byte block in the allocated region is left in the used status so that it can serve as an end of block marker.

The memory allocated for this region can be modified by changing the definition for the following in `cyfxtx.c`.

```
#define CY_U3P_BUFFER_HEAP_BASE
((uint32_t) CY_U3P_MEM_HEAP_BASE + CY_U3P_MEM_HEAP_SIZE)

#define CY_U3P_BUFFER_HEAP_SIZE
(CY_U3P_SYS_MEM_TOP - CY_U3P_BUFFER_HEAP_BASE)
```

The default cyfxtx.c in the SDK reserves the last 32 KB of memory (0x40078000 to 0x40080000) for the optional second stage bootloader application built using the fx3_boot.a library. If the second stage boot application is not required, this memory can be reclaimed by changing the CY_U3P_SYS_MEM_TOP value to 0x40080000.

5.1.3.7 Support for FX3 parts with reduced memory

Some FX3 parts such as the CYUSB3011 and CYUSB3012 only have 256 KB of memory available. When these parts are being used, the memory map described must be modified to fit into the available space.

A linker script suitable for these parts has been provided in the <FX3_SDK_INSTALL>/firmware/common/fx3_256k.ld file. The RTOS heap and buffer heap placement can be modified by enabling the CYMEM_256K pre-processor definition in the cyfxtx.c file.

The high-level memory map provided by the fx3_256k.ld file is:

Table 5-1. High-level Memory Map

Memory Type	Base Address	Size
DMA descriptor area	0x40000000	12 KB
Code area	0x40003000	128 KB
Data area	0x40023000	24 KB
RTOS heap	0x40029000	28 KB
Buffer heap	0x40030000	32 KB*
2-stage boot area	0x40038000	32 KB*

* The buffer heap allocation can be increased to 64 KB, if the 2-stage boot solution is not in use.

As in the above case, the values programmed into these files are only the default values that work with the SDK examples. They can be modified as required, subject to the constraint that CY_U3P_SYS_MEM_TOP should be set to 0x40040000 or lesser.

5.2 API Library

A full fledged API library is provided in the FX3 SDK. The API library and the corresponding header files provide all the APIs required for programming the different blocks of the FX3. The APIs provide for the following:

- Programming each of the FX3/FX3S device blocks - USB, GPIF II, Storage, Serial peripherals and GPIO
- Programming the DMA engine and setting up of data flows between these blocks
- The overall framework for application development, including system boot and init, OS entry, and application init
- ThreadX OS calls as required by the application
- Power management features
- Programming low level DMA engine
- Debug capability

5.2.1 USB Block

The FX3 device has a USB-OTG PHY built-in and is capable of:

- USB peripheral - super speed, high speed, and full speed
- USB host - high speed and full speed only
- Charger detection

The USB driver provided as part of the FX3 firmware is responsible for handling all the USB activities. The USB driver runs as a separate thread and must be initialized from the user application. After initialization, the USB driver is ready to accept commands from the application to configure the USB interface of the FX3.

The USB driver handles both the USB device mode and the USB host mode of operation.

5.2.1.1 USB Device Mode

The USB device mode handling is described in the following sections.

USB Descriptors

Descriptors must be formed by the application and passed on to the USB driver. Each descriptor (such as Device, Device Qualifier, String, and Config) must be framed as an array and passed to the USB driver through an API call.

Endpoint Configuration

When configured as a USB device, the FX3 has 32 endpoints. Endpoint 0 is the control endpoint in both IN and OUT directions, and the other 30 endpoints are fully configurable. The endpoints are mapped to USB sockets in the corresponding directions. The mapping is normally one-to-one and fixed – endpoint 1 is mapped to socket 1 and so on. The only exception is when one or more USB 3.0 bulk endpoints are enabled for bulk streams. In this case, it is possible to map additional sockets that are not in use to the stream enabled endpoints.

Endpoint descriptors are formed by the application and passed to the USB driver which then completes the endpoint configuration. An API is provided to pass the configuration to the USB driver, this API must be invoked for each endpoint.

Enumeration

The next step in the initialization sequence is USB enumeration. After descriptor and endpoint configuration, the Connect API is issued to the USB driver. This enables the USB PHY and the pull-up on the D+ pin. This makes the USB device visible to a connected USB host and the enumeration continues.

Setup Request

By default, the USB driver handles all Setup Request packets that are issued by the host. The application can register a callback for setup requests. If a callback is registered:

- It is issued for every setup request with the setup data
- The application can perform the necessary actions in this callback
- The application must return the handling status whether the request was handled or not. This is required as the application may not want to handle every setup request
- If the request is handled in the application, the USB driver does not perform any additional handling
- If the request is not handled, the USB driver performs the default handling

Class/Vendor-specific Setup Request

Setup request packets can be issued for vendor commands or class specific requests such as MSC. The application must register for the setup callback (described earlier) to handle these setup request packets.

When a vendor command (or a class specific request) is received, the USB driver issues the callback with the setup data received in the setup request. The user application needs to perform the requisite handling and return from the callback with the correct (command handled) status.

Events

All USB events are handled by the USB driver. These include Connect, Disconnect, Suspend, Resume, Reset, Set Configuration, Speed Change, Clear Feature, and Setup Packet.

The user application can register for specific USB events. Callbacks are issued to the user application with the event type specified in the callback.

- The application can perform the necessary event handling and then return from the callback function.
- If no action is required for a specific event, the application can simply return from the issued callback function.

In both cases, the USB driver completes the default handling of the event.

Stall

The USB driver provides a set of APIs for stall handling.

- The application can stall a given endpoint
- The application can clear the stall on a given endpoint

Re-enumeration

- When a reset is issued by the USB host, the USB driver handles the reset and the FX3 device re-enumerates. If the application has registered a callback for USB event, the callback is issued for the reset event.
- The application can call the ConnectState API to electrically disconnect from the USB host. A subsequent call to the same API to enable the USB connection causes the FX3 device to re-enumerate.

- When any alternate setting is required, the endpoints must be reconfigured (UVC is an example where the USB host requests a switch to an alternate setting). The USB on the FX3 can remain in a connected state while this endpoint reconfiguration is being performed. A USB disconnect, followed by a USB connect is not required.
- The USB connection must be disabled before the descriptors can be modified.

Data Flows

All data transfers across the USB are done by the DMA engine. The simplest way of using the DMA engine is by creating DMA channels. Each USB endpoint is mapped to a DMA socket. The DMA channels are created between sockets. The types of DMA channels, data flows, and controls are described in DMA Mechanism on page 44.

5.2.1.2 Host Mode Handling

If a host mode connection is detected by the USB driver, the previously completed endpoint configuration is invalidated and the user application is notified. The application can switch to host mode and query the descriptors on the connected USB peripheral. The desired endpoint configuration can be completed based on the descriptors reported by the peripheral. When the host mode session is stopped, the USB driver switches to a disconnect state. The user application is expected to stop and restart the USB stack at this stage.

5.2.1.3 Bulk Streams in USB 3.0

Bulk streams are defined in the USB 3.0 specification as a mechanism to enhance the functionality of Bulk endpoints, by supporting multiple data streams on a single endpoint. When the FX3 is in USB 3.0 mode, the bulk endpoints support streams and burst type of data transfers. All active streams are actually mapped to USB sockets. Additional sockets that are not in use can be mapped to the stream enabled endpoints.

5.2.1.4 USB Device Mode APIs

The USB APIs are used to configure the USB device mode of operation. These include APIs for

- Start and stop the USB driver stack in the firmware
- Setting up the descriptors to be sent to the USB host
- Connect and disconnect the USB pins
- Set and get the endpoint configuration
- Get the endpoint status
- Set up data transfer on an endpoint
- Flush and endpoint
- Stall or Nak an endpoint
- Get or send data on endpoint 0
- Register callbacks for setup requests and USB events

5.2.1.5 USB Host Mode APIs

The USB Host Mode APIs are used to configure the FX3 device for USB Host mode of operation. These include APIs for

- Start and stop the USB Host stack in the firmware
- Enable/disable the USB Host port
- Reset/suspend/resume the USB Host port
- Get/set the device address

- Add/remove/reset an endpoint
- Schedule and perform EP0 transfers
- Setup/abort data transfers on endpoints.

5.2.1.6 *USB OTG Mode APIs*

The USB OTG Mode APIs are used to configure the USB port functionality and peripheral detection. These include APIs for

- Start and stop the USB OTG mode stack in firmware
- Get the current mode (Host/Device)
- Start/abort and SRP request
- Initiate a HNP (role change)
- Request remote host for HNP

5.2.2 GPIF II Block

The GPIF II is a general-purpose configurable I/O interface, which is driven through state machines. As a result, the GPIF II enables a flexible interface that can function either as a master or slave in many parallel and serial protocols. These may be industry standard or proprietary.

The features of the GPIF II interface are as follows:

- Functions as master or slave
- Provides 256 firmware programmable states
- Supports 8-bit, 16-bit, and 32-bit data path
- Enables interface frequencies of up to 100 MHz

Figure 5-3. GPIF II Flow

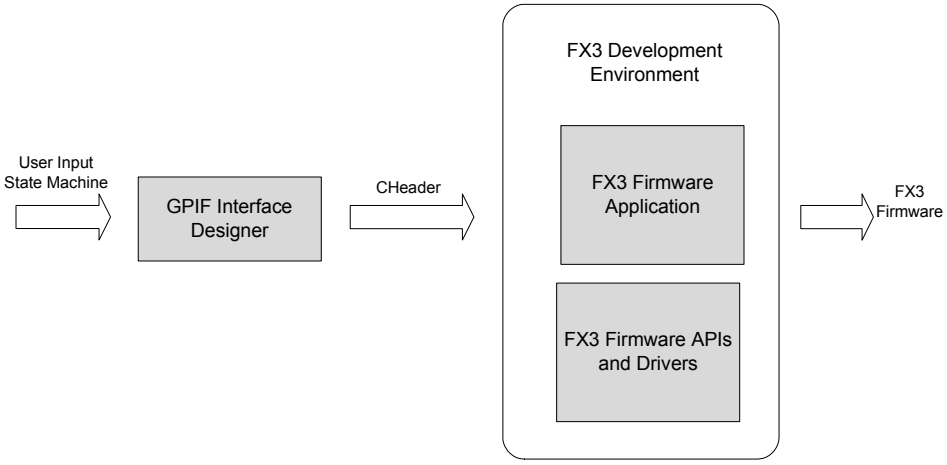


Figure 5-3 illustrates the flow of the GPIF II interface:

- The GPIF II Interface Design tool allows to synthesize the configuration by specifying the state machine.
- The configuration information for the GPIF II state machine is output as a C Header file by the tool.
- The header file is used along with the FX3 applications and API libraries to build the FX3 firmware.

On the FX3 device, the GPIF II must be configured before activating the state machine.

1. Load the state machine configuration into GPIF memory
2. Configure the GPIF registers
3. Configure additional GPIF parameters such as comparators and pin directions
4. Initiate the GPIF state machine

Each of these actions must be achieved by calling the appropriate GPIF API from the user application. The GPIF II driver provides calls for the user to setup and utilize the GPIF II engine. Because the GPIF II states can hold multiple waveforms, there is also a provision to switch to different states. These state switches are initiated through specific calls.

The GPIF II can be configured as a master or as a slave. When the GPIF II is configured as a master, GPIF II transactions are initiated using the GPIF II driver calls. The driver supports a method of executing master mode command sequences - initiate a command, wait for completion, and repeat the sequence, if required. When a transaction is complete, an event can be signaled.

A set of GPIF II events are specified. The user application needs to implement an interrupt handler function that is called from the ISR provided by the firmware library. Notification of GPIF II related events are only provided through this handler.

The programmed GPIF II interface is mapped to sockets. All data transfers (such as in USB) are performed by the DMA engine.

5.2.2.1 GPIF II APIs

The GPIF II APIs allow the programmer to set up and use the GPIF II engine. These include APIs to

- Initialize the GPIF state machine and load the waveforms
- Disable the GPIF state machine
- Start the GPIF state machine from a specified state
- Switch the GPIF state machine to a required state
- Pause and resume the GPIF state machine
- Configure a GPIF socket for data transfers
- Read or write a specified number of words from/to the GPIF interface

5.2.3 Serial Interfaces

The FX3 device has a set of serial interfaces: UART, SPI, I^2C, I2S, and GPIOs. All these peripherals can be configured and used simultaneously. The FX3 library provides a set of APIs to configure and use these peripherals. The driver must be first initialized in the user application. Full documentation of all Serial Interface registers is provided in Chapter 9. The source code for the serial interface drivers and access APIs is provided in the FX3 SDK package, in the *firmware/lpp_source* folder.

Each peripheral is configured individually by the set of APIs provided. A set of events are defined for these peripherals. The user application must register a callback for these events and is notified when the event occurs.

5.2.3.1 UART

A set of APIs are provided by the serial interface driver to program the UART functionality. The UART is first initialized and then the UART configurations such as baud rate, stop bits, parity, and flow control are set. After this is completed, the UART block is enabled for data transfers.

The UART has one producer and one consumer socket for data transfers. A DMA channel must be created for block transfers using the UART.

A direct register mode of data transfer is provided. This may be used to read/write to the UART, one byte at a time.

UART APIs

These include APIs to

- Start of stop the UART
- Configure the UART
- Setup the UART to transmit and receive data
- Read and write bytes from and to the UART

5.2.3.2 *I²C*

The I²C initialization and configuration sequence is similar to the UART. When this sequence is completed, the I²C interface is available for data transfer.

An API is provided to send the command to the I²C. This API must be used before every data transfers to indicate the size, direction, and location of data.

The I²C has one producer and one consumer socket for data transfers. A DMA channel must be created for block transfers using the I²C.

A direct register mode of data transfer is provided. This may be used to read/write to the I²C, one byte a time. This mechanism can also be used to send commands and/or addresses to the target I²C peripheral.

I²C APIs

These include APIs to

- Initialize/De-initialize the I²C
- Configure the I²C
- Setup the I²C for block data transfer
- Read/Write bytes from/to the I²C
- Send a command to the I²C

5.2.3.3 *I2S*

The I2S interface must be initialized and configured before it can be used. The interface can be used to send stereo or mono audio output on the I2S link.

DMA and register modes of access are provided.

I2S APIs

These include APIs to

- Initialize/de-initialize the I2S
- Configure the I2S
- Transmit bytes on the interface (register mode)
- Control the I2S master (mute the audio)

5.2.3.4 GPIO

A set of APIs are provided by the serial interface driver to program and use the GPIO. The GPIO functionality provided on the FX3 is a serial interface that does not require DMA.

Two modes of GPIO pins are available with FX3 devices - Simple and Complex GPIOs. Simple GPIO provides software controlled and observable input and output capability only. Complex GPIOs contain a timer and supports a variety of timed behaviors such as pulsing, time measurements, and one-shot.

GPIO APIs

These include APIs to

- Initialize/de-initialize the GPIO
- Configure a pin as a simple GPIO
- Configure a pin as a complex GPIO
- Get or set the value of a simple GPIO pin
- Register an interrupt handler for a GPIO
- Get the threshold value of a GPIO pin

5.2.3.5 SPI

The SPI has an initialization sequence that must be first completed for the SPI interface to be available for data transfer. The SPI has one producer and one consumer socket for data transfers. A DMA channel must be created for block transfers using the SPI.

A direct register mode of data transfer is provided. This may be used to read/write a sequence of bytes from/to the SPI interface.

SPI APIs

These include APIs to

- Initialize/de-initialize the SPI
- Configure the SPI interface
- Assert/de-assert the SSN line
- Set up block transfers
- Read/write a sequence of bytes
- Enable callbacks on SPI events
- Register an interrupt handler

5.2.4 Storage APIs

The Storage APIs are intended only for applications running on the FX3S device. These APIs allow user applications to initialize, identify, configure and exchange data with SD, eMMC or SDIO peripheral devices.

The Storage APIs are compiled into a separate API library (cyu3sport.a) and need to be linked with the application only if they are required. The Storage API library provides APIs to:

- Initialize the storage peripherals connected on one or both of the storage ports.
- Identify the device type and query device properties such as memory capacity, locked status, and so on.
- Perform read or write accesses to SD/eMMC storage devices. Only the DMA mode of transfer is supported by the Storage controller, and the DMA channel used for the transfer has to be configured separately.

- Partition a SD/eMMC device into multiple logical volumes that can be accessed independently.
- Use the SD/eMMC erase commands to erase some of the blocks on the storage device.
- Set/clear passwords and perform lock/unlock operations on SD cards and eMMC devices.
- Query SDIO device properties such as the number of functions, card capabilities, and so on.
- Perform byte-wise (CMD52) transfers from/to SDIO cards.
- Perform block-wise (CMD53) transfers from/to SDIO cards.
- Send a custom (vendor-specific) command to the peripheral and receive the response; and complete any associated data transfer.
- Support for identifying peripheral hotplug and providing event notifications.

5.2.5 DMA Engine

The FX3 DMA architecture is highly flexible and programmable. It is defined in terms of sockets, buffers, and descriptors. The complexity of programming the DMA of FX3 is eliminated by the high level libraries.

A higher level software abstraction is provided, which hides the details of the descriptors and buffers from the programmer. This abstraction views the DMA as being able to provide data channels between two blocks.

- A data channel is defined between two blocks
- One half is a producing block and the other half is a consuming block
- The producing and consuming blocks can be:
  - A USB endpoint
  - A GPIFII socket
  - Serial interfaces such as UART and SPI
  - CPU memory

The size and number of buffers required by the channel must be specified. Each DMA buffer is forwarded from the producer to the consumer when it has been filled with data or when an early completion is signaled by the producer. The reason for a partial buffer forwarding can be the receipt of a short packet from the USB host, a buffer COMMIT signal received on the GPIF II port, or a firmware initiated wrap-up action on the DMA channel. The term packet is used to denote the contents of one buffer of data, which is forwarded from the producer to the consumer.

The following types of DMA channels are defined to address common data transfer scenarios.

Table 5-2 summarizes the various DMA channel types supported by the FX3 firmware library.

Table 5-2. DMA Channel Types

DMA Channel Type	Description	Example
AUTO	Automatic DMA channel where the data flow is completely handled by the FX3 device hardware. Firmware only needs to set up the channel with the desired amount of buffering.	Data acquisition applications where data from the sensor is to be sent to the PC host without any modification.
AUTO – SIGNAL	Automatic DMA channel with signaling. This is the same as an auto channel with respect to the actual data flow. The only difference is that the firmware is notified when each data buffer is being forwarded through the FX3 device.	Used if the application wants to track statistics about the data being forwarded.
MANUAL	Manual DMA channel requires firmware intervention for the forwarding of each data buffer. Firmware is notified when a complete buffer of data is made available by the PRODUCER socket, and is responsible for sending this to the CONSUMER with or without modification.	USB Video Class (UVC) streaming applications where a UVC header has to be added (or removed) from each data buffer that is forwarded.
MANUAL OUT	Channel used by firmware to send data out through a CONSUMER socket. The size and content of each DMA buffer is determined by firmware.	Debug logging channel used to send verbose log messages out through an UART interface.
MANUAL IN	Channel used by firmware to receive data from a PRODUCER socket. The data is not expected to be sent out of the device and can be processed by the firmware.	Channel used to receive command packets (such as USB mass storage commands) from a PC host.
AUTO MANY-TO-ONE	This is a complex data channel where data received from two different PRODUCER sockets is multiplexed on to a single stream sent out through a CONSUMER socket. As in the case of an AUTO channel, the data forwarding is completely handled by hardware.	Used to implement the read path for a RAID-0 or striped storage solution using a pair of SD cards.

Table 5-2. DMA Channel Types

DMA Channel Type	Description	Example
AUTO ONE-TO-MANY	This is a complex data channel where data received from one PRODUCER socket is de-multiplexed onto two different CONSUMER sockets. The data forwarding is done automatically by hardware on a strictly alternating basis.	Used to implement the write path for a RAID-0 or striped storage solution using a pair of SD cards.
MANUAL MANY-TO-ONE	Channel that allows firmware controlled forwarding of data from multiple PRODUCERs to a single CONSUMER.	Used in UVC applications with a direct image sensor interface. Using the MANY-TO-ONE channel mode allows FX3 to continuously receive data despite the latencies associated with switching between buffers
MANUAL ONE-TO-MANY	Channel that allows firmware controlled forwarding of data from a single PRODUCER to multiple CONSUMERs.	Can be used in UVC-based display applications.
MULTICAST	Channel that allows data from a PRODUCER to be sent to multiple CONSUMERs. The same data is sent to all of the CONSUMERs. This channel can only operate in MANUAL mode, as it is not possible to implement a MULTICAST AUTO channel with the FX3 hardware.	Can be used to implement a RAID-1 (mirrored) mass storage implementation using a pair of SD cards.

5.2.5.1 *Automatic Channels*

An automatic DMA channel is one where the data flows between the producer and consumer uninterrupted when the channel is set up and started. There is no firmware involvement in the data flow at runtime. Firmware is only responsible for setting up the required resources for the channel and establishing the connections. When this is done, data can continue to flow through this channel until it is stopped or freed up.

This mode of operation allows for the maximum data through-put through the FX3 device, because there are no bottlenecks within the device.

Two flavors of the auto channel are supported: auto channel and auto channel with signaling

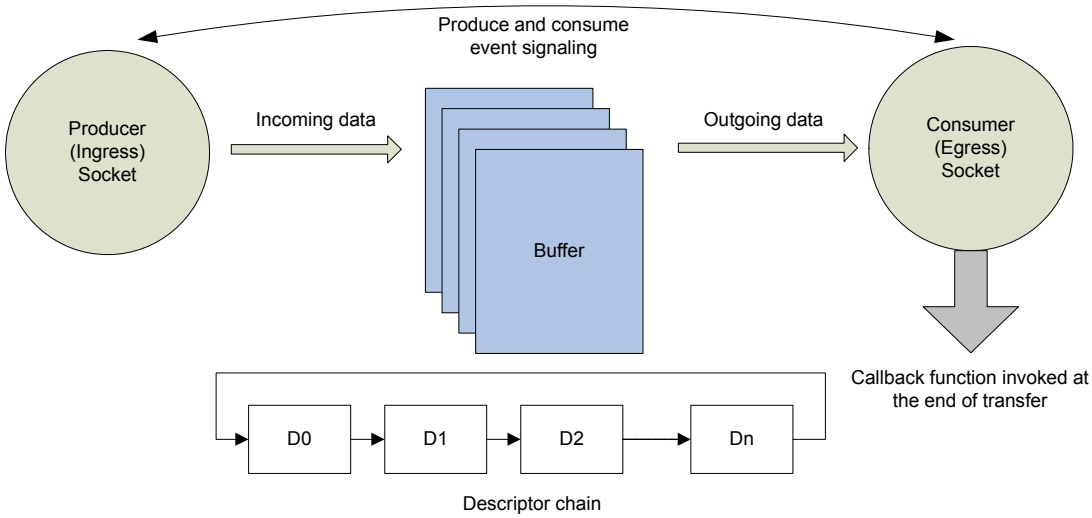
Auto Channel

This channel is defined as DMA_TYPE_AUTO. This is the pure auto channel. It is defined by a valid producer socket, a valid consumer socket, and a predetermined amount of buffering; each of these is a user programmable parameter.

The buffers are of equal size, the number of buffers is specified at channel creation time. Internally, the buffers are linked cyclically by a descriptor chain.

This type of channel can be set up to transfer finite or infinite amount of data. The user application is notified through an event callback when the specified amount of data is transferred.

Figure 5-4. Auto Channel

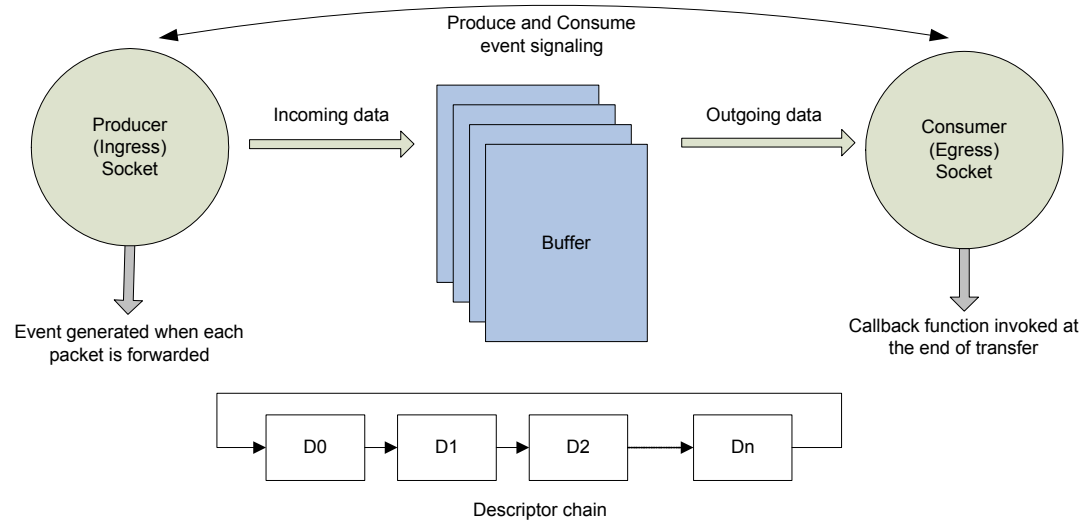


Auto Channel with Signaling

This channel is a minor variant of the Auto channel. The channel set up and data flow remains the same. The only change is that an event is raised to the user application every time a buffer is received from the PRODUCER socket. The buffer pointer and the data size are communicated to the application. This is useful for data channels where the data needs to be inspected for activities such as collection of statistics.

- The actual data flow is not impeded by this inspection; the DMA continues uninterrupted.
- The notification cannot be used to modify the contents of the DMA buffer.

Figure 5-5. Auto Channel with Signaling



Many-to-One Auto Channel

This channel is defined as DMA_TYPE_AUTO_MANY_TO_ONE and is a variation of the auto channel. It is defined by more than one valid producer sockets, a valid consumer socket, and a predetermined amount of buffering; each of these is a user programmable parameter.

This type of channel is used when the data flow from many producers (at least 2) has to be directed to one consumer in an interleaved fashion.

One-to-Many Auto Channel

This channel is defined as DMA_TYPE_AUTO_ONE_TO_MANY and is a variation of the auto channel. It is defined by one valid producer socket, more than one valid consumer socket, and a predetermined amount of buffering; each of these is a user programmable parameter.

This type of channel is used when the data flow from one producer has to be directed to more than one consumer (at least 2) in an interleaved fashion.

5.2.5.2 Manual Channels

These are a class of data channels that allow the FX3 firmware to control and manage data flow:

- Add and remove buffers to and from the data flow
- Add and remove fixed size headers and footers to the data buffers. Note that only the header and footer size is fixed, the data size can vary dynamically.
- Modify the data in the buffers provided the data size itself is not modified.

In manual channels, the CPU (FX3 firmware) itself can be the producer or the consumer of the data.

Manual channels have a lower throughput compared to the automatic channels as the CPU is involved in every buffer that is transferred across the FX3 device.

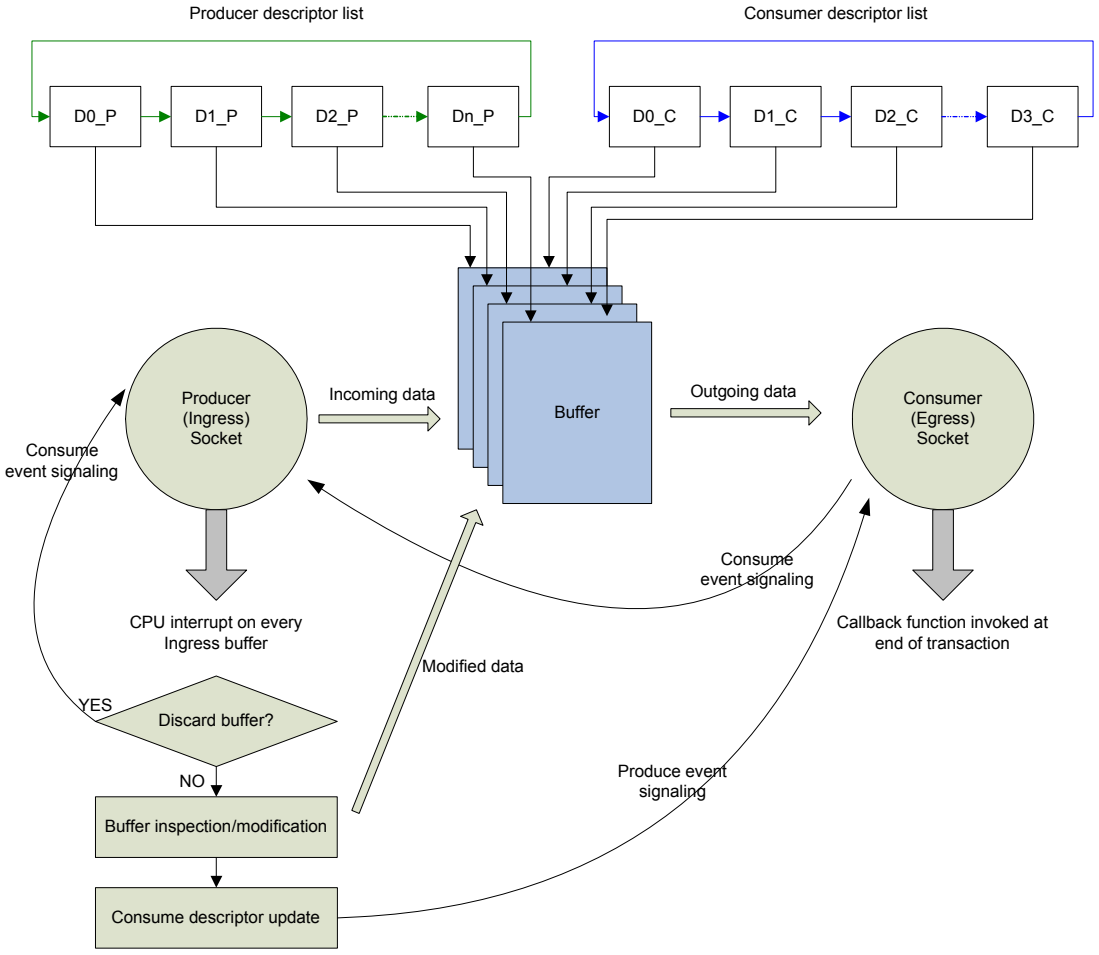
Manual Channel

The channel DMA_TYPE_MANUAL is a pass through channel with CPU intervention. Internally, the channel has two separate descriptor lists, one for the producer socket and one for the consumer socket. At channel creation time, the user application must indicate the amount of buffering required and register a callback function.

When the channel is operational, the registered callback is invoked when a data buffer is committed by the producer. In this callback, the user application can:

- Change the content of the data packet (size cannot be changed)
- Commit the packet, triggering the sending out of this packet
- Insert a new custom data packet into the data stream
- Discard the current packet without sending to the consumer
- Add a fixed sized header and/or footer to the received packet. The size of the header and footer is fixed during the channel creation
- Remove a fixed sized header and footer from the received packet

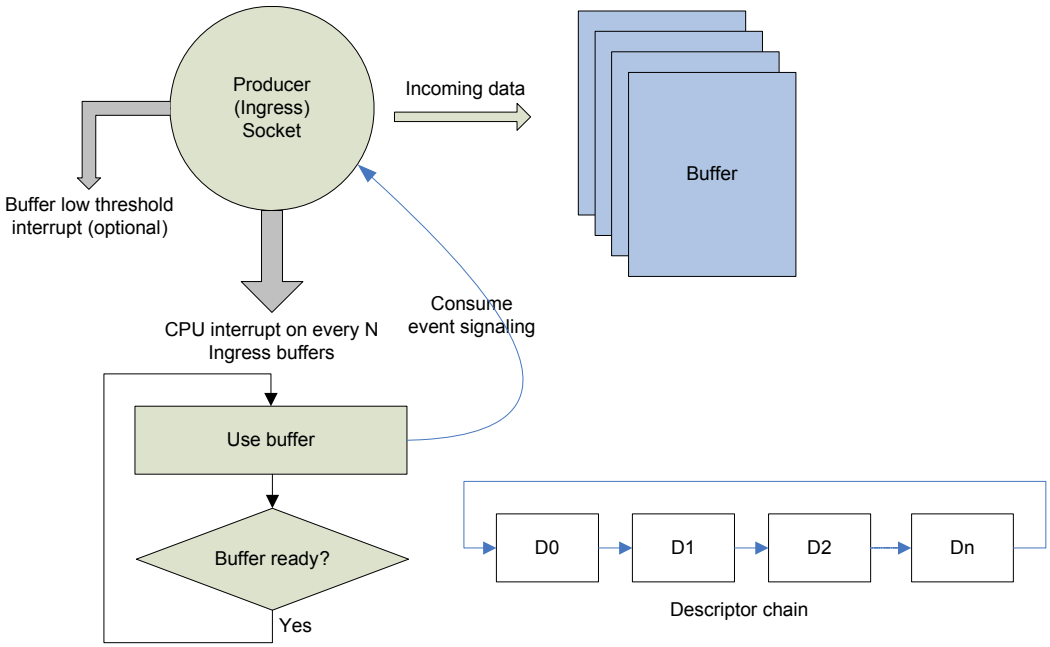
Figure 5-6. Manual Channel



Manual In Channel

The DMA_TYPE_MANUAL_IN channel is a special channel where the CPU (FX3 firmware) is the consumer of the data. A callback must be registered at channel creation time and this is invoked to the user application when a specified (default is one) number of buffers are transferred.

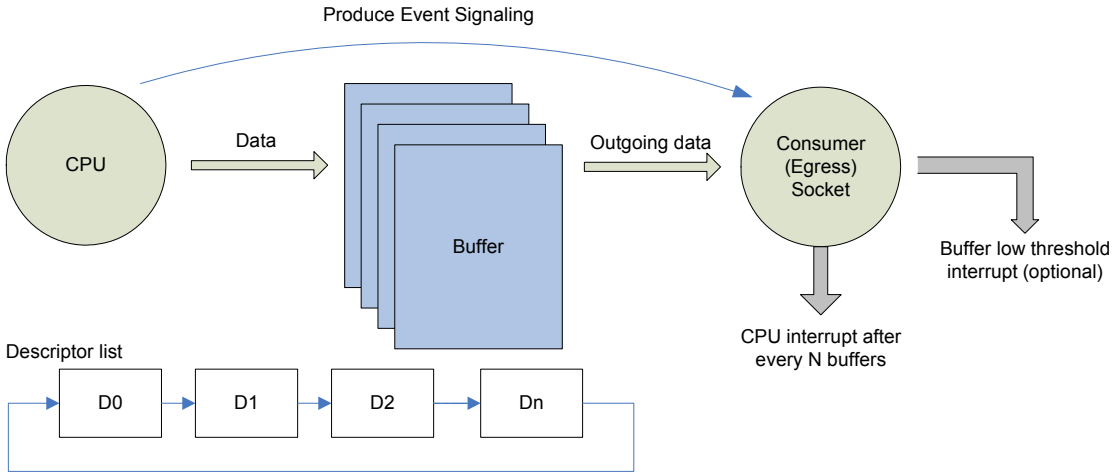
Figure 5-7. Manual In Channel



Manual Out Channel

The DMA_TYPE_MANUAL_OUT channel is a special channel where the CPU (FX3 firmware) is the producer of data. The user application needs to get the active data buffer, populate the buffer, and then commit it.

Figure 5-8. Manual Out Channel



Many-to-One Manual Channel

This channel is defined as DMA_TYPE_MANUAL_MANY_TO_ONE and is a variation of the manual channel. It is defined by more than one valid producer socket, a valid consumer socket, and a predetermined amount of buffering; each of these is a user programmable parameter.

This type of channel is used when the data flow from many producers (at least 2) has to be directed to one consumer in an interleaved fashion with CPU intervention.

One-to-Many Manual Channel

This channel is defined as DMA_TYPE_MANUAL_ONE_TO_MANY and is a variation of the manual channel. It is defined by one valid producer socket, more than one valid consumer socket, and a predetermined amount of buffering; each of these is a user programmable parameter.

This type of channel is used when the data flow from one producer has to be directed to more than one consumer (at least 2) in an interleaved fashion with CPU intervention.

Multicast Channel

This channel is defined as DMA_TYPE_MULTICAST. It is defined by one valid producer socket, more than once valid consumer socket, and a predetermined amount of buffering; each of these is a user programmable parameter.

This type of channel is used when the data flow from one producer has to be directed to more than one consumer. Here all the consumers receive the same data.

5.2.5.3 DMA Buffering

The buffering requirements of the DMA channels are handled by the channel functions. The amount of buffering required (size of buffer and number of buffers) must be specified at the time of channel creation. If channel creation is successful, the requisite buffers are successfully allocated. The buffers are allocated from the block pool. The FX3 user application does not have to allocate any buffers for the DMA channels.

5.2.5.4 DMA APIs

These consist of APIs to

- Create and destroy a DMA channel
- Set up a data transfer on a DMA channel
- Suspend and resume a DMA channel
- Abort and reset a DMA channel
- Receive data into a specified buffer (override mode)
- Transmit data from a specified buffer (override mode)
- Wait for the current transfer to complete
- Get the data buffers from the DMA channel
- Commit the data buffers for transfer
- Discard a buffer from the DMA channel

5.2.6 RTOS and OS Primitives

The FX3 firmware uses ThreadX, a real-time operating system (RTOS). The firmware framework invokes the RTOS as part of the overall system initialization.

All the ThreadX primitives are made available in the form of an RTOS library. The calls are presented in a generic form; the ThreadX specific calls are covered with wrappers. These wrappers provide an OS independent way of coding the user application.

These include APIs for:

- Threads
  - Thread create and delete
  - Thread suspend and resume
  - Thread priority change
  - Thread sleep
  - Thread information
- Message queues
  - Queue create and delete
  - Message send and priority send
  - Message get
  - Queue flush
- Semaphores
  - Semaphore create and destroy
  - Semaphore get and put
- Mutex
  - Mutex create and destroy
  - Mutex get and put
- Memory allocation
  - Memory alloc and free
  - Memset, memcopy and memcmp
  - Byte pool creation
  - Byte alloc and free
  - Block pool creation
  - Block alloc and free
- Events
  - Event creation and deletion
  - Event get and set
- Timer
  - Timer creation and deletion
  - Timer start and stop
  - Timer modify
  - Get/set time current time (in ticks)

5.2.7 Debug Support

The FX3 device supports the standard JTAG interface for ARM processors. Any of the standard JTAG debug probes (such as Segger J-Link) can be used to do run-time debugging of the FX3 firmware.

JTAG debugging is not always viable for USB applications as stopping at a breakpoint can cause USB transfers to time out. Debug logs are a valuable tool to track down issues in applications where JTAG debugging is not possible. The FX3 SDK supports a flexible debug logging scheme. All the drivers and firmware functions implement a logging scheme where optional debug logs are written into a reserved buffer area. This debug log can then be read out to an external device and analyzed.

The debug APIs provide the following functions:

- Start and stop the debug logging mechanism
- Print a debug message (a debug string)
- Log a debug message (a debug value which gets mapped to string)
- Flush the debug log to an external device (for example, UART)
- Clear the debug log
- Set the debug logging level (performed during init)

Typically, the UART interface on the FX3 device is used for the debug log output. However, if the UART interface is not available or used for other purposes, the logs can be output using any of the other interfaces such as a USB endpoint or a GPIF II socket. The only requirement is that the data be read out from the other end regularly so that free buffers for logging are always available.

Verbose logs can be output using the CyU3PDebugPrint() function, as demonstrated in all of the firmware examples. This function only supports the %c, %d, %u, %x, and %s conversion specifications and does not support floating point output. The function does not support any flags, precision, or type modifiers. The standard C library sprintf routine can be used to format the output string if any of these features are required.

Adding detailed verbose logs is not viable in most performance-critical applications. The SDK supports a coded log mechanism using the CyU3PDebugLog function for use in such applications. This function queues up the log data into internal memory buffers and sends them out when the buffer is full.

User code can also use the debug logging mechanism and use the debug log and print functions to insert debug messages.

5.2.8 Power Management

Power management support is provided. APIs are provided to place the FX3 device into the low-power Suspend and Standby modes. The conditions that would cause the device to resume to the active state must be specified.

Standby mode can only be initiated after shutting down all of the FX3 device blocks, except GPIO. When the device resumes from standby mode, the startup procedure is as in the case of a warm reset.

5.2.9 Low Level DMA

The DMA architecture of the FX3 is defined in terms of sockets, buffers and descriptors. Each block on the FX3 (USB, GPIF, Serial IOs) can support multiple independent data flows through it. A set of sockets are supported on the block, where each socket serves as the access point for one of the data flows. Each socket has a set of registers that identify the other end of the data flow and control parameters such as buffer sizes. The connectivity between the producer and consumer is

established through a shared data structure called a DMA descriptor. The DMA descriptor maintains information about the memory buffer, the producer and consumer addresses etc.

The low level DMA APIs provide for:

- Sockets
- Set/get the socket configuration
- Set other options on a given socket
- Check if a given socket is valid
- Buffers
- Create/destroy buffer pools
- Allocate/free buffers
- Descriptors
- Create/destroy descriptor lists and chains
- Get/Set a descriptor configuration
- Manipulate the descriptor chain

5.2.10 MIPI-CSI2 Configuration APIs

The MIPI-CSI2 configuration APIs are only intended for applications running on the CX3 device. The APIs allow user applications to initialize, configure, and perform power management for the interface. All MIPI-CSI2 configuration APIs communicate with the interface over the I2C bus.

The MIPI-CSI2 configuration APIs are compiled into a separate API library (cyu3mipicsi.a) and need to be linked with the application only if they are required. The API library provides APIs to:

- Initialize the MIPI-CSI2 receiver interface.
- Configure the MIPI-CSI2 interface clocks and interface settings.
- Configure the fixed-function GPIF-2 bus widths.
- Place the MIPI-CSI2 receiver in low-power standby mode.
- Reset the MIPI-CSI2 receiver.
- Drive the MIPI-CSI2 XRESET and XSHUTDOWN signals to the image sensor.

User code needs to be written to configure the image sensor. Configuration for two sensors (Aptina AS0260 and Omnivision OV5640) is provided in the form of library files, which are used in the CX3 example projects that are part of the SDK.

6. FX3 APIs

The FX3 APIs consist of APIs for programming the main hardware blocks of the FX3. These include the USB, GPIF II, Storage, DMA, and the serial I/Os. Refer to the corresponding sections of the FX3API Guide for information about these APIs.

7. FX3 Application Examples

The FX3 SDK includes various application examples in source form. These examples illustrate the use of the APIs and firmware framework putting together a complete application. The examples illustrate the following:

- Initialization and application entry
- Creating and launching application threads
- Programming the peripheral blocks (USB, GPIF, serial interfaces)
- Programming the DMA engine and setting up data flows
- Registering callbacks and callback handling
- Error handling
- Initializing and using the debug messages
- Programming the FX3 device in Host/OTG mode

7.1 DMA Examples

The FX3 DMA engine is independent of the peripheral used. The DMA APIs provide the mechanism for data transfer to and from the FX3 device.

These examples are essentially bulk loop-back examples where data received from the USB host PC through the OUT EP is sent back through the IN EP. These examples explain the different DMA channel configurations.

7.1.1 USBBulkLoopAuto

This example demonstrates the use of DMA AUTO channel to loop back data between the USB endpoints without any firmware intervention in actual data transfer. This type of channel provides the maximum throughput and is the simplest of all DMA configurations.

Location: `firmware/dma_examples/cyfxbulklpauto`

7.1.2 USBBulkLoopAutoSignal

This example demonstrates the use of the DMA AUTO_SIGNAL channel with signaling to loop back data between the USB endpoints without any firmware intervention in actual data transfer. This type of channel is similar to the AUTO channel, except for the event signaling provided for every buffer received by FX3.

Location: `firmware/dma_examples/cyfxbulklpautosig`

7.1.3 USBBulkLoopManual

This example demonstrates the use of the DMA MANUAL channel to loop back data between USB endpoints with CPU intervention. In this type of channel, the CPU has to explicitly commit the

received data. The CPU can also modify the data received before sending it out of the device. The data manipulation is done in place and does not require any memory-to-memory copy.

Location: `firmware/dma_examples/cyfxbulklpmanual`

7.1.4 USBBulkLoopManualInOut

This example demonstrates the use of the DMA MANUAL IN + DMA MANUAL OUT channel to loop back data between USB endpoints. The data received from the OUT endpoint through the MANUAL_IN channel is copied to IN endpoint through the MANUAL_OUT channel. In this type of channel, the data has to be explicitly copied from the OUT endpoint to the IN endpoint by the firmware.

Location: `firmware/dma_examples/cyfxbulklpmaninout`

7.1.5 USBBulkLoopAutoOneToMany

This example demonstrates the use of multi-channel DMA AUTO ONE TO MANY to loop back data between USB 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`

7.1.6 USBBulkLoopManualOneToMany

This example demonstrates the use of multi-channel DMA MANUAL ONE to MANY loop back data between USB endpoints. This is similar to the USBBulkLoopAutoOneToMany example, except for the fact that the data has to be committed explicitly by the CPU and the CPU can modify the data before being sent out.

Location: `firmware/dma_examples/cyfxbulklpmanonetomany`

7.1.7 USBBulkLoopAutoManyToOne

This example demonstrates the use of multi-channel DMA AUTO MANY TO ONE to loop back data between USB endpoints. The data received from two OUT endpoints is committed in an interleaved fashion to a single IN endpoint. Data interleaving is done automatically by the FX3 hardware without any firmware intervention.

Location: `firmware/dma_examples/cyfxbulklpautomanytoone`

7.1.8 USBBulkLoopManualManyToOne

This example demonstrates the use of multi-channel DMA MANUAL MANY TO ONE to loop back data between USB endpoints. This is similar to the USBBulkLoopAutoManyToOne example, except for the fact that the data has to be committed explicitly by the CPU and the CPU can modify the data before being sent out.

Location: `firmware/dma_examples/cyfxbulklpmanmanytoone`

7.1.9 USBBulkLoopMulticast

This example demonstrates the use of multi-channel DMA MULTICAST to loop back data between USB endpoints. In this case, data coming in on an OUT endpoint is sent out on two different IN endpoints. Both IN EPs shall receive the same data. MULTICAST channels only support a MANUAL mode of operation where the firmware has to forward each packet.

Location: `firmware/dma_examples/cyfxbulklpmulticast`

7.1.10 USBBulkLoopManualAdd

This example demonstrates the use of DMA MANUAL channel to modify data flowing through FX3 by adding fixed-size 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`

7.1.11 USBBulkLoopManualRem

This example demonstrates the use of DMA MANUAL channels where a header and footer is 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 the header and footer does not require data copying.

Location: `firmware/dma_examples/cyfxbulklpman_removal`

7.1.12 USBBulkLoopLowLevel

The DMA channel is a helpful construct that enables simple data transfer. The low-level DMA descriptor and DMA socket APIs enable finer constructs. This example uses these APIs to implement a simple bulk loop back example.

Location: `firmware/dma_examples/cyfxbulklplowlevel`

7.1.13 USBBulkLoopManualDCache

The data cache is disabled by default in the FX3 device. The data cache is useful when the CPU does a large amount of data modifications. But enabling the D-cache adds constraints for managing the data cache. This example demonstrates how DMA transfers can be done with the data cache enabled.

Location: `firmware/dma_examples/cyfxbulklpmandcache`

7.2 Basic Examples

The FX3 SDK includes basic USB examples that are meant to be a programming guide for the following:

- Setting up the descriptors and USB enumeration
- USB endpoint configuration
- USB reset and suspend handling

7.2.1 RTOSExample

This example demonstrates the use of the ThreadX RTOS services such as Threads, Mutexes, Semaphores, and Event Flag Groups. The example also provides a method to get a visual indication of RTOS object state changes through GPIOs on the FX3 Development Kits.

The application does not perform any actual data transfers but makes use of multiple threads, which operate on the other services such as Mutexes, Semaphores, and Event Flag Groups.

Location: `firmware/basic_examples/cyfx_rtos_example`

7.2.2 BulkLpAutoCpp

This example demonstrates the use of C++ with FX3 APIs using the DMA AUTO channel to loop back data between the USB BULK endpoints.

Location: `firmware/basic_examples/cyfxbulklpauto_cpp`

7.2.3 USBBulkLoopAutoEnum

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 the user to handle the device enumeration by directly responding to USB control requests. This example is similar to the USBBulkLpAuto example but demonstrates how the USB descriptor requests can be handled from the firmware application. All control requests are handled from the USB Setup Callback function provided by the application.

Location: `firmware/basic_examples/cyfxbulklpautoenum`

7.2.4 USBBulkSourceSink

The example demonstrates the use of FX3 as a data source and a data sink using bulk endpoints. 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 sourcing the data to the Bulk IN endpoint.

Location: `firmware/basic_examples/cyfxbulksrcsink`

7.2.5 USBIsoSourceSink

The example demonstrates the use of FX3 as a data source and a data sink using ISO endpoints. This is similar to the USBBulkSourceSink example but makes use of ISO endpoints. Multiple alternate interface settings are provided in this example to control the data rates.

Location: `firmware/basic_examples/cyfxisosrcsink`

7.2.6 USBIsochLoopAuto

This example demonstrates the loop back of data through ISO endpoints using the DMA AUTO channel. This example is similar to the USBBulkLoopAuto except for the fact that the endpoints used here are isochronous instead of bulk.

Location: `firmware/basic_examples/cyfxisolpauto`

7.2.7 USBIsochLoopManualInOut

This example demonstrates the loop back of data through ISO endpoints using a pair of DMA MANUAL IN and DMA MANUAL OUT channels. This example is similar to the USBBulkLoopManualInOut except for the fact that the endpoints used here are isochronous instead of bulk.

Location: `firmware/basic_examples/cyfxisolpmaninout`

7.2.8 USBBulkStreams

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

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

Location: `firmware/basic_examples/cyfxbulkstreams`

7.2.9 USBFlashProg

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

This example enumerates as a vendor-specific USB device with no endpoints, and provides a set of pre-defined vendor commands for read/write operations through which the I2C EEPROM and SPI flash devices can be programmed.

The binary produced by compiling this application is used by the Cypress Control Center tool to program I2C EEPROM and SPI Flash.

Location: `firmware/basic_examples/cyfxflashprog`

7.2.10 USBCDCDebug

This example demonstrates how the virtual COM port USB-CDC interface can be used to obtain debug data from a FX3 device.

This example makes use of an USB Bulk 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/cyfxcdcdebug`

7.2.11 USBDebug

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

This example uses 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`

7.2.12 USBHost

This example demonstrates how FX3 can be used as a 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.

This example can only work on the FX3 device variants that support the OTG host mode.

Location: `firmware/basic_examples/cyfxusbhost`

7.2.13 USBOtg

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.

This example can only work on FX3 device variants that support the OTG host mode.

Location: `firmware/basic_examples/cyfxusbotg`

7.2.14 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 connecting a pair of FX3 devices 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.

This example can only work on FX3 device variants that support the OTG host mode.

Location: `firmware/basic_examples/cyfxbulklpotg`

7.2.15 LowPowertest

This example illustrates how the FX3 device can be configured to be placed in the low-power standby or suspend state. This example supports the use of the VBUS, UART_CTS, and USB D+ or SSRX pins as wake-up sources from the low-power state. This application also implements a Bulk Loopback function based on an AUTO DMA channel.

Location: `firmware/basic_examples/cyfxlowpowertest`

7.2.16 GpifToUsb

This example illustrates the use of a DMA channel to continuously transfer data from the GPIF port to the USB host. A stub GPIF state machine, which does not require any external devices, is used to continuously commit data into the DMA buffers. This state machine continues to push data into the DMA channel whenever the thread is in the ready state.

The device enumerates as a vendor-specific USB device with one Bulk-IN endpoint. The data committed by the GPIF state machine is continuously streamed to this endpoint without any firmware intervention.

Location: `firmware/basic_examples/cyfxgpiftousb`

7.2.17 USBIsoSource

This example illustrates the use of the FX3 firmware APIs to implement a variable data rate ISO IN endpoint using a DMA MANUAL_OUT channel. A constant data pattern is continuously loaded into the DMA MANUAL_OUT channel and sent to the host. Multiple alternate settings with different transfer rates are supported.

Location: `firmware/basic_examples/cyfxisosrc`

7.3 Serial Interface Examples

The serial interface examples demonstrate data accesses to the Serial I/O interfaces: GPIO, UART, I2C, SPI, and I2S.

7.3.1 GPIO Examples

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

7.3.1.1 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 footprint.

Location: `firmware/serial_examples/cyfxgpioapp`

7.3.1.2 GpioComplexApp

The FX3 device has eight complex GPIO blocks that can be used to implement various functions such as timer, counter, and PWM. The example shows how to use the more advanced GPIO features of the FX3 device. The application demonstrates:

- Driving an output pin with a Pulse Width Modulated (PWM) signal
- Measuring the pulse duration on an input using an internal timer
- Counting the number of edges on an input signal using a counter

Location: `firmware/serial_examples/cyfxgpiocomplexapp`

7.3.2 UART Examples

The FX3 UART block 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.

7.3.2.1 UartLpDmaMode

This example uses a MANUAL DMA channel to loop any data received through the UART receiver back to the UART transmitter.

Location: `firmware/serial_examples/cyfxuartlpdmamode`

7.3.2.2 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 to 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`

7.3.2.3 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`

7.3.3 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 to and from an I2C EEPROM device in the two modes of operation: 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 to and from the I2C EEPROM device.

7.3.3.1 UsbI2cDmaMode

This example shows how to use the I2C block in DMA mode to read and write data to and fromthe I2C EEPROM.

Location: `firmware/serial_examples/cyfxusbi2cdmamode`

7.3.3.2 UsbI2cRegMode

This example shows how to use the I2C block in register (firmware-programmed) mode to read and write data to and from the I2C EEPROM.

Location: `firmware/serial_examples/cyfxusbi2cregmode`

7.3.4 SPI Examples

As in the I2C example, FX3 supports a master-only SPI interface block. The SPI examples in the SDK demonstrate how the FX3 can be used to read and write data on an 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.

7.3.4.1 UsbSpiDmaMode

This example uses DMA channels to read and write data on the SPI flash memory.

Location: `firmware/serial_examples/cyfxusbspidmamode`

7.3.4.2 UsbSpiRegMode

This example uses the FX3 SPI block's register interface to read and write data on the SPI flash memory.

Location: `firmware/serial_examples/cyfxusbspiregmode`

7.3.4.3 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 I/Os. This example shows how a set of four GPIOs on the FX3 device can be used as the SPI Clock, MOSI, MISO, and SSN# pins.

Location: `firmware/serial_examples/cyfxusbspigpiomode`

7.3.5 I2S Examples

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.

7.3.5.1 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`

7.4 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 use an external image sensor, but repeatedly send out pre-defined image data that is stored as part of the firmware image.

7.4.1 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 the 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-µs microframe (Service interval).

Location: `firmware/uvc_examples/cyfxuvcinmem`

7.4.2 USBVideoClassBulk

This example is similar to the USBVideoClass example, but uses 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 than the ISOCHRONOUS example.

Location: `firmware/uvc_examples/cyfxuvcinmem_bulk`

7.5 Slave FIFO 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).

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

7.5.1 SlaveFifoAsync

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

Location: `firmware/slavefifo_examples/slfifoasync`

7.5.2 SlaveFifoAsync5Bit

Asynchronous mode Slave FIFO example using a 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`

7.5.3 SlaveFifoSync

Synchronous mode Slave FIFO example using a 2-bit FIFO address. As a 2-bit address is used, a maximum of four pipes on FX3 can be accessed by the FIFO master.

Location: `firmware/slavefifo_examples/slfifosync`

7.5.4 SlaveFifoSync5Bit

Synchronous mode Slave FIFO example using a 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`

7.6 USB Audio Class Example

7.6.1 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`

7.7 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.

7.7.1 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`

7.7.2 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 upconverts the stream to 24-bit RGB-888, and transmits to the host PC over USB.

Location: `firmware/cx3_examples/cycx3_rgb24_as0260`

7.7.3 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 from the image sensor over the CX3 to the host PC.

Location: `firmware/cx3_examples/cycx3_uvc_as0260`

7.7.4 Cx3UvcOV5640

This example implements a Bulk-only UVC 1.1 compliant 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`

7.8 Two-Stage Booter (boot_fw) 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 bootloader applications. However, this is a misnomer because the library supports almost all device features and can also be used for building full-fledged applications. A set of applications that demonstrate the capabilities of this firmware library are also provided with the SDK.

7.8.1 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`

7.8.2 FX3BootAppGcc

This example shows how the boot APIs can be used to implement a custom bootloader 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`

7.8.3 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`

7.9 Mass Storage Class Example

7.9.1 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`

7.9.2 FX3SMassStorage

This example provides a Mass Storage Class interface through which the USB host can access the SD cards or eMMC devices connected to the FX3S device. The example shows how to implement high-performance data transfers using the FX3S storage APIs, and also supports features such as device hotplug handling, partitioning (multi-volume) support, and more.

Location: `firmware/msc_examples/cyfx3s_msc`

7.9.3 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`

7.10 FX3S Storage Examples

The following examples demonstrate the use 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.

7.10.1 GpifToStorage

An 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`

7.10.2 FX3SFileSystem

An 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`

7.10.3 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`

7.11 GPIF-II Master Example

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.

7.11.1 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 and read the content of this memory segment using a pair of USB bulk endpoints.

Location: `firmware/gpif_examples/cyfxsrammaster`

7.12 FX2G2 Example

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

7.12.1 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`

7.13 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, and SPI to the processor.

7.13.1 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`

8. FX3 Firmware Application Structure

All FX3 firmware applications will consist of two parts:

- Initialization code - This will be mostly common to all applications
- Application code - This will be the application specific code

The synchronous slave FIFO application (SlaveFifoSync) is taken as an example to present the FX3 application structure. All the sample code shown below is from this example.

8.1 Firmware Application Structure

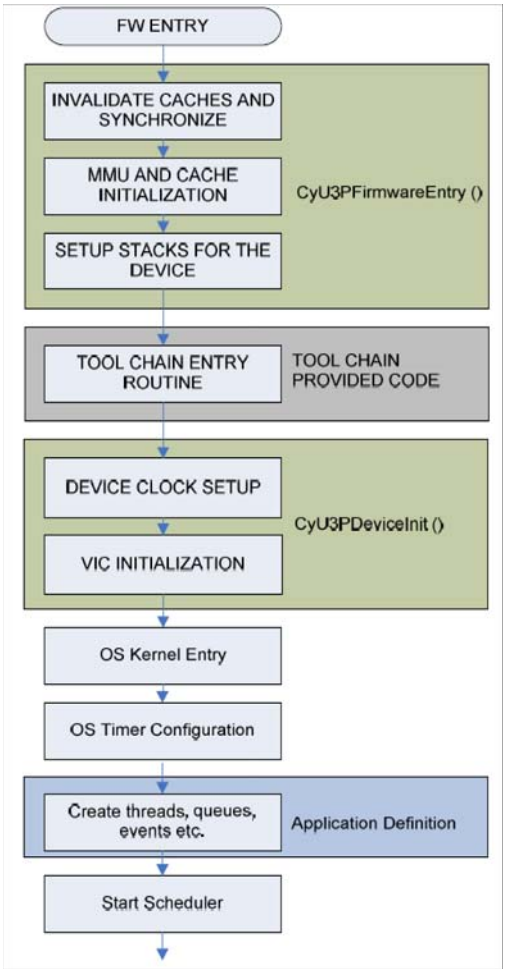
The Slave FIFO example comprises of the following files:

- cyfxgpif_syncsf.h: This file contains the GPIF-II descriptors for the 16-bit and 32-bit Slave FIFO interface.
- cyfxslfifousbdscr.c: This file contains the USB descriptors.
- cyfxslfifosync.h: This file contains the defines used in cyfxslfifosync.c. The constant CY_FX_SLFIFO_GPIF_16_32BIT_CONF_SELECT is defined in this file. Setting this to 0 will select 16 bit and 1 will select 32 bit. This constant is also used to configure the IO matrix for 16/32 bit GPIF in cyfxslfifosync.c.
- cyfxslfifosync.c: This file contains the main application logic of the Slave FIFO example. The application is explained in the subsequent sections.

8.1.1 Initialization Code

Figure 8-1 shows the initialization sequence of an FX3 application. Each of the main initialization blocks is explained here.

Figure 8-1. Initializing Sequence



8.1.1.1 *Firmware Entry*

The entry point for the FX3 firmware is CyU3PFirmwareEntry() function. The function is defined in the FX3 API library and is not visible to the user. As part of the linker options, the entry point is be specified as the CyU3PFirmwareEntry() function.

The firmware entry function performs the following actions:

1. Invalidates the caches (which were used by the bootloader)
2. Initialize the MMU (Memory Management Unit) and the caches
3. Initializes the SYS, FIQ, IRQ and SVC modes of stacks
4. The execution is then transferred to the Tool chain initialization (CyU3PToolChainInit()) function.

8.1.1.2 Tool Chain Initialization

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

As all the required stack initialization is performed by the firmware entry function, the toolchain init does not need to perform these operations. The only operation performed by the CyU3PToolChainInit function is to zero-out the contents of the BSS area, which holds all un-initialized data used by the firmware application.

The tool chain initialization function written for the GNU GCC compiler for ARM processors is presented as an example below.

```
.global CyU3PToolChainInit
CyU3PToolChainInit:
# Clear the BSS area
__main:
mov    R0, #0
ldr    R1, =_bss_start
ldr    R2, =_bss_end
1:cmp  R1, R2
strlo  R0, [R1], #4
blo    1b
b      main
```

In this function, only two actions are performed:

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

8.1.1.3 main() function

The main() function is the C programming language entry for the firmware application, and is defined in the *cyfxslfifosync.c* file. Four main actions are performed in this function.

1. Device initialization: This is the first step in the firmware.

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

As part of the device initialization:

   a. The CPU clock is setup. Passing NULL as the argument for CyU3PDeviceInit() selects the default clock configuration.
   b. The Vectored Interrupt Controller (VIC) is initialized
   c. The GCTL and the PLLs are configured.

The CyU3PDeviceInit() function is part of the FX3 library

2. Device cache configuration: The second step is to enable the device caches. The device has 8KB data cache and 8KB instruction cache. In this example, only the instruction cache is enabled as the data cache is useful only when there is a large amount of CPU based memory accesses.

When used in simple cases, it can decrease performance due to large number of cache flushes and cleans and it also adds complexity to the code.

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

3. I/O matrix configuration: The third step is the configuration of the I/Os that are required. This includes the GPIF and the serial interfaces (SPI, I2C, I2S, GPIO, and UART).

```
o_cfg.useUart = CyTrue;
io_cfg.useI2C = CyFalse;
io_cfg.useI2S = CyFalse;
io_cfg.useSpi = CyFalse;

  #if
     (CY_FX_SLFIFO_GPIF_16_32BIT_CONF_SELECT == 0)
     io_cfg.isDQ32Bit = CyFalse;
     io_cfg.lppMode = CY_U3P_IO_MATRIX_LPP_UART_ONLY;
  #else
     io_cfg.isDQ32Bit = CyTrue;
     io_cfg.lppMode = CY_U3P_IO_MATRIX_LPP_DEFAULT;
  #endif

/* No GPIOs are enabled. */
io_cfg.gpioSimpleEn[0] = 0;
io_cfg.gpioSimpleEn[1] = 0;
io_cfg.gpioComplexEn[0] = 0;
io_cfg.gpioComplexEn[1] = 0;
io_cfg.s0Mode = CY_U3P_SPORT_INACTIVE;
status = CyU3PDeviceConfigureIOMatrix (&io_cfg);
if (status != CY_U3P_SUCCESS)
{
  goto handle_fatal_error;
}
```

In this example:

a. The setting of CY_FX_SLFIFO_GPIF_16_32BIT_CONF_SELECT is used to configure the GPIF in 32- or16-bit mode.
b. GPIO, I2C, I2S and SPI are not used.
c. UART is used.
d. Both S ports are not available and are therefore disabled.

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

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

a. Initializes the OS

b. Sets up the OS timer
c. Starts and transfers control to the RTOS scheduler

8.1.1.4 *Application Definition*

The function CyFxApplicationDefine() is called by the FX3 library after the OS is invoked. In this function application specific threads are created.

In the Slave FIFO example, only one thread is created in the application define function. This is shown below:

```
/* Allocate the memory for the thread */
ptr = CyU3PMemAlloc (CY_FX_SLFIFO_THREAD_STACK);

/* Create the thread for the application */
retThrdCreate = CyU3PThreadCreate (&slFifoAppThread,
       /* Slave FIFO app thread structure            */
       "21:Slave_FIFO_sync",
       /* Thread ID and thread name                  */
       SlFifoAppThread_Entry,
       /* Slave FIFO app thread entry function       */
       0,
       /* No input parameter to thread               */
       ptr,
       /* Pointer to the allocated thread stack      */
       CY_FX_SLFIFO_THREAD_STACK,
       /* App Thread  stack size                     */
       CY_FX_SLFIFO_THREAD_PRIORITY,
       /* App Thread priority                        */
       CY_FX_SLFIFO_THREAD_PRIORITY,
       /* App Thread pre-emption threshold           */
       CYU3P_NO_TIME_SLICE,
       /* No time slice for the application thread   */
       CYU3P_AUTO_START
       /* Start the thread immediately               */
       );
```

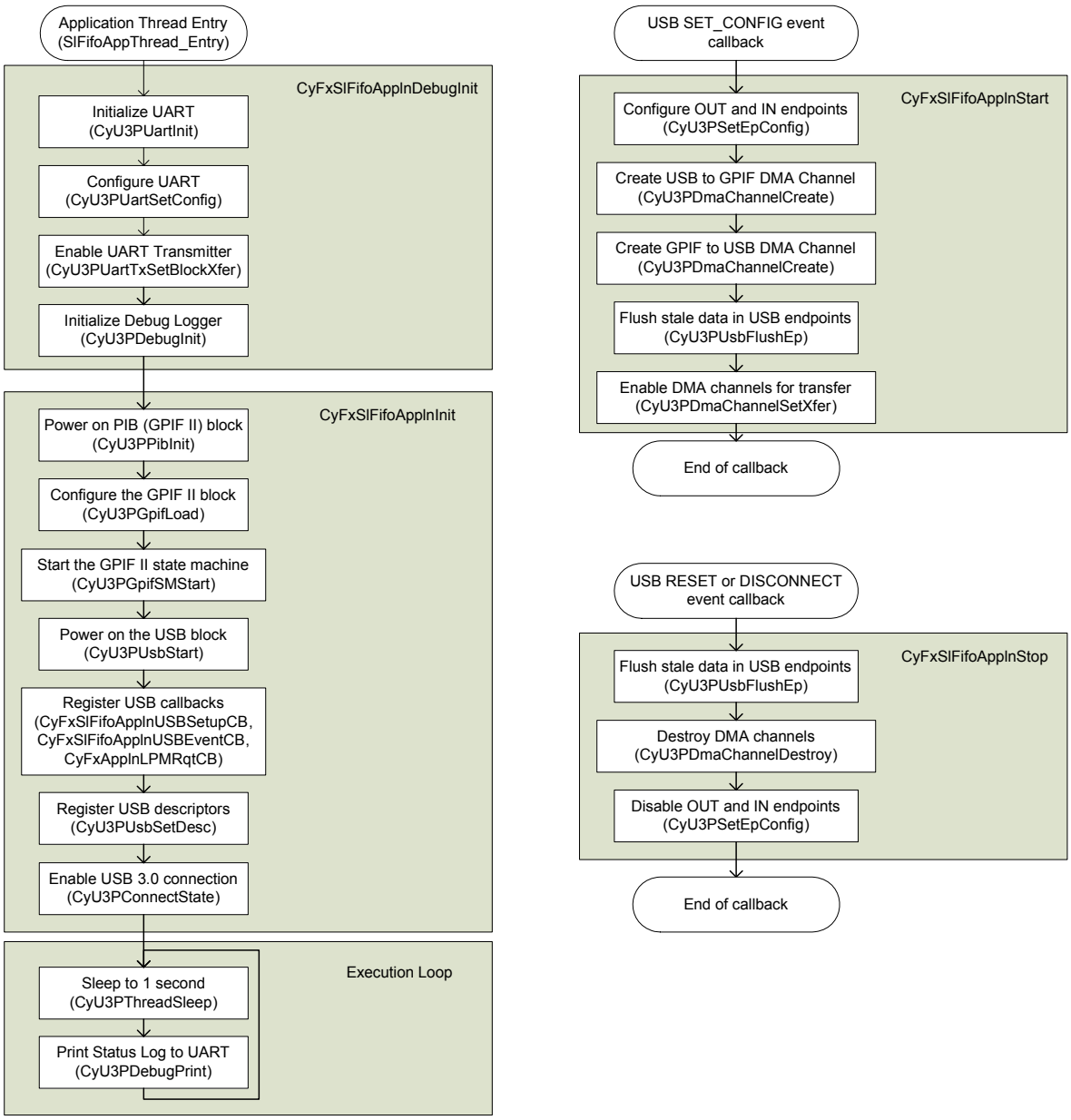
Note that more threads (as required by the user application) can be created in the application define function. All other FX3 specific programming must be done only in the user threads.

8.1.2 Application Code

The Slave FIFO application code is composed of three parts:

- The application thread function performs the application-specific initialization operations such as GPIF-II and USB block configuration.
- The CyFxSlFifoApplnStart function is called when a USB SET_CONFIGURATION request is received, and sets up the endpoints and DMA channels required for the USB <-> GPIF data transfers.
- The CyFxSlFifoApplnStop function is called when a USB reset or disconnect is detected and disables the USB endpoints and frees up the DMA channels.

Figure 8-2. Slave FIFO Application Code



8.1.2.1 *Application Thread*

The Application entry point for the Slave FIFO example is the SlFifoAppThread_Entry () function.

```
void
SlFifoAppThread_Entry (uint32_t input)
{
    /* Initialize the debug module */
    CyFxSlFifoApplnDebugInit();

    /* Initialize the slave FIFO application */
    CyFxSlFifoApplnInit();
```

```
    for (;;)
    {
       CyU3PThreadSleep (1000);
       if (glIsApplnActive)
       {
      /* Print the number of buffers received so far from the USB host. */
           CyU3PDebugPrint (6, "Data tracker: buffers received: %d, \
                       buffers sent: %d.\n", glDMARxCount, glDMATxCount);
       }
    }
}
```

The main actions performed in this thread are:

1. Initializing the debug mechanism
2. Initializing the main slave FIFO application

Each of these steps is explained below

8.1.2.2 *Debug Initialization*

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

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

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

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

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

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

- Finally the Debug module is initialized. The two main parameters are:
  - The destination for the debug prints, which is the UART socket

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

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

8.1.2.3 *Application initialization*

The application initialization consists of the following steps:

- GPIF-II Initialization
- USB Initialization

GPIF-II Initialization

The GPIF-II block is first initialized.

```
/* Initialize the P-port Block */
pibClock.clkDiv = 2;
pibClock.clkSrc = CY_U3P_SYS_CLK;
pibClock.isHalfDiv = CyFalse;
pibClock.isDllEnable = CyFalse;
apiRetStatus = CyU3PPibInit (CyTrue, &pibClock);
```

The slave FIFO descriptors are loaded into the GPIF-II registers and the state machine is started.

```
/* Load the GPIF configuration for Slave FIFO sync mode. */
apiRetStatus = CyU3PGpifLoad (&Sync_Slave_Fifo_2Bit_CyFxGpifConfig);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "CyU3PGpifLoad failed, \
                         Error Code = %d\n", apiRetStatus);
    CyFxAppErrorHandler(apiRetStatus);
}

/* Start the state machine. */
apiRetStatus = CyU3PGpifSMStart (SYNC_SLAVE_FIFO_2BIT_RESET,

SYNC_SLAVE_FIFO_2BIT_ALPHA_RESET);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "CyU3PGpifSMStart failed, \
                         Error Code = %d\n", apiRetStatus);
    CyFxAppErrorHandler(apiRetStatus);
}

/* Start the state machine. */
apiRetStatus = CyU3PGpifSMStart (SYNC_SLAVE_FIFO_2BIT_RESET,
                SYNC_SLAVE_FIFO_2BIT_ALPHA_RESET);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "CyU3PGpifSMStart failed, \
                         Error Code = %d\n", apiRetStatus);
    CyFxAppErrorHandler(apiRetStatus);
}
```

USB Initialization

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

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

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

```
/* The fast enumeration is the easiest way to setup a USB connection,
 * where all enumeration phase is handled by the library. Only the
 * class/vendor requests need to be handled by the application. */
CyU3PUsbRegisterSetupCallback (CyFxSlFifoApplnUSBSetupCB, CyTrue);

/* Setup the callback to handle the USB events. */
CyU3PUsbRegisterEventCallback (CyFxSlFifoApplnUSBEventCB);
```

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

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

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

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

- The USB pins are connected. The FX3 USB device is visible to the host only after this action. Hence it is important that all setup is completed before the USB pins are connected.

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

8.1.2.4 Endpoint Setup

The endpoints are configured on receiving a SET_CONFIGURATION request. Two endpoints: 1 IN and 1 OUT are configured as bulk endpoints. The endpoint maxPacketSize is updated based on the USB connection speed.

```
CyU3PUSBSpeed_t usbSpeed = CyU3PUsbGetSpeed();

/* First identify the usb speed. Once that is identified,
 * create a DMA channel and start the transfer on this.
 * Based on the Bus Speed configure the endpoint packet size. */
switch (usbSpeed)
```

```
{
case CY_U3P_FULL_SPEED:
    size = 64;
    break;
case CY_U3P_HIGH_SPEED:
    size = 512;
    break;
case CY_U3P_SUPER_SPEED:
    size = 1024;
    break;
default:
    CyU3PDebugPrint (4, "Error! Invalid USB speed.\n");
    CyFxAppErrorHandler (CY_U3P_ERROR_FAILURE);
    break;
}

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

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

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

8.1.2.5 USB Setup Callback

This callback function is called whenever the USB driver in the FX3 firmware framework needs the application to respond to a USB control request (SETUP command). Since the fast enumeration model is used, only class- and vendor-specific requests and standard requests addressed to an interface or endpoint need to be handled by the application. Other standard requests are handled by the firmware library. This example does not implement class- or vendor-specific requests.

```
CyBool_t
CyFxSlFifoApplnUSBSetupCB (uint32_t setupdat0,
                                                uint32_t setupdat1)
{
    /* Fast enumeration is used. Only class, vendor and unknown
     * requests are received by this function. These are not
     * handled in this application. Hence return CyFalse. */
    return CyFalse;
}
```

8.1.2.6 USB Event Callback

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

```
/* This is the callback function to handle the USB events. */
void
CyFxSlFifoApplnUSBEventCB (CyU3PUsbEventType_t evtype,
                                              uint16_t evdata)
{
    switch (evtype)
    {
    case CY_U3P_USB_EVENT_SETCONF:
        /* Stop the application before re-starting. */
        if (glIsApplnActive)
        {
            CyFxSlFifoApplnStop ();
        }
        /* Start the loop back function. */
        CyFxSlFifoApplnStart ();
        break;
    case CY_U3P_USB_EVENT_RESET:
    case CY_U3P_USB_EVENT_DISCONNECT:
        /* Stop the loop back function. */
        if (glIsApplnActive)
        {
            CyFxSlFifoApplnStop ();
        }
        break;
    default:
        break;
    }
}
```

8.1.2.7 DMA Setup

- The Slave FIFO application uses 2 DMA Manual channels. These channels are setup once a Set Configuration is received from the USB host. The DMA buffer size is fixed based on the USB connection speed.

```
 /* Create a DMA MANUAL channel for U2P transfer.
 * DMA size is set based on the USB speed. */
dmaCfg.size = size;
dmaCfg.count = CY_FX_SLFIFO_DMA_BUF_COUNT;
dmaCfg.prodSckId = CY_FX_PRODUCER_USB_SOCKET;
dmaCfg.consSckId = CY_FX_CONSUMER_PPORT_SOCKET;
dmaCfg.dmaMode = CY_U3P_DMA_MODE_BYTE;

/* Enabling the callback for produce event. */
dmaCfg.notification = CY_U3P_DMA_CB_PROD_EVENT;
dmaCfg.cb = CyFxSlFifoUtoPDmaCallback;
dmaCfg.prodHeader = 0;
dmaCfg.prodFooter = 0;
dmaCfg.consHeader = 0;
dmaCfg.prodAvailCount = 0;
apiRetStatus = CyU3PDmaChannelCreate (&glChHandleSlFifoUtoP,
                                CY_U3P_DMA_TYPE_MANUAL, &dmaCfg);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "CyU3PDmaChannelCreate failed, \
                         Error code = %d\n", apiRetStatus);
    CyFxAppErrorHandler(apiRetStatus);
}
/* Create a DMA MANUAL channel for P2U transfer. */
dmaCfg.prodSckId = CY_FX_PRODUCER_PPORT_SOCKET;
dmaCfg.consSckId = CY_FX_CONSUMER_USB_SOCKET;
dmaCfg.cb = CyFxSlFifoPtoUDmaCallback;
apiRetStatus = CyU3PDmaChannelCreate (&glChHandleSlFifoPtoU,
                                CY_U3P_DMA_TYPE_MANUAL, &dmaCfg);
if (apiRetStatus != CY_U3P_SUCCESS)
{
    CyU3PDebugPrint (4, "CyU3PDmaChannelCreate failed, \
                         Error code = %d\n", apiRetStatus);
    CyFxAppErrorHandler(apiRetStatus);
}
```

- The DMA channel transfers are enabled

```
/* Set DMA Channel transfer size */
apiRetStatus = CyU3PDmaChannelSetXfer (&glChHandleSlFifoUtoP,
                                CY_FX_SLFIFO_DMA_TX_SIZE);
/* Set DMA Channel transfer size */
apiRetStatus = CyU3PDmaChannelSetXfer (&glChHandleSlFifoPtoU,
                                CY_FX_SLFIFO_DMA_TX_SIZE);
```

8.1.2.8 DMA Callback

In this example, there are two data paths

- U to P
- P to U

A DMA callback is registered for the DMA Produce Events on each path. This event will occur when a DMA buffer is available (with data) from

- The USB OUT endpoint
- The GPIF-II socket

In the DMA callback, this buffer is committed, passing on the data to the

- The GPIF-II socket
- USB In Endpoint

The two call back functions are shown below

```
/* DMA callback function to handle the produce events for U to P
 * transfers. */
void
CyFxSlFifoUtoPDmaCallback (CyU3PDmaChannel *chHandle,
                  CyU3PDmaCbType_t type,CyU3PDmaCBInput_t *input)
{
    CyU3PReturnStatus_t status = CY_U3P_SUCCESS;
    if (type == CY_U3P_DMA_CB_PROD_EVENT)
    {
        /* This is a produce event notification to the CPU.
        * This notification is received upon reception of
        * every buffer. The buffer will not be sent out
        * unless it is explicitly committed. The call shall
        * fail if there is a bus reset/usb disconnect or if
        * there is any application error. */
        status = CyU3PDmaChannelCommitBuffer (chHandle,
                            input->buffer_p.count, 0);
        if (status != CY_U3P_SUCCESS)
        {
            CyU3PDebugPrint (4, "CyU3PDmaChannelCommitBuffer failed, \
                                 Error code = %d\n", status);
        }
        /* Increment the counter. */
        glDMARxCount++;
    }
}
```

9. FX3 Serial Peripheral Register Access

9.1 Serial Peripheral (LPP) Registers

The EZ-USB FX3 device implements a set of serial peripheral interfaces (I2S, I^{2}C, UART, and SPI) that can be used to talk to other devices. This chapter lists the FX3 device registers that provide control and status information for each of these interfaces.

9.1.1 I2S Registers

The I2S interface on the FX3 device is a master interface that is can output stereophonic data at different sampling rates. This section documents the control and status registers related to the I2S interface.

Name	Width (bits)	Address	Description
I2S_CONFIG	32	0xE0000000	Configurations and modes register
I2S_STATUS	32	0xE0000004	Status register
I2S_INTR	32	0xE0000008	Interrupt request (status) register
I2S_INTR_MASK	32	0xE000000C	Interrupt mask register
I2S_EGRESS_DATA_LEFT	32	0xE0000010	Left channel egress data register
I2S_EGRESS_DATA_RIGHT	32	0xE0000014	Right channel egress data register
I2S_COUNTER	32	0xE0000018	Sample counter register
I2S_SOCKET	32	0xE000001C	Socket register
I2S_ID	32	0xE00003F0	Block Id register
I2S_POWER	32	0xE00003F4	Power, clock and reset control register

Refer to Section 10.18 of EZ-USB® FX3™ Technical Reference Manual for details.

9.1.2 I^{2}C Registers

The FX3 device provides an I^{2}C master block that can be configured to talk to various slave devices at different transfer rates. This section documents all of the registers related to the I^{2}C interface.

Name	Width (bits)	Address	Description
I2C_CONFIG	32	0xE0000400	Configuration and modes register
I2C_STATUS	32	0xE0000404	Status register
I2C_INTR	32	0xE0000408	Interrupt status register
I2C_INTR_MASK	32	0xE000040C	Interrupt mask register
I2C_TIMEOUT	32	0xE0000410	Bus timeout register
I2C_DMA_TIMEOUT	32	0xE0000414	DMA transfer timeout register
I2C_PREAMBLE_CTRL	32	0xE0000418	Specify start/stop bit locations during preamble (command and address) phase.

Name	Width (bits)	Address	Description
I2C_PREAMBLE_DATA0	32	0xE000041C	Data to be sent during preamble phase – Word 0
I2C_PREAMBLE_DATA1	32	0xE0000420	Data to be sent during preamble phase – Word 1
I2C_PREAMBLE_RPT	32	0xE0000424	Settings for repeating the preamble for polling the device status.
I2C_COMMAND	32	0xE0000428	Command register to initiate I2C transfers
I2C_EGRESS_DATA	32	0xE000042C	Write data register
I2C_INGRESS_DATA	32	0xE0000430	Read data register
I2C_BYTE_COUNT	32	0xE0000438	Desired size of data transfer
I2C_BYTES_TRANSFERRED	32	0xE000043C	Remaining size for the current data transfer
I2C_SOCKET	32	0xE0000440	Selects DMA sockets for I2C data transfers
I2C_ID	32	0xE00007F0	Block ID register
I2C_POWER	32	0xE00007F4	Power and reset register

Refer to Section 10.19 of EZ-USB® FX3™ Technical Reference Manual for details.

9.1.3 *UART Registers*

The FX3 device implements a UART block that can communicate with other UART controllers at different baud rates and supports different communication parameters, parity settings, and flow control. This section documents the UART related configuration and status registers.

Name	Width (bits)	Address	Description
UART_CONFIG	32	0xE0000800	Configuration and modes register
UART_STATUS	32	0xE0000804	Status register
UART_INTR	32	0xE0000808	Interrupt status register
UART_INTR_MASK	32	0xE000080C	Interrupt mask register
UART_EGRESS_DATA	32	0xE0000810	Write data register
UART_INGRESS_DATA	32	0xE0000814	Read data register
UART_SOCKET	32	0xE0000818	Socket selection register
UART_RX_BYTE_COUNT	32	0xE000081C	Receive byte count register
UART_TX_BYTE_COUNT	32	0xE0000820	Transmit byte count register
UART_ID	32	0xE0000BF0	Block ID register
UART_POWER	32	0xE0000BF4	Power and reset control register

Refer to Section 10.20 of EZ-USB® FX3™ Technical Reference Manual for details.

9.1.4 SPI Registers

The SPI block on the FX3 device is a SPI master interface that can be configured with different word sizes, clock frequencies and modes of operation. While the block automatically supports the use of default Slave Select (SSN) pin, the firmware can use other GPIOs on the FX3 device to talk to other slaves.

This section documents the SPI related configuration and status registers.

Name	Width (bits)	Address	Description
SPI_CONFIG	32	0xE0000C00	Configuration and modes register
SPI_STATUS	32	0xE0000C04	Status register
SPI_INTR	32	0xE0000C08	Interrupt status register
SPI_INTR_MASK	32	0xE0000C0C	Interrupt mask register
SPI_EGRESS_DATA	32	0xE0000C10	Write data register
SPI_INGRESS_DATA	32	0xE0000C14	Read data register
SPI_SOCKET	32	0xE0000C18	Socket select register
SPI_RX_BYTE_COUNT	32	0xE0000C1C	Receive byte count register
SPI_TX_BYTE_COUNT	32	0xE0000C20	Transmit byte count register
SPI_ID	32	0xE0000FF0	Block ID register
SPI_POWER	32	0xE0000FF4	Power and reset register

Refer to Section 10.21 of EZ-USB® FX3™ Technical Reference Manual for details.

9.2 FX3 GPIO Register Interface

FX3 device has 61 I/Os, which can be individually configured as GPIOs or as dedicated peripheral lines. The GPIO itself can be simple or complex according to the use case. A simple GPIO can act as an output line or an input line with interrupt capability. A complex GPIO (pin) can act as a timer/counter or PWM in addition to its input and output functionality.

Any I/O can be configured as a simple GPIO, but only eight of the I/Os can be used as complex GPIOs. Only one of the I/Os with the same result for (GPIO number % 8) can be chosen as a complex GPIO.

9.2.1 Simple GPIO Registers

The following table lists key registers of the simple GPIO interface.

Table 9-1. Simple GPIO Registers

Address	Qty	Width	Name	Description
0xE0001100 + GPIO_NUM * 0x04	61	32	GPIO_SIMPLE	Simple general purpose i/o register (per GPIO)
0xE00013D0	1	32	GPIO_INVALUE0	GPIO input value vector
0xE00013D4	1	32	GPIO_INVALUE1	GPIO input value vector
0xE00013E0	1	32	GPIO_INTR0	GPIO interrupt vector
0xE00013E4	1	32	GPIO_INTR1	GPIO interrupt vector
0xE00013F0	1	32	GPIO_ID	Block identification and version number
0xE00013F4	1	32	GPIO_POWER	Power, clock, and reset control

Refer to Section 10.22 of EZ-USB® FX3™ Technical Reference Manual for details.

9.3 Complex GPIO (PIN) Registers

The following table lists key registers of the complex GPIO interface.

Table 9-2. Complex GPIO Registers

Address	Qty	Width	Name	Description
0xE0001000 + (GPIO_ID MOD 8) * 0x10	8	128	PIN	General purpose I/O registers (one pin)[a]
0xE00013E8	1	32	GPIO_PIN_INTR	GPIO interrupt vector for PINs

a. See table 3 for further break up of each 128 bit register.

The following table lists the pins registers for complex GPIO interface.

Table 9-3. Complex GPIO Registers

Offset	Width	Name	Description
0xE0001000 + (GPIO_ID MOD 8) * 0x10	32	PIN_STATUS	Configuration, mode and status of I/O Pin.
0xE0001004 + (GPIO_ID MOD 8) * 0x10	32	PIN_TIMER	Timer/counter for pulse and measurement modes.
0xE0001008 + (GPIO_ID MOD 8) * 0x10	32	PIN_PERIOD	Period length for revolving counter/timer (GPIO_TIMER).
0xE000100C + (GPIO_ID MOD 8) * 0x10	32	PIN_THRESHOLD	Threshold for measurement register.

Refer to Section 10.23 of EZ-USB® FX3™ Technical Reference Manual for details.

10. FX3 P-Port Register Access

FX3's processor interface is a General Programmable parallel interface (GPIF) that can be programmed to operate with:

1. 8-bit address
2. 8-, 16-, or 32-bit data bus widths
3. Burst sizes: 2 thru 2^14
4. Buffer sizes: 16 * n; for n 1 thru 2048: Max buffer size of 32 KB
5. Buffer switching overhead 550 - 900 ns per buffer

The PP Register protocol enables data transfers between Application Processor and FX3's parallel port.

The PP Register protocol uses a special register map of 16-bit registers that can be accessed through the GPIF interface. These addresses are not accessible internal to the device.

The PP register protocol as seen through GPIF uses the following mechanisms:

P-port addresses in the range 0x80-FF expose a bank of 128 16-bit registers. Some of these registers are used to implement the mechanisms mentioned below, and others control configuration, status and behavior of the P-Port.

Accesses to any address in the range 0x00-7F are used for FIFO-style read or write operations into the 'active' socket (to be explained later).

Special registers implement the 'mailbox' protocol. These registers are use to transfer messages of up to 8-bytes between the Application Processor and the FX3 CPU.

10.1 Glossary

Egress	Direction of transfer - out of FX3 to external device
Ingress	Direction of transfer – into FX3 from external device
Long transfer	DMA transfer spanning multiple buffers
Short transfer	DMA transfer spanning one buffer
Full Transfer	DMA transfer comprising of full buffer
Partial transfer	DMA transfer comprising of less than full buffer
Zero Length transfer	Transfer of a zero length buffer (packet)
ZLB	Zero length buffer

10.2 Externally Visible PP Registers

The following table lists key registers of the P-port used in initialization and data transfer across the P-port.

Table 10-1. PP Register

Description		Processor Port Register Map in GPIF space	
Offset	**Width**	**Name**	**Description**
0x80	16	PP_ID	P-Port Device ID Register
0x81	16	PP_INIT	P-Port reset and power control
0x82	16	PP_CONFIG	P-Port Configuration Register
0x88	16	PP_INTR_MASK	P-Port Interrupt Mask Register
0x89	16	PP_DRQR5_MASK	P-Port DRQ/R5 Mask Register
0x8A	32	PP_SOCK_MASK	P-Port Socket Mask Register
0x8C	16	PP_ERROR	P-Port Error Indicator Register
0x8E	16	PP_DMA_XFER	P-Port DMA Transfer Register
0x8F	16	PP_DMA_SIZE	P-Port DMA Transfer Size Register
0x90	64	PP_WR_MAILBOX	P-Port Write (Ingress) Mailbox Registers
0x99	16	PP_EVENT	P-Port Event Register
0x9A	64	PP_RD_MAILBOX	P-Port Read (Egress) Mailbox Registers
0x9E	32	PP_SOCK_STAT	P-Port Socket Status Register

Refer to Section 10.8 of EZ-USB® FX3™ Technical Reference Manual for details.

10.3 INTR and DRQ Signaling

INTR signal is derived by a bitwise OR of (PP_EVENT & PP_INTR_MASK) registers. This allows AP to selectively program PP_INTR_MASK for desired exception and socket events.

Typically, status bit DMA_WMARK_EV or DMA_READY_EV is made available as a DRQ signal by configuring PP_DRQR5_MASK. It is possible to combine DMA_READY or DMA_WMARK with a handshake DACK signal from AP, if required. This is done using a programmable GPIF state machine. For the rest of this document use of DACK is not considered.

The polarity of both DRQ and INTR signals is configurable.

10.4 Transferring Data In and Out of Sockets

The following section describes how the Application Processor transfers blocks of data into and out of active sockets using the PP Register protocol. All GPIF PP-Mode-based transfers use the same internal mechanisms, but may differ in the usage of DRQ/INTR signaling.

This section discusses how read and write transfers are implemented for full buffer, partial buffer, zero-length packets and long transfers. A distinction is made between short (single buffer) and long (multiple buffer) transfers. For long transfer it is assumed that a higher level protocol (for example, through use of mailboxes) has communicated the need for transfer of large size between AP and ARM CPU.

All transfers are based on the following command, config and status bits on PP_* interface:

- **PP_SOCK_STAT.SOCK_STAT[N]**. For each socket this status bit indicates that a socket has a buffer available to exchange data (it has either data or space available).
- **PP_DMA_XFER.DMA_READY**. This status bit indicates whether the P-Port is ready to service reads from or writes to the active socket (the active socket is selected through the PP_DMA_XFER register). PP_EVENT.DMA_READY_EV mirrors PP_DMA_XFER.DMA_READY with a short delay of a few cycles.
- **PP_EVENT.DMA_WMARK_EV.** This status bit is similar to DMA_READY, but it de-asserts a programmable number of words before the current buffer is completely exchanged. It can be used to create flow control signals with offset latencies in the signaling interface.
- **PP_DMA_XFER.LONG_TRANSFER.** This config bit indicates if long (multi-buffer) transfers are enabled. This bit is set by Application Processor as part of transfer initiation.
- **PP_DMA_XFER.DMA_ENABLE.** This command and status indicates that DMA transfers are enabled. This bit is set by Application Processor as part of transfer initiation and cleared by FX3 hardware upon transfer completion for short transfers and by Application Processor for long transfers.

10.4.1 Bursting and DMA_WMARK

AP transfers data on the interface in bursts of 1 or more words at a time. The burst size for transfers may be configured to be any power of 2 words.

The DMA_WMARK status is generated in the GPIF controller counting back a configurable number of words from the end of the last burst needed to transfer all the data for a buffer (this may be fewer bursts than fit the maximum buffer size).

Burst size and watermark distance is programmable across a range of values.

10.4.2 Short Transfer - Full Buffer

A full size write (ingress) transfer is defined as a transfer that fills the entire buffer space available. If this buffer space is known in advance (due to higher level protocol), it is not required to read DMA_SIZE.

A full size read (egress) transfer is defined as a transfer that reads all the available data from the buffer – this may be less than the actual max buffer size.

Note

In the figures that illustrate transfers:

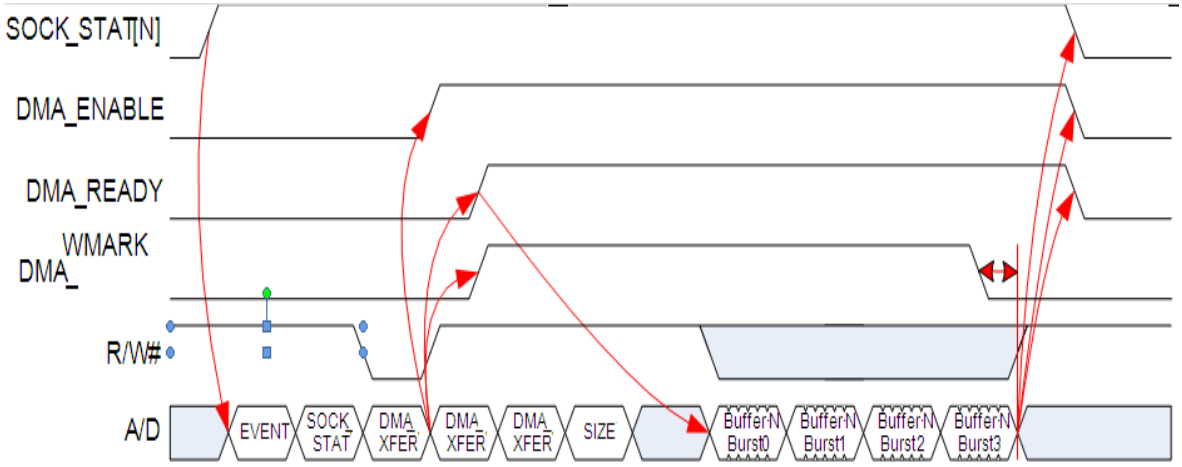
Values of SOCK_STAT[N], DMA_ENABLE, DMA_READY and DMA_WMARK are shown as signals; these are really values of status bits in PP_* registers.

R/W# indicates read or write operation on the interface. Interface protocol specific (such as, async, sync SRAM, or ADMux) signals are not shown.

A/D depicts Address and Data bus independent of interface protocol.

Full size transfer is illustrated in the following figure.

Figure 10-1. Full-sized DMA Transfers



1. AP configures PP_SOCK_STAT[N] becomes active when data or space is available.
2. AP configures PP_SOCK_MASK and PP_INTR_MASK register to enable interrupt when socket is ready with data or empty-buffer for data transfer.
3. Application processor detects this interrupt signaling and reads registers PP_EVENT and PP_SOCK_STAT_x to know which socket to work on.
4. AP activates socket by writing socket-number, direction and DMA_ENABLE=1 to PP_DMA_XFER. This causes DMA_ENABLE to assert[1] in a few cycles. AP uses LONG_TRANSFER=0 for short (single buffer) transfer.
5. Optional register reads are performed from PP_DMA_XFER until PP_DMA_XFER.SIZE_VALID asserts, after which PP_DMA_SIZE is read to determine buffer/data size. Multiple reads may be required before the SIZE_VALID asserts.
6. PP_EVENT.DMA_READY asserts after socket has been activated to indicate data transfer can start. This can occur before, during or after the DMA_SIZE read mentioned above.
7. DMA_SIZE bytes are transferred in an integral number of full bursts. The number of bursts is rounded up and data beyond the size of buffer for ingress is ignored; reads beyond the size of data for egress return 0.
8. PP_EVENT.DMA_WMARK de-asserts shortly after the watermark has passed (see burst section above). Due to pipelining of the interface it may take a couple of cycles before this signal physically de-assert in the GPIF state machine.

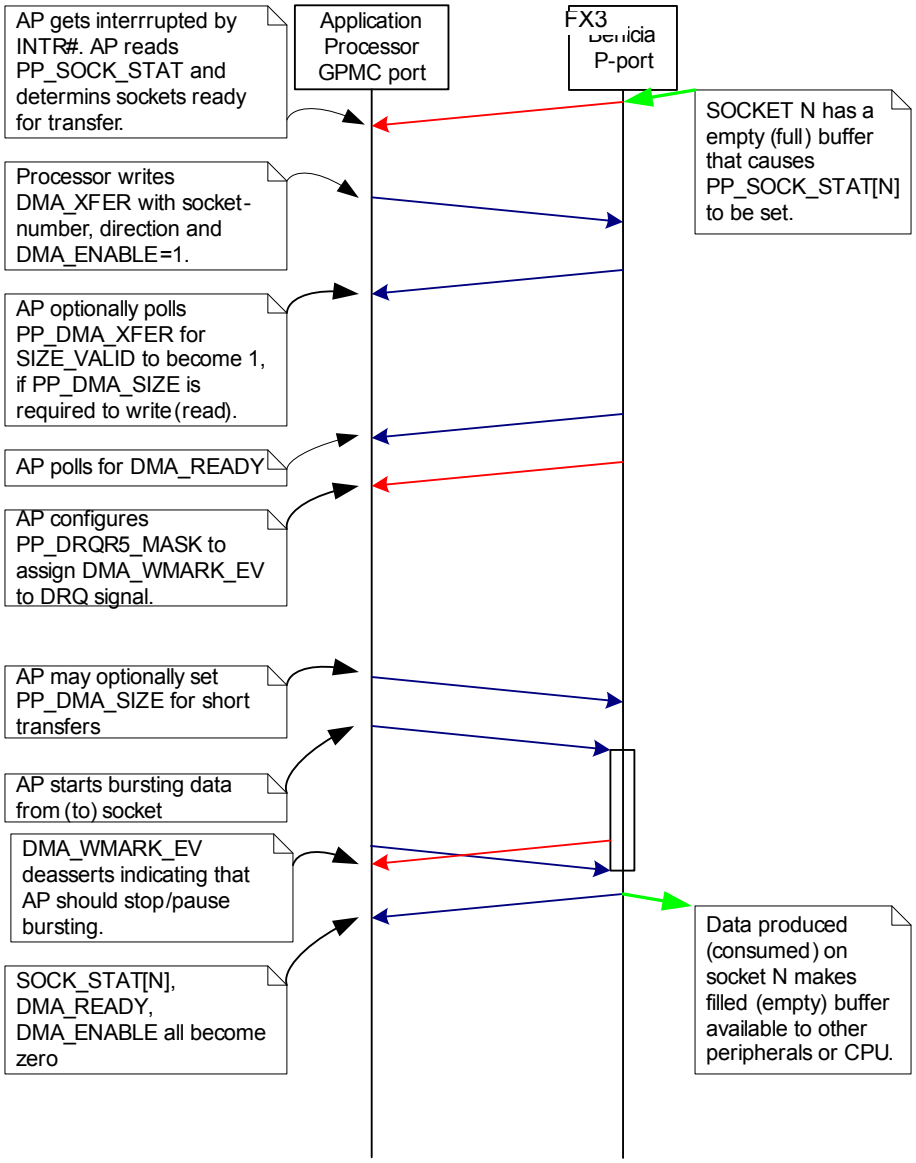
SOCK_STAT[N], DMA_READY and DMA_ENABLE all become zero a few cycles after the last word of the last burst has been transferred.

If data words are written when DMA_READY=0 (beyond the end of the buffer or after an error condition occurred), data will be ignored. Reads under these circumstances will return 0. These reads or writes themselves represent an error condition if one is not already flagged. Upon (any) error DMA_ERROR becomes active and DMA_READY de-asserts.

1. Asserting a status bit implies setting status bit to 1; and de-asserting a status bit implies setting the status-bit to 0.

The following diagrams illustrate the sequence of events at the P-Port for read and write transfers:

Figure 10-2. Short Transfer - DMA Transfer Sequence



10.4.3 Short Transfer – Partial Buffer

A partial write (ingress) transfer is defined as a transfer that writes fewer bytes than the available space in the buffer.

A partial read (egress) transfer is defined as a transfer that transfers less than the number of bytes available in the current buffer.

The normal mechanism for a partial transfer is to write the number of bytes to transfer into DMA_SIZE. This can be done only after a DMA_XFER.SIZE_VALID asserted. It is also possible to explicitly terminate a transfer by clearing DMA_ENABLE in DMA_XFER. Note that in that case, it is not possible to transfer an odd number of bytes.

Note that simply reading or writing fewer than DMA_SIZE bytes does **not** terminate the transfer – the remaining bytes can be read at any time. Only when DMA_SIZE bytes have been transferred or when DMA_ENABLE is explicitly cleared (by writing to DMA_XFER) does the transfer end.

The following diagrams illustrate both the normal partial and aborted transfer:

Figure 10-3. Partial DMA Transfers

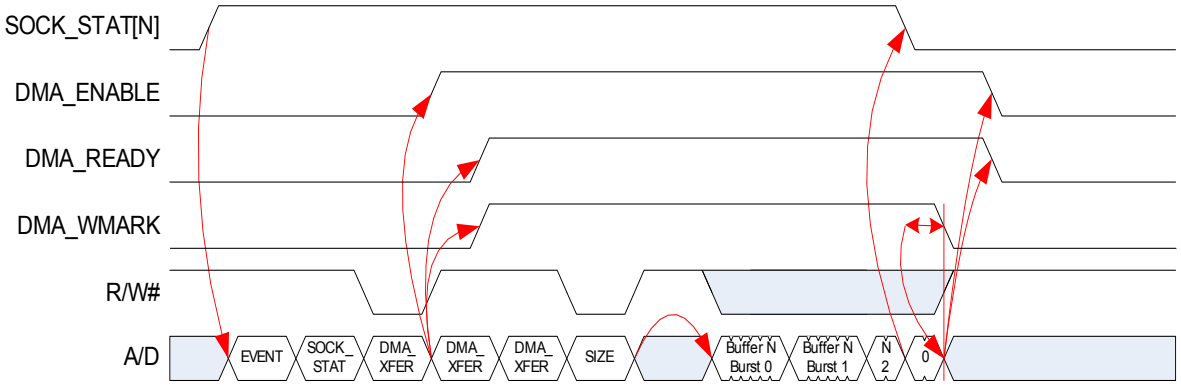
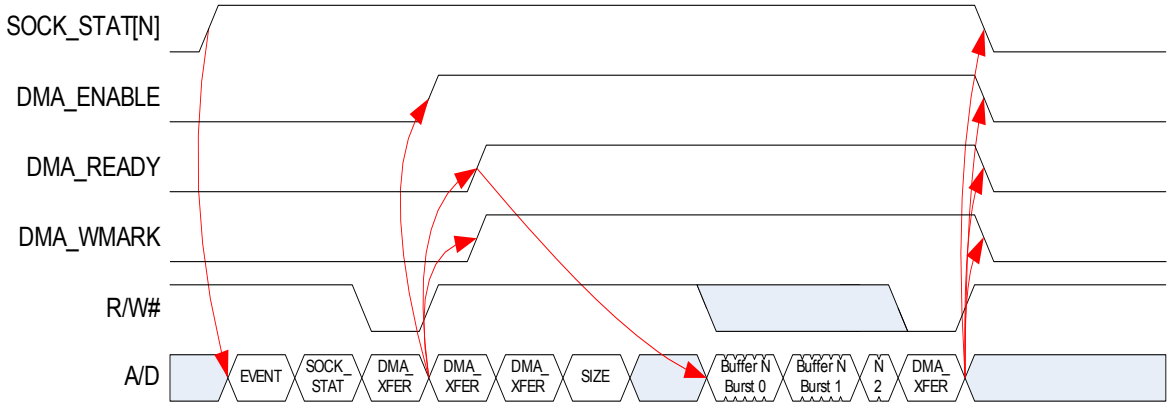


Figure 10-4. Aborted DMA Transfers



The following should be noted:

- DMA_WMARK de-asserts when either its watermark position is reached or shortly after the transfer is aborted, whichever occurs earlier.
- SOCK_STAT[N] de-asserts (and so will the INTR based on it) shortly after all data for the current buffer is exchanged. However, DMA_READY and DMA_ENABLE remain asserted until the last burst if fully completed (or the transfer is aborted).

A transfer can be aborted in the middle of a burst, assuming the AP is capable of transferring a partial burst.

10.4.4 Short Transfer – Zero Length Buffers

When a zero byte buffer is available for read (egress), no data words are transferred and DMA_READY will never assert. AP observes this by reading DMA_SIZE=0.

When a zero length buffer needs to written (ingress), the AP will write DMA_SIZE=0. The transfer terminates automatically with no data exchanged.

The AP can observe the completion of a ZLB transfer by polling the DMA_XFER register. This should not take more than a couple of cycles.

Figure 10-5. Zero Length Read (Egress) Transfer

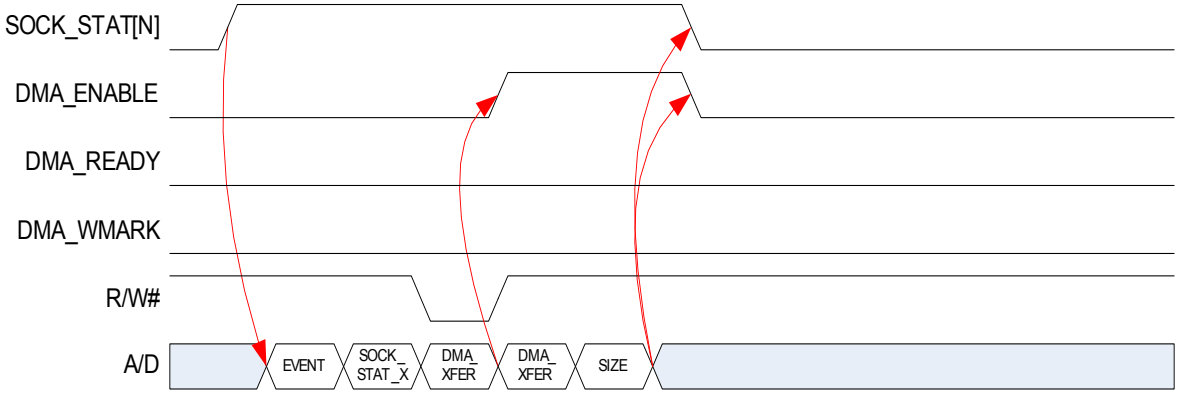
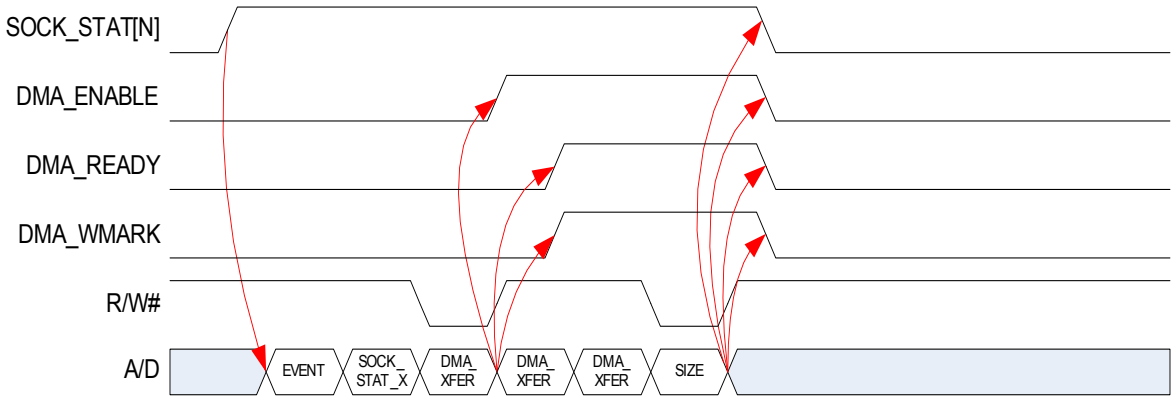


Figure 10-6. Zero Length Write (Ingress) Transfer



The following should be noted:

For read transfers, DMA_ENABLE de-asserts shortly after a DMA_XFER read with SIZE_VALID=1.

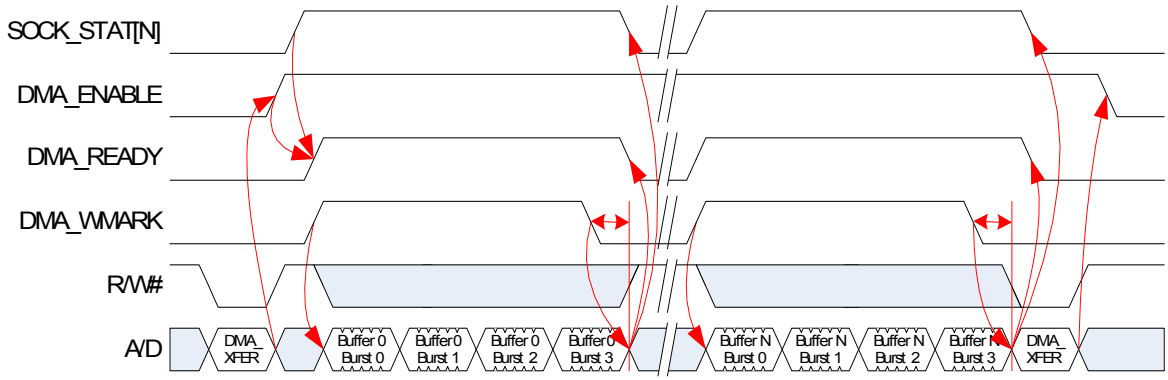
For write transfers, all signals de-assert shortly after the write of DMA_SIZE=0.

10.4.5 Long Transfer – Integral Number of Buffers

A long transfer is coordinated between AP and FX3 CPU using a higher layer protocol; for example, built on mailbox messaging. The length of transfer is conveyed to FX3 CPU which configures buffers and sockets required for transfer.

The following diagram illustrates the long transfer:

Figure 10-7. Long Transfer With Integral Number Of Buffers



The following should be noted:

The transfer is setup in the P-port socket by the FX3 CPU, resulting in SOCK_STAT[N] asserting at some point into the transfer to initiate transfer of the first buffer.

The AP initiates the transfer by writing DMA_ENABLE=1, LONG_TRANSFER=1 along with DMA_SOCK and DMA_DIRECTION to DMA_XFER. This may take place before, during or after SOCK_STAT[N] asserts.

When both SOCK_STAT[N] and DMA_ENABLE are asserted, DMA_READY and DMA_WMARK assert.

The AP now transfers data in full bursts until DMA_WMARK de-asserts. Each time this happens, the AP must wait until DMA_READY and DMA_WMARK re-assert.

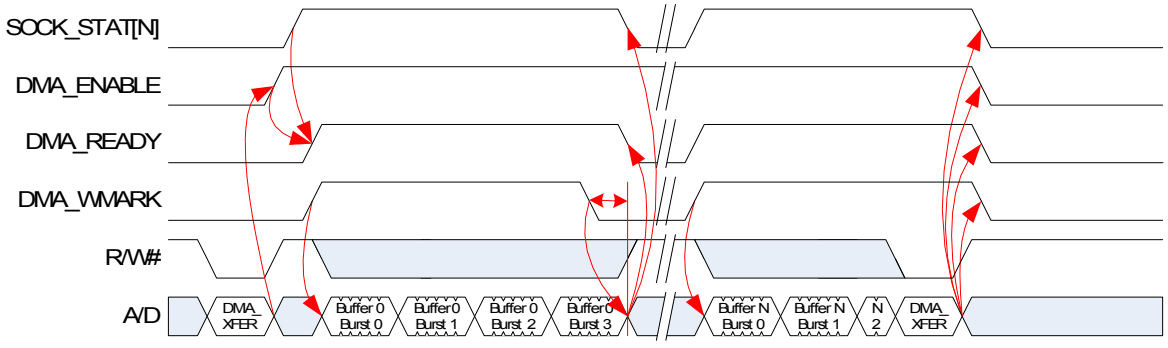
When enough data is transferred, the AP must terminate the transfer by writing DMA_ENABLE=0.

10.4.6 Long Transfer – Aborted by AP

A long transfer can be aborted by AP by writing DMA_ENABLE=0 at any time and follow it with a mailbox message to wrap up the partially written buffer.

The following diagram illustrates the working of an aborted long transfer:

Figure 10-8. Aborted Long Transfer



The following should be noted:

DMA_WMARK de-asserts when either DMA_ENABLE is cleared or the configured water mark position is reached, whichever occurs sooner.

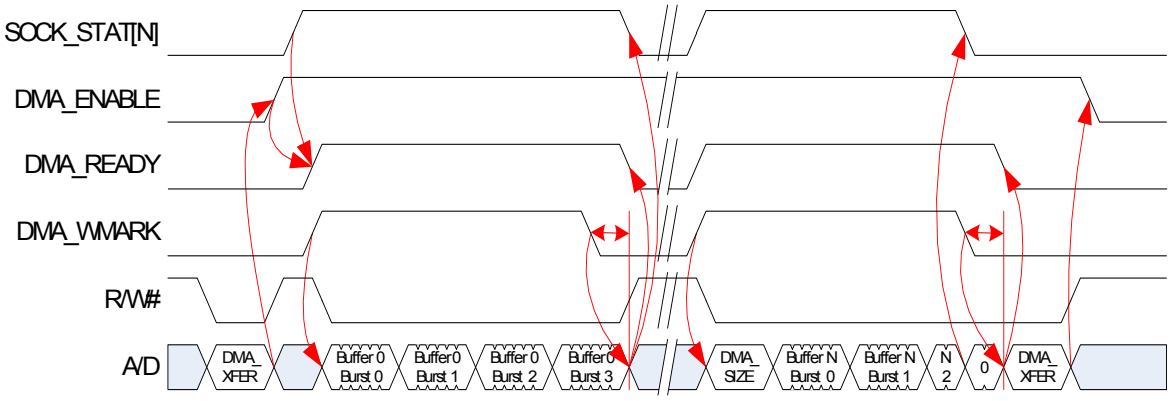
The AP may abort a transfer in the middle of a burst or at the end of a burst. Note that it is not possible to implement a transfer of a non-integral number of bursts using the AP abort mechanism. This requires the adjustment of the DMA_SIZE register as illustrated in the next section.

10.4.7 Long Transfer – Partial Last Buffer on Ingress

When a long ingress transfer has a partial last buffer, this buffer can be preceded by an adjustment by AP of DMA_SIZE. If no such adjustment is made and the transfer is shorter than the coordinated transfer size (that is set into the DMA Adapter's trans_size by firmware), it is possible to exchange whole words/bursts only.

The following diagram illustrates this concept:

Figure 10-9. Ingress Long Transfer with a Partial Last Buffer



The following should be noted:

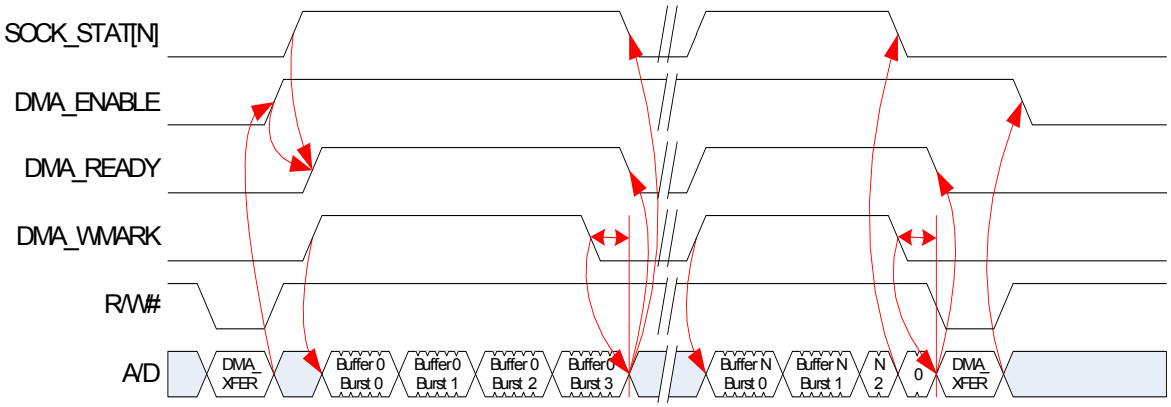
Before transferring the last buffer, the AP adjusts DMA_SIZE. AP must assure that DMA_READY=1 or DMA_WMARK=1 before writing to DMA_SIZE. This can be done using the signals directly (for example, through INTR/DRQ signaling) or by explicitly polling for DMA_SIZE.VALID=1.

If no adjustment of DMA_SIZE is made, but the long transfer itself was indicated to have a non-integral number of bursts, the DMA Adapter will truncate any data written by AP beyond the end of the transfer. In other words, this mechanism is not required and AP may terminate the transfer after writing the last full burst.

10.4.8 Long Transfer – Partial Last Buffer on Egress

On egress, a partial last buffer results in early de-assertion of DMA_WMARK but is otherwise no different from a transfer of a whole number of buffers. The following diagram illustrates this:

Figure 10-10. Egress Long Transfer - with Partial Last Buffer



10.4.9 Odd-Sized Transfers

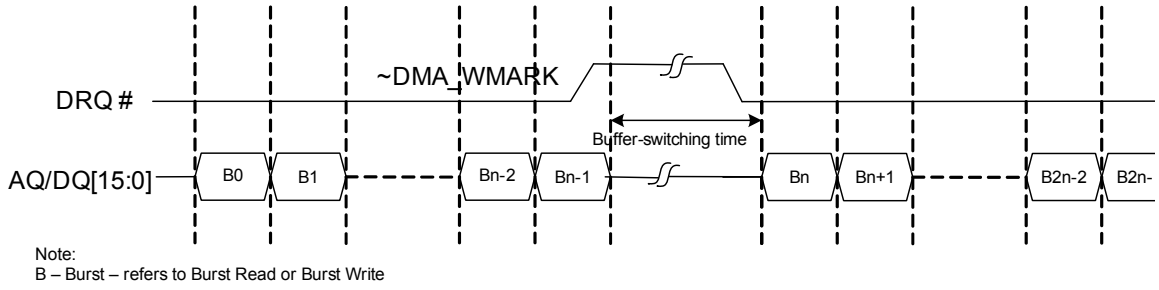
Whenever a packet is transferred that consists of an odd number of bytes, all words transferred are full words except the last one. The last word will contain only the valid bytes padded with 0. In big endian mode this will be the MSB in little endian mode this will be the LSB. The other bytes will contain 0 on read and will be ignored on write.

This scheme is independent of how the memory buffer is aligned in memory inside of FX3 (which is irrelevant to the application processor). In other words the first word of a transfer is always a full word, even if the buffer is misaligned in internal memory.

10.4.10 DMA transfer signalING on ADMUX interface

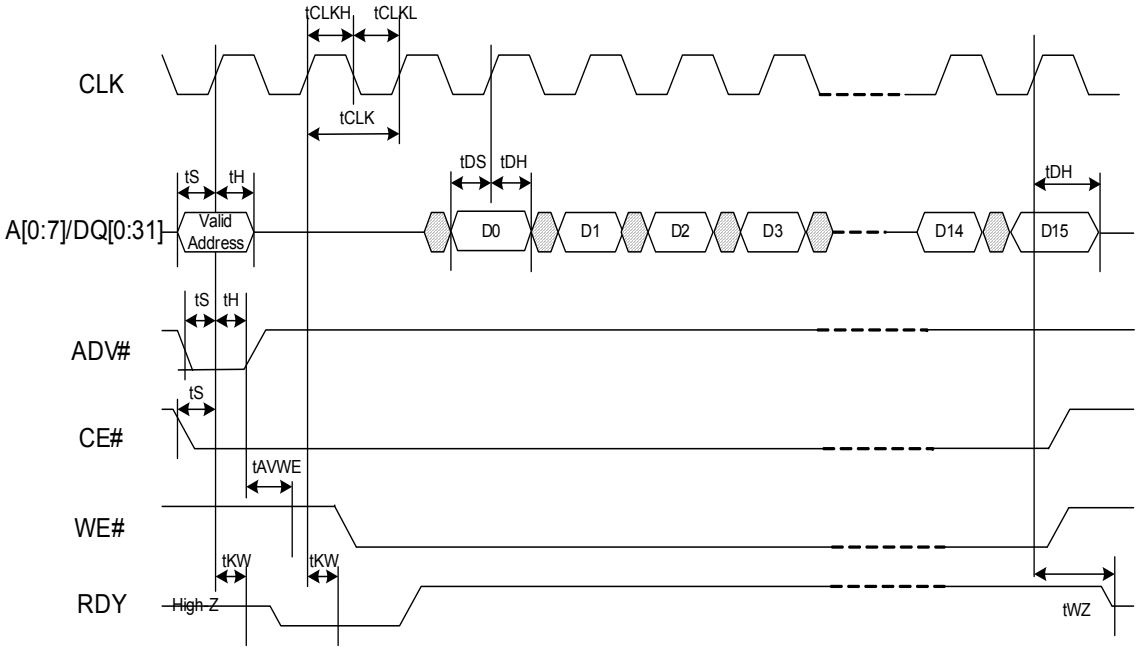
The figure illustrates DRQ# signaling on P-port interface for a long transfer. DMA_WMARK is mapped to DRQ# signal. Note that DRQ is programmed active-low in this example. The buffer-switching time is illustrated as the time from the last data cycle for a buffer to the first cycle of next buffer.

Figure 10-11. Long Transfer Using DMA_WMARK Mapped to DRQ# Signal



Each burst, B*x*, in Figure 10-11 is comprises of one address and burst-size data cycles. One example for a burst-of-16-read on ADMux interface is illustrated in Figure 10-12. Note that the RDY signal shown in the figure is the link level ready signal. This RDY signal is different from the higher level DMA control DMA_READY/DRQ signaling.

Figure 10-12. Burst of 16 Read on ADMux Interface



Note:
1. 2-cycle latency is shown.
2. RDY active high shown. RDY can be programmed to either active low or active high

11. FX3 Boot Image Format

The FX3 bootloader is capable of booting various sources based on the PMODE pin setting. The bootloader requires the firmware image to be created with the following format:

11.1 Firmware Image Storage Format

Binary Image Header	Length (16-bit)	Description
wSignature	1	Signature 2 bytes initialize with “CY” ASCII text
bImageCTL;	½	For I2C/SPI EEPROM boot: Bit0 = 0: execution binary file; 1: data file type Bit3:1 (I2C size. Not used when booting from SPI EEPROM) 7: 128 KB (Micro chip) 6: 64 KB (128K ATMEL) 5: 32 KB 4: 16 KB 3: 8 KB 2: 4 KB **Note** Options 1 and 0 are reserved for future usage. Unpredictable result will occurred when booting in these modes. Bit5:4(I2C speed on I2C Boot): 00: 100 kHz 01: 400 kHz 10: 1 MHz 11: 3.4MHz (reserved) Bit5:4(SPI speed on SPI boot): 00: 10 MHz 01: 20 MHz 10: 30 MHz 11: 40 MHz (reserved). **Note** On I2C boot, bootloader power-up default will be set at 100 kHz and it will adjust the I2C speed if needed. On SPI boot, bootloader power-up default is set to 10 MHz and it will adjust the SPI speed if needed. Bit7:6: Reserved should be set to zero
bImageType;	½	bImageType=0xB0: normal FW binary image with checksum bImageType=0xB1: Reserved for security image type bImageType=0xB2: Boot with new VID and PID
dLength 0	2	1st section length, in long words (32-bit) When bImageType=0xB2, the dLength 0 will contain PID and VID. Boot Loader will ignore the rest of the any following data.
dAddress 0	2	1st sections address of Program Code not the I2C address. **Note** Internal ARM address is byte addressable, so the address for each section should be 32-bit align
dData[dLength 0]	dLength 0*2	All Image Code/Data also must be 32-bit align
…		More sections
dLength N	2	0x00000000 (Last record: termination section)

Binary Image Header	Length (16-bit)	Description
dAddress N	2	Should contain valid Program Entry (Normally, it should be the startup code or the RESET Vector) Note if bImageCTL.bit0 = 1, the Boot Loader will not transfer the execution to this Program Entry. If bImageCTL.bit0 = 0, the Boot Loader will transfer the execution to this Program Entry: This address should be in ITCM area or SYSTEM RAM area Boot Loader does not validate the Program Entry
dCheckSum	2	32-bit unsigned little endian checksum data will start from the 1st sections to termination section. The checksum will not include the dLength, dAddress and Image Header

12. FX3 Development Tools

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

The firmware development environment will help the user to develop, build and debug firmware applications for FX3. The third party ARM® software development tool provides an integrated development environment (IDE) with compiler, linker, assembler and debugger (via JTAG). Free GNU tool-chain and Eclipse IDE (to be used with the GNU tool-chain) is provided.

12.1 GNU Toolchain

The GNU Toolchain provided as part of the FX3 SDK comprises of

- GCC compiler (gcc) – version 4.8.1
- GNU Linker (ld) – version 2.23.52
- GNU Assembler (as) – version 2.23.52
- GNU Debugger (gdb) – version 7.6.50

These executables are invoked by the Eclipse IDE.

12.2 Eclipse IDE

The Eclipse IDE for C/C++ Developer is provided as part of the FX3 SDK. This IDE comprises of the base Eclipse platform (4.3.2) and the C/C++ Development Toolchain (8.3.0). The eclipse plug-ins required for firmware development and debugging are bundled with the IDE.

- GNU ARM Eclipse Plug-in (2.4.2)

This plug-in ties the Eclipse IDE to the GNU ARM tool-chain, and supports managed firmware builds and debug sessions using Segger J-Link and OpenOCD.

- Java™ Platform, Standard Edition Runtime Environment Version 7

The JRE is required by eclipse.

Refer to the EZUSBSuite User Guide document in the `doc/firmware` folder of the FX3 SDK installation for detailed instructions on using the Eclipse IDE for firmware development and debugging.

13. FX3 Host Software

13.1 FX3 Host Software

A comprehensive host side (Microsoft Windows) stack is included in the FX3 SDK. This stack includes:

- Cypress generic USB 3.0/2.0 driver (WDF) on Windows 7 (32/64 bit), Windows Vista (32/64 bit), and Windows XP (32 bit only)
- 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

13.1.1 Cypress Generic Driver

The Cypress driver is general-purpose, understanding primitive USB commands, but not implementing higher-level, USB device-class specific commands. The driver is ideal for communicating with a vendor-specific device from a custom USB application. It can be used to send low-level USB requests to any USB device for experimental or diagnostic applications.

Refer to *CyUSB.pdf* in the Cypress SuperSpeed USBSuite installation for more details.

13.1.2 CYAPI Programmer's Reference

CyAPI.lib provides a simple, powerful C++ programming interface to USB devices. It is a C++ class library that provides a high-level programming interface to the CyUsb3.sys device driver. The library is only able to communicate with USB devices that are served by this driver.

Kindly refer to the CyAPI.pdf in the Cypress SuperSpeed USBSuite installation for more details.

13.1.3 CYUSB.NET Programmer's Reference

CyUSB.dll is a managed Microsoft .NET class library. It provides a high-level, powerful programming interface to USB devices and allows access to USB devices via library methods. Because CyUSB.dll is a managed .NET library, its classes and methods can be accessed from any of the Microsoft Visual Stuido.NET managed languages such as Visual Basic.NET, C#, Visual J# and managed C++.

Refer to the CyUSB.NET.pdf in the Cypress SuperSpeed USBSuite installation for more details.

13.1.4 Cy Control Center

USB ControlCenter is a C Sharp application that is used to communicate with Cypress USB devices that are served by CyUSB3.sys device driver.

Refer to the CyControlCenter.pdf in the Cypress SuperSpeed USBSuite installation for more details.

14. GPIF™ II Designer

GPIF™ II Designer provides a graphical user interface to configure the processor port of EZ-USB FX3 to connect to external devices. The interface between EZ-USB FX3 and the external device can be specified as a state machine, and the GPIF-II Designer tool will generate the register configuration in the form of a header file that can be readily integrated with the FX3 firmware application using the FX3 API library. Refer to the GPIF II Designer User Guide for more details on the tool.

The FX3 SDK includes the GPIF-II configuration header files for various Slave-FIFO modes. Please see the *Getting Started Guide* and the Application Note *AN65974 - Designing with EX-USB FX3 Slave FIFO Interface* for details on using these configuration headers. Contact the Cypress USB support team for any queries on GPIF-II configuration.