USB 3.0 - SUPERSPEED DATA ACQUISITION SYSTEMS

A PROJECT REPORT

Submitted by

BL.EN.U4ECE18138	SATHYASRI S.
BL.EN.U4ECE18140	SHARIQ AKHTAR P.P.
BL.EN.U4ECE18157	TALLURI SAYANTH KISHORE
BL.EN.U4ECE18182	YASH RAJESH UMALE

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

AMRITA SCHOOL OF ENGINEERING, BANGALORE

AMRITA VISHWA VIDYAPEETHAM

BENGALURU 560 075

JANUARY 2022

AMRITA VISHWA VIDYAPEETHAM

AMRITA SCHOOL OF ENGINEERING, BENGALURU – 560035

BONAFIDE CERTIFICATE

This is to certify that the project report titled **"USB 3.0 – SuperSpeed Data Acquisition Systems"** submitted by

BL.EN.U4ECE18138	-	Sathyasri S.
BL.EN.U4ECE18140	-	Shariq Akhtar P.P.
BL.EN.U4ECE18157	-	Talluri Sayanth Kishore
BL.EN.U4ECE18182	-	Yash Rajesh Umale

in partial fulfillment of the requirements for the award of the degree **Bachelor of Technology** in **Electronics and Communication Engineering** is a bonafide record of the work carried out under our guidance and supervision at Amrita School of Engineering, Bangalore and Indian Space Research Organisation (ISRO).

Dr. Navin Kumar Project Guide HoD, Dept. of ECE, ASE-B	Mr. Srinidhi M S Engr SF, DSAS, DSD, PDMG	Ms. R Srividhya Head, DSAS, DSD, PDMG	Mr. Om Sai Ayyappa Reddy Sci/Engr-SD, DSAS, DSD, PDMG

Dr. Navin Kumar
Chairperson, Dept. of ECE

This project report was evaluated by us on ……

Ms. Priya B.K.	Dr. Salija P

भारत सरकार
अन्तरिक्ष विभाग
यू.आर. राव उपग्रह केन्द्र
पोस्ट बॉक्स नं. १७९५, ओल्ड एयरपोर्ट रोड
विमानपुरा डाक, बेंगलूरु - ५६० ०१७, भारत
दूरभाष :
फैक्स :

Government of India
Department of Space
U.R. RAO Satellite Centre
Post Box No. 1795, Old Airport Road
Vimanapura Post, Bengaluru - 560 017, India
Telephone :
Fax :

N0.020/ 2.3(2)/21-WS

August 09, 2021

To,

Chairman, Department of Electronics & Communication Engineering
Amrita School of Engineering
Bengaluru Campus, Kasavanahalli, Carmelaram P.O
Bengaluru - 560035

Dear Sir/Madam,

Sub: Project Work for BE/B.Tech students at U R RAO Satellite Centre

Please find enclosed list of the student of your college (batch no. 1 & 2) who has been provisionally accepted for project work in this Organization from **09th August 2021 to 30th April 2022.** The concerned students may please be intimated accordingly.

In this connection, it is requested to send the undersigned by post/speed post, a certificate issued by the Head of the Institution as to the conduct and bonafides of these students with:

a) College should send the bonafide certificate for confirmation through email or by post/courier only.
b) Signature of the student attested on the same certificate.

Above documents & information should reach the undersigned **not later than by 10.08.2021.** It is requested not to send the above documents by Courier.

The students may be instructed/informed to/that:

i. Telephonically confirm, if the undersigned has received above documents.
ii. Students are strictly not allowed inside the campus due to prevailing COVID situation.
iii. Students have to carry out the project work remotely (virtually) under the guidance of the guide allotted from the Organization.
iv. Students have to complete the project work in the prescribed duration. No further extension will be entertained.
v. Upon completion of project and confirmation by the respective guide, the project completion certificate will be sent to the students by post.

The project work will commence from **11th, August 2021.**

Yours faithfully,

(Anantha V)
Administrative officer (W)
Phone 080-25084110

Encl: List of student allotted BE project work

Batch No.	Students Name	College Name	Discipline
1	Shariq Akhtar P P	Amrita School of Engg.	Electronics & Comm. Engg.
1	Sathyasri S	Amrita School of Engg.	Electronics & Comm. Engg.
1	Talluri Sayanth Kishore	Amrita School of Engg.	Electronics & Comm. Engg.
1	Yash Rajesh Umale	Amrita School of Engg.	Electronics & Comm. Engg.
2	Suraj S M	Amrita School of Engg.	Electronics & Comm. Engg.
2	Peddinti Guru Sai Phanindra	Amrita School of Engg.	Electronics & Comm. Engg.
2	Sumukh D N	Amrita School of Engg.	Electronics & Comm. Engg.

College Address:
Chairman, Department of Electronics & Communication Engineering
Amrita School of Engineering
Bengaluru Campus, Kasavanahalli, Carmelaram P.O
Bengaluru- 560035, Karnataka

Acknowledgements

This project reflects upon the work done collaboratively by the scientists at ISRO and the guides at Amrita Vishwa Vidyapeetham. This report cannot be collated without giving credit where it is due, and we'd like to acknowledge the same.

We were extremely privileged to have received industry grade mentorship and assistance from our external guides, Ms. R Sreevidya (Head, DSAS, DSD, PDMG), Mr. Om Sai Ayyappa Reddy (Sci/Engr-SD, DSAS, DSD, PDMG) and Mr. Srinidhi M S (Engr SF, DSAS, DSD, PDMG) for their undying support and constant guidance.

The success of the project is a result of applying solutions based on constant feedback and advice provided by our team of internal guides.

It has been a privilege to have Dr. Navin Kumar and Dr. Salija P assist us from the commencement of this project. They have been supportive and have never hesitated to spoon – feed or rectify our recurring errors. Their high standards of wisdom, knowledge and delivery of work have inspired us to never settle for less than the best.

We also acknowledge and exercise our gratitude towards our parents and members of the family, who have supported us morally as well as economically. Furthermore, our gratitude goes to all our friends, who have directly or indirectly helped us in the completion of this project.

Finally, we extend our gratitude to Amrita Vishwa Vidyapeetham for giving us the opportunity to work in collaboration with ISRO in close company of such illustrious and inspirational individuals.

Abstract

The process of sampling signals that measure real-world physical conditions and converting the resulting samples into digital numeric values that can be manipulated by a computer or sensors is known as **data acquisition**.

Data acquisition systems, or **DAQ**, are critical in product testing. With the invention and development of data acquisition systems (DAQs) capable of collecting data from a wide range of sensors.

DAQ is classified into two types:

- Analog Data Acquisition Systems
- Digital Data Acquisition Systems

This report deals with recording the theoretical, design and implementational aspects of the project responsible for data acquisition from flight – worthy solid state recorders, and providing a test bench to execute a series of checks to gauge the health of the device. Furthermore, the project deals with implementing compression/ decompression algorithms among other additional features to optimize and provide greater throughput.

After substantial research, it is found that the most optimized approach (as discussed in the remainder of the report) involves the use of a USB 3.0 microcontroller along with an FPGA and a host PC to control data transfer and control lines. The desired outcome of the project is to successfully conduct loopback transfer within the microcontroller, and to design the requisite image/ firmware for the same.

Index – Contents, Figures and Tables

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

1.1 Overview

Space exploration is the use of astronomy and space technology to explore outer space. While the exploration of space is carried out mainly by astronomers with telescopes, its physical exploration though is conducted both by unmanned robotic space probes and human spaceflight.

Common rationales for exploring space include advancing scientific research, national prestige, uniting different nations, ensuring the future survival of humanity, and developing military and strategic advantages against other countries. The data received from Earth observational satellites, for instance, enables us to learn and prepare for weather and environmental changes, emergency response and resource management.

There are 4 different types of space missions that're widely undertaken for exploration: **flyby, orbiter, rovers** and **human space explorations.** Most of these missions logged data using **magnetic tape recorders**, thereby limiting the data gathering and storage capabilities of these missions. The recorders' mechanical systems also made them vulnerable to failure; backup recorders took up valuable onboard real estate and added to cost and launch weight. As these recorders proved to constitute a life-limiting component, a replacement was found in solid-state technology.

Solid-state, or flash memory devices are crafted entirely from solid materials in which the only moving parts are electrons. A **solid-state recorder (SSR)** stores information in binary digits (0s and 1s) in transistors [26], and since there are no moving parts that could fail, and the information in the recorder can only be altered by voltage changes in the transistors, solid-state technology offers a more reliable, durable, and compact method of data storage.

The SSRs provide a wide array of capabilities to accommodate a variety of data management requirements in terms of density, data rates, sensor channels, and data integrity [9]. A core feature of the SSR technology is ease of integration, facilitating the addition of advanced data management with minimal impact to the overall system architecture. The availability of multiple interfaces to these SSRs (for telemetry, telecommand, record/ playback, relays, voltage/ temperature measurements) makes them highly versatile, and conventional hardware setups **that only toggled the record/ playback interfaces** of these recorders need an upgrade so that the other low-frequency interfaces [30] are utilized in an implicit manner. These interfaces, which are currently treated as separate modular units and

deliverables, need to be continuously monitored and manually triggered [19] within the SSR.

The **Cypress FX3 SuperSpeed Explorer Kit** offers a one-package solution for the integration of all aforementioned interfaces, and further supports super-speed data transfers over USB 3.0 to other peripherals within the kit [17]. The project focusses on programming the FX3 board and the adjoint custom simulator over an FPGA [22] to run tests on the SSR over multiple interfaces in order to check the reliability of the recorder.

1.2 Objectives

Main objective:

To develop an end-to-end system that performs a series of tests on a solid-state recorder across multiple interfaces to gauge the device's reliability and sustainability for space-flight exploration and autonomous missions.

Sub-objectives:

- Configuration of the GPIF state machine of the Cypress FX3 SuperSpeed Explorer Kit to enable slave FIFO transactions between an external processor/ FPGA and the kit

- Programming the firmware of the Cypress FX3 SuperSpeed Explorer Kit in order to facilitate data transfers between interfaces and the buffers within the kit
    - Handling the data and instruction caches
    - Setting up the Serial Peripheral Interface (SPI) for transfers
    - Assigning channel parameters for DMA transfers within the FX3
    - Creating callback functions for P2U and U2P transfers over DMA channels
    - Creating callback functions to govern USB events, loopbacks and throughput
    - Defining USB descriptors to set rules on data transfers

- Configuring the FPGA using VHDL to accept commands from the USB host via FX3 and run tests on the SSR

- Designing the requisite user interface for a GUI-based or console-based application to perform the aforementioned checks while maintaining total abstraction.

1.3 Methodology

The development of a system to determine the flight-worthy nature of an SSR requires a thorough understanding of digital logic, ARM architecture and AMBA buses, along with programming in VHDL for the configuration of an external FPGA to run these tests.

The **Cypress FX3 SuperSpeed Explorer Kit** uses the AHB architecture for internal DMA and system data transfers, along with the GPIF and SPI interfaces for our application. The overall proposed architecture for the project is as follows:

- The USB host (either 2.0 or 3.0, usually a PC) is connected to the FX3 via its USB port. The PC will run an application responsible for providing the user with options to run specific tests on the SSR.

- These options are propagated via lanes from the origin USB endpoint in the PC to the destination USB endpoint within the FX3.
  USB Descriptors govern the nature of data transfer and perform callback operations based on the occurrence of USB events.

- While the USB host is connected to the FX3 on the USB front, the FX3 is further connected gluelessly to an external processor or FPGA via the **configurable GPIF pins**.

- The FPGA hosts a custom board simulator that is responsible for running tests across multiple interfaces, such as telecommand, telemetry, record/ playback interfaces.

- The FPGA is further connected to the Solid-State Recorder, which happens to be the subject of the tests.

From the overall structure of the project, the outline of the project along with its tasks becomes evident:

- A **slave-FIFO interface** needs to be implemented, and hence the GPIF pins of the FX3 need to be configured accordingly.

- The FX3 device firmware is important to
  - Define descriptors for USB, SPI, UART or GPIF transfers
  - Configure data transfer over DMA channels
  - Handle read and write operations to FX3's system memory
  - Assign callback functions to events on the USB or GPIF ports
  - Manage and maintain miscellaneous peripherals and debug mechanisms

The FX3 SDK shipped with the hardware is used to develop the firmware and all requisite dependencies along with it. The **GPIF II Designer** helps with creating config files for integrating the programmed GPIF ports with the firmware.

- Develop VHDL code for the FPGA in order to enable the conversion of logic-based tests from the USB host to ARM level instructions.

- Create the GUI for a Windows application to run these checks while maintaining maximum abstraction on the underlying hardware.

Firmware development relies on the third-party support extended by the FX3 SDK, and the internal functionality of the FX3 is handled by the **ThreadX RTOS** wrapped by the Cypress modules along with the usage of **AHB buses** for data and instruction transfer between CPU and RAM within the FX3.

1.3.1 About Universal Serial Bus (USB) – 2.0 and 3.0

The Universal Serial Bus (USB) has gained wide acceptance as the connection method of choice for PC peripherals. It provides easy attachment, an end to configuration hassles and true plug-and-play operation.

1.3.2 Implementation: Configuring the programmable GPIF Pins

FX3 has a fully configurable General Programmable Interface (GPIF™ II) that can interface with any processor, ASIC, image sensor, or field-programmable gate array (FPGA). It provides easy and glueless connectivity to popular industry interfaces such as synchronous slave FIFO, and can be configured using the **GPIF II Designer Tool**.

1.4 Expected Result

At the stage of project completion, the deliverables expected are as follows:

- An end-to-end software application that enables the user to view any connected solid-state recorders, evaluate its health, run a variety of approved checks and view its reliability in a quantitative manner

- A hardware setup consisting of a USB host (PC), a Cypress FX3 USB SuperSpeed Explorer Kit, an FPGA mounted on a custom board simulator and interfaced with the FX3 with an objective to effectively test the readiness of any given SSR and facilitate high-speed transactions.

1.5 Time Table and Schedule

Project Phase	Task	Tentative Completion
Phase - I	Configuration of the GPIF state machine and I/O matrix	October 2021 (Completed)
	Testing the GPIF state machine with timing diagrams, streamer module and USB Control Utility	October 2021 (Completed)
	Provide DMA transfer of data for record and playback within the FX3	November 2021 (Completed)
	Programming the firmware of the Cypress FX3 SuperSpeed Explorer Kit in order to facilitate data transfers.	November 2021 (Completed)
	Testing Bulk-IN data transfers via the FX3 to the FPGA or USB host and vice versa and recording observations.	December 2021 (Completed)
	Verifying the loopback transfer of data from USB host or external master to the FX3	December 2021 (Completed)

Table 1.1: Abstract Project Plan and Time Table (Phase – I)

Project Phase	Task	Tentative Completion
Phase - II	Implementing a 1553 Bus Controller and building software to target the same	January – March 2022
	Generating and receiving command packets for 1553 BC, pulse command and simulator FPGA to forward over an SPI interface	February – March 2022
	Configuring the FPGA using VHDL to accept commands from the USB host via FX3 and run tests on the SSR	March 2022
	Implementing compression and decompression algorithms to handle image and video data from SSR on the fly via FPGA	
	Implementing the command receiver to receive and interpret commands over SPI to the FPGA	April 2022
	Defining and executing appropriate record/ playback protocols for data retrieval from SSR	
	(Optional) – Use FPGA compute resources to perform video analysis and image processing	
	Redesign FX3 firmware and FPGA configurations to facilitate real-time video streaming to FPGA/ from SSR	April – May 2022
	Designing an application using Visual Studio to issue telecommand, telemetry and toggle compression for incoming or outgoing data	

Table 1.2: Abstract Project Plan and Time Table (Phase – II)

Chapter 2

State of Art

2.1 Introduction

Data transfer in the field of space exploration is a highly complex task with multiple technological barriers as far as wired and wireless connections are concerned. In order to ensure the reliability and self-sufficiency of components pertaining to data storage and collection in aerial and space missions, the project introduces a fully encapsulated system for testing solid-state recorders with appropriate industry-approved tests.

The following chapter covers the core principles and building blocks within one of the most fundamental components of the system: **Cypress FX3 SuperSpeed Explorer Kit**. Since the initial phase requires a thorough understanding and implementation of logic over the FX3 microcontroller, all concepts pertaining to ARM9 and AHB architecture, DMA transfers, USB and GPIF peripherals will be discussed in detail with slight references to an equally important component, the **Field Programmable Gate Array (FPGA)** responsible for issuing commands for running tests on the SSR.

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 developers to add USB 3.0 to any system. The microcontroller acts as an intermediate between the USB host and the external processor (FPGA) as earlier depicted in Figure 1.1.

In order to accurately configure and define initial conditions and functions for the smooth and synchronized execution of peripherals, a background analysis for all participating components is included further.

2.2 Project Background with Related Literature and Work

2.2.1 About the Cypress FX3 SuperSpeed Explorer Kit

Cypress EZ-USB FX3 is the next generation **USB 3.0 peripheral controller**, which provides highly integrated and flexible features that enable developers to add USB 3.0 functionality to any system, as demonstrated in paper [16] by Anuja et al. It is designed to offer signalling rates of **5 Gbps** (10 times greater than USB 2.0's signalling rate of **480 Mbps**).

As the FX3 is USB 3.0 compliant, in addition to operating at the new SuperSpeed data rate, it can also operate at the High-Speed and Full-Speed USB data rates. FX3 integrates the USB 3.0 and USB 2.0 physical layers along with a **32-bit ARM9 microprocessor** for powerful data processing and for building custom USB SuperSpeed applications using [14] as a development guide.

A bird's eye view of the internal mechanism of the FX3 is depicted below.

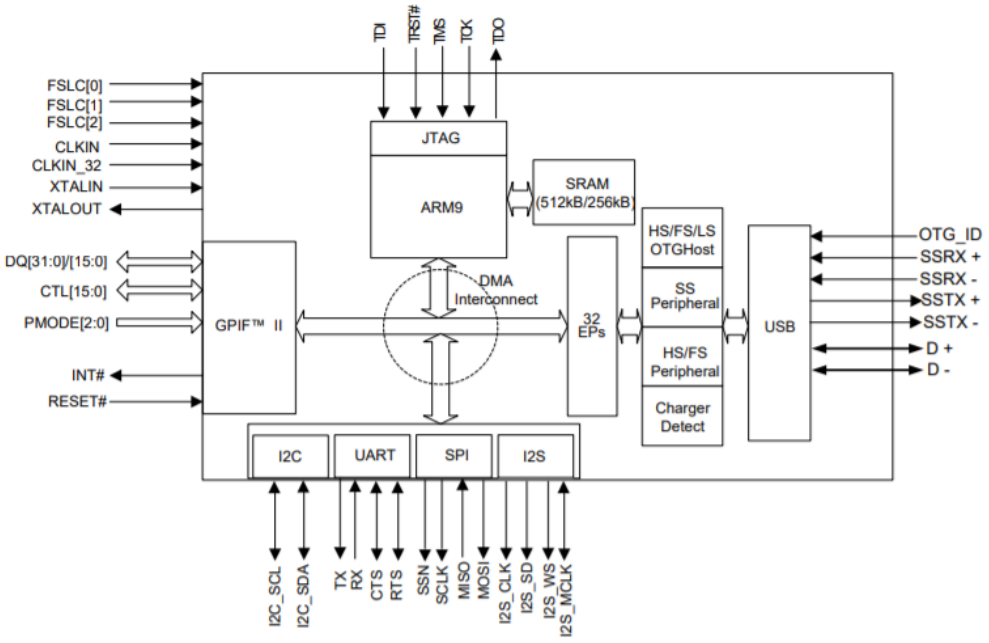


Figure 2.1: *Block Diagram of the FX3 SuperSpeed Explorer Kit*

As summarised by Y. Gong et al [26] in their research for a test bench, the kit:

- Integrates USB 3.0 and 2.0 physical layers along with a **32-bit microprocessor** for powerful data processing and for building custom USB SuperSpeed applications.

- Contains **512 kB of on-chip SRAM** for code and data, and provides interfaces to connect to serial peripherals such as UART, I2C, SPI and I2S.

- Provides a **JTAG interface** for firmware debugging via the ARM9 SoC.

- Enables efficient and flexible **DMA transfers** between peripherals, and thus requires firmware to configure data access between them.

The main function of the FX3 device in a system is to transfer high-bandwidth data between a USB host and a peripheral device, like a camera or scanner, as demonstrated by Y. Gong et al [26]. The presence of a powerful **ARM9 processor** on-chip also allows FX3 to access the data stream and efficiently process data.

In systems where FX3 is not required to perform data processing, the ARM9 firmware only initializes and manages data transfers between two interfaces – USB and a data consuming/providing device.

The next few sections focus on the core principles and concepts associated with the interfaces and ports embedded within the FX3 [23], which has a highly flexible, programmable interface known as the **General Programmable Interface Generation 2 (GPIF II)** in addition to **I2C**, **SPI**, **UART**, and **I2S** serial interfaces.

2.2.2 Decoding the Universal Serial Bus (USB) – 2.0 and 3.0

A USB system is an asynchronous serial communication host-centric design, consisting of a single host and multiple devices and downstream hubs connected in a tiered-star topology as described in the research conducted by Guoren et al [28].
It consists of

- Half duplex two-wire signalling featuring unidirectional data flow with negotiated directional bus transitions.
- Support for low-speed, full-speed, high-speed and super-speed (USB 3.0 only) data rates, where:
  - Low speed – 1.5 Mbps (Defined by USB 1.0)
  - Full speed – 12 Mbps (Defined by USB 1.1)
  - High speed – 480 Mbps (Defined by USB 2.0)
  - Super speed – 5 Gbps (Defined by USB 3.0 and 3.1)

  All high-speed and super-speed devices are **backward compatible** and are capable of falling back to full-speed operation if necessary.

Basic mechanism of a USB system:

A USB system has one master (the host computer), and devices implement specific **functions** and transfer data to and from the host. The host acts as the **bus arbiter** and is responsible for **detecting a device** and **initiating/ managing transfers between devices** [4]. A layered communication model view adopted by B. Janßen et al [29] to describe the USB system is presented below.

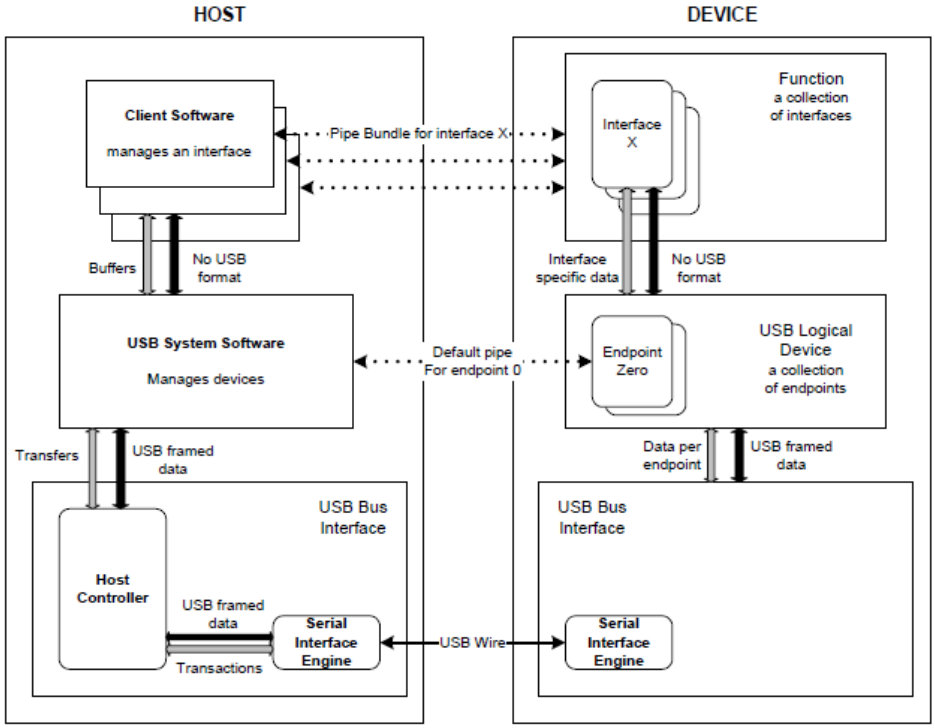


Figure 2.2: *Layered Communication Model for USB Systems*

Pipes and Endpoints:

USB data transfer occurs between the host software and a logical entity on a device called an **endpoint,** over a **pipe.** Each device should implement at least one bidirectional pipe (often referred to as **endpoint zero** or **default endpoint**).

An interface [9] is a collection of endpoints working together to implement a specific function. USB Endpoints are classified into four types based on their functionality:

- **Control endpoint:**
  Reserved for default endpoint, this endpoint is intended for device discovery and other control functions barring high performance data transfer.
  A three-stage protocol (request, data, handshake) is specified for detection of the USB device on the host.
- **Bulk endpoint:**
  Defined for variable latency, reliable, high performance data transfers. Handshake mechanism verifies the transfer readiness as well as successful completion.

- **Interrupt endpoint:**
  Defined for periodic, reliable, low performance data transfers. Handshake mechanism verifies the transfer readiness as well as successful completion.
- **Isochronous endpoint:**
  Defined for periodic, lossy, high performance data transfers. Handshakes are not used and any corrupt data packets are dropped by the receiver.

A USB device can implement one or more functions, and the capabilities of the device are reported to the host through a set of data blocks called **descriptors**.

Descriptors:

USB devices describe themselves to the host using a chain of bytes known as descriptors. As per analysis conducted by Fenxian Tian et al. on USB 3.0 data transmission [22], descriptors contain information such as the function the device implements, the manufacturer of the device, number of endpoints, and class specific information.

There are a number of descriptor types for USB devices which need to be defined within the FX3 firmware. In their paper [25], Y.J. Qian and K. Cui design a high speed data acquisition system by defining a variety of descriptors mentioned below:

- **Device descriptors:**
  Specifies the product ID (PID) and vendor ID (VID) of the device with the USB revision in order to recognize the device plugged into the host.
- **Configuration descriptors:**
  Contains information such as the device's wake-up feature, number of interfaces for the configuration and power used by a particular configuration.
- **Interface descriptors:**
  Specifies the number of endpoints associated with an interface. Each function of the device has an associated interface descriptor.
- **Endpoint descriptors:**
  Specifies the type of transfer, direction, polling interval and maximum packet size for each endpoint.

USB 3.0 – Improvements and Additions:

The primary goal of USB 3.0 is to provide the same ease of use, flexibility and hot-plug functionality but at a much higher data rate with a fair trade-off in terms of power management.

The USB 3.0 interface as described in the programmer's manual [4] consists of a physical SuperSpeed bus in addition to the physical USB 2.0 bus. The 3.0 standard defines a dual simplex signalling mechanism at a rate of 5 Gbits/s. Inspired by the **PCI Express** and **OSI** architectures, the USB 3.0 protocol is also abstracted into different layers.

- **Physical Layer:**
  - Bottom-most layer, consisting of the cable connecting the upstream and downstream ports
  - Contains separate shielded wires transmission and reception (dual simplex, thereby allowing simultaneous transmission and reception)

- **Link Layer:**
  - Responsible for maintaining a reliable and robust connection between the host and the device.
  - Used to maintain the link flow control and initiate a change in the power state, implements protocols for flow control.
  - Provides the correct framing of sequences of bytes into packets during transmission.

- **Protocol Layer:**
  - Defines end-to-end communication rules between host and device.
  - FX3 logic implicitly manages protocol details to enable better performance, efficiency and power conservation.

Feature	USB 2.0	USB 3.0
Signalling rate	480 Mbps	5 Gbps
Data transfers	Half duplex	Dual simplex
Number of pins in the USB cable	4 pins	9 pins

Table 2.1: *General Comparison between USB 2.0 and USB 3.0*

2.2.3 General Purpose Programmable Interface (GPIF):

The GPIF II is a programmable **state machine** [20] that provides the flexibility of implementing an industry-standard or proprietary interface. It can function either as a master or slave.
The GPIF II can implement both parallel and serial high-bandwidth interfaces, and provides easy and glueless connectivity to asynchronous SRAM and asynchronous/ synchronous **multiplexed interfaces**. As per the SDK documentation provided by Cypress, the GPIF II has the following features:

- Functions as master or slave.
- Offers **256 firmware programmable states**.
- Supports 8 bit, 16 bit, 24 bit and 32 bit parallel data bus.
- Enables interface frequencies up to 100 MHz.
- Supports 14 configurable control pins when a 32 bit data bus is used.
- Supports 16 configurable control pins when a 16/8 bit data bus is used.

The behaviour of the state machine described in the GPIF documentation [20] is defined by a **GPIF II descriptor**, essentially a set of programmable register configurations designed to meet the required interface specifications.

Each state is defined by 32 bytes in (SRAM) memory [9]. These 32 bytes define the properties of a state and the trigger conditions that can cause state (or I/O) transitions.

Transitions are caused by both external and internal triggers. Each state is programmed to perform certain actions. Y.J. Qian et al. [25] state that the transition conditions are checked on each clock edge or after a set number of clock cycles.

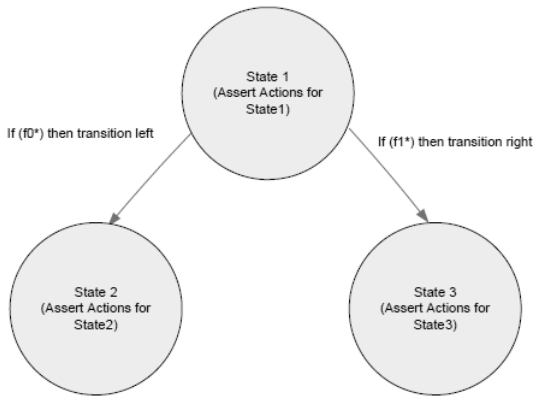


Figure 2.3: *Structure of the GPIF State Machine*

- f0* and f1* are logical functions of triggers
- f0 is checked first.
  If f0 = True, then GPIF II transitions from State 1 to State 2.
  If f0 = False, then f1 is checked.
- **If f1 = True**, then GPIF II transitions from State 1 to State 3.
- **If both f0 and f1 = False**, then GPIF II remains in State 1.
  At the next clock, the conditions are checked again.

GPIF II Actions:

Each state in a GPIF II state machine can be programmed to perform one or more actions described in the technical reference manual [9]. Actions performed in a state can be programmed to be performed once or repeated in every clock cycle until a state change occurs.

Actions can be internal, such as reading or writing to a buffer; or can also be external, such as driving an output high or low.

GPIF II Designer:

The GPIF II Designer tool provides a convenient graphical user interface (GUI) that allows you to define GPIF II state machines in graphical form [20]. The various GPIF II triggers and actions are available in an easy-to-use manner, allowing simple addition to the states in the diagram.

The tool converts the user graphical design into a C header file that can be integrated with the firmware application code using the firmware API framework described in slave FIFO documentation [27]. As a part of its initialization, FX3 firmware copies the settings in the header file into the appropriate GPIF II registers.

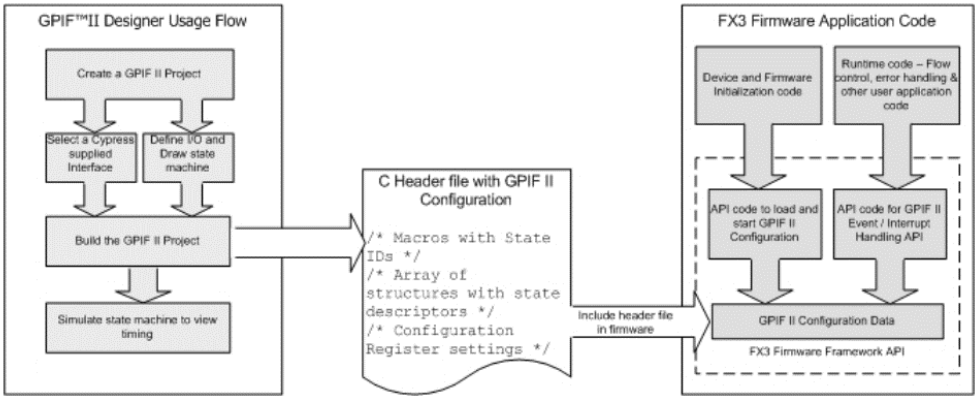


Figure 2.4: *Integration of the GPIF II Config within the FX3 Firmware*

Miscellaneous Concepts for effective usage of the GPIF II Designer:

The set of memory mapped register values that can be programmed to define the interface settings and behaviour of the GPIF II port of the FX3 is termed as **GPIF II configuration** [17]. Minyeol Seo et al, in their paper [19] explain **interface definition** as defining the electrical interface between the FX3 device and the FPGA, and extensively talk about components involved in the interface definition as follows:

- **Peripherals:**
  In addition to a programmable GPIF II interface, the EZ-USB FX3 device also implements a set of serial communication blocks to connect to external peripheral devices (I2S, I2C, SPI and UART).
- **Master and Slave:**
  A GPIF II interface allows the FX3 to interface with an external processor acting as a master or a slave as demonstrated by Deepika et al [21]. If the interface transactions with the external system are **initiated by FX3**, then FX3 is the master. FX3 is a slave if the interface transactions are initiated by the external processor.
- **Interface Clocking:**
  A GPIF II-based interface can be synchronous or asynchronous in nature. In the case of a synchronous project, the GPIF II state machine operates using the same clock that is used on the interface. The maximum frequency supported for the GPIF II interface clock is 100 MHz.
- **Multiplexing of Address/ Data:**
  A GPIF II interface gives a maximum of 32 bidirectional data lines, which can be time multiplexed with address lines [29]. In the case of multiplexed operation, the address bus will have the same width as the data bus. If the address bus is not time multiplexed with the data bus, the number of address lines available will depend on the width of the data bus.

2.2.4. Data transfer mechanism within the FX3

For data transfer between peripherals (USB to GPIF, SPI to UART etc.), the **DMA (direct memory access)** is the underlying backbone that connects all components of the FX3 with an **AHB bus**, as theorised in a paper for AMBA by Anuja et al. [16].

The FX3 device has an internal DMA fabric [25] that connects the GPIF interface to the internal system memory. All data transfers through the GPIF interface are initially done from or into an internal memory buffer.

The firmware application running on the FX3 is responsible for connecting this data path to the appropriate source or sink, such as the USB host or a serial peripheral.

Interconnect Fabric:

The **Advanced Microcontroller Bus Architecture – Advanced High Performance Bus (AMBA AHB)** forms the central nervous system of the

FX3 [18]. This fabric allows easy integration of processors and on-chip memories using low power macro cell functions, while providing high bandwidth communication link between elements [6]. The interconnect fabric of the AHB has the following components:

- **AHB Bus Masters** that can initiate read and write operations by providing an address and control information [7]. A bus can have at most one owner, decided by the **AHB arbiter**.

- **AHB Bus Slaves** that respond to read or write operations within a given address space range. These bus slaves signal back the success, failure or waiting of the data transfer to the master.
- **AHB Bridges** to translate traffic of different frequency, bus width and burst size and to create links between buses.

- **AHB Slave/ Master interface** macro cells to connect peripherals, memories and other elements to the bus [5].

The minimum data bus width is specified as 32 bits, but the bus width can be increased for realizing higher bandwidths.

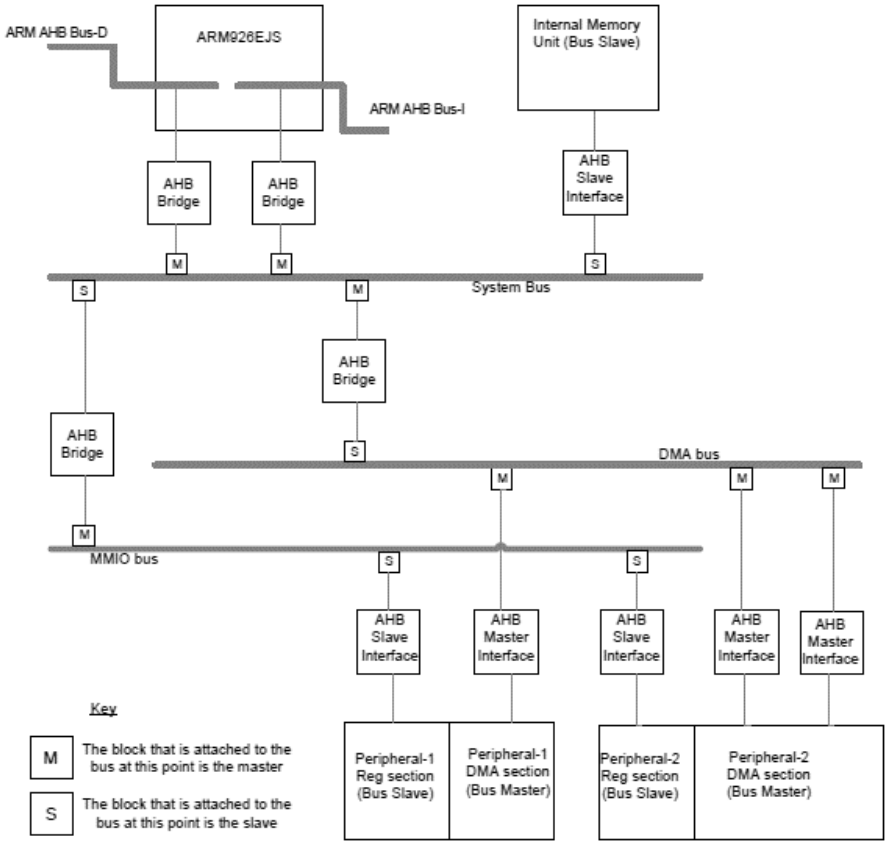


Figure 2.5: *Interconnect Fabric with AMBA AHB*

In order to fully understand the extent of valid usage of the AHB backbone, it is important to understand various flag configurations, which subsequently depend on the concept of **threads**, **sockets** and the **DMA channel**.

Sockets, Threads, Buffers and Descriptors:

A **socket** is a point of connection between a peripheral hardware block and the FX3 RAM [2]. Each peripheral hardware block on FX3, such as USB, GPIF, UART, and SPI, has a fixed number of sockets associated with it.

The P-port (GPIF port) of the FX3 supports a maximum of 32 sockets, which means that a total of 32 independent data transfers [23] can be performed across this interface.

Although the GPIF port on the FX3 provides 32 independently addressable sockets for data transfer, there are some restrictions on how the application can switch the active socket that is used for a transfer operation.

The FX3 device implements **four DMA transfer handlers** or **threads**, which are active concurrently.

Each of these threads can be associated with **one DMA socket at a time**, i.e. the application [9] can select a maximum of four sockets that are bound to these threads, and then switch between them with **no added latency**.

The firmware application is responsible for allocating memory buffers corresponding to all sockets that need to be used; and for connecting the socket to the source of sink of data on another port.

A **DMA descriptor** is a set of registers allocated in the FX3 RAM. It holds information about the address and size of a DMA buffer as well as pointers [10] to the next DMA descriptor.

Descriptors are used to synchronize between sockets; every buffer created in FX3's RAM has a descriptor associated with it, which contains info such as **its address, empty/full status** and the **next buffer/ descriptor** in the chain.

These descriptors are regularly monitored [3] and updated by peripheral's sockets.

A **DMA buffer** is a section of RAM used for intermediate storage of data transferred through the FX3 device. DMA buffers are allocated from the

RAM by the FX3 firmware and their addresses are stored as part of DMA descriptors.

DMA Architecture and Data Transfer:

For data transfer between peripherals (USB to GPIF, SPI to UART etc.) the DMA is the underlying backbone connecting all components with an AHB bus.

All data passes through the system memory (SYSMEM: 512 kB) of the FX3.

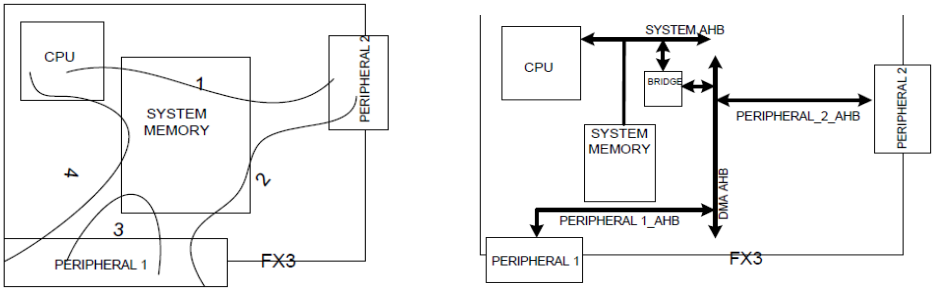


Figure 2.6: *System Memory and DMA Mechanism*

As per the depicted DMA mechanism:

- The ARM9 CPU on the FX3 uses the **System AHB** to access SYSMEM.
- Respective DMA paths of peripherals are linked to the backbone DMA AHB, using their corresponding **peripheral AHB** [2].
- An **AHB Bridge** connects the System AHB [18] to the DMA AHB, and is also essential for routing DMA traffic to SYSMEM.

When a DMA transfer occurs, there are three sub-components of DMA architecture that perform separate tasks to ensure accurate socket – thread – descriptor – buffer relationships:

- **DMA Adapter**
- **Thread Controller**
- **Peripheral Core**

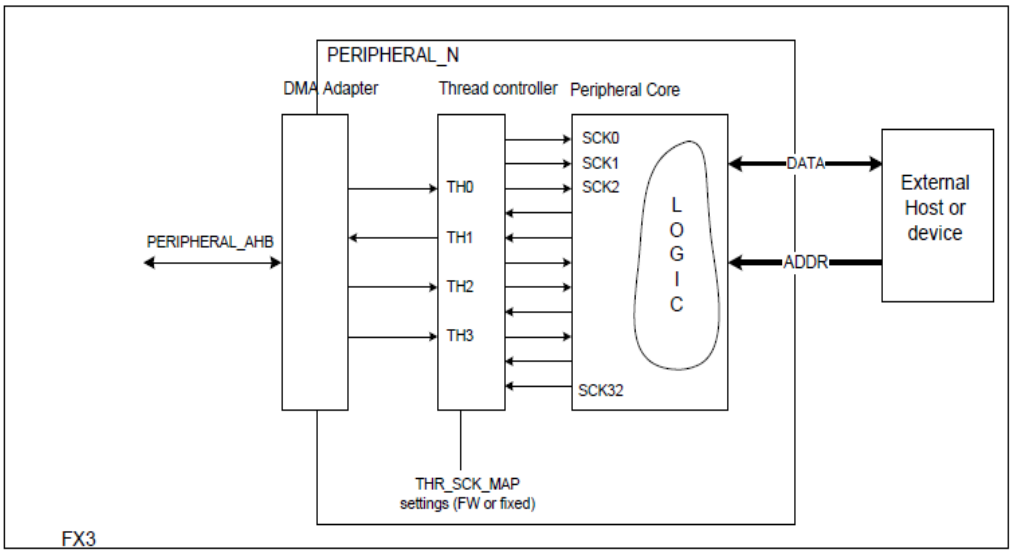


Figure 2.7: *DMA Components*

In practice,

- The peripheral attaches to the DMA fabric using AHB, and the width of the AHB determines the throughput of the peripheral.
- The **peripheral core** implements the logic of the peripheral (I2C, GPIF or USB). Data transfer between peripheral core and external device happens over two buses (address and data).
- The address specified by external host is used to index data to/from one of the **sockets** which are logically presented at the interface **as a chain of buffers.**

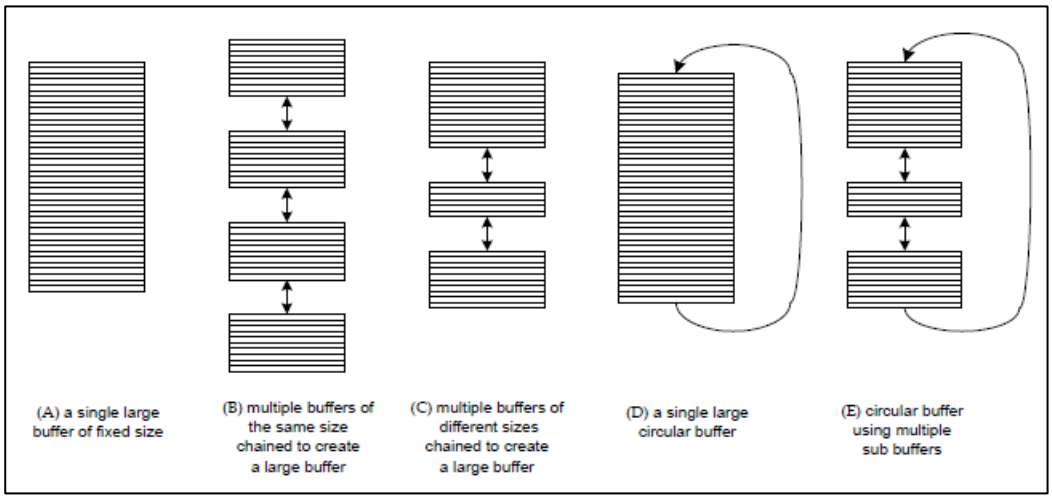


Figure 2.8: *DMA Socket Configurations*

- 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** [9], the requests of which are time multiplexed.

- 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 [16] and launches them on to the AHB.

Essentially, a DMA transfer is the **flow of data from producer socket on one peripheral to consumer socket** on another through buffers in SYSMEM.

In a nutshell, **at the producer socket:**

- The data source peripheral sends data over the producer socket, which fills the current buffer of the socket.
- When the last byte is being written into the buffer, the producer socket updates the descriptor of its current buffer and loads the next buffer in the chain.
- This continues until either the buffer chain ends, or the next buffer in the chain is not free.

At the consumer socket:

- The flag waits until the buffer it points to becomes filled
- Once available/filled, the destination peripheral is notified to start reading over the consumer socket
- When the last byte of the buffer is read, the consumer socket updates the descriptor to the next buffer in the chain.
- This continues until either the buffer chain ends, or the next buffer in the chain is unavailable/ unfilled.

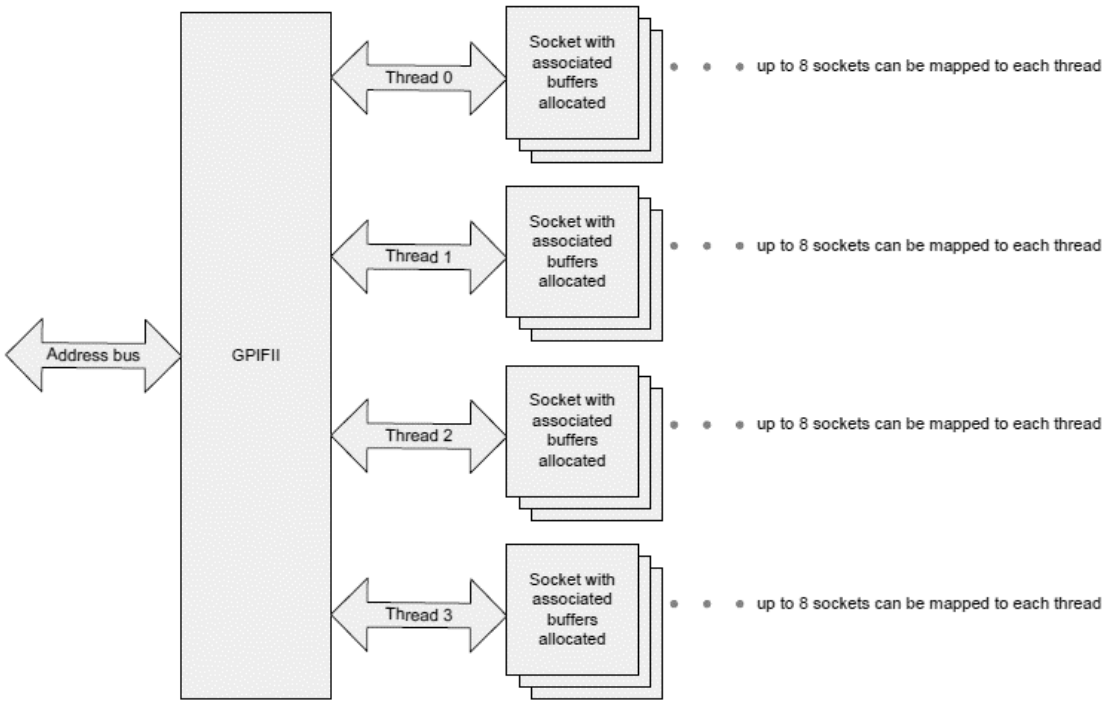


Figure 2.9: *GPIF Configuration with Threads*

Moreover, data transfer over DMA channels can further be classified [27] based on the nature of the DMA channel:

- **Automatic DMA channel implementation**
  For data transfer between the GPIF II block to the USB block, the GPIF socket can tell the USB socket that it has filled data in a DMA buffer and vice versa.
  The FX3 CPU does not modify any data in the data stream, nor is it involved in any phase of data transaction.

- **Manual DMA channel implementation**
  Alternatively, the GPIF socket can send an interrupt to the FX3 CPU to notify it that the GPIF socket filled a DMA buffer. The FX3 CPU can relay this information to the USB socket, which sends an interrupt to the FX3 CPU to notify it that the USB socket emptied a DMA buffer.
  The FX3 CPU can relay this information back to the GPIF socket. This is called the **manual DMA channel implementation**.
  This implementation is used when the FX3 CPU has to add, remove, or modify data in a data stream.

Flag Configuration:

Flags may be configured as empty, full, partially empty, or partially full signals. These are not controlled by the GPIF II state machine, but by the DMA hardware engine internal to EZ-USB FX3 [4].

Flags are associated with specific threads or the currently addressed thread and indicate the status of the socket mapped to that thread.

Flags indicate empty or full, based on the direction of the socket.

Therefore, a flag indicates empty/not empty status if data is being read out of the socket and full/not full status if data is being written into the socket.

- **Dedicated thread flag:**
  A flag that is dedicated only to a particular thread and always indicates the status of the socket mapped to the particular thread, is called a **dedicated thread flag**.
  The external processor must keep track of the dedicated links of flags to their corresponding threads, and **monitor the correct flag** every time a different thread is accessed.

- **Current thread flag:**
  A flag can be configured to indicate the status of the currently addressed thread. In this case, the GPIF II state machine samples the address on the address bus and then updates the flags to indicate the status of that thread.

- **Partial thread flag:**
  A flag can be configured to indicate the partially empty/full status of a socket. A **watermark value** must be selected such that the flag is asserted when the number of 32-bit words that may be read or written is less than or equal to the watermark value.

When using flags, there is always a significant latency involved while asserting read and write signals.

For **write transfers**, a **three-cycle latency** is incurred at the end of transfer. At the fourth clock edge, the external master can sample the flag low. For **read transfers**, a **two-cycle latency** is incurred at the end of transfer. The external master can sample the flag low at the third clock edge.

2.2.5 Synchronous Slave FIFO Interface

The synchronous slave FIFO interface is suitable for applications in which an external processor or device needs to perform data read/write accesses to FX3's internal FIFO buffers [17].

Register accesses are not done over the slave FIFO interface. The synchronous slave FIFO interface is generally the interface of choice for USB applications, to support high throughput requirements.

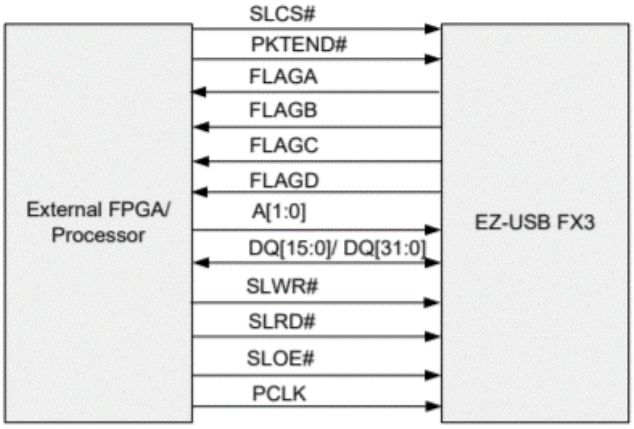


Figure 2.10: *Schematic depiction of the Slave FIFO Interface*

As per the schematic, data lines can either be allocated a 16 bit/ 32 bit bus for data transfer, and the address lines are restricted to 2 bit lines.

Hence, for most applications of the slave FIFO, there are only two address bits; **only four sockets can be toggled** at any instant of time.

In order to implement the synchronous 2-bit slave FIFO interface [28] within the FX3 for our project, a sequence of prerequisites need to be completed:

1. Developing the **state machine** to configure the GPIF for synchronous slave FIFO, using the GPIF II Designer.
    a. The FX3 has 4 physical threads for data transfer over GPIF, and **one socket is associated with one thread** at a time.
    b. The pin mapping is done by the Designer II Tool, via the **IO Matrix configuration** tab.
2. When a **32-bit data bus** is used, **14 configurable pins** need to be set up. Similarly, **16 configurable pins** need to be initialized for a 16/8 bit data bus.

These control pins dictate the switching states [27] of the state machine. **State transition** occurs upon the arrival of the input control signal.

3. **Behaviour of the state machine** is defined by GPIF descriptors.

4. Generate the GPIF config header file for integration with the firmware.

5. Define important subcomponents [4] of the firmware:
    a. Debug mechanism for the slave FIFO application
    b. Thread creation and initializing the **ThreadX RTOS** within the FX3.
    c. Configuring the Serial Peripheral Interface (SPI) along with the GPIF ports
    d. Defining the USB descriptors
        - SuperSpeed Device Descriptor
        - High Speed Device Descriptor
        - String Descriptor
        - Configuration Descriptor
        - Endpoint Descriptor
    e. Setting up the DMA channel for P2U (GPIF to USB) transfers and vice versa
        - Initializing DMA descriptors
        - Defining callback functions to handle transfer events
        - Managing flags

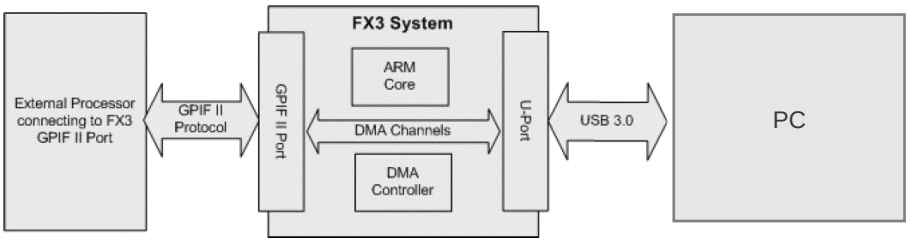


Figure 2.11: *Overview of the FX3 System for Slave FIFO*

6. Create an image file encapsulating the entire firmware using the Eclipse IDE.
7. Develop a user-friendly GUI to simulate and observe data transfers over the FX3, and work towards FPGA configuration.

Chapter 3

Design and Development

3.1 Introduction

This section deals with the details of implementation and development of the FX3 firmware along with the integration of software-designed state machines onto the GPIF and SPI interfaces.

The GPIF II Designer is used for designing the **IO matrix**, which is responsible for assigning the mapping of pins as per the slave FIFO interface requirements. Furthermore, a **state machine** is designed using the tool in order to define the sequence of commands and execution, while ensuring conditional state transitions based on signals from an external processor.

A config file generated by the GPIF II Designer is included as a header within the FX3 firmware, which is later compiled and packaged as an image file for the hardware by the Eclipse IDE. The firmware handles all the internal data transfer and cache management processes mentioned in the previous chapters.

3.2 System Architecture

3.2.1 About the Slave FIFO architecture

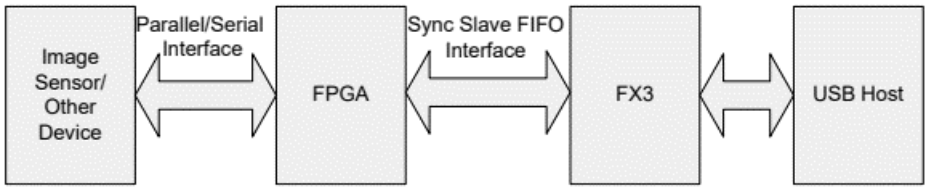


Figure 3.1: *Basic Outline of Slave FIFO Architecture*

In a typical slave FIFO setup, an external processor such as an FPGA is connected to the FX3 via the GPIF ports, such that the GPIF pins have been mapped to serve the standard slave FIFO commands effectively.

GPIF II allows the FX3 to connect directly to external peripherals such as FPGAs, image sensors, ADCs, or any other high bandwidth devices. It provides external pins that can operate as inputs/outputs, a data bus, an interface clock (PCLK), and an interrupt line.

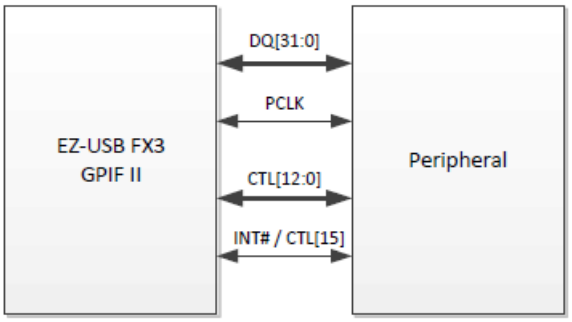


Figure 3.2: *Interface between the FX3 and Peripheral*

Pin	In/ Out	Description
CTL[12:0]	I/O	Programmable Control Signals
DQ[31:0]	I/O	Bidirectional data bus
PCLK	I/O	Interface Clock
INT# / CTL[15]	I/O	Interrupt or Control Signal

Table 3.1: *GPIF II Interface Signals*

Programming the GPIF II State Machine:

The GPIF II port known as the **P-port of FX3** provides a parallel interface with maximum of **32 bidirectional data lines**. The data lines can be split into or time **multiplexed** into address lines.

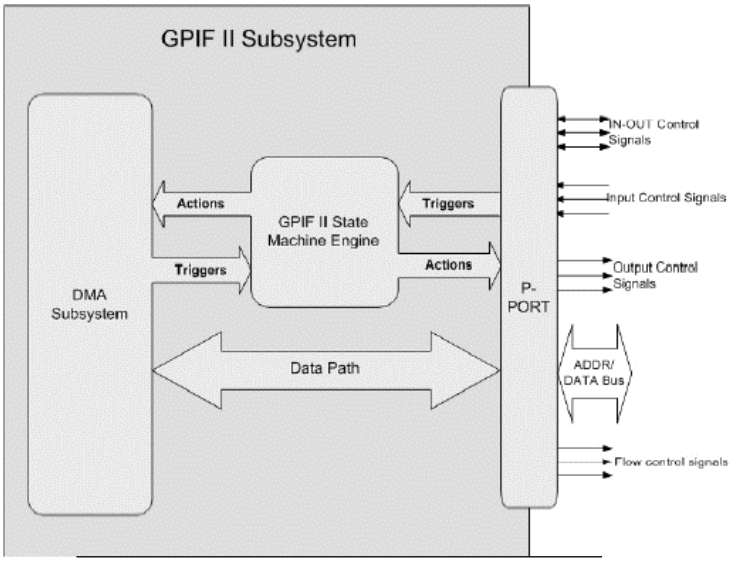


Figure 3.3: *The GPIF II Subsystem*

System Overview:

The main programmable block (FX3) 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/ I^2S)
- Set up, control and monitor data flow between peripherals
- Perform necessary operations such as data inspection, data modification, header/ footer information addition/ deletion.

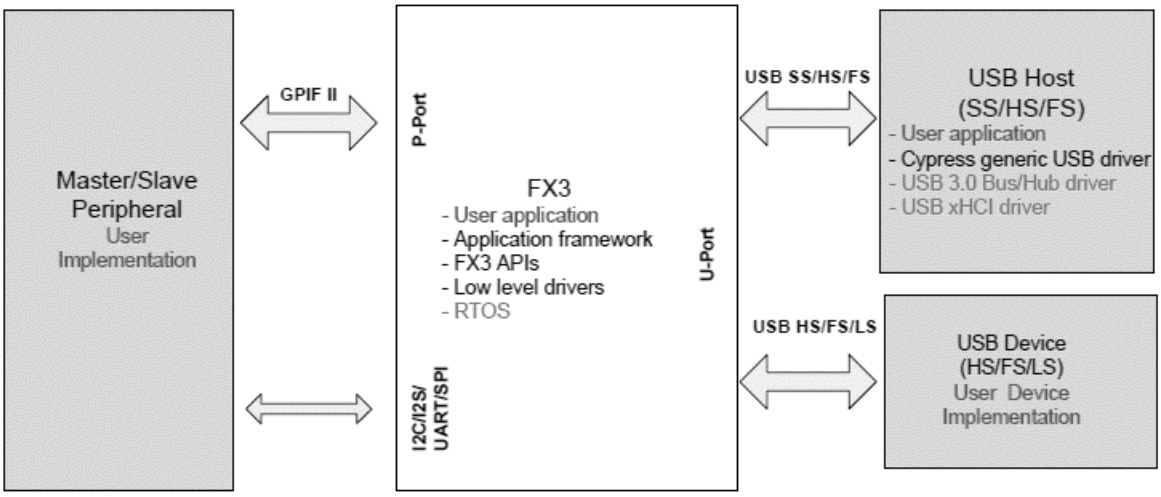


Figure 3.4: *Programming View of the FX3*

The FX3 comes bundled with a complete software development solution end-to-end architecture as depicted below:

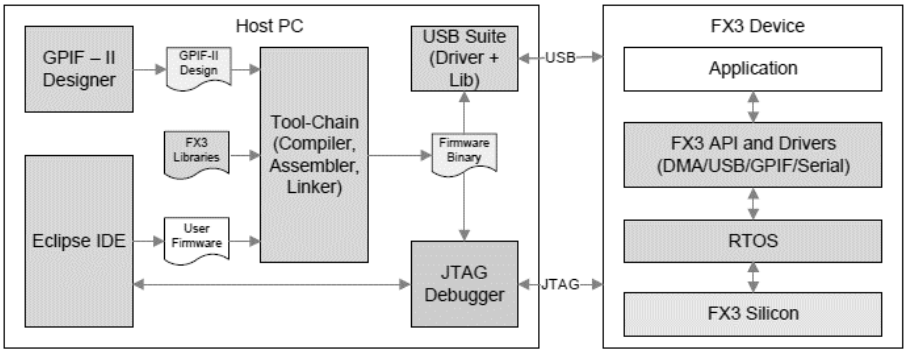


Figure 3.5: *Internal Architectural Components of the FX3 Software Development Kit*

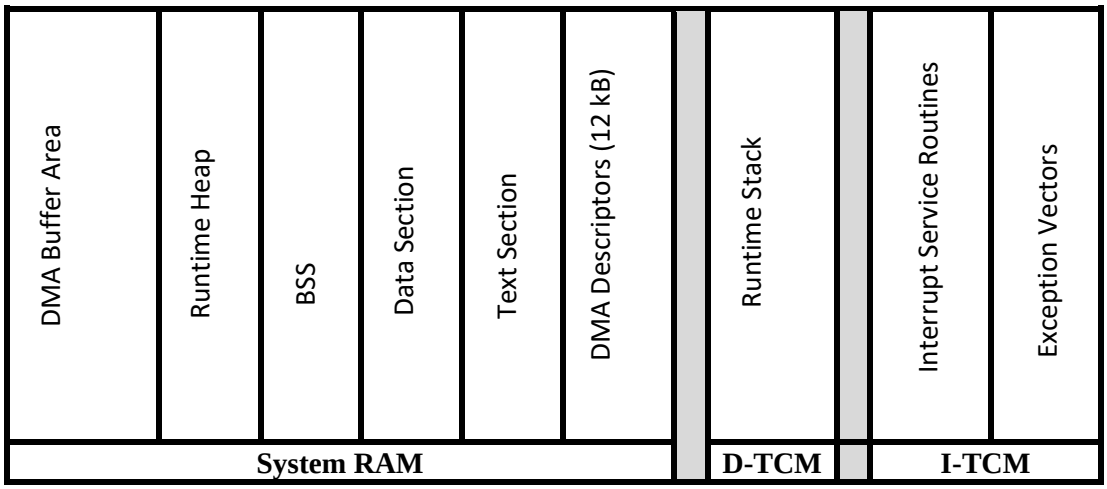
Memory Map Usage:

The tightly coupled memory (TCM) and system RAM regions on the FX3 device are general purpose and can be used by the firmware application **with the following restrictions:**

- The initial part of the I-TCM (256 bytes starting at address 0) is reserved for setting up the ARM exception vectors
- The initial part of the system RAM (from address 0x40000000) is reserved for setting up DMA transfer related data structures
- The TCM regions cannot be used for direct data transfers, as the DMA engine does not support data transfers from/to these regions

Section Name	Description
Vectors	ARM exception vectors
Text	Executable code for the application
Data	Explicitly initiated global data used by the application
BSS	Uninitialized global data used by the application
Stacks	Stack regions for the ARM processor operating modes
Heap	Runtime heap for dynamic allocation of new variables
DMA Descriptors	Reserved for DMA transfer-related data structures
DMA Buffer	Reserved for DMA data buffers used by the application

Table 3.2: *Memory Regions used by the FX3 Application*



3.2.2 Logic Diagram for Synchronous Slave FIFO

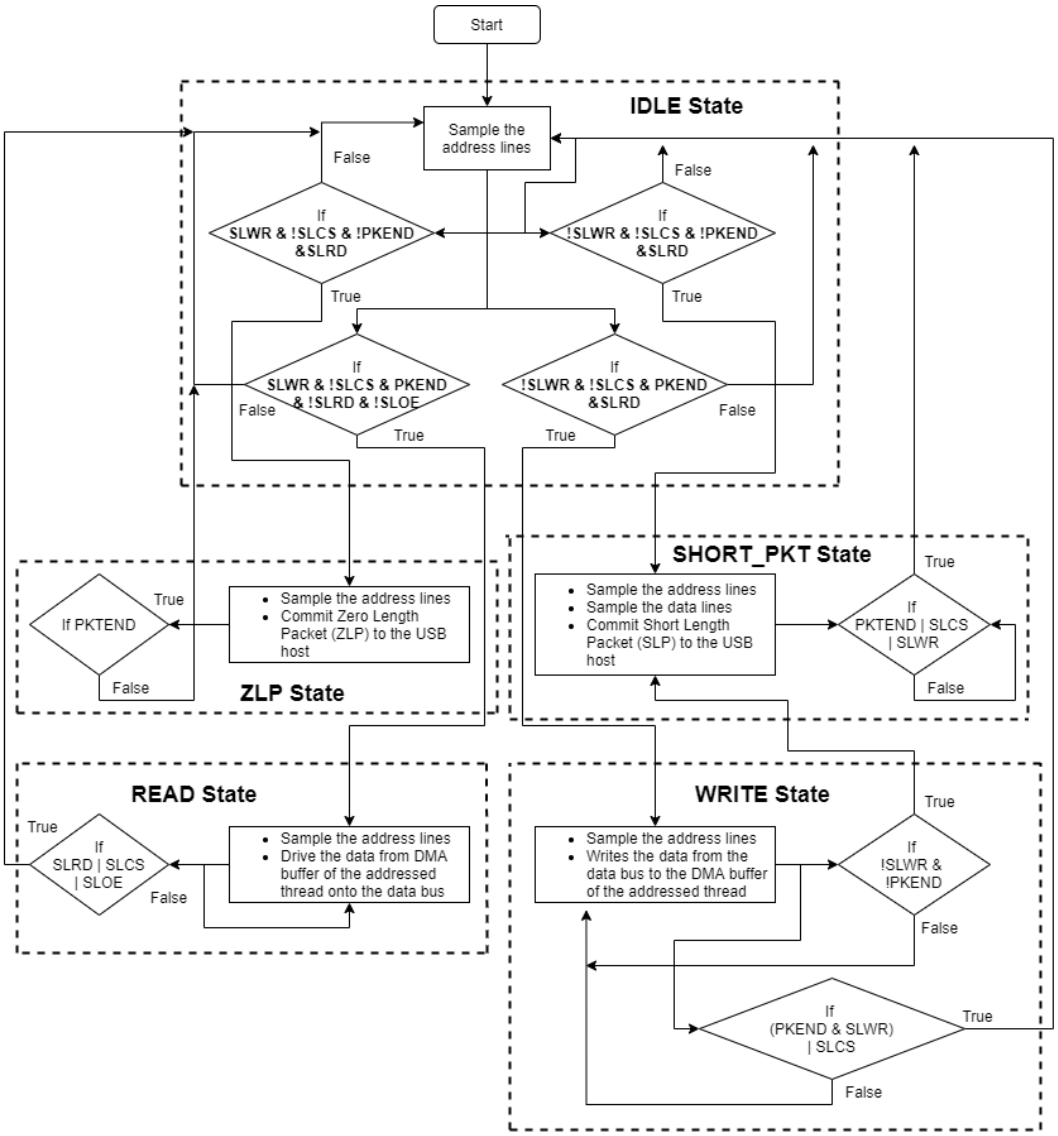


Figure 3.6: *Logical Interpretation to State Machine for Slave FIFO*

In order to configure the GPIF II interface, the following initial states need to be defined:

- **Interface type:** Slave
- **Communication type:** Synchronous
- **Endianness:** Little Endian
- **Data bus width:** 8 bit, 16 bit or 32 bit
- Address or data bus multiplexed
- **2 address pins** for selecting among 4 available threads
- **Active low signals** for **Read**, **Write**, **Chip Select** and **Output Enable**.

Signal Name	Signal Description
SLCS#	Chip Select signal for the Slave FIFO interface. It must be asserted to perform any access to the interface.
SLWR#	Write strobe for the Slave FIFO interface. It must be asserted to perform write transfers to the Slave FIFO.
SLRD#	Read strobe for the Slave FIFO interface. It must be asserted to perform read transfers from the Slave FIFO.
SLOE#	Output Enable signal, causing data bus of the Slave FIFO interface to be driven by the FX3. It must be asserted to perform read operations.
PKTEND#	The packet end signal which must be asserted to send a short packet to the USB host.

Table 3.3: *Slave FIFO Interface Signal Description*

3.2.3 DMA Architecture and System Interconnects

Non-CPU-intervened data transfers between a peripheral and CPU or between two different peripherals or between two different gateways of the same peripheral are collectively referred to as DMA in FX3.

All the data in the DMA subsystem flows through the system memory.

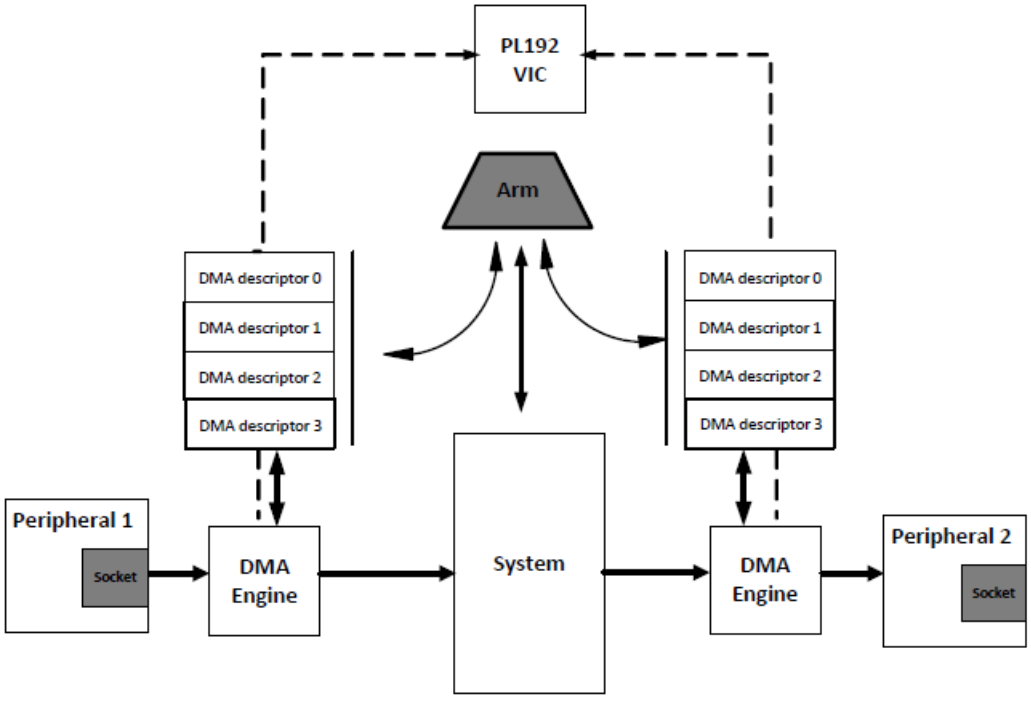


Figure 3.7: *FX3's DMA System*

DMA Descriptors are DMA instructions in a set of registers allocated in the FX3 RAM, containing information about the address and size of the DMA buffer as well as pointers to the next DMA Descriptor. These pointers create DMA descriptor chains.

A **DMA buffer** is a section of RAM used for intermediate storage of data transferred through the FX3 device.

DMA buffers are allocated from the system RAM by the FX3 firmware; their addresses are stored as part of DMA descriptors. Every buffer created in the system memory has a descriptor associated with it that contains buffer information such as

- its address
- empty/ full status, and
- next buffer/descriptor in the chain.

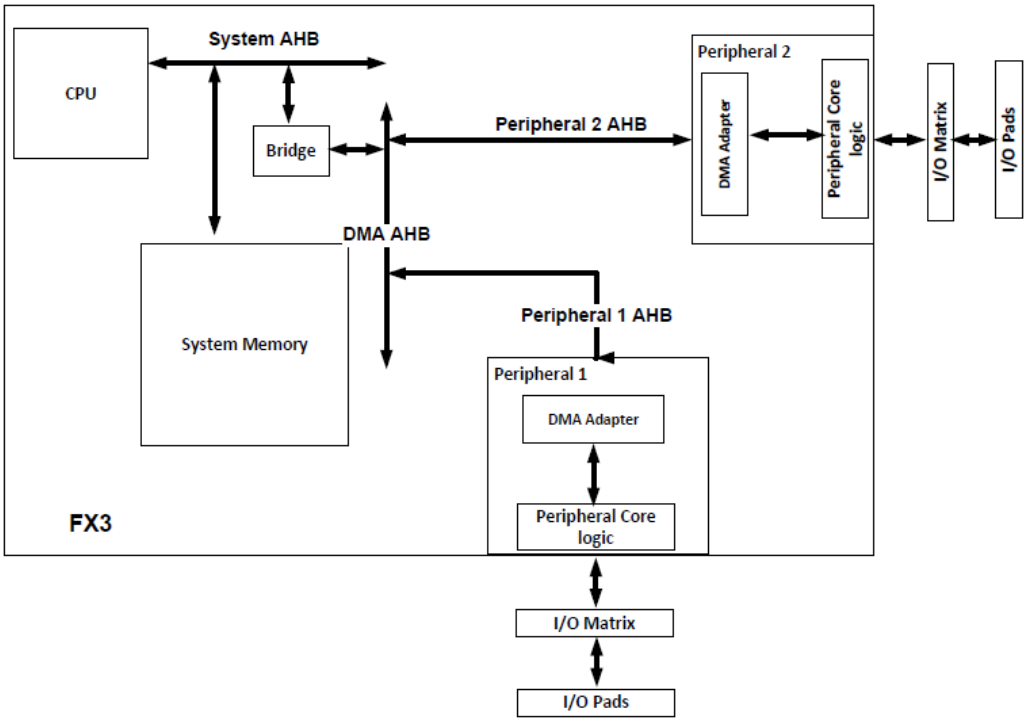


Figure 3.8: *Block Diagram of the FX3 DMA Subsystem*

A **socket** is a point of connection between a peripheral hardware block and the FX3 RAM.

The number of **parallel data channels** through a peripheral is equal to the number of its sockets. The socket implementation includes a **set of registers** that point to the active DMA descriptor and the enable or flag interrupts associated with the socket.

Chapter 4

Implementation and Demonstration

4.1 Design and Implementational Logic for Slave FIFO

4.1.1 Hardware Setup

The project consists of an **FX3 Development Kit** interconnected with an **evaluation kit (FPGA)** using a **Samtec to FMC interconnect board**. The interconnect board mates with the Samtec connector on the FX3 and the FMC connector on the FPGA.

The figure shown below uses the FX3 with a **Xilinx Spartan 6 SP601 evaluation kit.**

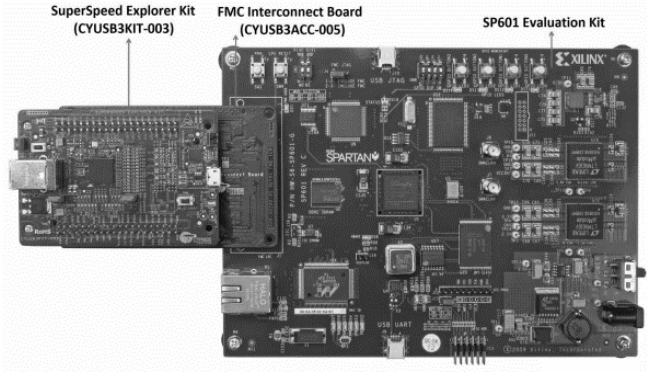


Figure 4.1: *SuperSpeed Explorer Kit connected to SP601 board*

For execution and testing of the slave FIFO interface, the following components are included in the firmware:

- **Loopback transfer**
  The FPGA first reads a complete buffer from FX3 and then writes it back to the FX3. The USB host issues OUT/IN tokens to transmit and receive data.

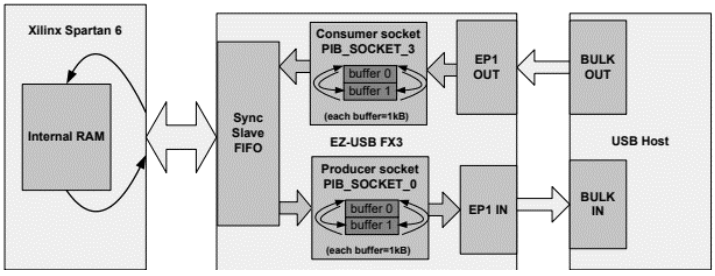


Figure 4.2: *Loopback Transfer*

- **Streaming (IN) data transfer**
  The FPGA does one-directional transfers: continuously writes data to FX3 over synchronous slave FIFO.
  The USB host should issue **IN tokens** to receive data.

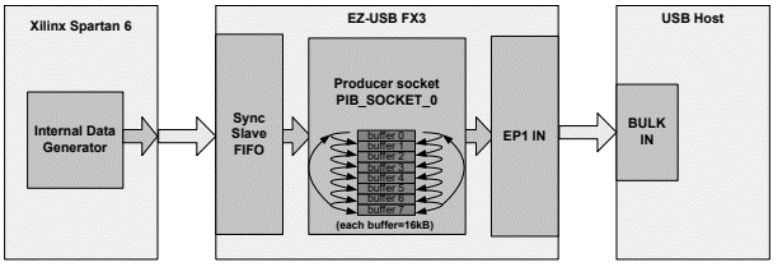


Figure 4.3: *Stream IN Transfer Setup - Buffer count and size optimized for performance*

- **Streaming (OUT) data transfer**
  The FPGA does one-directional transfers: continuously reads data from FX3 over synchronous slave FIFO.
  The USB host should issue **OUT tokens** to provide this data.

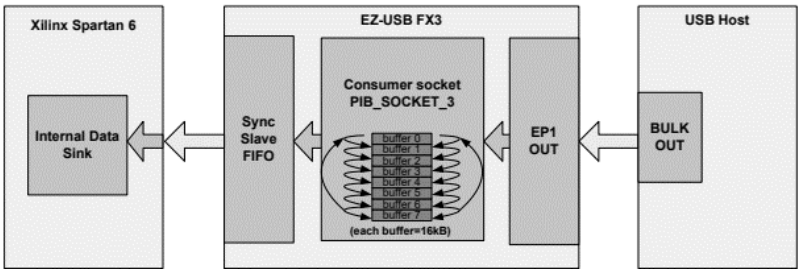


Figure 4.4: *Stream OUT Transfer Setup - Buffer count and size optimized for performance*

4.1.2 Configuring the I/O Matrix for Slave FIFO Interface

The **interface definition** window allows developers to define the I/O level external interface by displaying a graphical view of the FX3 GPIF port interface..

The I/O matrix helps in configuring

- **FX3 peripherals** such as I2C, UART, SPI
- **Data bus width**
- **Endianness**
- **Flags** to indicate the status of the thread in terms of current buffer conditions
- **Pin mappings** as per the desired interface

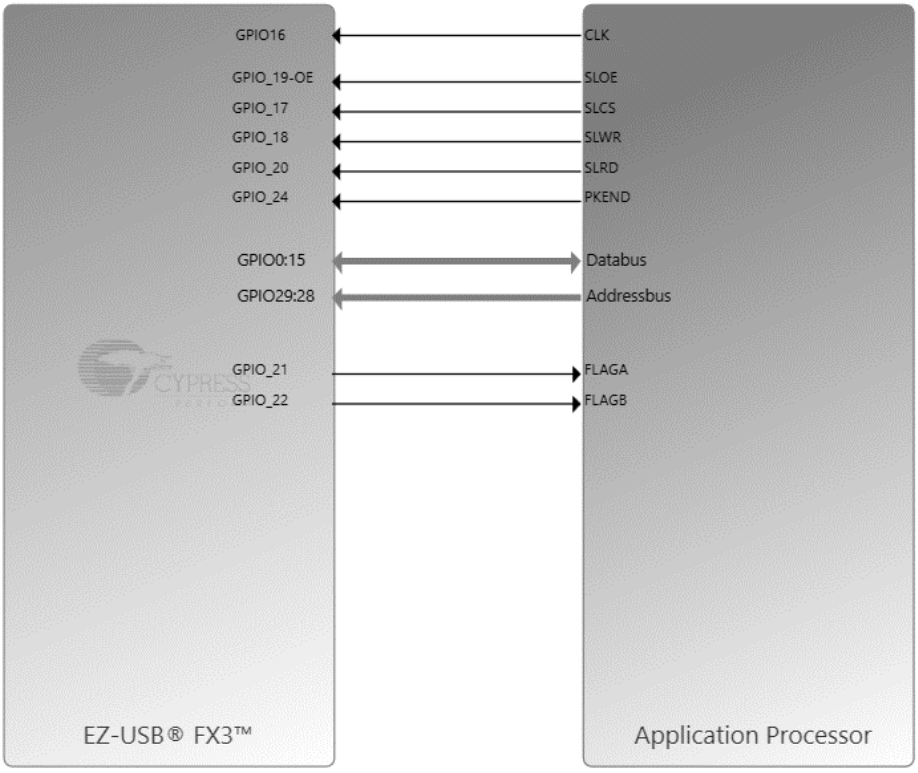


Figure 4.5: *I/O Matrix Configuration in* ***GPIF II Designer*** *for Slave FIFO Interface*

The interface definition window and I/O matrix helps define the electrical interface in terms of data and address buses that are used, the number and direction of control signals and the interface clocking parameters.

These parameters have been defined in the I/O matrix configuration panel shown above.

After completing the interface definition, a **state machine** canvas is used to enter a state machine diagram matching the processor interface, wherein states can be added from a predefined menu.

4.1.3 GPIF II State Machine

GPIF II is a programmable finite state machine, where every state of the state machine can be programmed to carry out instructions on the P-port and the DMA system.

Such instructions, known as **GPIF II actions**, are used to create a data path between an external device connected on P-port and any other peripheral port of FX3 including the USB SuperSpeed port.

Transition from one state to other is caused by Boolean expressions formed using the signals on the P-port or events generated as a result of actions.

For the synchronous slave FIFO, a state machine is designed based on the expected behaviour and outcomes, along with the requisite input control signals to the GPIF port as displayed.

The expected behaviour can be visualized as a flowchart in Figure 3.6.

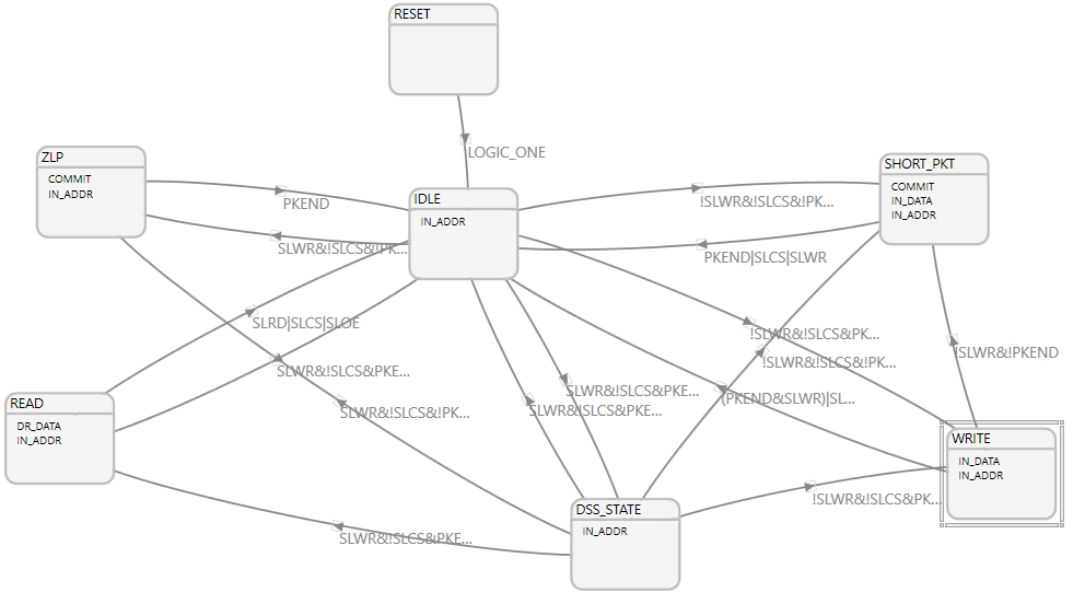


Figure 4.6: *State Machine for Slave FIFO*

Each state in a GPIF II state machine is programmed to perform one or more **GPIF II actions**, which are instructions used to create a data path between an external device connected to the GPIF port and any other peripheral port of the FX3.

Actions performed in a state can be programmed to be performed once or in every clock cycle till the state change occurs. The following table highlights some of the GPIF II actions used in the state machine.

Action	Parameters	Description
IN_DATA	1. **Data Sink** (Register or socket) 2. **Thread number** associated with data sink 3. **Sample data** from data bus 4. **Write data** into data sink	Samples data from data bus and moves to the destination specified. Destination can be Registers or Sockets.

IN_ADDR	1. **Address select** – Thread/ Socket 2. Enable PP Register Access Only (Boolean)	Select a thread or socket after sampling the address word from the bus.
DR_DATA	1. Update new value from data source 2. Data source 3. Remove data from data source	Drives data on to the bus from the source specified. Source can be Registers or Sockets.
COMMIT	1. Thread Number (0 to 3)	Commit or wraps up the buffer associated to the selected thread and socket. The buffer will be transferred to the consumer side of the pipe.

Table 4.1: *GPIF II Actions used for designing State Machine for Slave FIFO*

4.2 FX3 Firmware Application Structure

All FX3 firmware applications consist of two parts:

- Initialization code – Usually common for all applications
- Application code – Code specific to the Slave FIFO application

The typical file structure for the firmware application would be as follows:

- **A header file containing the GPIF II descriptors** for 16-bit and 32-bit slave FIFO interface
- **A C program containing USB descriptors**
- **A header file containing defines** used in the main application. Several important constants essential for setting the initial conditions for execution are defined in this file.
- **A C program containing the main application logic** for Slave FIFO.

Initialization Code:

1. The CPU clock is setup.
2. **Vectored Interrupt Controller** (VIC) is initialized.
3. Device caches are enabled. (8 kB data cache and 8 kB instruction cache)
4. I/Os are configured based on the header files included.
5. The inherent **ThreadX RTOS** is invoked, and the OS timer is initialized before transferring the control over to the RTOS scheduler.

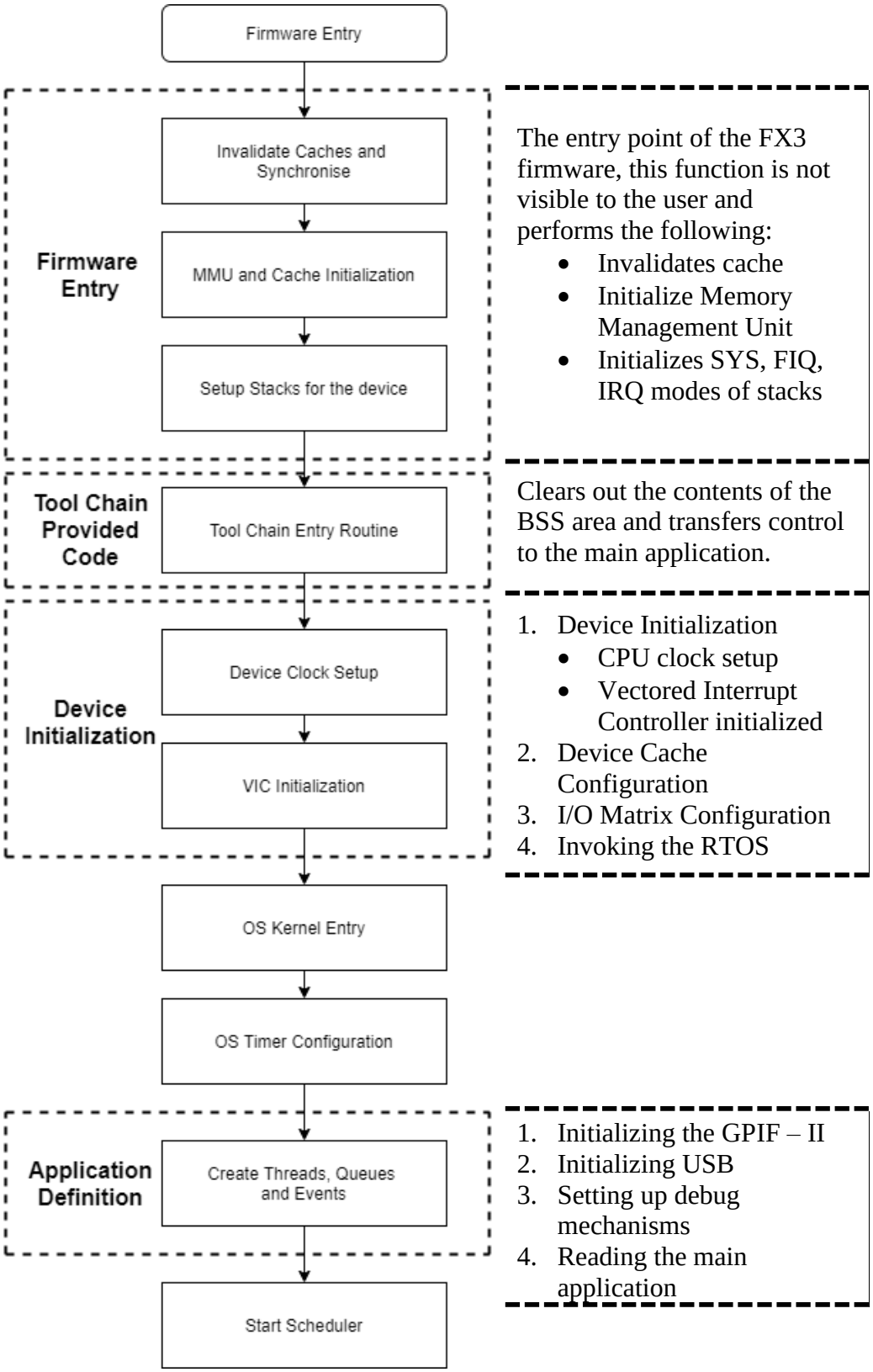


Figure 4.7: *Sequence of Initialization*

The application definition function is called by the FX3 library after the OS is invoked, following which **application-specific threads** are created. Additional threads can be created in the function and all FX3 specific programming is done in these user-defined threads.

4.2.1 Application Code:

The Slave FIFO application code comprises of three parts:

- The **application thread** function to perform application – specific initialization operations.
- The **application start** function to set up endpoints and DMA channels required for USB – GPIF data transfers.
- The **application stop** function to free up DMA channels and disable USB endpoints when a USB reset or disconnect is detected.

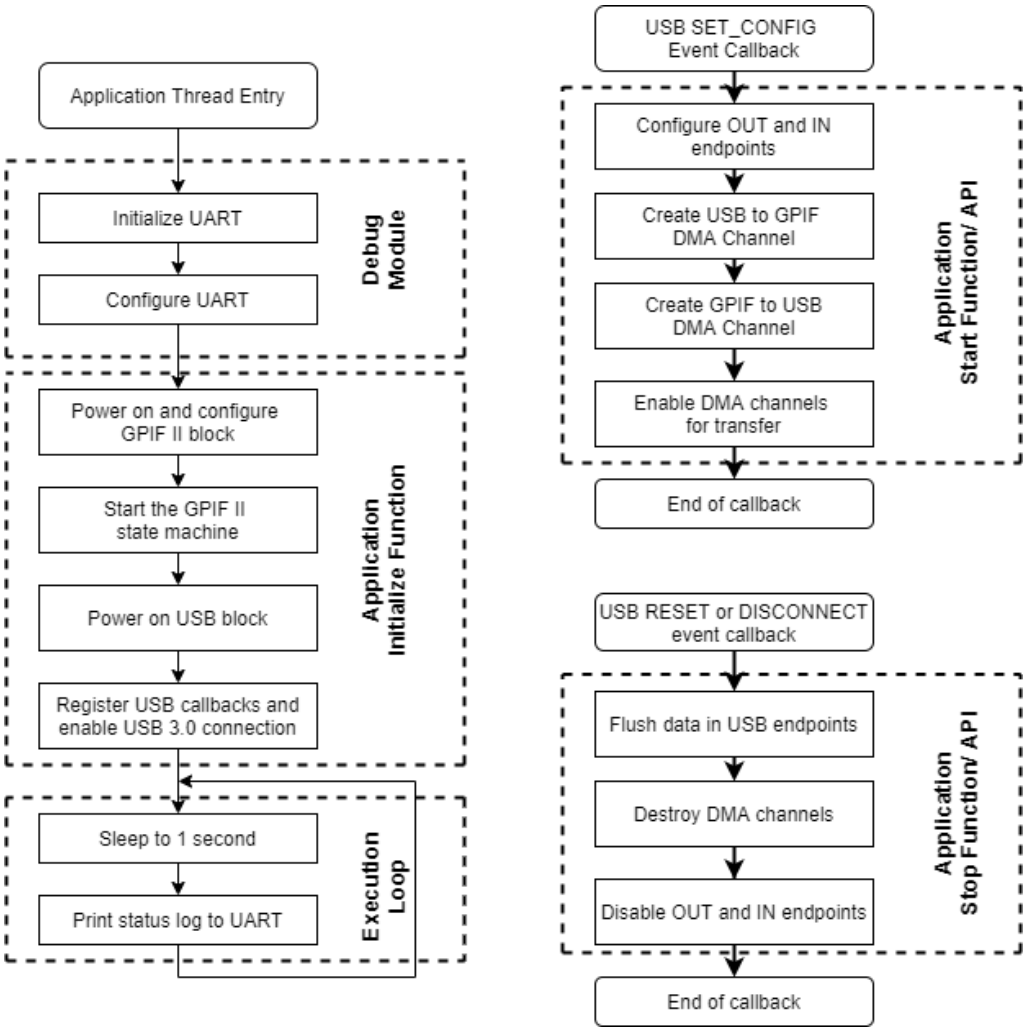


Figure 4.8: *Slave FIFO Application Code Structure*

4.2.2 Application Thread:

The application entry point for the slave FIFO interface execution is the application thread. The main actions performed in this thread are

- **Initializing the debug mechanism**

  The debug module uses the UART to output debug messages. The UART

  1. needs to be configured before the debug mechanism is initialized.
  2. needs to be initialized with the requisite **baud rate, DMA settings** and other essential parameters.
  3. has the transfer size set to infinite, so that debug prints are not limited to any size.

  The debug module is hence initialized, with two main parameters: the destination for the debug prints (UART socket) and the verbosity of the bug.

- **Initializing the main slave FIFO application**

  The application initialization consists of the following steps:

  1. **GPIF II Initialization**

     The GPIF II block is initialized and **slave FIFO descriptors** are loaded into the GPIF II registers.
     The state machine execution is initiated.

  2. **USB Initialization**

     a. The USB stack in the FX3 library is first initialized.

     b. Callbacks are registered for USB setup requests and USB events.

     c. USB descriptors are defined and set for each type of descriptor within the firmware.

Post-peripheral initialization also requires defining a variety of other sub-components that need to be set up for end-to-end communication. The **complete code for the FX3 firmware application** containing the aforementioned threads along with upcoming definitions can be found here.

4.2.3 Other prerequisite definitions:

- **Endpoint Setup**
  Endpoints are configured on receiving a request to detect USB connections. Two endpoints (IN and OUT) are configured as **bulk**

endpoints and the **maximum packet size** is updated based on the USB connection speed.

- **USB Setup Callback**
  This callback function is used whenever the USB driver in the FX3 firmware framework needs the application to respond to a USB control request.

- **USB Event Callback**
  The USB events of interest are
  - Set Configuration (to detect the presence of a new USB)
  - Reset
  - Disconnect

  The slave FIFO loop is initialized on receiving a **set configuration** event and is stopped on a **USB reset** or **disconnect**.

- **DMA Setup**
  The slave FIFO application uses 2 DMA Manual channels. These channels are set up once a **set configuration** is received from the USB host. The DMA buffer size is fixed based on the USB connection speed.
  The DMA channel transfers are enabled.

- **DMA Callback**
  There exist two data paths within the FX3:
  - USB to GPIF II (U to P)
  - GPIF II to USB (P to U)

  A DMA callback is registered for the DMA produce events on each path. The event occurs when a DMA buffer (with data) is available from
  - The USB OUT endpoint
  - The GPIF II socket

  In the DMA callback, this buffer is committed, passing on the data to the
  - GPIF II socket
  - USB in endpoint

The firmware application is packaged with all dependencies and include files as a fully functional image which can be burned onto the FX3 memory for execution.

Chapter 5

Results and Discussions

5.1 Introduction

One of the main reasons USB was chosen as the high-speed data link between the FPGA and PC was to leverage the simplicity of a VCP (virtual communication ports). Most programs or scripting languages such as MATLAB and Python are able to interact and control a VCP with one line of code.

Any other device would have required writing drivers specific for each OS.

The SSR's different interfaces (telemetry, telecommand, pulse command, record data, playback data) each have separate modular hardware units to perform these respective operations. The FX3 SuperSpeed Explorer Kit integrates all of these operations onto a single unit, coupling the advantage of USB 3.0's high speed data transfer mechanism.

As a precursor to testing solid state recorders, we first implement the Slave FIFO interface to the GPIF pins of the FX3 and configure the microcontroller as explained in detail in the previous chapters. The observed results, along with their reasons, mechanisms and irregularities with the expected results will be noted and discussed upon in detail during the course of this chapter.

In a nutshell, to summarise the work and expected results for phase 1 of the project:

- The GPIF II Designer is used to configure the I/O matrix
- The state machine for slave FIFO interface is designed and commissioned for integration with the FX3 firmware
- Read-write operations are observed with the configured GPIF interface as functions of time and input control signals
- A comprehensive firmware application is designed to handle USB/ GPIF callbacks and events, along with DMA configurations, descriptor definitions, thread and process handling along with setting up debug mechanisms.

The above list is a highly condensed, yet all-comprising version of the multitude of concepts and implementations discussed in the previous chapters. This chapter aims to deliver results on the basis of the same list in a detailed manner, in addition to further theoretical background and leverage.

5.2 GPIF II Designer: I/O Matrix and State Machine

The GPIF II block is configured by writing to a set of device registers and configuration memories.

The GPIF II Designer utility generates the configuration data corresponding to the communication protocol to be implemented based on the graphical state machine. This toolgenerates the GPIF II configuration information in the form of a C header file that defines a set of data structures.

For achieving the **asynchronous 2-bit slave FIFO interface**, the I/O matrix utilizes the following pin mapping scheme:

EZ – USB FX3 pin	Synchronous slave FIFO interface with 16-bit data bus	Synchronous slave FIFO interface with 32-bit data bus
GPIO[0:7]	Data pins [0:7]	Data pins [0:7]
GPIO[8:15]	Data pins [8:15]	Data pins [8:15]
GPIO[16]	GPIF Clock	GPIF Clock
GPIO[17]	Chip select (SLCS)	Chip Select (SLCS)
GPIO[18]	Write (SLWR)	Write (SLWR)
GPIO[19]	Output Enable (SLOE)	Output Enable (SLOE)
GPIO[20]	Read (SLRD)	Read (SLRD)
GPIO[21:23]	Flag [A:C]	Flag [A:C]
GPIO[24]	PKTEND	PKTEND
GPIO[25]	FLAGD	FLAGD
GPIO[28]	Address bit 1 (A1)	Address bit 1 (A1)
GPIO[29]	Address bit 0 (A0)	Address bit 0 (A0)
GPIO[33:40]	Available as plain GPIO	Data pins [16:23]
GPIO[46:49]	UART pins	Data pins
GPIO[53]	SPI SCK	SPI SCK
GPIO[55]	MISO (Master in Slave out)	UART pin
GPIO[56]	MOSI (Master out Slave in)	UART pin

Table 5.1: *Pin mapping for Slave FIFO Interface*

With the pin mapping as shown, an I/O matrix is developed and paired with the state machine displayed in Figure 3.13. The GPIF II Designer Tool is used to perform read and write operations under different scenarios to simulate and observe the timing diagrams of the output control signals with respect to state transitions, flags and input control signals.

The following sections display the expected and resultant output for read and write strobes to the GPIF II interface.

5.2.1 Slave FIFO Read Sequence and Interface Timing

An external processor or FPGA (functioning as master of the interface) may perform single-cycle or burst data accesses to FX3's internal FIFO buffers.

The external master drives the **two-bit address** on the ADDR lines and asserts the **read or write strobes**.

The FX3 asserts the flag signals to indicate empty or full conditions of the buffer.

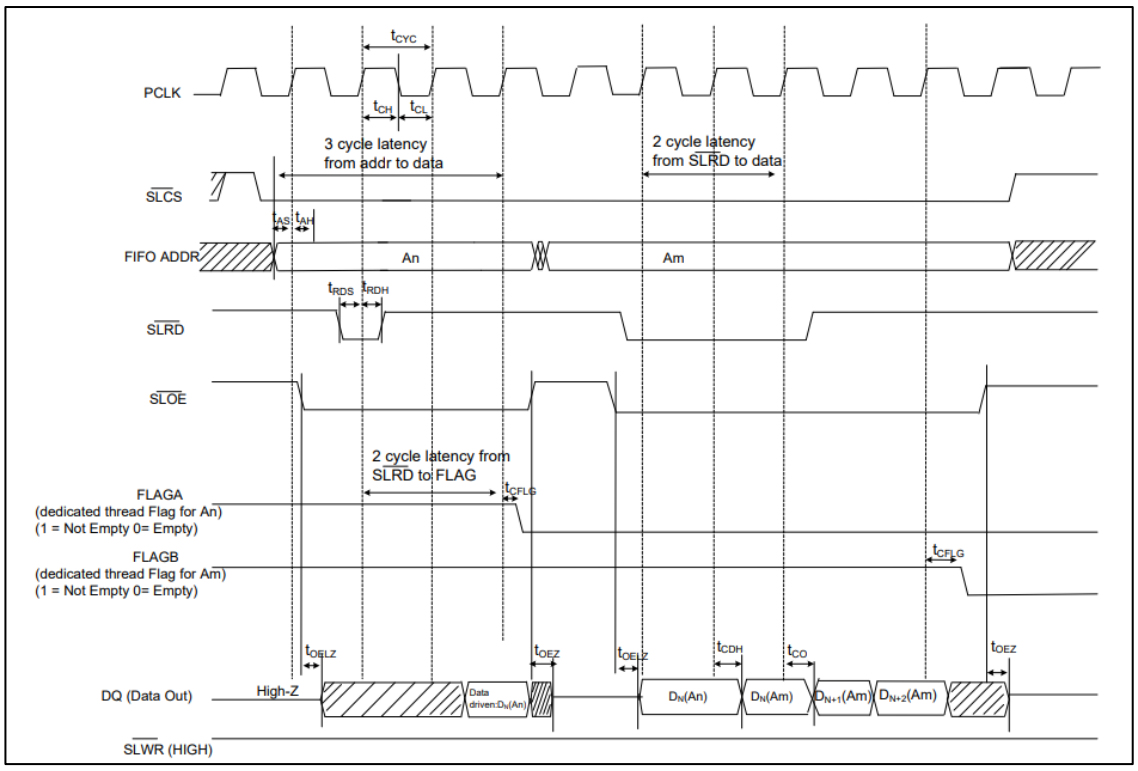


Figure 5.1: *Synchronous Slave FIFO Read Sequence* ***(Note that all signals are active low)***

As observed from the timing diagram of the ideal read sequence, the order of execution for performing synchronous slave FIFO is as follows:

- The FIFO address is stable and SLCS (chip select) is asserted
- SLOE (output enable) is asserted, and it signals the FX3 to drive the data bus.
- SLRD (read) is asserted.

The FX3 FIFO pointer is updated on the rising edge of the clock while SLRD (read) signal is asserted. This action starts the propagation of data from the newly addressed FIFO to the data bus.

Note: To drive the data bus, SLOE (Output Enable) must also be asserted in concurrence with SLRD (read) signal.

The external processor monitors flag signals for flow control. Flag signals are configured to show empty/ full/ partial status for dedicated or current threads.

5.2.2 Slave FIFO Write Sequence and Interface Timing

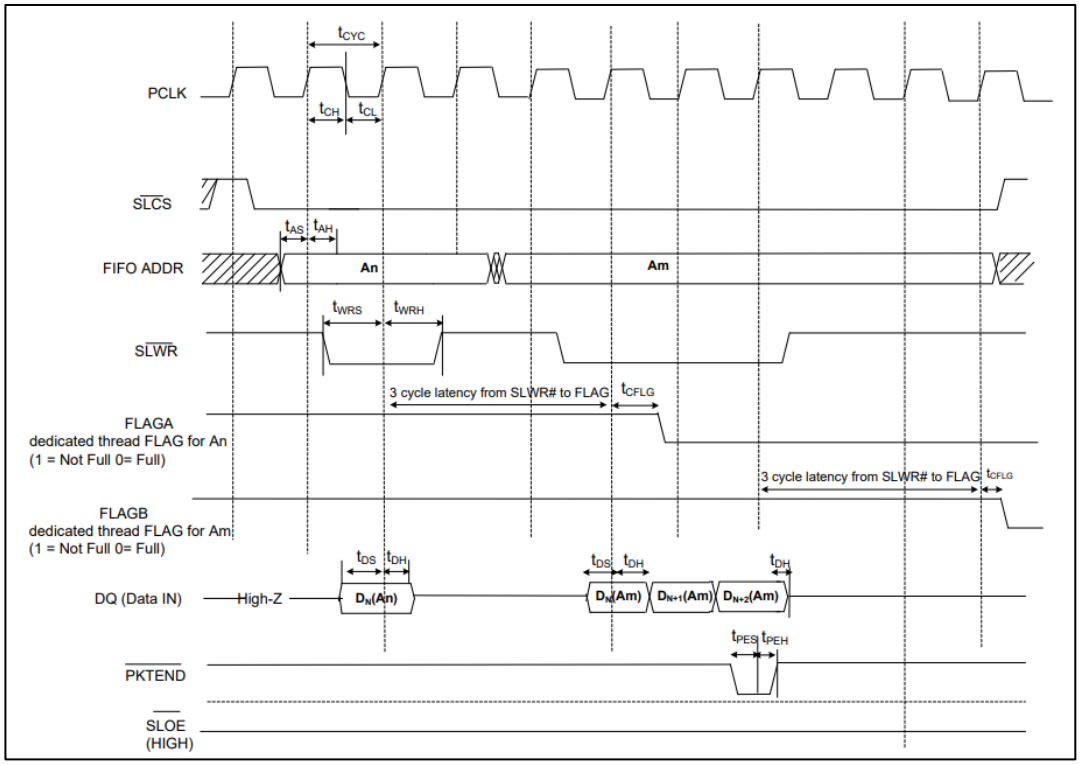


Figure 5.2: *Synchronous Slave FIFO Write Sequence* ***(Note that all signals are active low)***

Similar to read sequences, the order of execution to perform writes to the synchronous slave FIFO interface is:

- FIFO address is stable and the signal SLCS (chip select) is asserted.
- External master/ peripheral outputs the data onto the data bus.
- SLWR (write) is asserted.

- While SLWR is asserted, data is written to the FIFO. On the rising edge of the clock, the FIFO pointer is incremented.
- The FIFO flag is updated after a slight delay from the rising edge of the clock.

5.2.3 Analysing the Signal Timing of the GPIF II Interface

After entering the state machine corresponding to the P-port interface, the Timing window can be used to simulate the relative timing of the input and output signals.

To perform a timing simulation,

- Select the specific state machine
- Create a new scenario and a corresponding time frame
- Define a state navigation path

The following figures depict the timing simulation observed on the Designer tool.

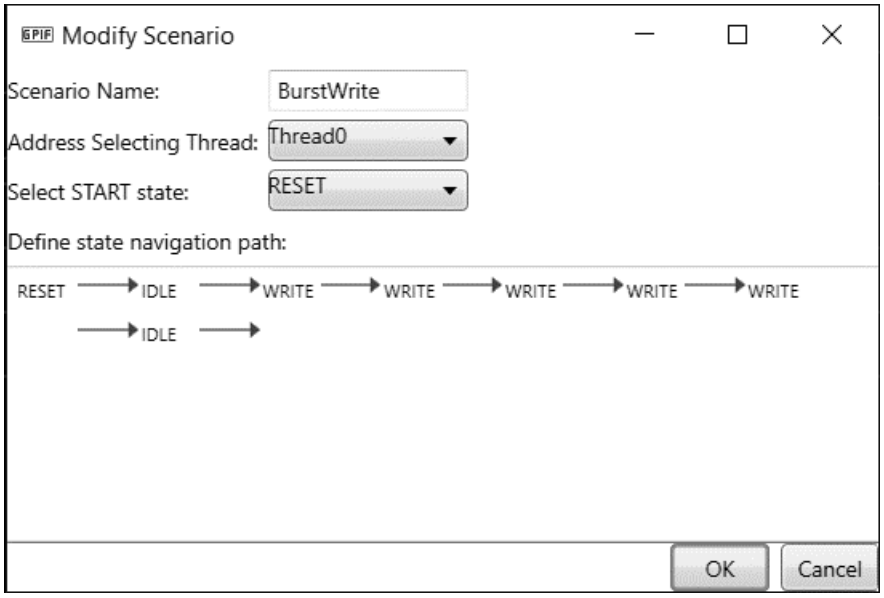


Figure 5.3: *Scenario Dialog Box for a Write Operation*

The tool continuously provides a dropdown menu showing all possible next states for the currently selected state, and the user can define a path by repeatedly selecting desired states.

The timing scenario shown above is saved and the output is obtained as follows:

As per the above timing simulation:

- **SLOE** (Output Enable) is high throughout the simulation

- **SLCS** (chip select) and **SLWR** (write) strobes are toggled synchronously and cause no impact when both flags are low.
- Around 70 ns, when all three signals (SLCS, SLWR and SLOE) are high along with both flags, the address bus is **enabled on the rising edge of clock**.
- There's a significant two-cycle delay (60 ns – 70 ns) observed due to flag updates per clock cycle.

Similarly, consider another scenario with customized state transitions.

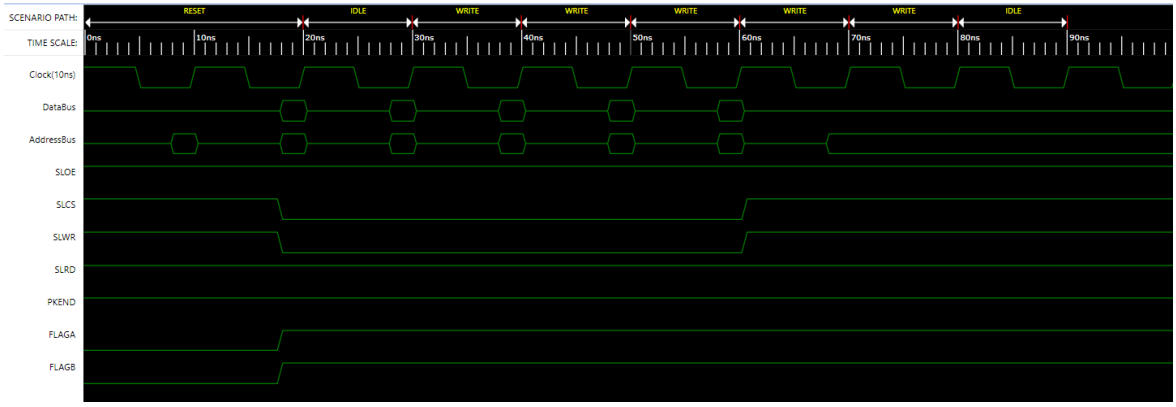


Figure 5.4: *Timing Simulation observed for Write Sequence* ***(Scale – X axis: 100 ns)***

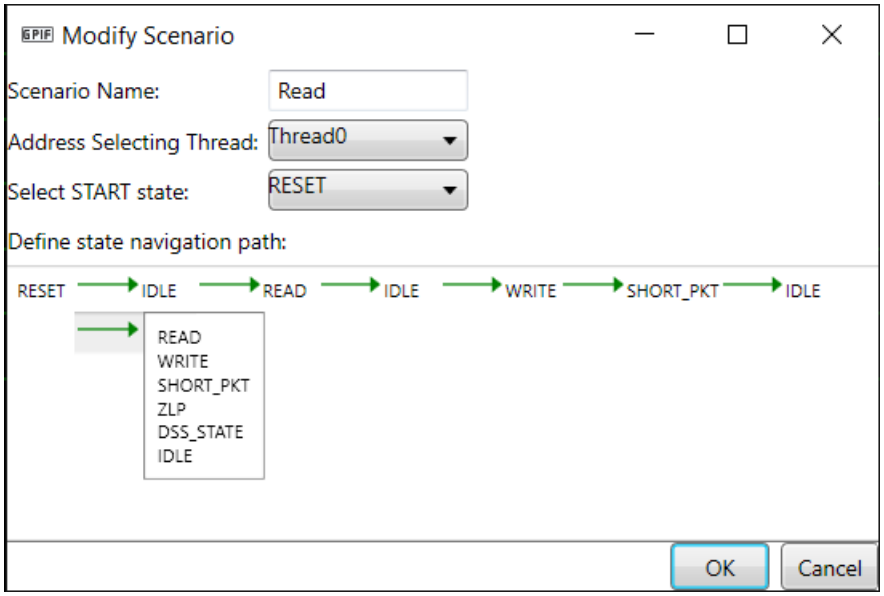


Figure 5.5: *Scenario Dialog Box for a custom operation.*
Note that the final state is IDLE, and the dropdown is shown for representational purposes

For the above designed scenario, the timing simulation is observed as follows:

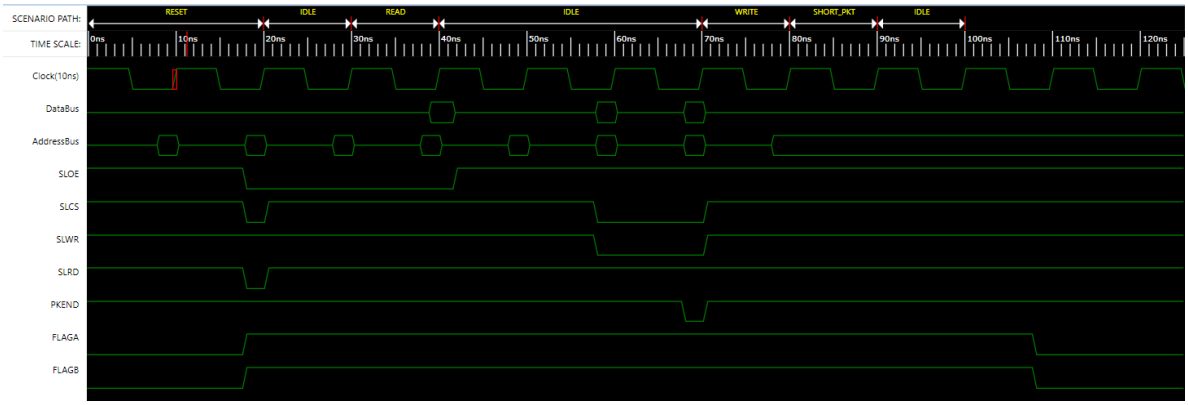


Figure 5.6: *Timing Simulation observed for a Custom Sequence* ***(Scale – X axis: 125 ns)***

A short packet can be committed to the USB host by using the PKTEND signal. The external processor is designed to assert the PKTEND signal along with the last word of data (**at 70 ns**) and the SLWR pulse corresponding to the last word.

Also, note that there is no specific signal to indicate a short packet sourced from the USB for read sequences.

5.2.4 Streaming transfers onto the FX3

The FX3 firmware application is developed to facilitate data transfer between peripherals, and the SDK provides a couple of useful tools to track and gauge the extent of data transfer within the microcontroller.

The **USB Control Centre Utility**, along with the **C++ Streamer tool** can be used to quantitatively measure the average throughput observed at the USB and GPIF endpoints. For example, in case of a bulk IN transfer, here's the procedure on measuring peak and average throughput at the USB 3.0 port:

- The FX3 must be programmed with a firmware image pertaining to input stream IN interface. It can be programmed from the PC using the Control Centre Utility in the SDK.
- After downloading the firmware onto the FX3, data transfers can be initiated from the Control Centre Utility and the Streamer utility can gauge the observed throughput.
- The external processor/ FPGA will start writing continuously to FX3's socket as soon as **flag A** goes high.
- From the PC, issue continuous bulk IN transfers and start the process.

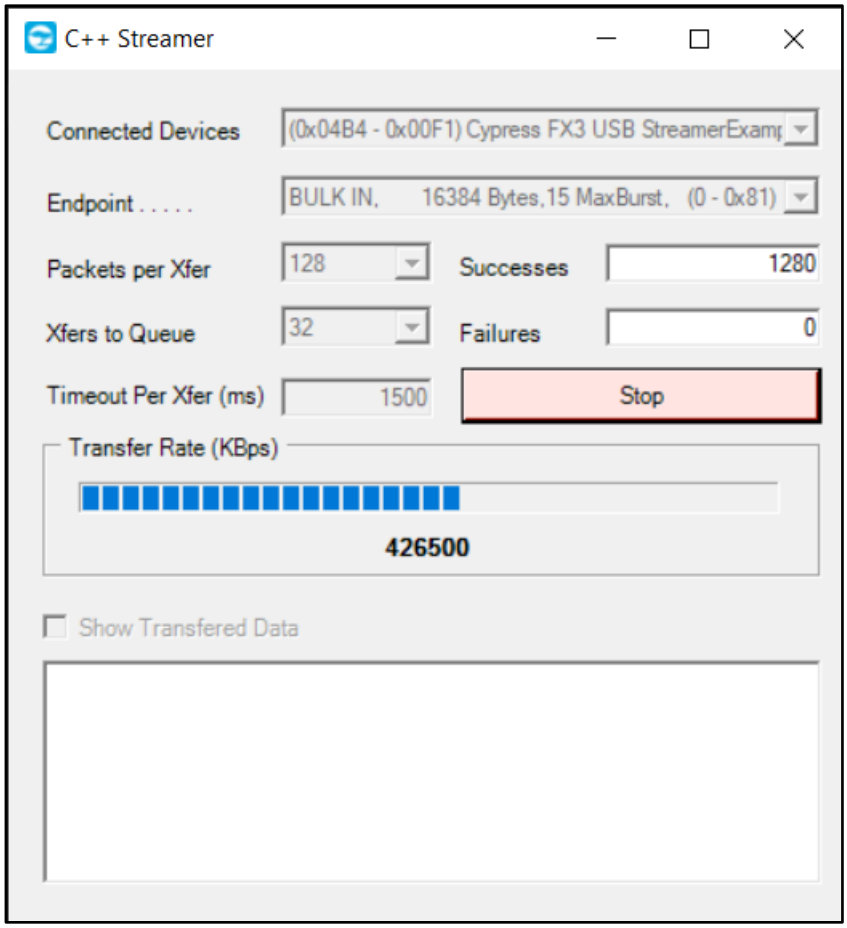


Figure 5.7: *Stream IN performance displayed in the streamer utility*

Chapter 6

Conclusion and Future Enhancements

6.1 Closing Remarks

The current phase of the project focusses on configuration of the FX3 SuperSpeed Explorer Kit for integration with another external system for performing high speed data transfers and testing **solid-state recorders** for inclusion in space missions.

Based on the implementational details and literature survey conducted, it is possible to consider qualified alternatives to the FX3 microcontroller which consist of technological specifications and capabilities far exceeding those of the prescribed SuperSpeed Explorer Kit.

For instance, using an **Nvidia Jetson Nano** or a **Raspberry Pi 4** may seem like powerful alternatives given their superior CPU and RAM capacity, but they're not included in the system for a variety of reasons; the major one being that these devices are standalone powerful devices which do not contain extensive SDK support or pin configuration access for implementing interfaces such as **slave FIFO** on GPIO pins.

Moreover, the FX3 fulfills with all the project's requirements by

- Providing an appropriate software development kit for firmware building
- Including tools to gauge and define hardware pins and state machines
- Granting full access to internal slave FIFO buffers for additional programmability
- Integrating USB 3.0 with a variety of other serial peripherals for multiple functionalities

The next phase of the project deals with programming an external processor (FPGA) in Verilog, in a manner similar to that observed with the FX3 in the previous chapters. Moreover, the firmware will be combined and shipped within a single package as an **image file**, subsequently abstracted by a GUI-driven user-friendly application built using Visual Studio and UI/UX tools such as CSS and Bootstrap.

6.2 Future Enhancements and Prospects

This project designs a high-speed data transmission system based on FPGA and USB 3.0 synchronous Slave FIFO mode, where the FPGA retains core control of the system. Compared with general data transmission systems, this model greatly

increases the data storage capacity and increases **transmission rate** to an average of 422 MB/s and **a logical resource occupancy** of less than 1%, which meets the real-time requirements for high-speed data transmission.

As an additional measure to ensure speed and reliability at a significant tradeoff with portability, the system can be coupled with NVMe SSDs and PCs with USB 3.1/ Type-C support to observe far greater throughput between peripherals.

We will also be dealing with the MILS STD 1553 bus in the upcoming phase of the project. This 1553 bus will be implemented on the FPGA in order to perform various types of transfers between the bus controller and the remote terminals in order to achieve the desired outcome. In the near future, we would investigate the various types of transfers, the various types of broadcast transactions, and their implementation.

References and Related Work

[1] S. Segars. The ARM9 family-high performance microprocessors for embedded applications. *Proceedings International Conference on Computer Design*, 1998.

[2] Q. Yi, M. Shi, M. Chen and G. Wang. Research and design of embedded microprocessor based on ARM architecture. *13th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)*, 2016.

[3] Ryzhyk, Leonid. (2006). The ARM Architecture.

[4] Cypress FX3 Programmer's Manual, Doc. #001-64707, 2018 Edition

[5] Ms. Radhika Koti, Ms. Divya Meshram. An Overview of Advance Microcontroller Bus Architecture Relate on APB Bridge. *International Journal of Scientific and Research Publications*, April 2013.

[6] P. Giridhar and P. Choudhury. Design and Verification of AMBA AHB. *1st International Conference on Advanced Technologies in Intelligent Control, Environment, Computing & Communication Engineering (ICATIECE)*, 2019.

[7] Divya M and Dr. K. A. Radhakrishna Rao. AHB design and verification AMBA 2.0 using System Verilog. *International Journal of Advance Research, Ideas and Innovations in Technology*, 2018.

[8] Kommirisetti Bheema Raju, Bala Krishna Konda. Design and Verification of AMBA APB Protocol. *International Journal of Engineering Research and Application*, January 2017.

[9] EZ - USB FX3 Technical Reference Manual. Spec No. 001-76074, May 2017

[10] Mukunthan J, A Daniel Raj, Keerthivadana S, Kiruthika R, Shruthi T, Snegasri S. *Design and Implementation of AMBA APB Protocol. IOP Conf. Series: Materials Science and Engineering*, 2021

[11] Abhinandan Aggarwal, Prof. (Dr.) Neelam Sharma. Implementation of AMBA AHB protocol using Verilog HDL. *International Journal on Recent and Innovation Trends in Computing and Communication*, 2016.

[12] EZ - USB FX3 API Reference Guide, Version 1.3.4, 2018 Edition

[13] Varsha Vishwakarma, Abhishek Choubey, Arvind Sahu. Implementation of AMBA AHB protocol for Wide Narrow BUS-SLAVE combination using

VHDL. *International Journal of Computational Engineering Research*, March 2012.

[14] Cypress EZ - USB FX3 SDK Suite User Guide, Version 1.3.4, 2018 Edition

[15] A V S R L Bharadwaj, L. Padma Sree. Implementation of On-chip Network Protocol AMBA AHB. *International Journal of Advanced Research in Computer and Communication Engineering*, July 2015.

[16] Shashank Ahire, Gaurav Gawali, Anuja Nanhe and K. Sivasankaran. Implementation of High Data Rate AMBA AHB for On Chip Communication. *International Journal of Applied Engineering Research*, 2014.

[17] Cypress EZ - USB FX3: Getting Started with FX3 SDK, Version 1.3.4, 2018 Edition

[18] L. Deeksha and B. R. Shivakumar. Effective Design and Implementation of AMBA AHB Bus Protocol using Verilog. *International Conference on Intelligent Sustainable Systems (ICISS)*, 2019.

[19] Minyeol Seo, Ha Seok Kim, Ji Chan Maeng, Jimin Kim and Minsoo Ryu. An Effective Design of Master-Slave Operating System Architecture for Multiprocessor Embedded Systems. *Advances in Computer Systems Architecture, 12th Asia-Pacific Conference, ACSAC*, 2007.

[20] GPIF II Designer 1.0, Doc. #001-75664, 2015 Edition

[21] Deepika, Jayanthi K Murthy. Interrupt Enabled Priority Based Master Slave Communication using SPI Protocol. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, July 2020.

[22] Fenxian Tian, Juan Li, Xinxin Sun and Jun Yang. Design and implementation of USB3.0 Data Transmission System based on FPGA. *3rd International Conference on Computer Engineering, Information Science & Application Technology (ICCIA 2019), Advances in Computer Science Research*, 2019.

[23] CYUSB3KIT-003 SuperSpeed Explorer Kit User Guide, Doc. #001-93186, 2014 Edition

[24] Joseph P. Lakeman. An FPGA based Test bench with high-speed Data Acquisition for Mixed Signal Application, *Thesis Paper*, August 2015.

[25] Y. J. Qian and K. Cui. Design of high-speed CCD data acquisition system based on FPGA and USB 3.0. *International Conference on Information and Communication Technology Convergence (ICTC)*, 2015.

[26] Y. Gong and F. Yu. Design of high-speed real-time sensor image processing based on FPGA and DDR3. *3rd IEEE International Conference on Computer and Communications (ICCC)*, 2017.

[27] AN65974 - Designing with the EZ-USB FX3 slave FIFO interface

[28] Li Guoren, Du Yaoyao, Zeng Bing, Wei Long, Sun Yunhua, Wang Peilin, Li Xiaohui, & Wei Shujun. PET system based on USB3.0 high-speed data transmission. *Nuclear Electronics and Detection Technology*, 2013.

[29] B. Janßen, M. Hübner and T. Jaeschke. An AXI compatible cypress EZ-USB FX3 interface for USB-3.0 SuperSpeed. *International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, 2014.

[30] Russell E. Averill. Nonvolatile Solid State Data Recorder for Space Applications.

Appendix

Term	Description
USB 3.0	USB 3.0, also called SuperSpeed USB, is the third generation of Universal Serial Bus standard. It is capable of transmitting data at a maximum rate of 5 Gigabits per second (Gbps). This makes it 10 times faster than the previous than the USB 2.0 standard.
Cypress EZ-USB FX3 SuperSpeed Explorer Kit	Cypress EZ-USB FX3 is the industry's only SuperSpeed USB 3.0 peripheral controller that enables developers to add USB 3.0 device functionality to any system.
Firmware	A firmware is a tangible electronic component with embedded software instructions etched directly into a piece of hardware. It operates without going through APIs, the operating system, or device drivers—providing the needed instructions and guidance for the device to communicate with other devices.
Software Development Kit (SDK)	An SDK is a collection of software development tools in one installable package. They facilitate the creation of applications by having a compiler, debugger and a software framework.
GPIF II Interface	The GPIF II is a programmable **state machine** that provides the flexibility of implementing an industry-standard or proprietary interface. It can function either as a master or slave.
UART	UART (Universal Asynchronous Receiver/Transmitter) is usually an individual integrated circuit (IC) used for serial communications over a computer or peripheral device serial port.
SPI	The serial peripheral interface (SPI) is a communication interface used to send data between multiple devices. These devices are organized into a master and slave configuration, in which the master has control over the slaves and the slaves receive instruction from the master.

FIFO	FIFO (first in, first out) is a method for organising the manipulation of a data structure where the oldest (first) entry, or head of the queue, is processed first.
ARM	An ARM processor is one of a family of CPUs based on the RISC (reduced instruction set computer) architecture developed by Advanced RISC Machines (ARM).
RAM Buffer	The RAM buffer is normally used for input/output processes for sending or receiving data, for example, input from keyboard or output to printer.
Sockets	A socket is the point of connection between a peripheral hardware block (USB or UART or GPIF) and FX3 RAM.
Threads	A thread is a dedicated data path in the GPIF II block that connects the external data pins to a socket.
Descriptors	A DMA descriptor is a set of registers allocated in the FX3 RAM. It holds information about the address and size of a DMA buffer as well as pointers to the next DMA descriptor.
Callbacks	A callback function is a function that is passed as an argument to another function, to be “called back” at a later time based on the occurrence of an event or a trigger signal.
DMA	Direct memory access (DMA) is the process of transferring data without the involvement of the processor itself.
Flags	Flags are associated with specific threads or the currently addressed thread and indicate the status of the socket mapped to that thread.