Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

UNIDAD ZACATENCO

Departamento de Computación

# Una Interfaz Generica para Proteger Datos en Memorias USB

**Tesis que presenta**

**Marco Antonio Soto Hernández**

**Para Obtener el Grado de**

**Maestro en Ciencias**

**En Computación**

**Director de la Tesis:**

**Dr. Debrup Chakraborty**

**México, D.F.** **Febrero, 2014**

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

UNIDAD ZACATENCO

Departamento de Computación

# A Generic Interface for Securing Data in USB Memories

by

**Marco Antonio Soto Hernández**

**Thesis Advisor:**

**Dr. Debrup Chakraborty**

**México, D.F.** **Febrero, 2014**

# Abstract

Portable computing devices have increased a lot both in numbers and varieties in the past few years. These devices provides the convenience of performing several tasks even when the user in "on the go". This mobility has also given rise to new issues in data security. In this thesis we focus on the security of portable memory devices, which are ubiquitous. Because of their small size and portability, there is a high risk of them getting lost, which means the stored data gets compromised.

There are encryption enabled memory devices, which are already available in the market. In these devices, the encryption facility comes along with the specific memory, and thus, such memories are more expensive. Here we plan the solution through a different approach: we aim to design a generic device, which would be capable of enabling encryption in *any* USB memory device. This means, the encryption device can be attached to any ordinary USB memory. We call our device as the Middleman, and present its complete design and implementation.

MiddleMan acts as an interface between a USB memory and a host computer. Externally, it has two USB ports; a host computer and an USB memory can be attached to any of these two ports. There is an encryption algorithm built in within MiddleMan. All bulk data transfers from the host to the memory gets encrypted when it goes through the MiddleMan. Transfers from the memory to the host gets decrypted. This ensures that the data stored in the memory is always encrypted. Internally, the MiddleMan has a functionality of both as an USB device and also as a host. To the memory device, it poses itself as a host, and in turn to the host, it poses like a device, this enables transparent communication to the user of the USB memory. MiddleMan uses a low cost tweakable enciphering scheme as the encryption algorithm, thus it guarantees stronger security than most commercially available encryption enabled USB memories.

This work also includes a prototype implementation of the MiddleMan. As a proof of concept, we implement the full functionality of the MiddleMan in a Spartan 3E FPGA board. For the purpose of this implementation, we also required to enhance the USB support provided by the board. To do this we designed a special extension for the Spartan 3E board, which we call S3USB. This device extends the functionality of the bare board in several respects and can be of independent interest.

We tested the prototype in various computing platforms and found it to work satisfactorily. We also did some controlled performance tests and found it to be efficient. The slowdown that take place in the MiddleMan is within acceptable limits

in a typical usage scenario.

# Resumen

Los dispositivos de cómputo pórtatil se han incrementado mucho en cuanto a número y variedad desde hace algunos años. Estos dispositivos nos ofrecen una forma conveniente de realizar varias tareas incluso cuando el usuario va "sobre la marcha". Esta mobilidad tambien ha generado un incremento las cuestiones referentes a la seguridad de datos. En esta tesis nos enfocamos en la seguridad de los dispositivos de memoria portátiles, los cuales son ubicuos. Debido a su pequeño tamaño y portabilidad, existe in alto riezgo de que se extravíen, lo que significa que los datos almacenados son comprometidos.

Existen dispositivos de memoria con capacidades de cifrado, los cuales se encuentran disponibles en el mercado. En estos dispositivos, el cifrado se encuentra adentro del dispositivo, lo cual incrementa su costo. Aquí hemos planeado una solución por medio de un enfoque diferente: nosotros nos concentramos en el diseño de un dispositivo genérico, el cual pueda ser capaz de cifrar cualquier memoria USB. Nosotros hemos llamado a este dispositivo MiddleMan, además presentamos su completo diseño e implementación.

MiddleMan actúa como una interface entre una memoria USB y una computadora huésped. Posee dos puertos externos; una computadora huésped y una memoria USB pueden ser conectados a cualquiera de estos dos puertos. Existe un algoritmo de cifrado implementado dentro de MiddleMan. Todas las transferencias en masa desde el huésped hasta la memoria son cifradas a travez de MiddleMan. Transferencias de la memoria al huésped son descifradas. Esto nos asegura que los datos almacenados en la memoria siempre permanezcan cifrados. Internamente, MiddleMan funciona como un dispositivo USB y también como un huésped. Ante el dispositivo de memoria, este se presenta como un host, mientras que para el huésped, se presenta como un dispositivo, esto logra comunicaciones transparentes para el usuario de la memoria USB. MiddleMan utiliza un esquema de cifrado ajustable como su algoritmo de cifrado y por lo tanto, garantiza una seguridad más fuerte que las soluciones comerciales.

Este trabajo también incluye una implementación prototipo de MiddleMan. Como prueba de concepto, hemos implementado todas las funcionalidades de MiddleMan en una tarjeta FPGA Spartan 3E. Para propósitos de esta implementación, también requerimos de una mejora en el soporte USB de la Spartan 3E. Para esto diseñamos una extensión especial para dicha tarjeta y la llamamos S3USB. Esta extensión incrementa la funcionalidad de la tarjeta FPGA en varios aspectos y este diseño es de interés por si solo.

Hemos probado el prototipo en varias plataformas de cómputo y encontramos que funciona satisfactoriamente. También hemos hecho algunas pruebas de rendimiento controladas y encontramos que es efficiente. El alentamiento que toma lugar dentro de MiddleMan, se encuentra dentro de los límites aceptables para un escenario de uso típico.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| AES | Advanced Encryption Standard |
| ACK | Acknowledge in a USB handshake |
| CBW | Command Block Wrapper |
| CRC | Cyclic Redundancy Code |
| CSW | Command Status Wrapper |
| EEFF | Effective Relative Permittivity |
| ENPX | Endpoint X, where X is a number |
| FIFO | Fist IN First Out memory system |
| FPGA | Field Programmable Gate Array |
| FSM | Finite State Machine |
| IN | USB IN transaction |
| IP | Intelectual Property |
| MLUH | Multilinear Universal Hash |
| NAK | Negative acknowledge in a USB handshake |
| NRZI | Non Return Zero Invert |
| OTG | USB On The Go |
| OUT | USB OUT transaction |
| PCB | Printed Circuit Board |
| PHY | Physical Layer |
| PID | USB Packet Identifier |
| RFID | Radio Frequency Identification |
| SIE | Serial Interface Engine |
| TES | Tweakable Enciphering Schemes |
| SOF | USB Start Of Frame |
| UPRC | USB Packet Replication Core |
| STES | Small TES |
| ULPI | UTMI Low Pin Interface |
| USB | Universal Serial Bus |
| UTMI | USB 2.0 Transceiver Macrocell Interface |

# Chapter 1

# Introduction

Data storage is an important part in computer systems. There are several ways of storing data, ranging from internal hard drives in desktop/laptop computers to portable flash based storage devices. Even the popular portable digital devices in the market like smart phones, tablets, cameras, etc., have some way to store data. The increase of portable devices and the improvements in their connectivity to the Internet has raised numerous security concerns regarding data stored in various kinds of storage media. The convenience of portability which most devices offer today can also be a problem: the smaller the device, the easier it can be lost.

In the last few years there has been significant activity towards designing cryptographic algorithms suitable for storage security [20, 21, 18, 11, 19, 32, 10], also there have been some standardization efforts [3, 4]. Thus in terms of designing secure and efficient cryptographic algorithms suitable for securing stored data there has been significant improvements. On the other hand, in open literature much is not reported about practical deployments of these algorithms in specific devices. Though, there are commercially available storage devices with encryption facilities, the intricacies of their designs are trade secrets. This prevents easy reproducibility of those designs and also prevents large scale security analysis of such devices.

In this thesis we design a novel device for enabling encryption in an external USB memory. We name the device as MiddleMan. The basic functionality of MilddleMan is to act as a special interface between an USB memory and the host computing system (say a desktop or a laptop computer). MiddleMan is equipped with an encryption algorithm and all bulk data transfers from the computer to the memory are encrypted when passing through the MiddleMan. Transfers from the memory to the computer get decrypted. Thus, the data that resides in the memory is always encrypted. The MiddleMan is neither a part of the USB memory nor the host computing system, thus *any* USB memory device can be connected to *any* computer using the device in the middle to obtain functionality of encryption in the memory.

In this thesis we do a careful design of the MiddleMan and also do a prototype implementation of the design in a Spartan 3E FPGA board. We decided to use an FPGA, because it provides high flexibility in terms of low level, fast hardware applications. In addition, the Spartan 3E board also provides a low cost FPGA

solution. We have performed extensive testing with the prototype using different computing environment and we found the device to work satisfactorily.

## 1.1 Storage Security and the MiddleMan

Bulk storage in modern computing devices is provided through hard disks or NAND type flash memories. Though the technology behind these two options are quite different, they are similar in that both of them are organized as sectors. A sector is the smallest addressable part of these storage systems and the host computing device reads or writes on these medias at sector level granularity.

There exist a rich cryptographic literature which addresses the problem of encrypting sector oriented storage media. Also, there is a consensus in the cryptographic community that a class of encryption algorithms, called tweakable encryption schemes (TES) are best suited for the application [20]. To date there are numerous proposals of TES which are efficient and proven secure in a well accepted and reasonable security model.

The specific application area where TES can be deployed is called "in-place disk encryption". In this application the encryption algorithm is part of the disk/memory controller and it encrypts sectors before it writes to the disk and decrypts sectors before it sends them to the operating system. This model is general and can be applied to any storage media organized into sectors, irrespective of the other high level systems like operating systems, file systems, etc.

There are two relevant active standards which specifies cryptographic algorithms for storage encryption. The IEEE 1619-2007 [3] specifies an algorithm called XTS-AES [30], and the IEEE 1619.2-2010 [4] specifies two algorithms EME2 [21, 18] and XCB [31]. EME2 and XCB are TES[1]. XTS-AES is not a TES, and the security guarantees that XTS-AES provides are much less than those provided by any secure TES, in particular XTS encryption is vulnerable to tampering which any TES can resist to some extent. Security limitations of XTS are widely known and some details regarding this can be found in [2, 17].

The advantage of XTS over any known TES is its efficiency, as it uses far less operations than any known TES and thus when implemented either in software or hardware would be much more efficient both in terms of speed and area compared to any TES. But it is to be noted that there have been extensive studies on implementation of TES both in software [17] and hardware [29, 7, 10, 28] which suggests that a TES when implemented using reasonable resources can exceed the data rates of modern disk/memory controllers. Hence there is no efficiency barrier for the use of TES in the required application, and given that TES are more secure it should be preferred to the XTS algorithm.

[1] Recently there was some attacks reported on XCB [9, 22], which questions the so far known security properties of XCB. These observations thus puts into doubt the security of XCB and also the standard which specifies it.

In this work we are interested in encryption of USB memories. To apply in-place disk encryption in case of USB memories, the most suited design strategy would be to implement the encryption algorithm within the memory controller. Thus, the encryption unit would be part of the memory stick. Such products are commercially available [25], for example, Kingston, which is a major player in the market of USB memories, has a spectrum of offerings of secure USB drives [1]. As expected the details of these designs are not publicly available, but they reveal the encryption algorithms that they use. The secure USB drives marketed by Kingston either use XTS (as in Data Traveller Vault Privacy 3.0 and Data Traveller 6000) or CBC (as in Data Traveller Vault Privacy Managed and Data Traveller 4000). It is curious that these commercially available products do not implement any TES, which is known to be the best suited algorithms for this application. The reason behind this is probably both CBC and XTS are much less complex than most known TES. It is to be noted that CBC is a privacy only block cipher mode, which does not provide adequate security for disk encryption and as already discussed, XTS though standardized for disk encryption purposes, cannot match the security levels provided by a secure TES.

The design that we propose in this thesis is different from the commercial products described above. We aim to achieve a generic device for USB memory encryption, which would be independent of both the memory device and the host. Our proposal, the MiddleMan, acts as a encryption interface between any host and any USB bulk storage device. The most important advantage of this philosophy is that we do not require to depend on the USB memory vendor for its security, we can use an encryption algorithm of our choice to encrypt our memory. Moreover, a single encryption device can be used to encrypt multiple memories.

To achieve the above mentioned objective, the MiddleMan has both the functionality of an USB host and a device. In simple terms, the MiddleMan accepts two USB connections, with one it connects to the memory device and with the other it connects to a host (for example a PC). To the memory device, it acts like a host and thus receives bulk data transfers and redirects to the real host. To this real host the MiddleMan posses as an USB device. MiddleMan has an encryption algorithm residing within it: the host to the memory device transfers are encrypted and the transfers from the memory device to the host are decrypted. While encrypting/decrypting the MiddleMan follows the philosophy of in-place disk encryption, i.e., it performs sector wise encryption/decryption of all bulk transfers which passes through it.

The basic design of MiddleMan can support any encryption algorithm. In our implementation we use a specific TES which is called STES [10]. STES is a recent proposal which is very different from existing TES. The difference is that STES was designed with the goal that when implemented suitably, it would have a very small hardware and power footprint. This characteristic of STES is very important when is used in MiddleMan, as this gives us a secure, efficient and low cost encryption algorithm.

The design of MiddleMan was done so that a big part of the design can be implemented within a reconfigurable fabric like a field programmable gate array (FPGA).

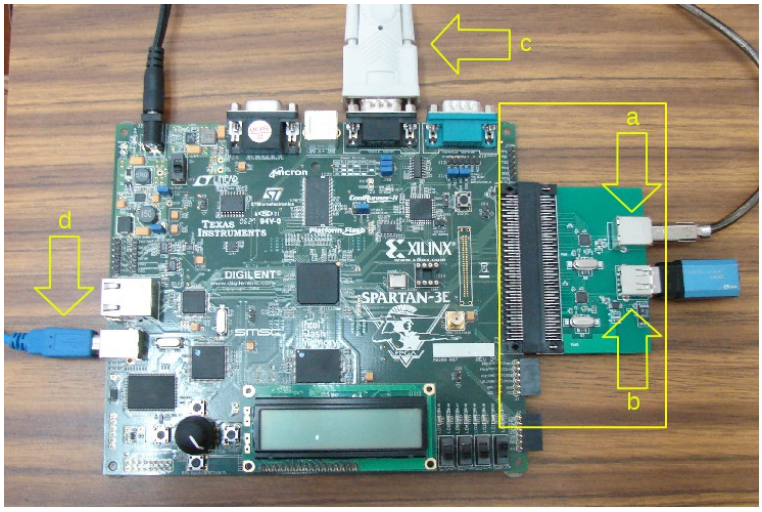


Figure 1.1: The prototype of the MiddleMan. The yellow rectangle marks the hardware extension S3USB that we attached with the Spartan 3E board for implementing the physical layer. **a**) Device USB port, **b**) Host USB port, **c**) Serial port for debugging, **d**) FPGA configuration port.

With the advent of small, low power and cheap FPGA families like Spartan , Lattice etc., it may be possible that a practical realization of MiddleMan has an FPGA in it. This would enable reconfigurability, then at least the built-in encryption algorithm may be suitably changed according to diverse security needs or local security legislation. Thus, MiddleMan provides a highly flexible and generic way to implement in place encryption in a variety of USB mass storage devices, which can be practically useful. Moreover, to the best of our knowledge there does not exist a device or a design which is even similar to the design philosophy and goals of the MiddleMan.

This thesis also includes a prototypical implementation of most functionalities of MiddleMan. The implementation was done in a Spartan 3E starter board. This board is equipped with some rudimentary USB facilities, but these were not enough for our purpose. Thus, we needed to extend the board with some custom designed hardware where we implement the USB physical layer. We call this special add on to the board as S3USB. This extended board can be of independent interest, and can serve as a test platform for prototypical implementations of other applications, which uses the USB protocol.

To achieve the intended functionality, we had to implement the whole USB protocol from scratch in an FPGA. There exist some open FPGA cores [27, 36], but we found them to be in sufficient for our purpose. In our design, we implement the full device USB protocol and additionally implement the part of the host protocol, which is required for our design. Finally we suitably aggregate an FPGA implementation of STES in our design to obtain the full functionality of MiddleMan.

The final prototype that we developed is shown in Figure 1.1. As is clear from the figure, for implementing the prototype we had to extend the Spartan 3E board by a

custom built circuit, the S3USB (marked by a yellow rectangle in the figure). The device and host USB ports (marked a and b respectively in the figure) are parts of S3USB. For using the device one needs to insert a USB memory in port **b** and connect it to the host (a PC, laptop, tablet etc.) through port **a**. Once such a connection is made, all data written to the USB memory by the host gets encrypted and any data read by the host from the USB memory gets decrypted by the MiddleMan. For enabling encryption functionality in any USB memory, the memory needs to be formatted after it is connected to the host through the MiddleMan. This ensures that the boot sector and the partition table in the memory are also encrypted. Thus if it is connected to any host without the MiddleMan, the host will not recognize the data inside the USB memory[2]. But when connected with the MiddleMan the host receives the data in a decrypted form, and thus usual functionality of an USB memory resumes.

As of now, the cryptographic key is hard coded in the device, thus for security, this device has to be physically secure, i.e., if an adversary has access to both the encrypted memory and the MiddleMan then no security remains. This is because we have not implemented any specialized key management in the system. The design of MiddleMan allows for other modes of key inputs and we suggest various ways in which keys can be managed in the device. We plan to implement them in the near future.

## 1.2 The Organization of the Thesis

This thesis is organized into a total of eight chapters including the current chapter. Next we give a brief overview of the contents of the following chapters.

In **Chapter 2** we discuss the general problem of storage encryption and tweakable enciphering schemes, which are the state of the art schemes for in-place storage encryption. In this Chapter we also present the detailed algorithm for STES [10], which is the algorithm that we selected for incorporation in the MiddleMan. The hardware design for STES is also briefly discussed in this Chapter.

In **Chapter 3** we discuss the basics of the USB protocol. As this protocol forms the heart of MiddleMan, we discuss all relevant aspects which we required for the design and implementation of our device.

In **Chapter 4** we give a design overview of the MiddleMan. In this chapter we carefully define the design goals of MiddleMan, and then do a systematic exploration of the design space and concretize our design decisions. Finally, we present the basic architecture of the MiddleMan. The design decisions are highly motivated by the technology we used to implement it. In this chapter we also discuss some basic implementational issues.

In Chapters 5 and 6 we provide the details of our design and implementations. In **Chapter 5** we discuss in detail the design and implementation of the extension

[2]Some operating systems like Windows, would ask to format the device as if there was no data, doing so will destroy the encrypted information

board for Spartan 3E. As discussed earlier we use the extension to implement some functionality of the USB physical layer. This chapter also discuss the details of the circuit diagram along with the specifications of the printed circuit board that we designed for the purpose.

In **Chapter 6** we discuss the details of the USB protocol along with the encryption algorithm implemented in the FPGA. This chapter provides all details related to how the device and host protocols run inside the MiddleMan. In this Chapter we also provide the detailed circuit diagrams of the various units that implements the whole functionality of the MiddleMan.

In **Chapter 7** we discuss some performance results for the device. We briefly discuss the various tools that were used to debug and test the device. Then, we present the performance of the device in terms of data transfer rates and FPGA utilization.

We conclude the thesis in **Chapter 8** and point out the limitations of our implementation and also discuss various ways to overcome these limitations and improve the design.

# Chapter 2

# Secure Storage of Data

In this chapter we discuss about storage encryption. The purpose of such an encryption is to protect data inside a persistent media. This data may be the target of a malicious entity accessing the device, either virtually trough malicious software or physically by stealing.

The main goal of this Chapter is to introduce tweakable enciphering schemes (TES) and to describe a specific TES called STES. For that purpose we need to fix some notations which we do in Section 2.1. Then, in Section 2.2 we discuss what storage encryption is and how we can achieve it by using TES. In Section 2.3 we provide a formal definition of TES, and also briefly mention some existing constructions. In section 2.4 we present STES, which is the scheme of choice for this work. The description of STES presented in this Chapter closely follows the original work where STES was first proposed [10]. In Section 2.5 we discuss a hardware architecture of STES which we used in our prototype.

## 2.1 Notations

**Binary strings:** By $\{0,1\}^*$ we denote the set of all binary strings and $\{0,1\}^n$ denotes the set of binary strings of length $n$. For $X, Y \in \{0,1\}^*$, by $X||Y$ we denote the concatenation of $X$ and $Y$. $|X|$ denotes the length of $X$ in bits. $\mathsf{bits}(X,i,j)$ denotes the binary string formed by the substring of $X$ extending from position $i$ to position $j$. By $X \lll r$ we mean a $r$ bit circular left shift of $X$, i.e. if $X = 011011$ then $X \lll 2 = 101101$.

**Finite fields and string representations:** By $\mathbb{F}_q$ we mean a finite field with $q$ elements. We shall often treat $n$-bit binary strings as elements in $\mathbb{F}_{2^n}$. For such a treatment we would see a $n$ bit string as a polynomial with coefficients in $\{0,1\}$, i.e. if $A = a_{n-1}a_{n-2}\ldots a_1a_0$ be a binary string then we will treat $A$ as the polynomial $A(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$, which is of degree at most $n-1$. The addition of elements $A, B \in \{0,1\}^n$ is defined as $A \oplus B$, where the operation $\oplus$ denotes the bit wise xor of the strings. For defining multiplication of strings $A$ and $B$ we consider them as polynomials $A(x), B(x)$ and define $AB = A(x) \cdot B(x) \bmod q(x)$ where $q(x)$

is an irreducible polynomial of degree $n$.

## 2.2 Storage Encryption

Common storage media like hard disks, flash memories etc. are organized into sectors of equal sizes. Securing data stored inside storage systems organized into sectors is best achieved through a paradigm called "in-place encryption". This type of encryption sees data as a collection of sectors, and it does not know about the high level organization (like files and directories) of the data. The best way to achieve in-place encryption is to implement the encryption algorithm within the controller of the storage device, i.e., the encryption scheme should be a part of the storage media itself not a part of the host computing system. In an in place encryption, when the operating system writes a block, the controller encrypts the transmitted data before being stored. Similarly, for a read procedure, the requested block is decrypted inside the memory controller before being sent to the operating system. The obvious performance guarantees required for such encryption schemes is that its throughput should match the data rates of the storage device to prevent visible delays in data transfers.

The most accepted solution to achieve storage encryption is by the use of Tweakable Enciphering Schemes (TES) [20], which we discuss in the next section.

## 2.3 Tweakable Enciphering Schemes

Let $\mathsf{Key}$, $\mathsf{Tweak}$, $\mathsf{Msg}$ be a finite non empty sets which we call as the key space, tweak space and message space respectively. A tweakable enciphering scheme is a function $\mathcal{E} : \mathsf{Key} \times \mathsf{Tweak} \times \mathsf{Msg} \to \mathsf{Msg}$. For $K \in \mathsf{Key}$, $T \in \mathsf{Tweak}$ and $M \in \mathsf{Msg}$ we denote $\mathcal{E}(K, T, M)$ by $\mathcal{E}_K^T(M)$. It is required that $\mathcal{E}$ has the following properties:

1. For every $K \in \mathsf{Key}$, and every $T \in \mathsf{Tweak}$, $\mathcal{E}_K^T : \mathsf{Msg} \to \mathsf{Msg}$ is a permutation, which in turn implies that there exist an inverse function $\mathcal{D} : \mathsf{Key} \times \mathsf{Tweak} \times \mathsf{Msg} \to \mathsf{Msg}$, such that for every $K \in \mathsf{Key}$, $T \in \mathsf{Tweak}$ and $M \in \mathsf{Msg}$, $\mathcal{D}_K^T(\mathcal{E}_K^T(M)) = M$.

2. $\mathcal{E}$ is length preserving, i.e., for every $K \in \mathsf{Key}$, $T \in \mathsf{Tweak}$ and $M \in \mathsf{Msg}$, $|\mathcal{E}_K^T(M)| = |M|$.

In addition, $\mathcal{E}$ is required to have some security properties. In formal terms, a secure TES is supposed to be a *strong pseudo-random permutation* [28]. As we would not further require to treat TES in a formal manner thus we do not try to formally define security properties of TES here. Informally, for a secure TES, the outputs of both $\mathcal{E}$ and $\mathcal{D}$ should "look" like random strings to any computationally bounded adversary.

For the purpose of disk encryption the set $\mathsf{Msg}$ should contain equal length strings of size same as the sector size of the storage media where the algorithm needs to be

applied, for current hard disks the sector sizes are 4096 bytes and for USB memories it is 512 bytes. The tweak is considered to be the sector address. The tweak provides some variability of the cipher text in the sense that if the same message is stored in two different sectors (with different sector addresses) then they get encrypted by distinct tweaks and hence the corresponding ciphertexts are different.

In the last decade there has been significant advancements in constructions of secure TES. The first complete construction of a TES to be used for disk encryption was CMC [20]. After this, several schemes appeared in the literature, and all the schemes use a block-cipher as the main cryptographic primitive. Currently available TES are classified in three main groups: *encrypt-mask-encrypt*, *hash-counter-hash* and *hash-encrypt-hash*. We briefly discuss these three paradigms of TES construction next:

- **Encrypt-Mask-Encrypt** type schemes have two layers of encryption with a lightweight masking layer in between. Examples from this class of schemes are CMC [20], EME [21] and EME* [18].

- **Hash-Counter-Hash** type schemes have two layers of universal hashing with a counter mode of encryption in between. XCB [31], HCTR [37], HCH [8] and HMCH [32] are some examples of this class of schemes.

- **Hash-Encrypt-Hash** uses two universal hash functions along with an Electronic Code Book encryption step in between. Examples of this class of schemes are PEP [11], TET [19] and HEH [33] etc.

In terms of efficiency, to encrypt a $m$ block message (where the block size is same as the block size of the underlying block cipher, for example, for AES the block size is 128 bits) the hash-encrypt-hash type schemes require about $2m$ block cipher calls. The other schemes require both block cipher calls along with finite field multiplications. HMCH and HEH require about one $m$ block cipher calls and $m$ finite field multiplications, all other schemes mentioned above in the hash-counter-hash and hash-encrypt-hash categories require about $m$ block cipher calls and $2m$ finite field multiplications.

Recently a TES called STES was proposed in [10]. STES is different from the previous proposals by the fact that it uses stream ciphers instead of block ciphers. The main motivation behind the design of STES was to have a secure TES which when implemented in hardware would have a small hardware footprint, such that the algorithm can be used for encryption in small and power constrained devices. In [10] STES was implemented with various datapaths, and the results demonstrate that STES provide excellent time-area trade-off and when implemented in certain class of FPGAs has very low power consumption. Because of these encouraging features of TES we decided to use STES in MiddleMan. In the next section we give an overview of the construction of STES which follows closely the description in [10].

## 2.4 STES Construction

The two main building blocks of STES are a special universal hash function called MLUH and a stream cipher.

A Stream Cipher is a function $\mathsf{SC}_K : \{0,1\}^\ell \rightarrow \{0,1\}^L$ where $\ell << L$. This function takes as input an initialization vector (IV) of small length $\ell$ and outputs a long and random looking string $L$.

STES construction also uses a particular hash function called Multilinear Universal Hash. A multilinear hash function with output length $b$ and datapath $d$ is defined as

$$\mathsf{MLUH}_K^{d,b}(M) = h_1||h_2||\cdots||h_b$$

where the input message $M$ is written as $M = M_1||M_2||\cdots||M_m$, the key $K$ is written as $K = K_1||K_2||\cdots||K_{m+b-1}$, such that $|M_i| = d$ for $1 \leq i \leq m$ and $|K_i| = d$ for $1 \leq i \leq m+b-1$, and

$$\left.\begin{array}{rcl} h_1 & = & M_1 \cdot K_1 \oplus M_2 \cdot K_2 \oplus ... \oplus M_m \cdot K_m \\ h_2 & = & M_1 \cdot K_2 \oplus M_2 \cdot K_3 \oplus ... \oplus M_m \cdot K_{m+1} \\ \cdot & \cdot & \cdots \\ h_b & = & M_1 \cdot K_b \oplus M_2 \cdot K_{b+1} \oplus .... \oplus M_m \cdot K_{b+m-1}. \end{array}\right\} \quad (2.1)$$

The additions and multiplications in the above equations are in $\mathbb{F}_{2^d}$.

STES uses a stream cipher with IV SC, and a multilinear hash function as shown in Figure 2.1. The algorithm also makes calls to an external function called Feistel which is shown in Figure 2.2.

The algorithm is parameterized on the key $K$ and tweak $T$, additionally it uses a string fStr. It has been mentioned in [10] that the string fStr can be a public constant. The algorithm assumes that the length of the IV of the stream cipher used is $\ell$ bits and the datapath of the MLUH (described in Eq. 2.1) is $d$.

## 2.5 Implementation of STES

In this work we did not implement the STES, but used an implementation done by Cuauhtemoc Mancillas-López. We only adapted the implementation for its proper use within the MiddleMan. For completeness we describe briefly the architecture of STES that we used, the description again is adopted from the original article [10].

For our implementation we instantiated the stream cipher using Trivium, which uses a 80 bit IV, and we used a MLUH with 8-bit data path. The architectural overview is shown in Figure 2.3. Next we describe the architecture with reference to the algorithm in Figure 2.1.

The circuit presented in Figure 2.3 consists of the following basic elements:

1. The MLUH constructed with 8-bit multipliers.

| $\boldsymbol{STES.Encrypt}^T_K(P)$ | $\boldsymbol{STES.Decrypt}^T_K(C)$ |
|---|---|
| 1. $b \leftarrow \frac{\ell}{d}$; | 1. $b \leftarrow \frac{\ell}{d}$; |
| 2. $b_1 \leftarrow \frac{\lvert P\rvert+\lvert T\rvert-2\ell}{d}$ ; | 2. $b_1 \leftarrow \frac{\lvert C\rvert+\lvert T\rvert-2\ell}{d}$ ; |
| 3. $\ell_1 \leftarrow (b_1+b-1)d$; | 3. $\ell_1 \leftarrow (b_1+b-1)d$; |
| 4. $\ell_2 \leftarrow (2b-1)d$; | 4. $\ell_2 \leftarrow (2b-1)d$; |
| 5. $\ell_3 \leftarrow \lvert P\rvert - 2\ell$; | 5. $\ell_3 \leftarrow \lvert C\rvert - 2\ell$; |
| 6. $P_1 \leftarrow \mathsf{bits}(P,1,\ell)$; /* $\lvert P_1\rvert = \ell$) */ | 6. $C_1 \leftarrow \mathsf{bits}(C,1,\ell)$; /* $\lvert C_1\rvert = \ell$) */ |
| 7. $P_2 \leftarrow \mathsf{bits}(P,\ell+1,2\ell)$; /* $\lvert P_2\rvert = \ell$ */ | 7. $C_2 \leftarrow \mathsf{bits}(C,\ell+1,2\ell)$; /* $\lvert C_2\rvert = \ell$ */ |
| 8. $P_3 \leftarrow \mathsf{bits}(P,2\ell+1,\lvert P\rvert)$; /* $\lvert P_3\rvert = \ell_3$ */ | 8. $C_3 \leftarrow \mathsf{bits}(C,2\ell+1,\lvert C\rvert)$; /* $\lvert C_3\rvert = \ell_3$ */ |
| 9. $\tau \leftarrow \mathsf{SC}_K^{\ell_1+\ell_2+\ell}(\mathsf{fStr})$; | 9. $\tau \leftarrow \mathsf{SC}_K^{\ell_1+\ell_2+\ell}(\mathsf{fStr})$; |
| 10. $\tau' \leftarrow \mathsf{bits}(\tau,1,\ell_1)$; | 10. $\tau' \leftarrow \mathsf{bits}(\tau,1,\ell_1)$; |
| 11. $\beta \leftarrow \mathsf{bits}(\tau,\ell_1+1,\ell_1+\ell)$; | 11. $\beta \leftarrow \mathsf{bits}(\tau,\ell_1+1,\ell_1+\ell)$; |
| 12. $\tau'' \leftarrow \mathsf{bits}(\tau,\ell_1+\ell+1,\ell_1+\ell+\ell_2)$; | 12. $\tau'' \leftarrow \mathsf{bits}(\tau,\ell_1+\ell+1,\ell_1+\ell+\ell_2)$; |
| 13. $Z_1 \leftarrow \mathsf{MLUH}^{d,b}_{\tau'}(P_3\|\|T) \oplus \beta$; | 13. $Z_2 \leftarrow \mathsf{MLUH}^{d,b}_{\tau'}(C_3\|\|T) \oplus (\beta \lll 1)$; |
| 14. $A_1 \leftarrow P_1$; | 14. $B_1 \leftarrow C_1 \oplus Z_2$; |
| 15. $A_2 \leftarrow P_2 \oplus Z_1$; | 15. $B_2 \leftarrow C_2$; |
| 16. $(B_1,B_2,W) \leftarrow \mathsf{Feistel}^{\ell,d}_{K,\tau''}(A_1,A_2,\ell_3)$; | 16. $(A_1,A_2,W) \leftarrow \mathsf{InvFeistel}^{\ell,d}_{K,\tau''}(B_1,B_2,\ell_3)$; |
| 17. $C_3 \leftarrow P_3 \oplus W$; | 17. $P_3 \leftarrow C_3 \oplus W$; |
| 18. $Z_2 \leftarrow \mathsf{MLUH}^{d,b}_{\tau'}(C_3\|\|T) \oplus (\beta \lll 1)$; | 18. $Z_1 \leftarrow \mathsf{MLUH}^{d,b}_{\tau'}(P_3\|\|T) \oplus \beta$; |
| 19. $C_1 \leftarrow B_1 \oplus Z_2$; | 19. $P_1 \leftarrow A_1$; |
| 20. $C_2 \leftarrow B_2$; | 20. $P_2 \leftarrow A_2 \oplus Z_1$; |
| ***return*** $(C_1\|\|C_2\|\|C_3)$; | ***return*** $(P_1\|\|P_2\|\|P_3)$; |

Figure 2.1: STES: A TES using SC and MLUH. The $\ell$-bit string fStr is a parameter to the whole construction. The length of the IV of SC is $\ell$ and the data path of MLUH is $d$. This figure has been taken verbatim from [10].

| $\mathsf{Feistel}^{\ell,d}_{K,\tau''}(A_1, A_2, i)$ | $\mathsf{InvFeistel}^{\ell,d}_{K,\tau''}(B_1, B_2, i)$ |
|---|---|
| 1. $H_1 \leftarrow \mathsf{MLUH}^{d,b}_{\tau''}(A_1)$; | 1. $H_2 = \mathsf{MLUH}^{d,b}_{\tau''}(B_2)$; |
| 2. $F_1 \leftarrow H_1 \oplus A_2$; | 2. $F_2 = H_2 \oplus B_1$; |
| 3. $(G_1, W) \leftarrow \mathsf{SC}^{\ell+i}_K(F_1)$; | 3. $G_2 = \mathsf{SC}^{\ell}_K(F_2)$; |
| 4. $F_2 \leftarrow A_1 \oplus G_1$; | 4. $F_1 = B_1 \oplus G_2$; |
| 5. $G_2 \leftarrow \mathsf{SC}^{\ell}_K(F_2)$; | 5. $(G_1, W) = \mathsf{SC}^{\ell+i}_K(F_1)$; |
| 6. $B_2 \leftarrow F_1 \oplus G_2$; | 6. $A_1 = F_2 \oplus G_1$; |
| 7. $H_2 \leftarrow \mathsf{MLUH}^{d,b}_{\tau''}(B_2)$; | 7. $H_1 = \mathsf{MLUH}^{d,b}_{\tau''}(A_1)$; |
| 8. $B_1 \leftarrow H_2 \oplus F_2$; | 8. $A_2 \leftarrow H_1 \oplus F_1$; |
| ***return*** $(B_1, B_2, W)$; | ***return*** $(A_1, A_2, W)$; |

Figure 2.2: The Feistel network (and its inverse) constructed using a stream cipher and a MLUH. The variable $\ell$ is the length of an IV for $\mathsf{SC}$ and $d$ is the data path of $\mathsf{MLUH}$. This definition is different from the usual Feistel construction: a positive integer $i$ is provided as an additional input and a binary string $W$ of length $i$ is returned as an additional output. This figure has been taken verbatim from [10].

2. Two stream cipher cores labeled **SC1** and **SC2**.

3. Two 80-bit registers **RegH1** and **RegH2** which are used to store the output of **MLUH**.

4. Four registers labeled **regF1**, **regF2**, **regKh** and **reg**$\beta$. All these registers are 80 bits long and are formed by ten registers each of eight bits connected in cascade, so that they can be used as a FIFO queue.

5. One special register **reg**$\beta$**1** which is able to store a 80-bit data and rotate it in one bit position. This register outputs 8-bit data each clock cycle when the control input *ce* is activated.

6. Seven multiplexers labeled **1**, **2**, **3**, **4**, **5**, **6** and **7**.

7. The control unit whose details are not shown in the Figure.

8. The connections between **MLUH** and the registers **RegH1**, **RegH2** have a data path of 80 bits. All other connections have a data path of 8 bits.

9. The input lines $M_i$, $IV$ and $K$ which receives the data and tweak, the initialization vector and the key respectively.

10. The output line $C_i$ which outputs the cipher.

The **MLUH** computes the MLUH, it receives as inputs message blocks $M_i$, tweak blocks $T_i$ and key blocks $K_i$ and give as output the result of MLUH in its output port S. The register **RegH1** and **RegH2** receive the output from S as input, in this case $|S| = 80$ bits. The registers **RegH1** and **RegH2** are designed to give eight bit

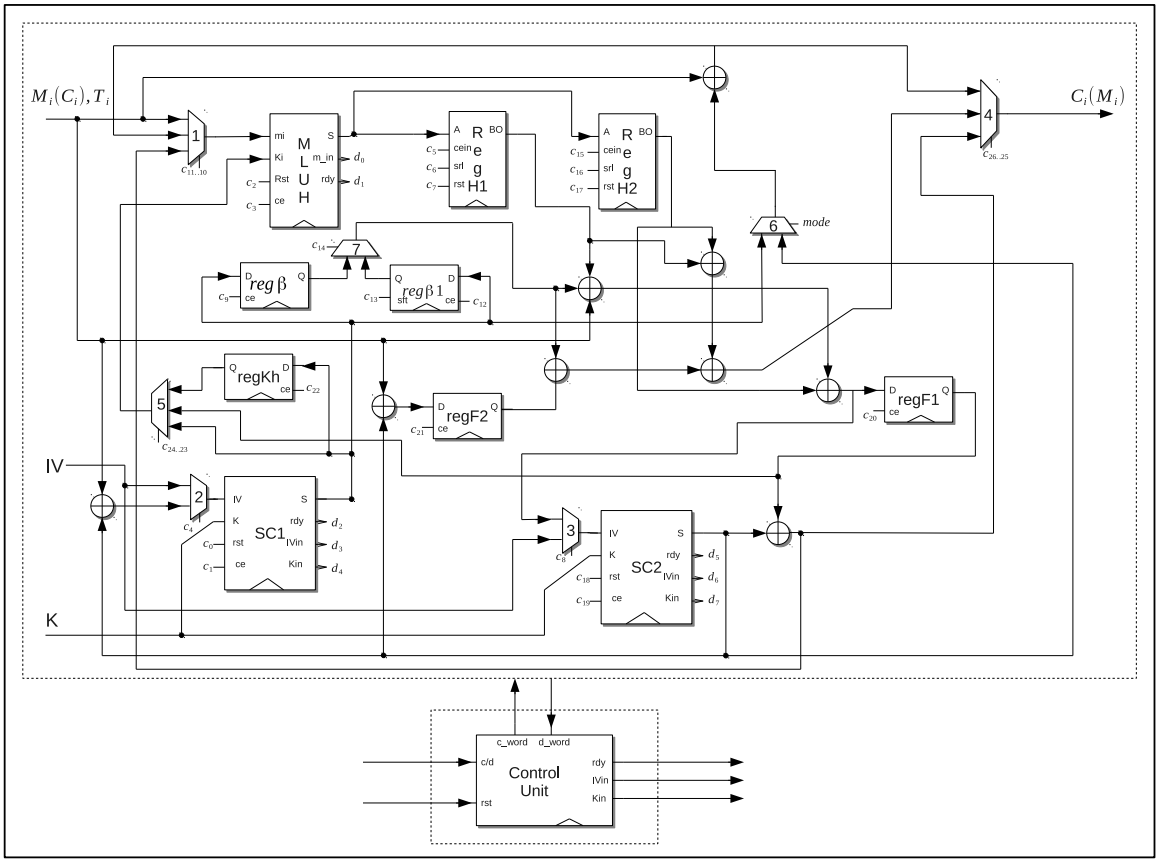


Figure 2.3: Architecture of STES

blocks as outputs in each clock cycle in their output port BO. The **MLUH** receives its input from the $3 \times 1$ multiplexer labeled **1**. Notice, that in the algorithm of STES, the MLUH is called on three different inputs. Multiplexer **1** helps in selecting these inputs. In the algorithm MLUH is called on two different keys $\tau'$ and $\tau''$, thus, **MLUH** can receive the key from two different sources: the key $\tau'$ is received directly from the output of the stream cipher **SC1** or **SC2**. The key $\tau''$ is received either directly from stream cipher **SC1** or from the register **regKh** which is used to store $\tau''$. To accommodate these selection of keys the input port Ki of **MLUH** receives the input from the $2 \times 1$ multiplexer **5**.

We use two stream ciphers **SC1** and **SC2**. Both take the key from the input line $K$ of the circuit. **SC1** receives the IV from multiplexer **2**, it selects between input line $IV$ or $F_1$. Multiplexer **3** feeds the IV to the stream cipher **SC2**, it selects between $IV$ or $F_2$.

In the algorithm of STES we can see that the output of MLUH is xored with the value of $\beta$, or $\beta \lll 1$, depending which hash is computed $Z_1$ or $Z_2$ and whether encryption or decryption mode is being executing. The selection between these two values is made with Multiplexer **7**.

In the encryption mode the stream $W$ is generated using **SC2** but in the decryption mode it is generated by **SC1**. Multiplexer **6** is used to select the correct stream cipher to produce the cipher text or plain text.

# Chapter 3

# The Universal Serial Bus

The Universal Serial Bus (USB) is a popular interface, which has been in use for several years. Almost all computer devices use this interface to provide various functionalities, ranging from basic I/O to video and audio devices.

In this chapter we provide general information on the most important aspects of the USB interface. We begin with an introduction to the bus in Section 3.1, this is important to understand the bus interconnection. Then, we discuss the internal USB components. In Section 3.2 we give a short introduction to the Physical Layer (PHY), in Section 3.3 we discuss about the Serial Interface Engine (SIE) and in Section 3.4 we discuss the USB Endpoints (ENP). These components are the building blocks for every USB device.

Section 3.5 introduces the transactions, which are the lowest level protocol of the USB communication. We also present the packet structure in this section. Transfers are build upon transactions, we discuss about the transfer types in Section 3.6.

Some tables and figures, which we used to explain several parts of the protocol, where partially taken from the USB specification [12] or the book by Axelson [6]. The description of the bus and its functionality that we provide here is not detailed, the reader may consult [12, 6] for more detailed information on the USB specification.

## 3.1 USB Basics

In this section we discuss about the basics of the USB specification. We begin by defining the bus entities and its topology in Section 3.1.1. Then we present the four speeds available in Section 3.1.2.

### 3.1.1 Components of the bus

USB is an interface created to achieve communications between a host entity and several devices. The purpose of this communication is to provide extra functionality to the host, which can be a computer or an embedded system.

Several entities can interact in the bus at the same time. We list some of these entities along with their basic functionality next:

- **Host**: Master and manager of the bus, it detects newly attached devices and configures them in order to begin communication. It schedules bus time for each of the devices attached to the bus.
- **Device**: This entity provides some functionality to the host. The most common functionality is the external data storage.
- **Hub**: The Hub routes communications between the Host and any other Hub or device attached to it's ports. It also manages bus speed conversion (low speed devices connected to a high speed bus). The Hub can have multiple ports where Devices or other Hubs can be attached. The host connects directly to a Hub called the "Root Hub".
- **Compound device**: this device is a combination of a Hub and a Device in a single entity.

The bus topology is a tiered star as shown in Figure 3.1. In this topology, each hub is a star center. Tiers are added by connecting additional Hubs in series, up to five tiers can be attached with a total number of 127 devices (including the root Hub). Despite the limit in the number of devices, it is impractical to have this many devices attached to the bus at the same time. The host assigns bus time based on the number of devices connected to the bus. Too many devices means less bus time for each of them.

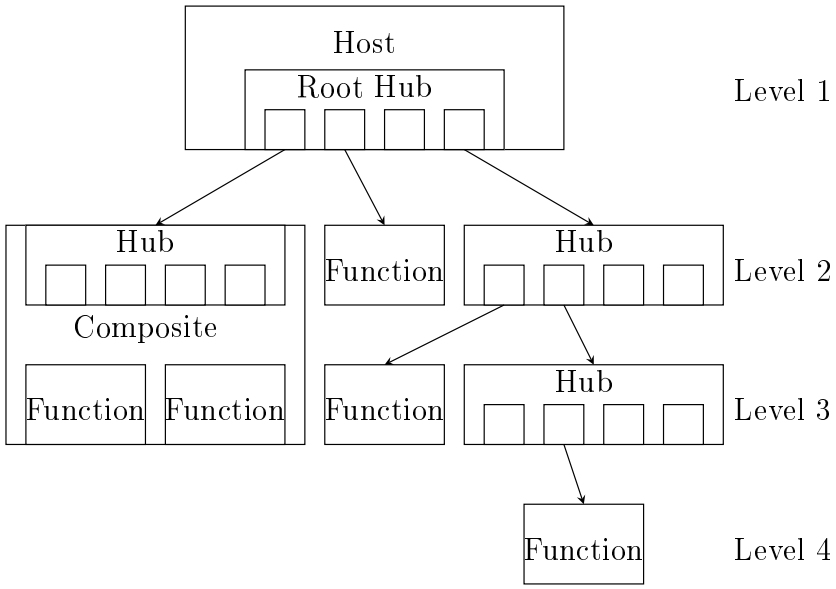


Figure 3.1: USB topology

Inside each entity there are three basic components, the Physical Layer (PHY), the Serial Interface Engine (SIE) and the Endpoint (ENP), their relationship is shown in Figure 3.2. The endpoint section is only available for Hubs and Devices.

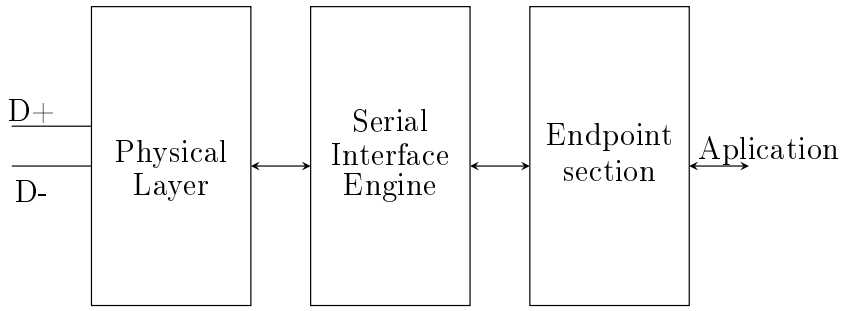


Figure 3.2: USB device components

### 3.1.2 USB Supported Speeds

The USB specification defines four speed configurations, these are shown in Table 3.1. The host detects the speed of a device by checking the D- and D+ lines shown in Figure 3.2. If a device has the D- line connected to 5V through a resistor, the host detects it as a low speed device. Full speed is detected if the device has D+ connected to 5V through a resistor instead. Every high speed device connects to the USB bus at full speed. When the host requests a device to enable high speed, if this device supports it, then it will respond to this request and the host enables high speed.

Table 3.1: USB speed configurations.

| **Speed** | **Data rate** | **Common applications** | **Supported USB versions** |
|---|---|---|---|
| Low speed | 1.5 Mbps | input peripherals | 3.0 and below |
| Full speed | 12 Mbps | Data transfer | 3.0 and below |
| High speed | 480 Mbps | Data, Audio and Video | 3.0,2.0 |
| Super speed | 5 Gbps | Data transfer | 3.0 |

The host requests high speed by using a protocol called "Chirp Protocol", it uses the signaling that will be explained in Section 3.2. The Chirp Protocol follows a sequence of K and J USB bus states, which we define in Table 3.2. We explain the basic steps next:

1. The host detects a full speed device, then it resets the bus.
2. The device responds to the reset by transmitting a full speed K state for no less than 1ms.
3. When the host detects the K state for at least 2.5$\mu$s, it waits for the device to stop transmission. After this, the host transmits a series of K and J states.

4. The device must detect the state sequence K-J-K-J-K-J. Each state must last 2.5$\mu$s. After the sequence has been detected, the device enables high speed.

USB 3.0 and super speed are not covered in this work, refer to [6] for more information on these standards.

## 3.2 The Physical Layer

The Physical Layer (PHY) is the component which deals with the physical connection in the USB bus. The responsibilities of this component are to detect the start and end of a packet in order to receive it properly. This component also decodes or encodes data transmitted from/to the bus. Encoding is explained later in this section.

The USB bus uses differential signaling techniques in order to transfer data. A differential signal is transmitted over a pair of wires which have opposite voltage levels, the USB naming convention for this pair of wires is D+ and D−. Table 3.2 shows the signal levels and possible bus states.

Table 3.2: USB 2.0 signaling

| **Bus state** | **Signal level** |
|---|---|
| Differential 1 | D+ =high, D− =low |
| Differential 0 | D+ =low, D− =high |
| Single ended 0 (SE0) | D+ =low, D− =low |
| Single ended 1 (SE1) | D+ =high, D− =high |
| J state:<br>Low speed<br>Full speed<br>High speed | <br>Differential 0<br>Differential 1<br>High speed differential 1 [1] |
| K state:<br>Low speed<br>Full speed<br>High speed | <br>Differential 1<br>Differential 0<br>High speed differential 0 [1] |

There are two types of PHYs defined by the USB specification: upstream and downstream. We will explained them next:

- A downstream facing PHY goes from a Host or Hub to a device. It has two resistors connected to ground on D+ and D- lines.

- An upstream facing PHY goes from a device to a Hub or Host. This PHY has a resistor on D- connected to 5V for low speed, for full speed, the resistor is connected from 5V to D+ instead.

[1] Full and high speeds differ in voltage levels, refer to USB 2.0 [12, pp. 145-147]

The USB specification requires the codification of bus data in the format called "non-return zero invert format" (NRZI). NRZI represents data by a level change instead of a logic level. In NRZI, a logic '0' is represented by a state change, a logic '1' is represented by not changing the previous state; Figure 3.3 shows a codification example. This codification must also stuff bits inside the data as follows: When there are six consecutive '1', the PHY must stuff a '0' before encoding, this guarantees regular changes in the bus state. Regular changes are needed by the PHY to keep synchronization of bus data.

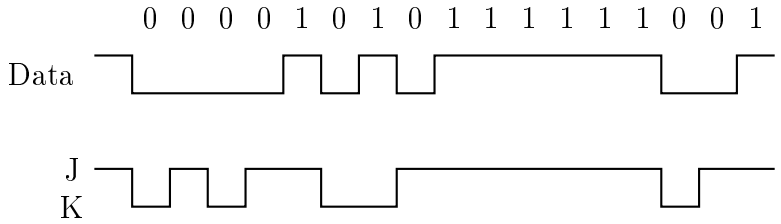


Figure 3.3: NRZI encoding

## 3.3 Serial Interface Engine

The Serial Interface Engine (SIE) has several functions inside a USB host or device entity. These functions are summarized next:

- **Data conversion**: The SIE receives data in serial format and converts it to parallel. It also converts transmitted data from parallel to serial.
- **Error checking**: The SIE performs error checking of the received data by calculating a Cyclic Redundancy Code (CRC). This code is compared against a CRC appended at the end of each received USB packet. The packet is discarded if the codes differ. CRC calculation procedure for USB packets is described in Appendix A.2.
- **Acknowledge successful transmissions**: The SIE must send an acknowledge (ACK) packet for every data packet received without errors, this process is called "Handshake".

## 3.4 Endpoints

The endpoint (ENP) is the main source and sink of data. In order to communicate with a device, a host controller addresses a specific endpoint inside a device. Endpoints can have two directions:

- **IN** endpoints store data to be sent to the host.

- **OUT** endpoints receive data from the host.

Full and high speed devices can have up to 32 endpoints (16 IN and 16 OUT). Every device must implement at least the default endpoint, which consists of the endpoint 0 in both directions. Configuration of any USB device is done through the default endpoint using a transfer type called "control transfer". We discuss this transfer type in Section 3.6. In addition to control transfers, Endpoints can be configured to respond to other types of transfers, which are also discussed in Section 3.6.

## 3.5 Protocol

This section describes the basics of the USB protocol. There are several concepts behind low level USB communication, which we present in this section. We discuss transactions and packet structures in Sections 3.5.1 and 3.5.2, these are the building blocks for USB transfers, which we introduce USB transfers in Section 3.6.

### 3.5.1 Transactions

Transactions are the lowest USB communication protocol. They are performed in the following sequence:

1. The host sends a token packet to the device
2. If the token is an OUT token, then the host send data. If the token is an IN token, then the device sends data.
3. The receiver of the data in the last step responds with an acknowledge packet. Only error free packets are acknowledged.

The host divides the bus time in intervals of 1ms called "Frames". during each frame, the host and device can exchange data. For high speed, the host divides each frame in eight parts of 125 $\mu$s. These parts are called "Micro frames". The host signals the start of each frame by sending a special token called Start of Frame (SOF). The device may use this packet to detect bus activity. If the host stops sending SOFs for more than 3 ms, the device must enter into a suspend mode, which consumes less power.

The structure of a transaction is shown in Figure 3.4. Transaction type is sent by the host in the token stage, there are three types of transactions:

- **IN**: Data travels from the device to the host. If the device has no data to send, then it responds with a handshake (acknowledge) packet.
- **OUT**: Data travels from the host to the device.
- **SETUP**: Data travels from the host to the device. This token is used to indicate that the data contains a command for device configuration.

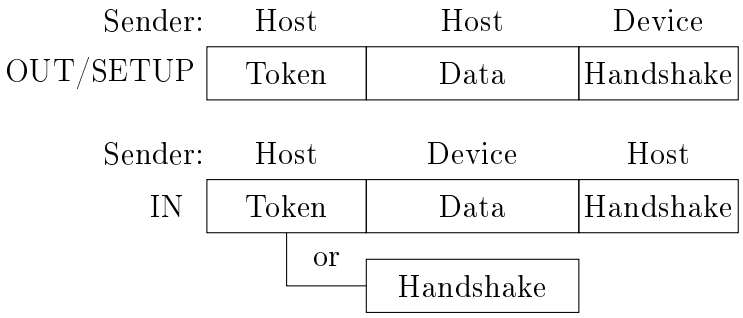


Figure 3.4: Stages of a transaction

### 3.5.2 Packet Structure

A USB transaction is build by three types of packets described in Section 3.5.1, each packet has its own purpose and structure. All USB packets begin with a Packet Identifier (PID) byte, this identifier tells a host or device the type of data received. The four less significant bits of the PID byte contain the identifier, the last four contain error protection bits. The error protection bits are calculated by complementing the value of the identifier bits. Table 3.3 shows the PID types and a small description.

#### Token packet

Token packets (shown in Figure 3.5) are only issued by the host. Their purpose is to begin a transfer and to address a specific device.

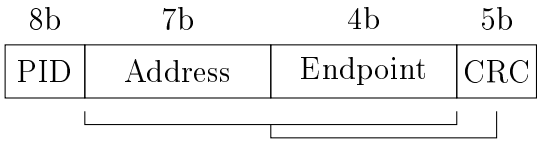


Figure 3.5: Fields inside a token packet

The PID field of a token packet identifies the type of token and the direction of the transfer. A device must decode the Address and Endpoint fields to determine if the transaction corresponds to this device.

The address field stores the USB device address, there can be up to 127 possible addresses including a default address '0'. When a device powers up or is reset, it's address register must be set to '0'. Once a device is configured, the host assigns a different address and the device must respond to any packet with an address field equal to the one it was assigned. The next field in the packet is the endpoint, this field is four bits long. It addresses 16 endpoints, the PID field is used to determine if the endpoint addressed is an IN or OUT endpoint. A SETUP token is only considered to address an OUT endpoint.

Table 3.3: USB Packet IDs.

| Type | Name | PID(bits 3 to 0) | Description |
|---|---|---|---|
| Token | OUT | 0001B | Host to function transaction |
| | IN | 1001B | Function to host transaction |
| | SOF | 0101B | Start of frame |
| | SETUP | 1101B | Host to function setup transaction |
| Data | DATA0 | 0011B | Data packet PID even |
| | DATA1 | 1011B | Data packet PID odd |
| | DATA2 | 0111B | Data PID for high speed isochronous transactions |
| | MDATA | 1111B | Data packet for high speed split and isochronous transactions |
| Handshake | ACK | 0010B | Receiver accepts error-free data packet |
| | NAK | 1010B | Receiving device cannot accept or transmit data |
| | STALL | 1110B | Endpoint halted or control request unsupported |
| | NYET | 0110B | No response yet from the receiver |
| Special | PRE | 1100B | Token to enable communications with low speed devices |
| | ERR | 1100B | Split transaction error handshake (reuses PRE value) |
| | SPLIT | 1000B | High speed split transaction token |
| | PING | 0100B | High speed flow control for a bulk or control endpoint |
| | Reserved | 0000B | Reserved PID |

The start of frame (SOF) token is a special type of packet, it can only be issued by the host and a device must not acknowledge its reception, Figure 3.6 shows this packet structure.

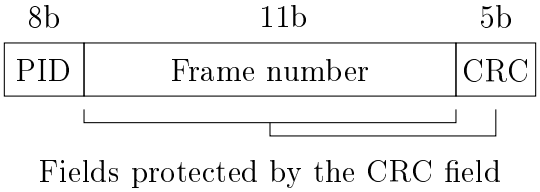


Figure 3.6: Fields inside a SOF packet.

### Data packet

Data packets, shown in Figure 3.7, are identified by any Data PID (shown in table 3.3). The data field can be up to 64 bytes for full speed and 512 bytes for high speed. The CRC field in this case is two bytes long (16 bits) and protects only the data field in the packet.

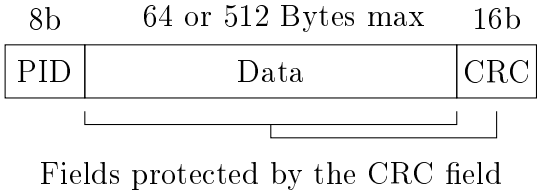


Figure 3.7: Fields inside a data packet.

### Handshake packet

The handshake packet is issued by the receiver of the data packet. It's purpose is to report the status of the current transaction. If a data packet is received with errors, no handshake is sent. The handshake consist of the PID field only (8 bits).

## 3.6 Transfer Types

This section describes the four transfer types defined for USB 2.0 (not to be confused with transactions, which we explained in Section 3.5.1). Each transfer have their own characteristics regarding latency and bus bandwidth. We must take care to select the proper transfer type when designing a USB device. We discuss all transfer types next:

**Control transfer**: This transfer is used to gather information, or send requests in order to configure a device. This transfer is received by the default control endpoint (Endpoint 0). Every device must support this endpoint.

A control transfer is shown in Figure 3.8, if consist of the following three transaction stages:

1. **Setup stage**: The host sends a setup transaction, it's structure is shown in Figure 3.4. A Command is sent in the data packet.
2. **Data stage**: If the command in the Setup stage requires the host or device to transmit more data, it is transmitted. This stage can have multiple IN or OUT transactions, every packet in this stage has the same direction.
3. **Status stage**: A transaction is performed in this stage. If there was no data stage or the direction of the previous transaction was from the

host to the device, then the status stage is an IN transaction. An OUT transaction is performed otherwise. This stage is used to report the status of the transfer.

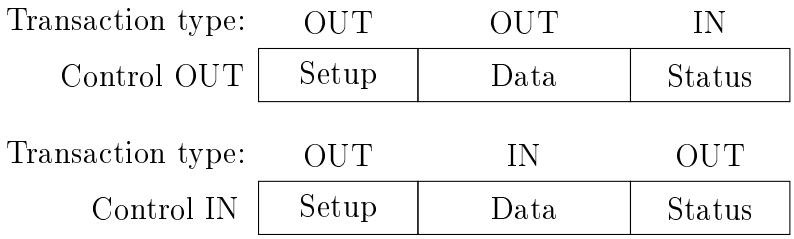


Figure 3.8: Stages of a control transfer.

Control transfer are used by the host to configure a USB device. This configuration process is called "Enumeration" and we discuss it in Appendix A.1.

**Bulk transfer**: This transfer may be used when there is the need to send big amounts of data, which are not time critical. For this type of transfers, the host allocates time for each packet when there is bus time available, therefore, a host can take much time to complete this transfer if the bus is busy. If the bus is otherwise idle, this type of transfer can be the fastest. Bulk transfers consist only of one transaction, which begins with an IN or OUT token.

**Interrupt transfer**: This transfer may be used for devices needing a critical time in their communications but don't require large amounts of data to transfer. This transfer is adequate for I/O devices like keyboards, mouse or game pads. The host guarantees a minimum bus bandwidth. The structure is similar to Bulk transfers.

**Isochronous transfer**: This transfer is used to transmit large amounts of data, which are time critical. The host guarantees a minimum bandwidth for these transfers. In order to increase streaming speed, any error checking and retry is disabled, therefore, this transfers are only suited for transmitting data where error can be tolerated like in sound and video applications. These transfers consist of a single transaction with a token and data stages, the handshake packet is not present.

## 3.7 USB Device Classes

There are several devices which perform similar functions, like button events in a mouse and keyboard, or data transfers to a printer or a network device. For such similar functions, the USB specification defines classes. Classes help to ease development and compatibility, in addition, the operating system can have standard drivers

for each class, this helps a device to operate properly without the requirement of vendor specific drivers. This thesis focuses only on the mass storage device class. Every external USB storage media use this class to communicate with a host.

We discuss the common architecture of a USB device in the mass storage class in Section 3.7.1. For more about USB classes refer to [12, 6].

### 3.7.1 USB Mass Storage Class

The mass storage class defines several configuration values and protocol definitions in order to store or retrieve data inside a persistent medium. This class uses bulk transfers, which we have discussed in Section 3.6. A USB device must support an additional command set in order to send and receive user data, in this case the Small Computer System Interface (SCSI) command set is used [34].

The USB defines a structure to transfer SCSI commands inside a data packet. This structure is called a "Command Block Wrapper" (CBW), the last field of this structure stores the SCSI command for the USB storage device to execute. In order to know if a command was successfully executed, the host request status. The device sends status information in a structure called "Command Status Wrapper" (CSW). Both the CBW and CSW can be consulted in more detail in [5].

## 3.8 USB Storage Architecture

The common architecture for a USB memory device is shown in Figure 3.9. This figure shows all the specific parts any USB storage media should implement.

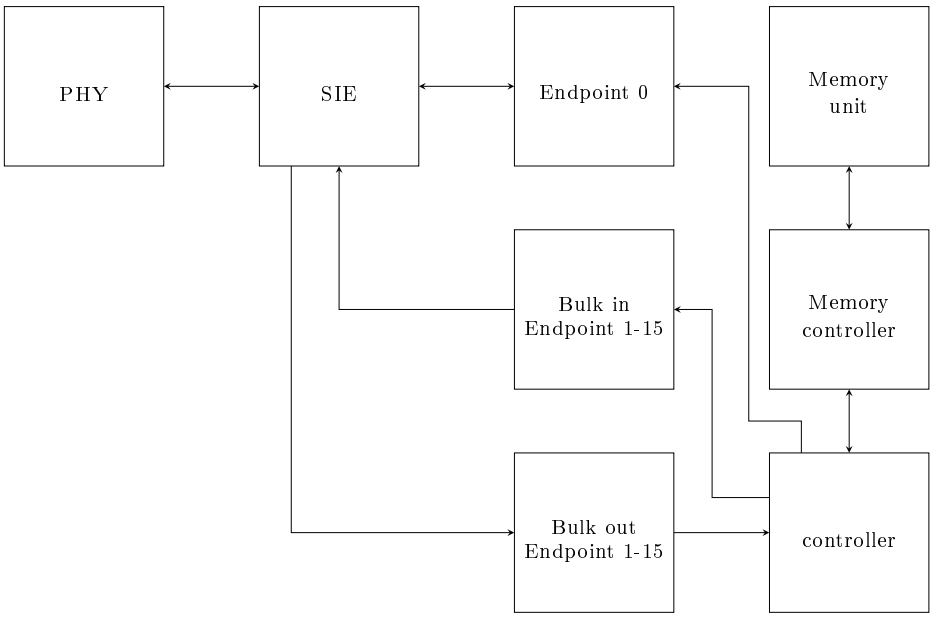


Figure 3.9: A common USB memory architecture.

The mass storage class definition specifies at least the implementation of two additional endpoints (besides endpoint 0). This is needed in order to transfer bulk packets in both directions.

Data arriving to the bulk OUT endpoint is processed by the Device's controller. Once a CBW arrives, the device executes it and then prepares the CSW in the bulk IN endpoint. For SCSI commands, which require data, one of two cases may happen:

- **The host writes data to the USB memory**: After the host has sent the CBW, it sends data packets. The USB memory must send a CSW when all data packets are received.
- **The host read data from the USB memory**: After the host has sent the CBW, the USB memory may send data. The USB memory must send a CSW after all the packets where sent.

# Chapter 4

# Design Overview

In this Chapter we give an overview of the design of MiddleMan. We provide a systematic exploration of the design space and argue why we chose a specific design option.

The diagrams and discussions of this chapter does not carry much details. The various design options discussed here are provided in a much elaborate manner in the following two chapters.

## 4.1 Design Goals and Decisions

The main goal of this work is to protect data inside an USB memory device. In order to achieve this protection, user data must be encrypted. This encryption can be performed with one of the following methods:

- **Host Encryption**: Data is encrypted by software within the Host PC before it is transmitted to the USB memory.
- **Device Encryption**: Data is transmitted to the USB memory, then it is encrypted within the memory.
- **External Encryption**: An encryption unit is attached between the Host and Device. Data is encrypted in this external unit.

These three cases are shown pictorically in Figure 4.1. Each method has its own advantages and drawbacks, which are summarized in Table 4.1. We used two properties in order to compare the three methods. Device compatibility refers to the number of USB memory devices that can be used with a particular method. For instance, a USB with internal data protection can not encrypt other USB memory devices. In contrast, host encryption can encrypt every USB memory device. Finally, Host power refers to the computational power a Host needs to have in order to encrypt. Data rate was not used for comparison, this is because it is highly dependent on the specific implementation rather than the method itself. For instance, a Host encryption method is faster on a high end PC rather than a digital camera or a smart phone.

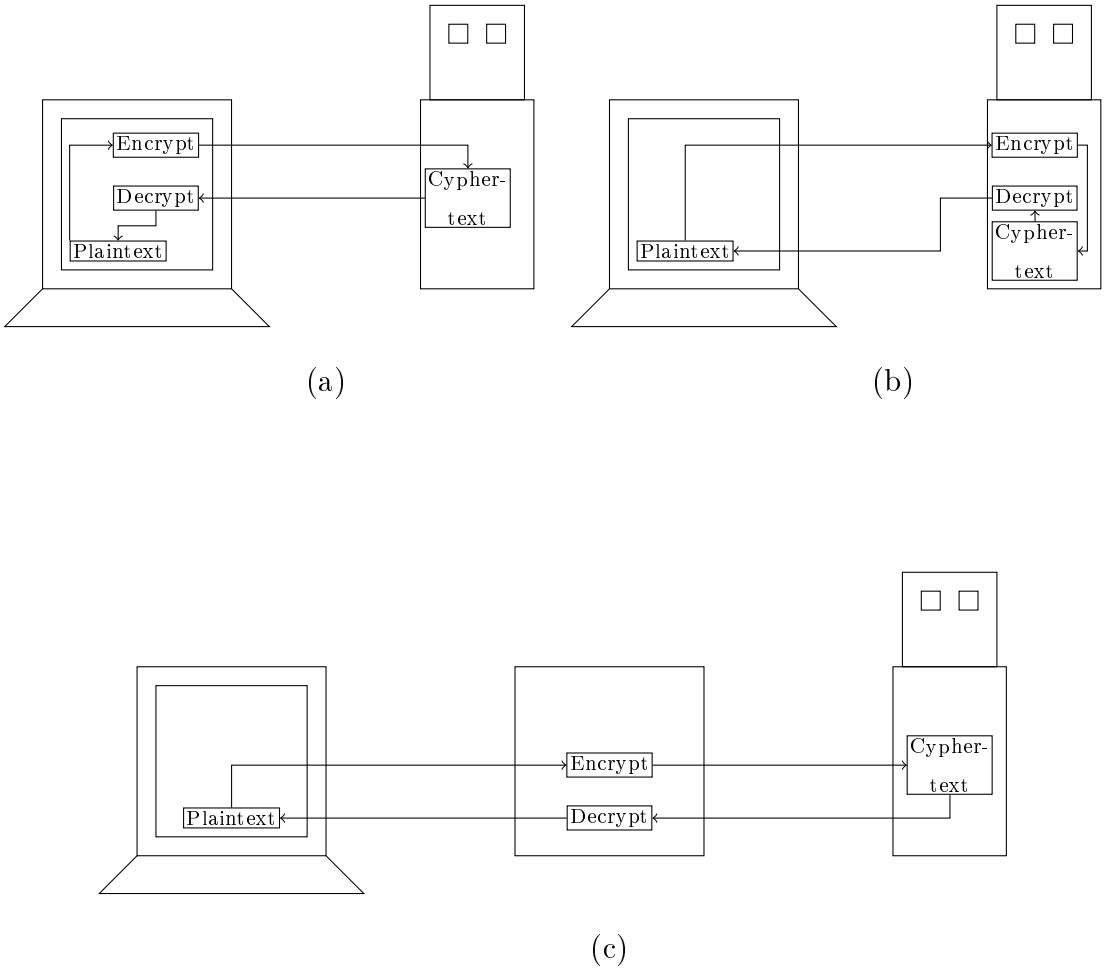


Figure 4.1: USB memory encryption methods: a) Within the PC, b) In the USB controller, c) In an external device.

Table 4.1: Comparison table between encryption methods of Figure 4.1.

| **Method** | **Device compatibility** | **Host power** |
|---|---|---|
| Host encryption | High | High |
| Device encryption | Low | Low |
| External encryption | High | Low |

We can see from Table 4.1 that the best method is the external encryption. Using this method we can achieve the best compatibility with the lowest host power usage. We decided to use this method to implement data encryption, in order to protect the information inside a USB memory device. We called this device the "MiddleMan".

## 4.2 Design of the MiddleMan

In order to design the MiddleMan, we must first understand what happens between a USB Host and Device. In Section 4.2.1 we will discuss the overall data flow of a USB memory device, the entities involved and what they expect to receive while performing USB communications. Then in section 4.2.2, we will introduce the MiddleMan's functionalities. The USB specification defines a device and a host, in this section we refer as Host PC to the USB host which controls the bus. We refer as USB memory to the device attached to the bus. The MiddleMan also implements a USB host controller in order to communicate with the USB memory, we refer to this controller as the USB host controller. Host PC is just a name to avoid confusion, in fact a Host can be any computer, tablet or similar device with at least one USB port available.

### 4.2.1 Functionality of a USB Memory Device

When a Host PC and USB memory are interacting in the bus, there are two types of data being exchanged: *control* data and *user* data. Control data is used to issue commands to the USB memory in order to get information and set a configuration. User data is the actual information that is stored inside a USB memory. Before a Host PC can store user data in a USB memory, it must gather information regarding this particular device. This information contains, among other things, the device's structure. This structure is the most important information inside a USB device. It tells the Host PC how the hardware is organized and what resources are available. Figure 4.2 shows the data flow between a USB Host and a memory device. This data flow takes place in the following manner:

1. The Host PC sends a command or data to the USB memory.
2. The USB memory sends an acknowledge.
3. The Host sends a data request to the USB memory.
4. The USB memory responds with data.
5. The Host sends an acknowledge.

The USB memory can send a negative acknowledge (NAK) to signal the Host PC that it does not have any data ready, NAKs send by the USB memory are omitted from the figure for simplicity.

### 4.2.2 Functionality of the MiddleMan

The MiddleMan must be able to operate without disrupting the USB functionality discussed in the previous section. Figure 4.3 shows the expected USB data flow when the MiddleMan is attached between the Host PC and the USB memory.

The flow of data in this case is as follows:

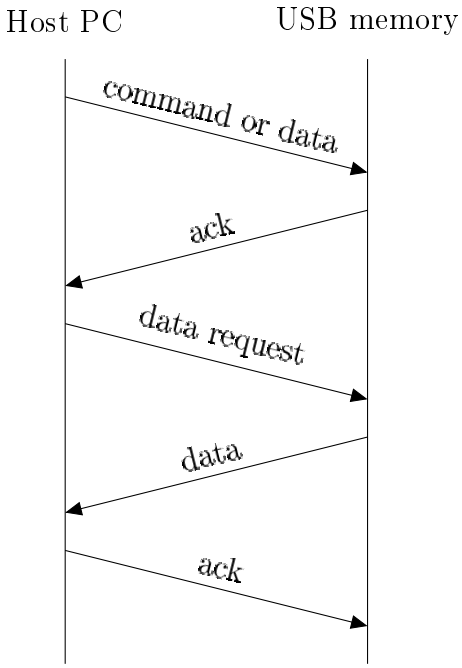


Figure 4.2: USB communications data flow.

1. The Host PC sends a command or data to the MiddleMan, the Host PC thinks it is the real USB memory.

2. The MiddleMan sends an acknowledge to the Host PC and sends the captured command or data towards the USB memory.

3. The USB memory sends an ACK upon reception of the command or data to the MiddleMan, the USB memory sees MiddleMan as if it were the original Host PC.

4. The Host PC sends a data request to the MiddleMan.

5. The MiddleMan responds with a negative acknowledge (NAK) and replicates the data request towards the USB memory.

6. The Host PC will retry the data request until MiddleMan responds with data.

7. Middle man will respond any following data request with NAKs, and will also retry the same data request to the USB memory until the USB memory responds with data, at this point MiddleMan will send this data to the Host PC on the next data request.

8. The host will send an acknowledge.

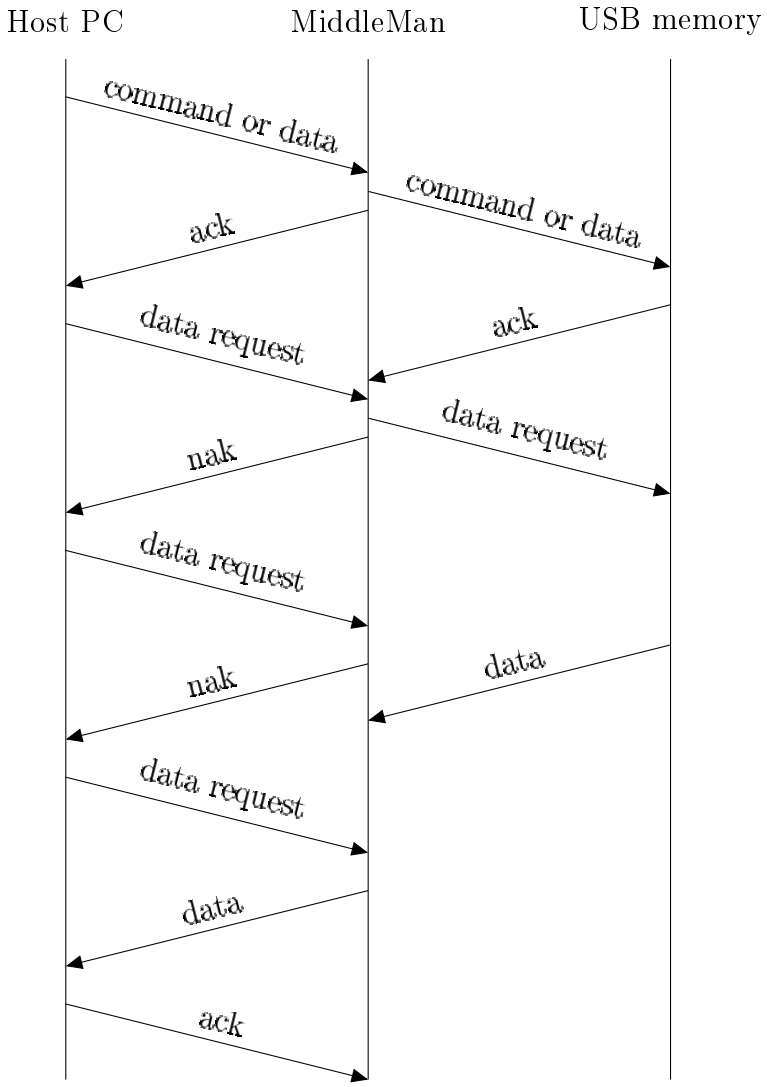


Figure 4.3: USB communications data flow with MiddleMan.

As we can see, the MiddleMan must behave as if, it was not connected to the bus and at the same time, it should be able to analyze and modify the information in transit.

Figure 4.4 shows all the components inside the MiddleMan. These components must be included regardless of the technology used to implement the design. In order to implement MiddleMan, we consider several options, every option uses an FPGA to implement the cryptographic part, along with some external components. Data flow inside MiddleMan's components is as follows:

1. The Host PC sends a command or data to the MiddleMan, USB device controller sends an ACK to the Host PC.

2. The USB host controller sends this command or data to the USB memory.

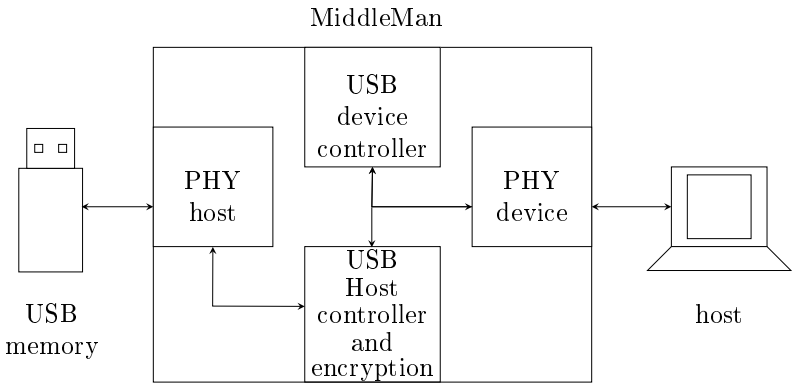


Figure 4.4: Components inside the MiddleMan implementation.

3. The USB memory sends an ACK upon reception of the command or data, which is received by the USB host controller.
4. The Host PC sends a data request to the MiddleMan, USB device controller responds with a NAK.
5. The USB host controller replicates this data request towards the USB memory.
6. The USB memory may respond with a NAK or data.
7. The USB host controller will retry the same data request if a NAK is received. If data is received, the USB host controller will store the data inside a buffer, and then it sends an ACK to the USB memory.
8. The Host PC will retry the data request until MiddleMan responds with data, the USB device controller will check the USB host controller's buffer for data, if there is no data it sends a NAK, it sends data otherwise.
9. The host will send an acknowledge, which is captured by the USB device controller.

### 4.2.3 Design Options for MiddleMan

We considered three options for the design of the MiddleMan. We discuss these three options below:

1. **FPGA only**:

   This option is shown in Figure 4.5. It uses the direct pin connections of the FPGA to drive the USB bus signals. The advantage of this choice is that just a basic external component is needed, also full flexibility is achieved (we can work with the smallest delay possible at low level). The drawback is the limitation in signal quality. This approach was tested and proved to have good reception but it has problems to transmit data back to the host PC.

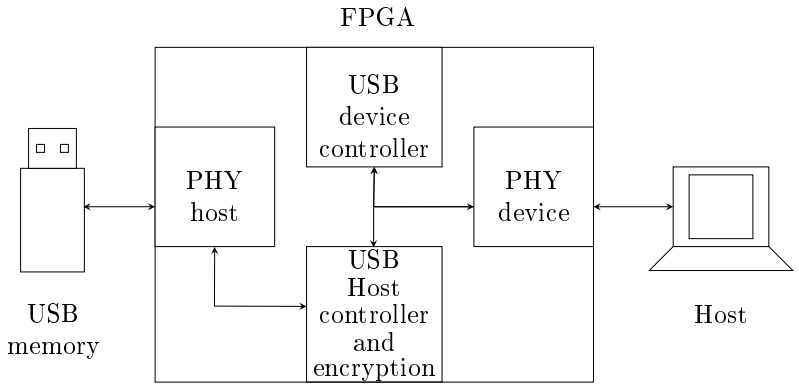


Figure 4.5: FPGA implementation.

2. **FPGA + Microcontroller**:

   This option is shown in Figure 4.6. It uses a USB capable microcontroller in order to handle USB communications and a FPGA to process the data. The microcontroller considered for this option was the PIC18F4550, which is capable of handling USB communications and ease the process by abstracting all low level details of the protocol from the user. The drawback of this option is the added delay to the processing time.

   There are two types of USB ports, one type is used by the host in order to detect devices, the other is used by the device to report speed. The PIC18F4550 has only one device port and therefore, cannot communicate with another USB device. A different microcontroller with host capabilities may also be chosen.

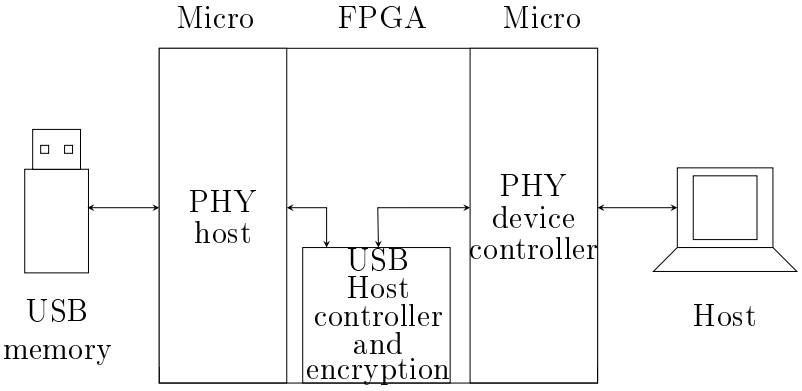


Figure 4.6: FPGA + microcontroller implementation.

3. **FPGA + Physical layer (PHY)**:

   This option is shown in Figure 4.7. It uses an external physical layer chip with an FPGA. The advantage is that we have low level control, the system delay is small and we can configure the physical layer chip to work as a host or device

port. The drawback is that the implementation of the circuit and an external manufacturer must be used in order to produce the board. In contrast with the previous option, the microcontroller board can be fabricated in place.

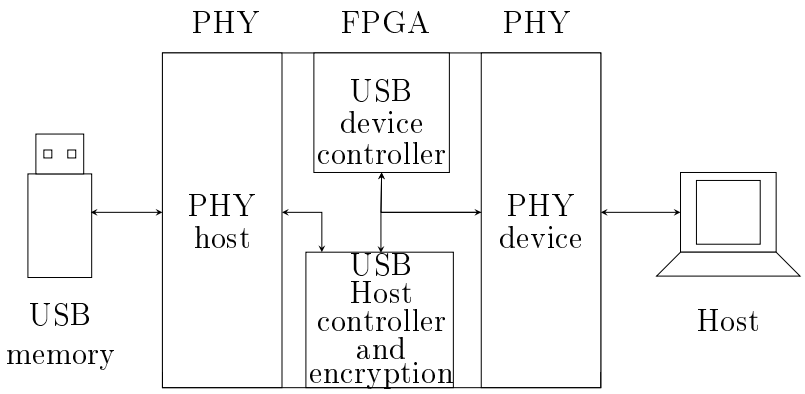


Figure 4.7: FPGA + physical layer implementation.

Table 4.2 shows a comparison between all the options considered for the design of the MiddleMan. In the table, **FPGA Usage** refers to the amount of physical resources used inside the FPGA, **System Delay** refers to the processing time of the MiddleMan, **Signal Reliability** refers to the quality of the signal inside the physical connections of the USB bus and **Flexibility** is the freedom we have to process low level information.

Table 4.2: Performance comparison between design options for the MiddleMan.

| **Option** | **FPGA usage** | **System Delay** | **Signal Reliability** | **Flexibility** |
|---|---|---|---|---|
| (FPGA only) | High | Low | Low | High |
| (FPGA+ Microcontroller) | Low | High | High | Low |
| (FPGA+PHY) | Medium | Low | High | High |

We decided to use the third option (i.e. FPGA+PHY), the reason is that this option provides the lowest system delay and the highest flexibility without sacrificing signal reliability. Choosing this option means we need an additional circuit implementing the physical layer. We built an external circuit and attached this as an extension to the Spartan 3E board. We call this additional circuit as S3USB.

## 4.3 Physical Layer Implementation

This part of the design implements communications between a USB host and device, using the Spartan 3E board as a router device. Communications between the physical layer and the FPGA board are performed through a 100 pin external port. The

Physical layer (PHY) communicates using an 8 bit data bus plus 5 bits for control. Our implementation uses two physical layer blocks for a total of 26 connection lines. Figure 4.8 shows the basic structure of all the components. The host physical layer needs an additional power source for an attached device, this power is taken from the FPGA board. The current provided to the device is controlled by a current limiter integrated circuit (IC) inside the host PHY block. This current IC protects the system from any device, which could consume too much current. We implemented the external PHY block by designing the S3USB board, according to the characteristics we have just discussed.

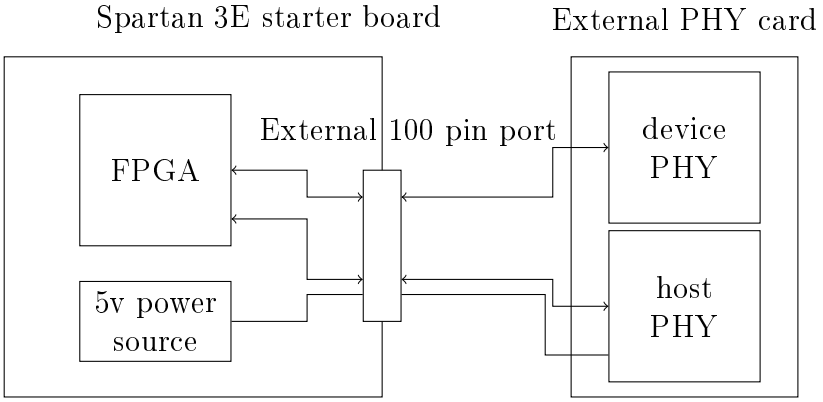


Figure 4.8: FPGA + physical layer external card.

The physical layer uses a standard protocol for the bus called "USB2.0 Transceiver Macrocell Interface extension, Low Pin Interface" (ULPI) [38]. We implement this protocol in the FPGA in order to send and receive data to the physical layer.

## 4.4 USB Protocol Implementation

For MiddleMan the whole USB protocol needs to be implemented. Some USB protocol implementations exists in the literature [27, 36]. These cores where not used for the design of the USB controller of the MiddleMan. The first reference [27], was designed to be used with a Cypress SX2 USB controller [15]. This controller is only capable of functioning as a USB device, this means that we can not communicate with a USB memory using this controller. The second reference [36] is a more generic design, which is intended to interface with devices using UTMI, a standard protocol for USB controller communications [39]. We didn't use this core as it has several components that are not necessary for the implementation of the MiddleMan, also the UTMI interface, although compatible with ULPI, adds more delay to the transmission of data when interacting with an ULPI bus. The implementation of the MiddleMan requires communication with a USB host and a USB memory device, while keeping the lowest delay possible. For these reasons, the implementations in both references were not used.

The physical layer explained in the previous section, deals only with raw data, which is transmitted or received on the bus. It provides every packet on the bus as it is, the designer must implement a way to handle the low level USB protocol, from filtering USB packets, to acknowledge for any packet received without errors.

The USB protocol is implemented inside the FPGA. Figure 4.9 shows the location of the block in the data processing path. To handle this protocol, we need a device and a host block. The device block handles communications with the main host PC. The host block handles communications with the USB memory device, and acts similarly to the main Host PC. Implementing a host is not an easy task, it requires control over many aspects of the USB specification, like power management and bus time division and administration. The host also detects newly attached devices, configures them and checks whether they are high speed capable. All of these are performed by a host in addition to perform the tasks of the low level USB protocol. In order for a host to communicate with a USB memory device, it must be compatible with Small Computer System Interface (SCSI) commands [34], and use a Command Block Wrapper (CBW) packet, discussed in Chapter 3.7.1 in page 25, to carry these commands through the bus. Implementing a full host inside a FPGA is impractical for this project as we have limited space. Instead of implementing a full featured host, we implemented only the basic functions. Bus time managing was omitted (as we only work with one USB memory at a time). The SCSI protocol required is implemented, but instead of generating the protocol, we can just use the packets received from the Host PC and re-transmit them to the USB device. These two simplifications save design and debug time, and reduces the space and complexity of a host controller. We name the resulting component as "Packet Replicator". This Packet Replicator receives and transmits data from every endpoint and controls them.

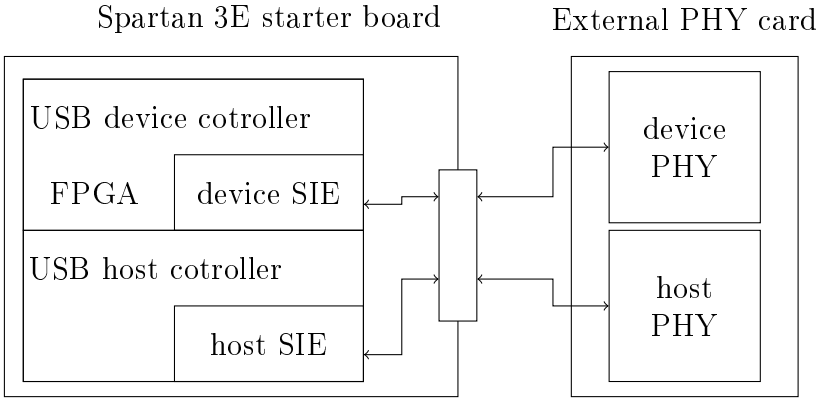


Figure 4.9: Location of the protocol implementation.

For MiddleMan to provide transparent communications in the bus, the device block executes all the commands addressed to the USB memory device and ultimately, it becomes a copy of the USB memory attached to it. At this point, the USB memory and the MiddleMan have the same USB address, name, logical units, storage size, etc.

The host PC can only see and communicate with the MiddleMan but it thinks it's communicating with the USB memory attached at the other side of the MiddleMan.

The USB Packet Replicator acts as as a regular USB endpoint, but instead of processing all the packets, it just transmits them to their destination. This process is described as follows:

1. The host PC transmits a packet.
2. The Packet Replicator receives it, if it is a regular packet then it is transmitted to the USB memory.
3. If the USB memory should respond to the packet, then the Packet Replicator waits for a response.
4. The Packet replicator tells the USB device controller that there is data available, and is sent to the host PC upon request.

This process should not take much time, and must work at low level. This is why we chose a FPGA to implement the design. The Packet Replicator inside the FPGA is capable of identifying all USB packets, and determine whether they contain a command or user data.

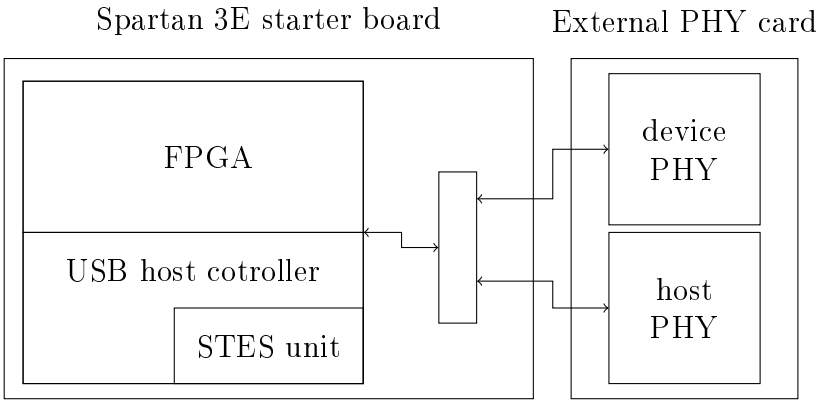


Figure 4.10: Location of the encryption implementation.

## 4.5 USB Data Encryption

This part is implemented inside the host controller as shown in Figure 4.10. the purpose of this unit is to encrypt user data only, allowing any other type of data to be retransmitted without any modifications. It encrypts data using the hardware implementation of STES described in Chapter 2 (in page 10). To encrypt data, STES needs a key and a destination sector. The key is stored in a register inside the host controller, it can be input using the FPGA buttons and switches, or by implementing

custom USB commands. The sector value is taken from the CBW. The system only encrypts or decrypts data after a SCSI write or read command is received. The data flow for encryption can be summarized as follows.

1. The Packet Replicator receives a write or read command, sector and data length are stored.

2. The command is replicated to the USB memory device.

3. The next packet is treated as user data and encrypted (or decrypted). The same is done for any other following packets until the Packet Replicator has received all data packets.

During the encryption of each sector of data, only the host PC waits for the proper response, the USB memory just receives or sends data upon request from the Packet Replicator. In both cases the host PC and the USB memory device are not aware of the Packet Replicator's presence.

Key management was not completely implemented at this point. We use a fixed key for the encryption process, various options for key management would be discussed in Chapter 8 (in page 89).

## 4.6 USB Storage Architecture Comparison

Section 3.8 (in page 25) discusses the common architecture for any USB storage device. In this section we will show how we implemented this architecture inside the MiddleMan's components shown in Figure 4.4.

Figure 4.11, shows the common architecture. This figure highlights the component locations mentioned in this chapter. The device controller component and the Host controller and encryption component share part of the common architecture's controller. The USB memory section can be replaced by any storage device. The common architecture uses a memory controller and a memory unit, in contrast, the proposed architecture uses a USB host port (host PHY included) to attach a second USB device.

The proposed architecture can be easily modified to encrypt other types of media that can be replaced in the USB memory component shown in the figure, from SD memory cards to external hard drives.

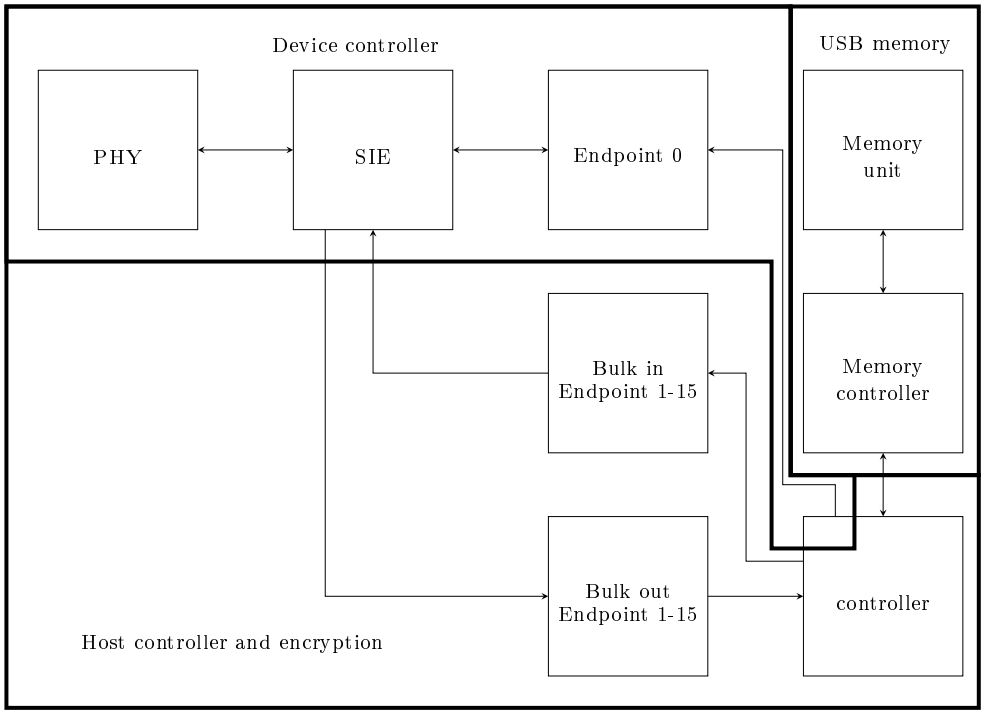


Figure 4.11: Common USB architecture with MiddleMan's block division.

# Chapter 5

# Hardware Design

The design of the MiddleMan requires two USB physical layers connected to an FPGA. In this Chapter we will discuss the design of an external device which effectively adds USB communications to an FPGA. We named this device S3USB.

The FPGA board we have chosen is the Spartan 3E starter board. We will only refer to it as FPGA through the rest of the chapter. This board features a 100 pin connector, which we used to connect the S3USB device.

The chapter organization is as follows: In Section 5.1 we will talk about all the design considerations for the development of S3USB. In Section 5.2 we discuss the connectivity between the FPGA and the S3USB board. In Section 5.3 we explain S3USB's functionality and how we designed the circuit. Finally, in Section 5.6, we will talk about the features and limitations of S3USB.

## 5.1 Design Considerations

Some FPGA development boards provide basic USB communications, which allow simple input peripherals to interact, others even have basic data transmission capabilities. The aforementioned options are not optimal for data storage applications like ours for the following reasons:

- They usually require a driver and/or software.
- Most of them use the available USB ports for configuration or basic I/O.
- Common FPGA boards provide only USB device ports. They cannot be used to implement a USB host controller.
- A common method to add USB capabilities to a device is by adding a dedicated USB microcontroller. Most FPGA boards, which have a USB port, perform USB communications through a microcontroller. The firmware inside these microcontrollers can not be changed easily.

Because of the limitations in existing FPGA boards we decided to build an USB board as an expansion to a FPGA. In this extended board we implement the MiddleMan USB communications. Thus we are following the FPGA+PHY option as discussed in Chapter 4 (in page 27). The board extension (which we call S3USB) is used to implement the physical layer.

There are some design considerations to be taken into account for a proper design of the FPGA+PHY approach, and the S3USB. We discuss them next:

- **Connectivity**: The Spartan 3E board provides several ports, which can be used to connect external devices. There are simple pin connectors, which have the advantage of easy interface prototyping. The drawback of these connectors is that they have low pin count and don't support high speed signaling. The board also provides a high speed connector with a 100 pin interface.

- **Power**: The Spartan 3E board provides 3.3V and 5V power sources for external logic. The S3USB board needs at least a 5 Volts to drive one USB port. Some components of the S3USB may use a 3.3 Volts source. We must be sure that the FPGA board can source the S3USB properly.

- **Compatibility**: The S3USB must be able to communicate with a USB host or device. It must be able to meet MiddleMan's requirements.

- **Working frequency**: S3USB must operate at a working frequency, which is compatible with the FPGA board. Higher frequencies may put some synchronization problems due to the distance from the FPGA to the connector where S3USB will be attached.

- **Routing**: Careful routing considerations must be taken to reduce signal noise. Routing deals with a proper placement of the copper tracks in a Printed Circuit Board.

- **Clocking**: Clocking the S3USB can be done by providing a clock signal from the FPGA or including independent crystal oscillators. FPGA clock sourcing reduces the number of components inside the S3USB but may generate noise in the clock signal. Crystal source uses more components but the clock signal may be more reliable.

We decided to attach our circuit in the 100 pin expansion port of the Spartan 3E starter board. Event though the connector is not easily available, this port was designed specifically for implementations with the characteristics of the S3USB board, as we will discuss next.

## 5.2 Spartan 3E Connectivity

The Spartan 3E starter board has several external connectors, which can be used to interface the FPGA with an external circuit. The best option for this project is the

external 100 pin port of the Spartan 3E board [40, p. 113], we will refer to this port as FX2. Most of the pins in this port are tied to ground to support high speed signaling. The FPGA provides two voltage sources to the FX2: a 3.3V and a 5V source. The former can be used to power the components, while the latter may be used as the source to drive the USB port of the S3USB.

Another important thing to consider for the design of S3USB is the clock source and the clock routing. We describe next two different possibilities of clock sources:

1. **FPGA as a clock source**:

   The FX2 connector provides some pins for clock input and output to the FPGA. These clock pins can be used to source a clock signal to S3USB. We can use the FPGA as a clock source but this should be done carefully. A clock feedback would be necessary for proper synchronization. This clocking approach reduces the external component count at the cost of low working frequency and design difficulty. Figure 5.1 shows a block diagram for this approach. If this approach is well implemented, we can achieve full synchronization of the FPGA and the S3USB board.

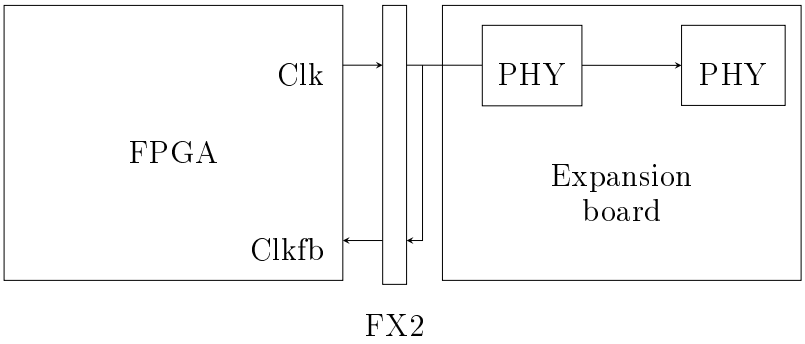


Figure 5.1: FPGA clock sourcing.

   FPGA clock sourcing requires a careful routing of the clock signal, also they must be properly terminated. Termination is a method to reduce noise by adding a circuit at the end. This prevents the signal to be reflected to the signal's source. Reflection adds to much noise to the clock inputs of the circuit components. Some techniques for proper termination can be found in [26, 14].

2. **Crystal oscillator source**:

   Another clock sourcing method, shown in Figure 5.2, provides a crystal oscillator for each clock input. This approach uses more components, particularly, one crystal, one resistor and two load capacitors [16]. This approach offers better clock stability for external components, in exchange of extra space, also we will end up working with an asynchronous system. We decided to use this method because calculations and tests are easier than designing a proper FPGA clock source. If the FPGA clock source design was bad, we must redesign the whole

PCB, wasting time and money. If the crystal oscillator fails, we can just change the crystal or faulty components.

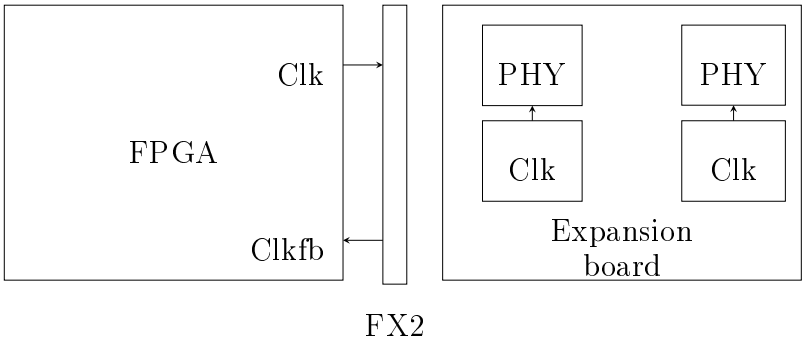


Figure 5.2: External clock sourcing.

The distance between the FPGA and the S3USB board must be considered. The optimal placement for any interconnected component is right next to each other, however, there are situations where this is not possible. This is the case of expansion boards, if the distance between the components can not be changed, then we must be careful in the routing designs to avoid synchronization errors.

## 5.3 Design of the S3USB Expansion Board

The final circuit schematic for the S3USB is shown in Figure 5.3. We will use the reference labels of this schematic to explain the functionality of each component through the rest of this chapter. Some components can have an associated number next to the reference label. This number is the actual part we used in the final design.

### 5.3.1 S3USB circuit functionality

The S3USB circuit in Figure 5.3, is based upon two USB physical layer components referred to as U2 and U3. These components are implemented with the USB3300 IC [35]. They capture USB serial data from ports J2 and J3, and buffers it to a digital bus connected to port J1. As discussed in Chapter 4, the MiddleMan implementation requires two physical layers to communicate with a USB memory and a Host PC. Components U2 and U3 are designed to be configured as a USB host or device and thus, they successfully implement the communication requirements of the MiddleMan.

Communications between the FPGA and components U2 and U3 are performed through port J1. Connections for each component are summarized in Table 5.1. This table shows pin to pin connections between the FPGA, connector J1 and components U2 and U3. Each component communicates through a parallel bus, which uses a protocol called ULPI. We will discuss this protocol in Chapter 6. The control pins of the bus are:

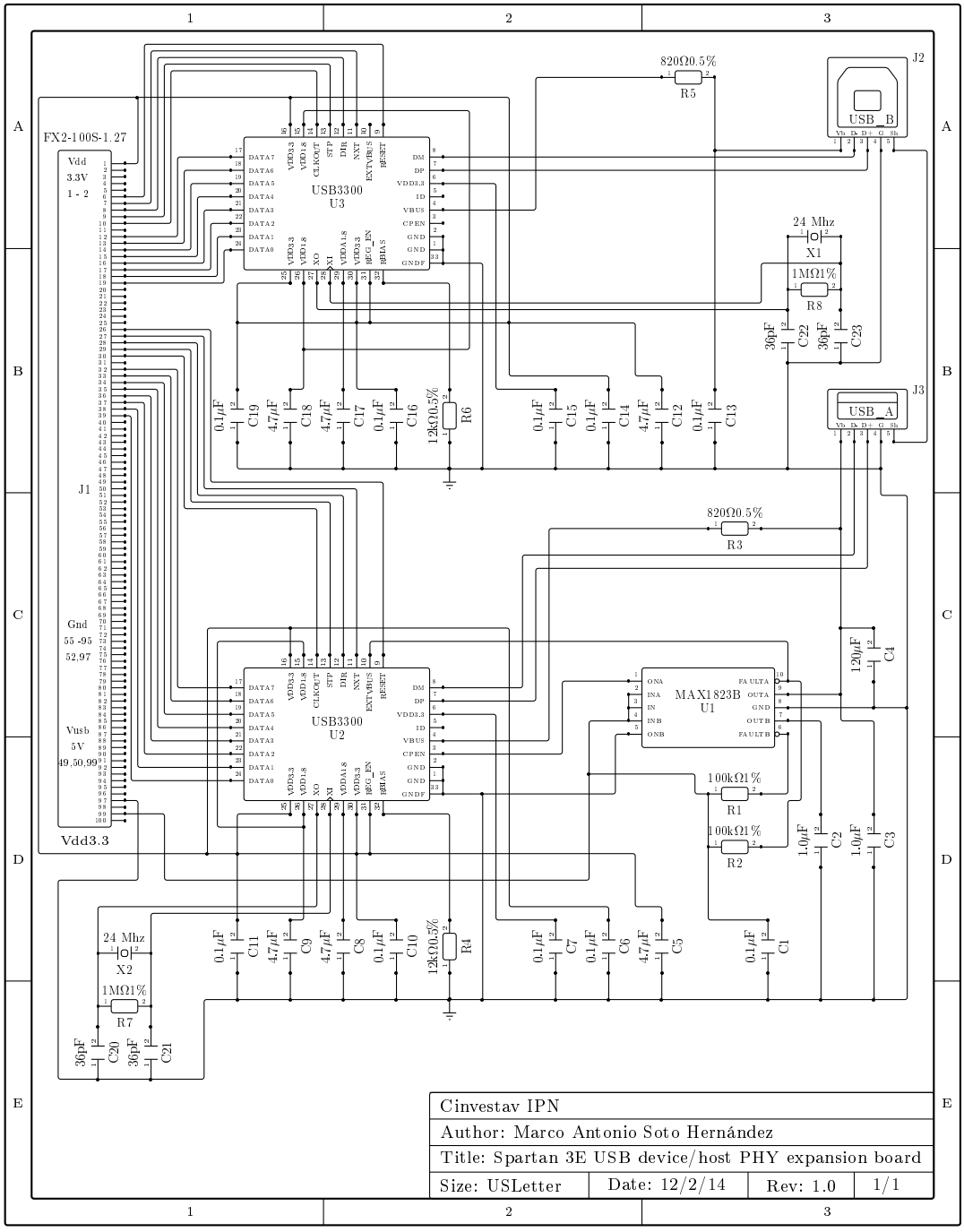


Figure 5.3: PHY schematic.

- **RESET**: This line reset the internal logic of units U2 and U3.
- **NXT**: This is the data throttle line. The FPGA must assert this line to insert the current value of the DATA port.
- **DIR**: This line is controlled by U2 and U3. It signals the current ownership of the bus: '0' FPGA, '1' U2 or U3.
- **STP**: Stop signal, the FPGA asserts STP when there is no more data to be inserted.
- **CLKOUT**: This is the clock source from which the bus is synchronized. It runs at 60 Mhz.

The S3USB board is divided in two blocks: the host and device block. The device block consist of components U3, J2, X1 and all the associated capacitors and resistors on coordinates A and B on Figure 5.3. This block was designed to work in a USB device mode. For this mode, the circuit only requires a USB port to communicate with a Host PC. The host block on the other hand, was designed to work in a USB host mode. It consists of components U1, U2, J3, X2 and all the associated capacitors and resistors on coordinates C through E.

Host mode is slightly more complex than device mode. It needs to provide a port to connect an external USB device and provide a suitable 5V source. The J3 port is sourced by the FPGA board through components U1 and C4. Capacitor C4 keeps the voltage source stable when a USB device is attached on port J3. Component U1 is an important IC, is used by U2 in order to manage bus power. Component U1 is a current limiter switch, if a device attached to J3 draws too much power, U1 turns off the power source in order to protect the internal components. The bus power will remain off until the device is removed. U2 can also enable or disable bus power by controlling "ONA" port of component U1. Voltage status on J3 is reported to U2 through component U1's "FAULTA" port, this status can be read by the FPGA. The part we choose to implement U1 is the MAX1823B [24], for U2 and U3 the part number is USB3300 [35].

## 5.4 Circuit Design

The S3USB circuit on Figure 5.3 was designed based on the application notes for components U1,U2 and U3. These application notes can be found in [35, 24].

The application notes contain the recommended configurations and values for most passive components in the S3USB schematic. The only components that where calculated are the capacitors associated with X1 and X2, Figure 5.4 gives a close up of these components.

Capacitors on Figure 5.4 are called "load capacitors". These capacitors help the crystal oscillator components X1 and X2 to start properly. Load capacitor values are defined by the oscillator's manufacturer and can be found on the device's data sheet.

Table 5.1: J1 connector pin-out.

| **Block** | **FPGA** | **FX2** | **U2/U3** |
|---|---|---|---|
| USB Device | B4 | 6 | RESET |
| | A4 | 7 | NXT |
| | D5 | 8 | DIR |
| | C5 | 9 | STP |
| | A6 | 10 | CLKOUT |
| | E7 | 12 | DATA(7) |
| | F7 | 13 | DATA(6) |
| | D7 | 14 | DATA(5) |
| | C7 | 15 | DATA(4) |
| | F8 | 16 | DATA(3) |
| | E8 | 17 | DATA(2) |
| | F9 | 18 | DATA(1) |
| | E9 | 19 | DATA(0) |
| USB Host | A13 | 26 | RESET |
| | B13 | 27 | NXT |
| | A14 | 28 | DIR |
| | B14 | 29 | STP |
| | C14 | 30 | CLKOUT |
| | A16 | 32 | DATA(7) |
| | B16 | 33 | DATA(6) |
| | E13 | 34 | DATA(5) |
| | C4 | 35 | DATA(4) |
| | B11 | 36 | DATA(3) |
| | A11 | 37 | DATA(2) |
| | A8 | 38 | DATA(1) |
| | G9 | 39 | DATA(0) |

Table 5.2: USB3300 bus interface signals.

| **Signal** | **Direction** | **Description** |
|---|---|---|
| CLKOUT | out | USB3300 reference clock. |
| NXT | out | Data throttle. |
| DIR | out | Bus direction controller by PHY. |
| STP | in | Stop signal. |
| DATA | io | Bidirectional data bus. |

For this design we used the oscillator found in [23]. The data sheet specifies a load capacitance of 20 pF for a proper functioning of the oscillator, however, we could not use this value directly on the circuit, we needed to account for the connection and parasitic capacitance. We will show how we include this extra capacitance next.

Table 5.3 summarizes all the parameters needed in order to calculate the compo-

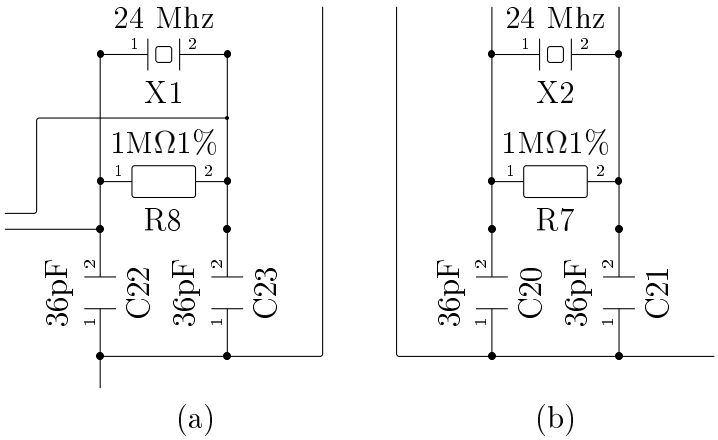


Figure 5.4: Close up of the oscillator networks of Figure 5.3. a) Device block oscillator, b) Host block oscillator. Pins 1 and 2 of components X1 and X2 connect directly to pins XO and XI of components U2 and U3.

nents of Figure 5.4. Reference [16] give us the following formula for the calculation:

$$C_L = \frac{C_{L1} \times C_{L2}}{C_{L1} + C_{L2}} + C_{stray}, \tag{5.1}$$

where $C_L$ is the total load capacitance specified in [23], $C_{L1}$ and $C_{L2}$ are the capacitors we need to calculate and $C_{stray}$ is an additional capacitance calculated as:

$$C_{stray} = C_i + C_p \tag{5.2}$$

$C_i$ is the total capacitance added from U3's XO and Xi pins and $C_p$ is the total parasitic capacitance generated by the connection traces. For this case, equation 5.1 is simplified by choosing $C_{L1} = C_{L2}$. Solving Equation 5.1 for $C_{L1}$ and substituting $C_{stray}$ from 5.2 yields:

$$C_{L1} = 2(C_L - (C_i + C_p)). \tag{5.3}$$

We use this formula to calculate capacitors C20, C21, C22 and C23. The capacitance value for $C_i$ is specified in U3's data sheet [35]. $C_p$ is calculated based on a type of transmission line called microstrip, which can be seen on Figure 5.5. It consist of a conductor line and a ground plane separated by a dielectric insulator. Parameters in the Table 5.3 were calculated using the formulas provided in [26, p. 430]. To calculate $C_p$ we use this formula:

$$\text{CMSTRIP} = \frac{\text{PMSTRIP}}{\text{ZMSTRIP}} \times x \tag{5.4}$$

Where PMSTRIP is the propagation delay measured in pico seconds, $ZMSTRIP$ is the characteristic impedance of the microstrip in Ohms and $CMSTRIP$ is the trace capacitance for one length unit. $C_p$ is obtained by multiplying $CMSTRIP$ times the length of the connection between X1 and U3. Parameters in Table 5.3 are organized

in order of dependency. To calculate the Effective Relative Permittivity (EEFF) and the Effective electrical trace width (WE); we need $\epsilon_r$, $h$, $w$ and $t$. To calculate the propagation delay (PMSTRIP) we need EEFF and so on.

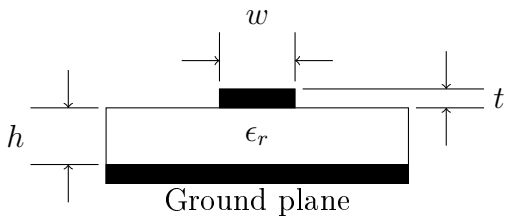


Figure 5.5: Micro strip parameters diagram.

Table 5.3: Parameters calculated for the capacitors of Figure 5.4.

| Parameter | Value | Units | Description |
|---|---|---|---|
| $C_L$ | 20 | pF | crystal's load capacitance |
| $\epsilon_r$ | 4.6 | n/a | Dielectric's relative permittivity. |
| $h$ | 0.03125 | in | Distance from trace to ground. |
| $w$ | 0.01 | in | Trace width. |
| $t$ | 0.00134 | in | Trace thickness. |
| $x$ | 0.4894 | in | Trace length. |
| EEFF | 10.317776118 | n/a | Effective relative permittivity. |
| WE | 0.0129542576 | in | Effective electrical trace width. |
| PMSTRIP | 0.272131613788334 | ns | Propagation delay |
| ZMSTRIP | 55.2912009238 | Ω | Microstrip characteristic impedance. |
| $C_p$ | 2.4087 | pF | Microstrip trace capacitance |
| $C_i$ | 3 | pF | USB3300 Xi,Xo capacitance. |
| $C_{stray}$ | 5.408723441758 | pF | $C_p + C_i$ capacitance. |
| $C_{L1}$ | 29.18255311648 | pF | Capacitance for components in Figure 5.4 |

## 5.5 PCB Design

The S3USB circuit was built in a two layer PCB. The front layer shown in Figure 5.6 is used to route signals, while the back layer shown in Figure 5.7 is used for the ground plane and power source connections.

There are three problems regarding the design of a PCB. The first and most important is the distance between interconnected components. This distance may cause synchronization issues. The reason for this is that at long distances, electric signals may arrive at different times [26, pp. 7-8]. To ensure proper synchronization

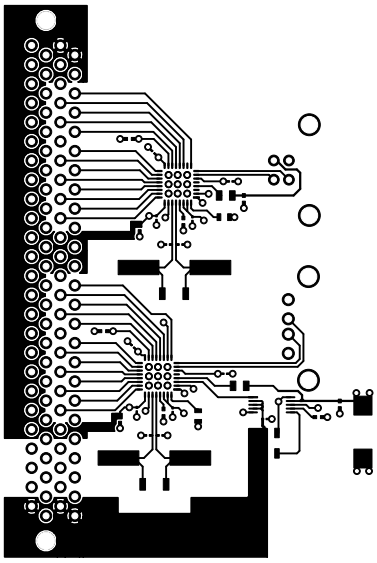

Figure 5.6: S3USB Front PCB layer (signal layer).

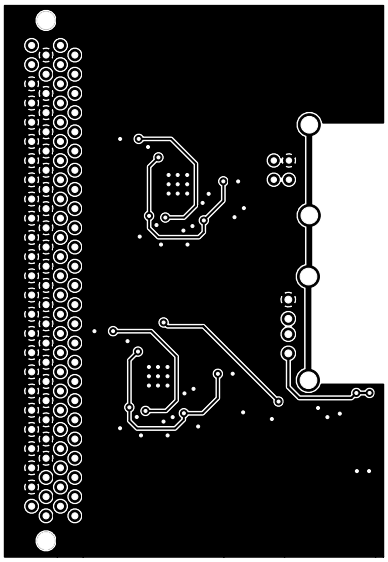

Figure 5.7: S3USB back PCB layer (ground and power layer).

on long distances, all signal traces must have similar lengths. The FPGA board provides similar trace lengths on port J1, we routed the circuit components as close as possible to this connector, this way we ensured similar length traces. We can compare the final component placement in Figures 5.3, 5.6 and 5.8.

The second problem is the signal routing of U2 and U3. These components provide an eight bit bus plus control lines. Having a bus means having several signal traces running in parallel to each other. Parallel traces can cause unwanted noise which may cause data errors. Noise is reduced by implementing a ground plane below the

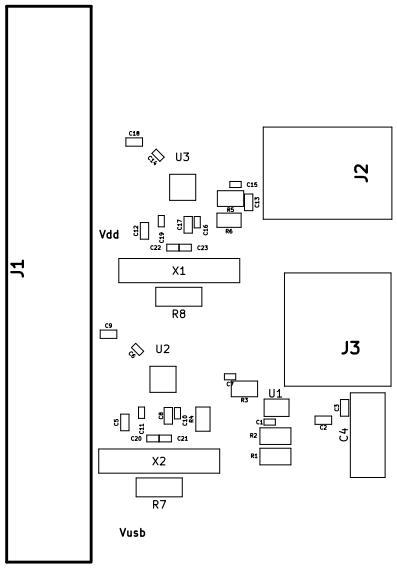


Figure 5.8: S3USB silk screen layer (component labels).

bus (shown in Figure 5.7), and by adding a proper distance between each signal trace (shown in Figure 5.6).

The last problem is the proper routing of power traces. Power traces must have low impedance, this means that they need to be wide. For this particular design, routing power lines was difficult, mainly because the power pins where obstructed by connector J1. Difficulty lies in the distance between each pin hole in Figure 5.6. PCB manufacturers have a minimum distance tolerance between traces. Putting power lines in between these holes can increase PCB's manufacture costs.

## 5.6 Some Characteristics of the S3USB Expansion Board

The S3USB expansion board shown in Figure 5.9, is a device intended to add two full featured USB 2.0 ports. This board was designed to address the lack of some features needed to implement a USB memory encryption device in a Spartan 3E board. The S3USB board has the following features:

- **Speed support**: USB 2.0 low, full and high speed support.
- **Unpowered type B port**: It can be configured as a device or host using an external power supply.
- **Powered type A port**: It can be configured as device or host with bus power supply and overload protection. This port can provide up to 700 mA.
- **Design flexibility and speed**: The USB protocol can be implemented inside the FPGA, allowing full control and fast responses.

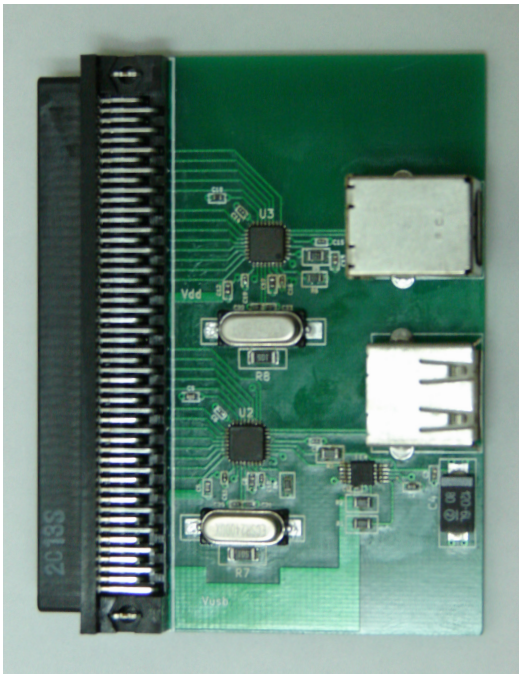


Figure 5.9: The S3USB expansion board.

The board's limitation is the maximum power supply for external devices. It is designed to provide up to 700 mA. This power source fully complies with the USB specification. The limitation lies in the fact that some devices actually draw more than this limit. The most obvious examples are the external USB hard drive devices.

Though we used the S3USB for a prototypical implementation of MiddleMan, it can be used to implement several other devices and thus, can be of independent interest. Some other possible uses of the board are explained in Chapter 8.

# Chapter 6

# Reconfigurable USB Architecture

In this chapter we discuss the part including the FPGA implementation of the USB memory encryption device. In Section 6.1 we present an overview of the part, which we implemented in an FPGA, and explain the general functionality of each unit in the implementation. In the subsequent sections we discuss the details of each implementation unit along which the specific block and state diagrams.

## 6.1 Architecture Overview

The MiddleMan architecture is divided in three layers. We already discussed the physical layer implementation in Chapter 5, and introduced the S3USB board, which we used for implementing the physical layer. The other two layers are the link and the Protocol layer, which are shown in Figure 6.1

The Link and Protocol layers are implemented inside the FPGA. We call this unit the "USB Packet Replication Core" (UPRC). The Link layer inside UPRC communicates directly to the S3USB with a standard protocol, which we will explain later in Section 6.2.1. The protocol layer receives USB data from the Link. This layer takes care of the low level communications in the USB. The UPRC block with its main components are shown in Figure 6.1.

Data flow within the MiddleMan is either from the host to the device or from the device to the host. We will explain these two cases separately:

- **Host to device**: In a host to device communication, the host first issues a token packet. This packet is then received by MiddleMan's physical layer. Upon reception of the token packet, the physical layer issues a reception command to the link layer. The link receives the data and forwards it to the protocol layer. On receiving data, the protocol layer sends an acknowledge to the host PC, and then it analyzes the token packet and decides if it is to be filtered or send to the device.

  Upon receiving the acknowledge from the protocol layer the host sends a data packet. The same protocol is followed for the data packet as in the case of the

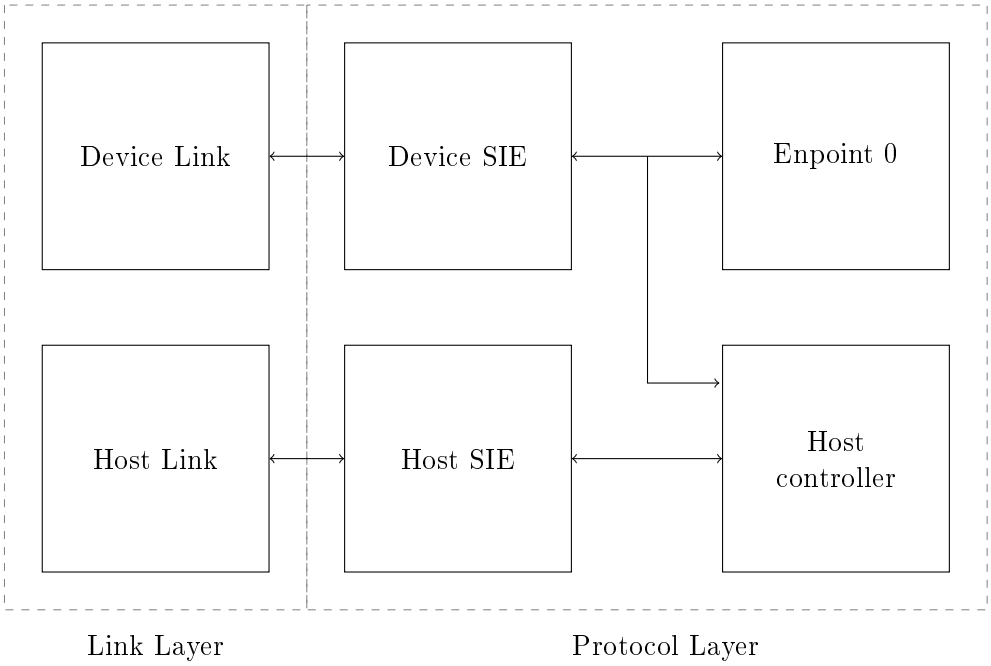


Figure 6.1: Block diagram of the UPRC architecture.

token packet for all layers except the protocol layer. Here the packet is analyzed to detect user data. If the packet contains user data, then the data inside the USB packet is encrypted and then sent to the device.

- **Device to host**: For a device to host communication, the process begins with the host sending a token to request data to the device. This token is processed in all layers in the same way as the previous case except for the protocol layer. When the token arrives at the protocol layer, it is sent to the USB device. At this point the protocol layer responds to the host with a negative acknowledge (NAK). This response tells the host to retry. The protocol layer will respond with a NAK to every successive token the host sends until the USB device responds. When this happens, the protocol layer will acknowledge the reception of the data packet from the USB device. This packet is also analyzed to determine if it is user data and decrypt it. When the host retries, the protocol layer will respond with the data packet.

The UPRC block is responsible for implementing the dataflow we explained above. Figure 6.19 shows the full implementation of the block diagram presented in Figure 6.1. Now we will discuss the overall functionality of the components in Figure 6.1 and it's corresponding schematic figure:

- **Host and device link**: These two devices communicate directly with the physical layer. They receive and transmit data from and to the S3USB board. Figure 6.19 shows the interface between their corresponding physical layers (shown as

bold text in the figure), and the connection with the components of the protocol layer. The host and device link are basically the same circuit, their details are shown in Figure 6.20.

- **Device and host Serial Interface Engine (SIE)**: These two hardware components communicate with their corresponding link components. They perform the USB protocol at the lowest level. These two components are implemented similarly. The details of the implementation are shown in Figure 6.21.

- **Endpoint 0**: This block consists of components descriptor ram and ENP0 shown in Figure 6.19. The descriptor ram stores the USB information regarding the MiddleMan. Endpoint 0 executes the most important device requests sent by the host PC, in order to configure the MiddleMan. It takes the data in the descriptor ram when the host PC requests a descriptor. In our implementation, descriptor data requested to Endpoint 0 is not used by the device SIE, descriptor data is taken from the USB memory instead. The descriptor ram was left in order to implement an alternative mode of operation in the case the MiddleMan does not have a USB memory attached,though this functionality was not used. The implementation details of the Endpoint 0 is shown in Figure 6.22.

- **Host controller**: This component interacts with both the device and host SIEs. It receives packets from both sides, and performs data encryption. The full architecture is shown in Figure 6.23. The crypto unit inside this figure is the main cryptographic core. It is based on the STES scheme proposed in [10]. The architecture of the crypto unit is separately shown in Figure 6.24.

## 6.2 The Link Layer

The link layer connects directly to the S3USB board through the 100 pin connector provided by the FPGA. There are two components inside this layer:

- **Device Link**: This component communicates with the S3USB section, which interacts with the host PC.

- **Host Link**: This component communicates with the USB device.

To understand the Link layer implementation, we must first introduce the communications bus that is used by the physical and link layers. After that, we present the implemented architecture.

### 6.2.1 The ULPI bus

The physical and link layer communicate using a bus called ULPI [38]. This is a standard bus, which uses eight bits for data transfer and five bits for control. We

Table 6.1: The ULPI bus signals.

| | |
|---|---|
| **RESET** | Asynchronous reset |
| **NXT** | Data throttle, next signal |
| **DIR** | Bus ownership |
| **STP** | Stop signal |
| **CLKOUT** | Bus clock signal |
| **DATA** | 8 bit bidirectional data bus |

Table 6.2: S3USB Register space only some registers where omitted.

| | Address (6 bit) | | | |
|---|---|---|---|---|
| **Register name** | Read | Write | Set | Clear |
| Vendor ID low | 00h | - | - | - |
| Vendor ID high | 01h | - | - | - |
| Product ID low | 02h | - | - | - |
| Product ID high | 03h | - | - | - |
| Function control | 04h-06h | 04h | 05h | 06h |
| Interface control | 07h-09h | 07h | 08h | 09h |
| OTG Control | 0Ah-0Ch | 0Ah | 0Bh | 0Ch |

have discussed briefly the bus signals on Chapter 5, in page 41. In Table 6.1 we list the ULPI bus signals.

The ULPI specification not only defines the protocol and signals of the bus, it also defines how to implement a physical layer (PHY). Every physical layer compliant with the ULPI specification must have a set of registers. These registers store information about the device and are also used for configuration.

The physical layer integrated circuit used for the S3USB board implements the registers shown in Table 6.2. These registers can be accessed for read, write, set and clear operations. Operations are performed by addressing the register, i.e._ we can write to the Function Control register by writing the desired value to the address 04h. We will explain the purpose of the most important registers only.

- **Function Control**: This register stores the physical layer's configuration regarding the USB device functionality. The speed of the USB device (low, full and high), and USB suspend are configured here. The USB suspend is a feature that must be implemented by every device and allows to enter a low power consumption state when the USB bus is not active.

- **OTG control**: This register is used to configure the features of the physical layer when working in host mode. It configures the resistors used to detect low and full speed devices and enables bus power on the S3USB.

In order to read or write to a register inside the PHY, the ULPI protocol defines two types of commands: The TX and RX commands. TX commands are sent by

the link layer while the RX commands are sent by the PHY. A TX command is an eight bit string value, where bits 7 to 6 carry the specific command for the PHY to execute and bits 5 to 0 carry the command parameters. Table 6.3 summarizes all the commands.

Table 6.3: TX commands

| **Command name** | **Command (bits 7-6)** | **Parameters** | **Description** |
|---|---|---|---|
| Idle | 00b | 000000b | ULPI bus idle |
| Transmit | 01b | 000000b | USB transmit with no PID |
| | | 00XXXX | USB transmit, X is the PID |
| Register Write | 10b | XXXXXX | Register write,<br>X is the address |
| Register Read | 11b | XXXXXX | Register read, X is the address<br>X is the address |

Now we explain the ULPI commands in details. The waveforms for the TX commands are:

- **TX commands**: The waveforms for the different TX commands are shown in Figure 6.2. There are three different TX commands, which are explained as follows:

  1. **Transmit command**: As the name implies, it sends a USB packet to the PHY for transmission over the USB bus. The command is put into the DATA lines and the PHY asserts NXT when the command or data is accepted. The link layer must put the next value into the bus the cycle after NXT is asserted. When the link does not have more data to send, it must assert STP.
  2. **Write command**: This command writes a value inside any writable register on the PHY. First the link layer must put the write command into the bus. The address bits must contain a valid write address from Table 6.2. When the PHY accepts the command it asserts NXT, the link must put the write value on the next cycle, after this, the link must assert STP.
  3. **Read command**: The read command gets the value of any register inside the PHY. First the link must put the read command and register address into the bus. When the PHY accepts the command it asserts NXT, then DIR is asserted and the PHY takes control of the bus. The link waits one cycle, the PHY will put the valid register value on the following cycle.

- **RX commands**:

  RX commands are send by the host every time the PHY asserts the DIR signal. The link layer cannot send any TX commands while DIR is asserted. When

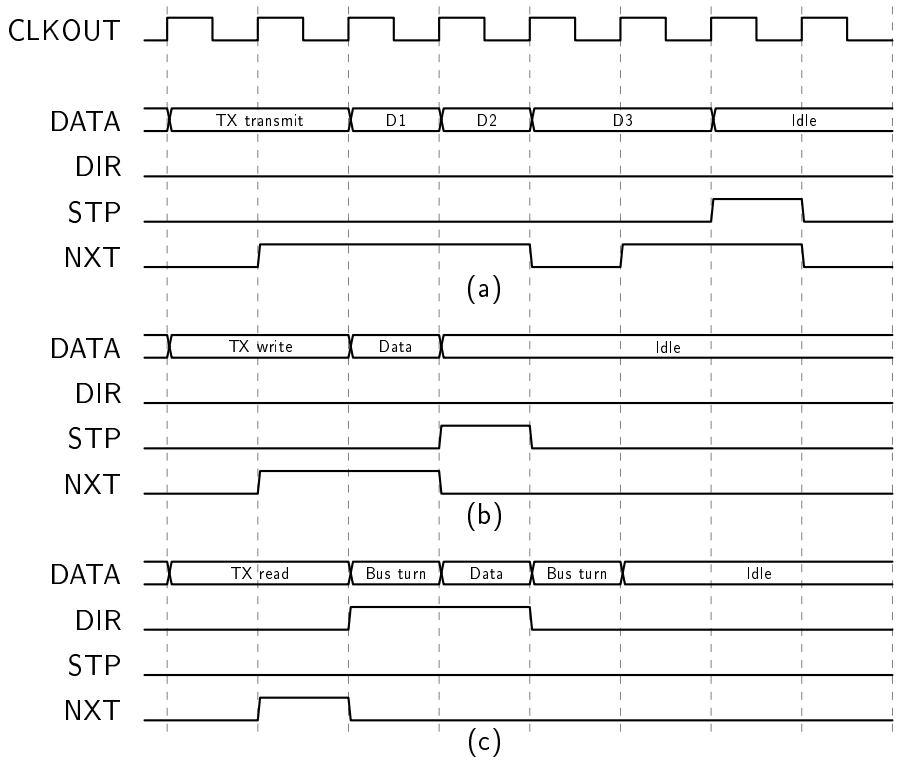


Figure 6.2: TX commands. a) Transmit command, b) Write command and c)Read command.

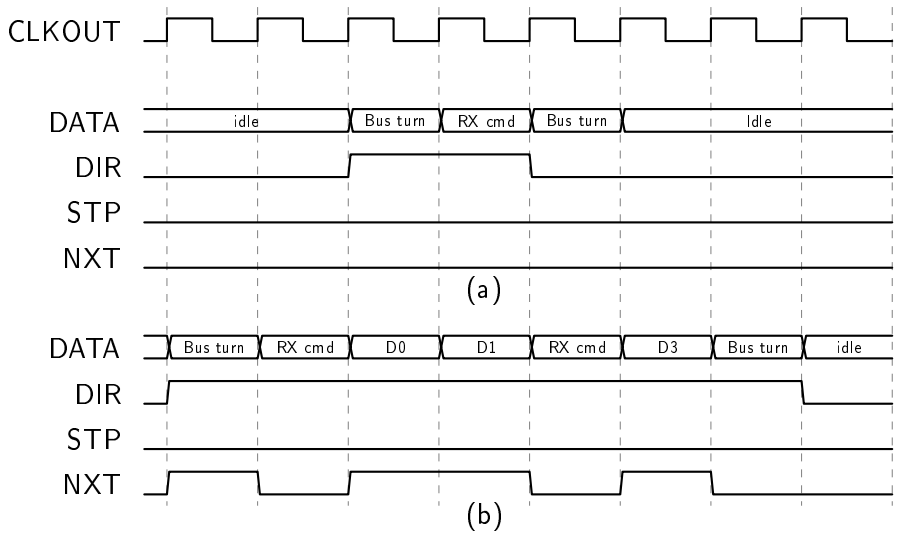


Figure 6.3: RX commands. a) RX command with the PHY status, b) RX command for USB packet receptions. RX commands are received between USB data to update the PHY status.

The RX command data carries the PHY status which is stored inside the link layer. Figure 6.3 shows the waveform for a RX command.

### 6.2.2 The Link Architecture

The link component interacts directly with the physical layer. It is connected directly to the S3USB board through the Spartan 3E expansion port. It consists of two FIFOs used for I/O, a command generation unit and a control unit as shown in Figure 6.4 (the details of the same figure are shown in Figure 6.20). The I/O FIFOs have two different clocks domains. The ULPI domain uses the PHY clock signal (CLKOUT) to synchronize the input bus lines, while the FPGA domain uses the clock generated inside the FPGA. The Link component can configure each USB3300 component inside the S3USB unit as a host or device port.

To transmit a TX command, the link layer's control unit uses the command generation block. This block is implemented as an array of multiplexers, which select the proper command or data to be inserted into the output FIFO. Data extraction from this FIFO is controlled by NXT DIR and STP signals. A more detailed link layer schematic can be seen on Figure 6.20(in page 75).

To receive RX commands, the ULPI bus state is extracted from the Input FIFO on each cycle. The control unit monitors the extracted value in order to detect USB packet receptions or PHY status updates. The bus state register stores the value of the last Rx command received. The Rx command contains the state of several parameters of the PHY like USB D+ and D- status or device attached to the port. RX commands are always sent to the link layer after a TX command or when the

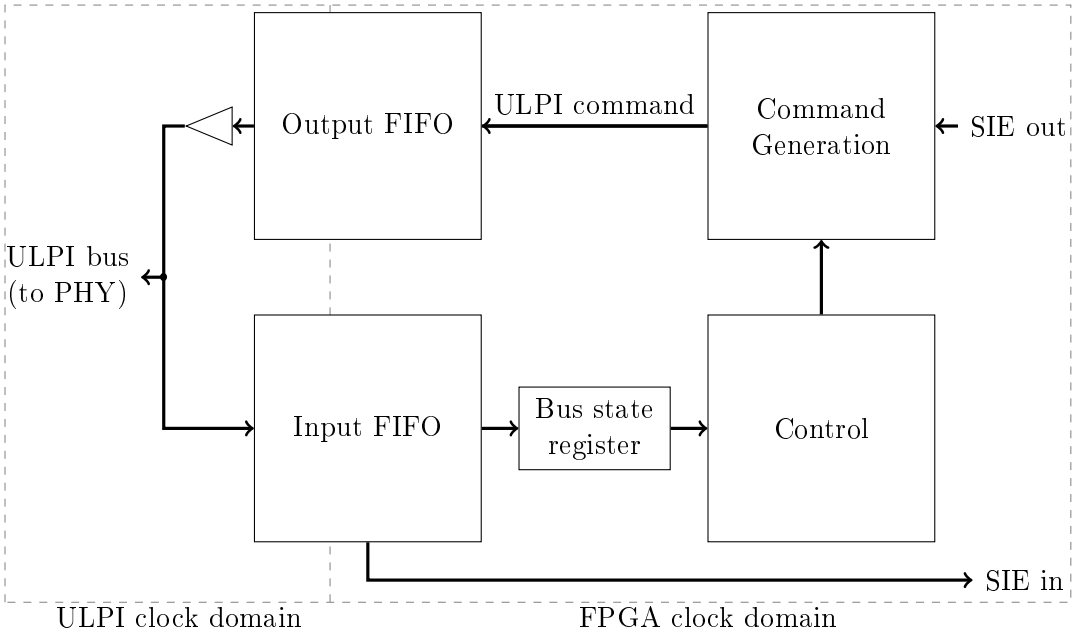


Figure 6.4: Block diagram of the ULPI link layer.

status of the PHY changes.

To keep the Link Layer interface simple, some ULPI features supported by the USB3300 unit [35] inside the S3USB where not implemented. These features are the suspend mode and interrupts. These features are not necessary for a basic implementation.

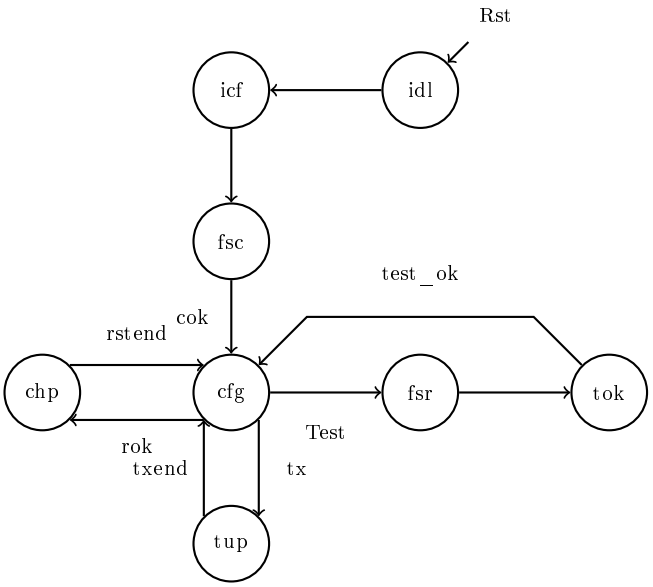


Figure 6.5: Link Layer state diagram.

Figure 6.5 shows the main Finite State Machine (FSM) for the Link Layer. The description of the states is as follows:

- **idl**: This is the idle default state.
- **icf**: This state initializes the configuration procedure for the S3USB.
- **fsc**: This state writes the configuration values to the register inside the S3USB.
- **cfg**: S3USB configured state.
- **fsr**: Reads configuration values from the S3USB to check it was properly configured.
- **tok**: This state compared the read values and ends the test procedure.
- **tup**: Performs a USB transmission.
- **chp**: USB bus reset protocol state. In device mode this state detects the reset and performs the protocol to enable high speed. In host mode this state controls the reset state on the bus and initiates the protocol to enable high speed. This protocol is called the "Chirp protocol". The USB specification [12] discuss this protocol in more detail.

The FSM of Figure 6.5 performs several TX write and read commands. These commands are implemented by two additional FSM. The TX write FSM of Figure 6.6 has the following states:

- **idl**: The default idle state.
- **txw**: Insert the TX write command.
- **txd**: Insert the TX command data. If the commands does not have data then this state is skipped.
- **stp**: Insert the TX command stop signal.
- **txi**: Insert the ULPI bus idle state.

This FSM is used by states **fsc**,**chp** and **tup** of Figure 6.5. The TX read command in Figure 6.7 varies slightly, the states for this command are the following:

- **idl**: The default idle state:
- **txr**: Insert the TX read command.
- **txe**: TX read command end. This states just adds a cycle to wait for turn around.
- **txi**: Insert the ULPI bus idle state.
- **txw**: Wait for the register data.

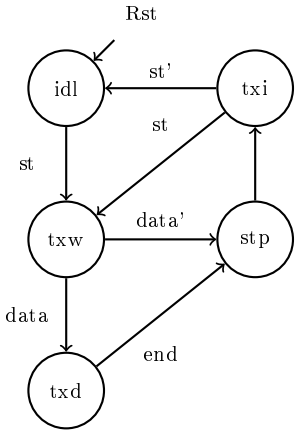


Figure 6.6: TX write command FSM.

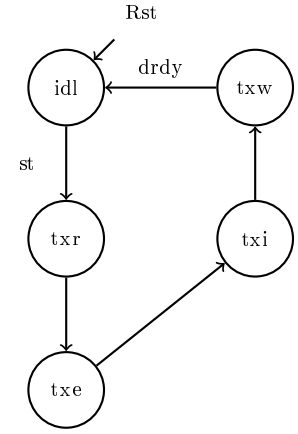


Figure 6.7: TX read command FSM.

## 6.3 The Protocol Layer

This is the top layer inside the MiddleMan. This layer implements the USB protocol and replicates every packet sent from the host or device. It consists of four units shown in Figure 6.1, which are described as follows:

- **Device SIE**: This unit implements the USB protocol for a USB device. Checks the Status from the Endpoint 0 and Host controller in order to send acknowledgments and data to the host PC.
- **Host SIE**: This unit implements the USB protocol for a USB host. It communicates with an external USB device.
- **Endpoint 0**: This endpoint is invisible to the host PC. It is included in order to configure the same address as the USB device attached at the host link.

- **Host controller**: This unit receives all the packets from the device SIE and replicates this information towards the USB device attached at the host link. This unit is also performs encryption of user data.

Data and control signals within the protocol layer can be seen in more detail on Figure 6.19. We will now discuss each unit's architecture.

### 6.3.1 Serial Interface Engine

Figure 6.8 shows the block diagram for the architecture of the SIE. This architecture is divided in two blocks, reception and transmission. Data is registered from both sides, the SIE acts as a pipeline between the link and protocol layer.

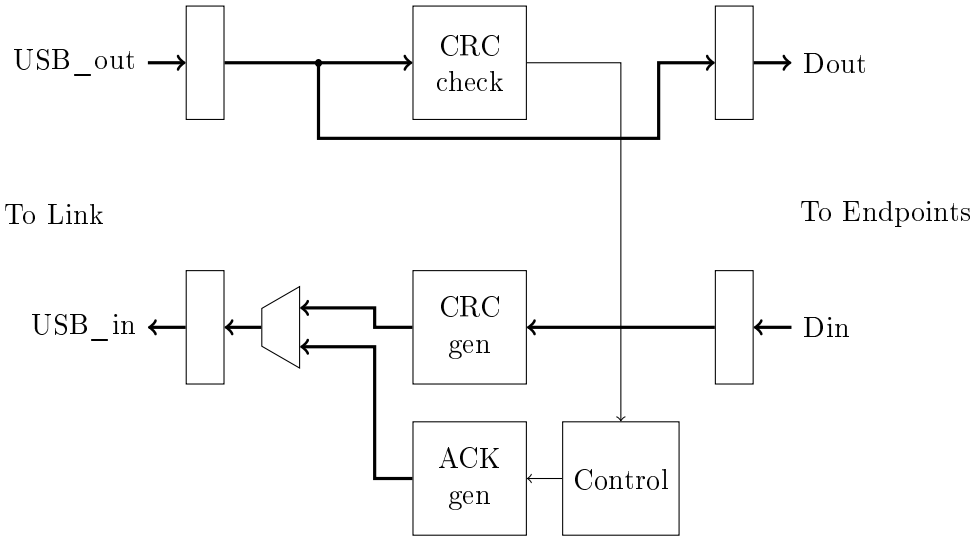


Figure 6.8: Serial Interface Engine architecture.

The transmission block checks the Host controller status. When the host PC requests data, the SIE will respond based on the host controller status in three ways:

- If the host controller has data available, then the SIE will send this data host controller.

- If the host controller is empty, the the SIE will respond to the host controller with a negative acknowledge, this is generated inside the ACK gen block.

- If the host controller has the stall flag asserted, then the SIE will send a stall packet to the host PC, this is also generated at the ACK block.

The SIE automatically calculates CRC codes when receiving and transmitting data. This is done inside the CRC gen and check blocks. The USB CRC generation

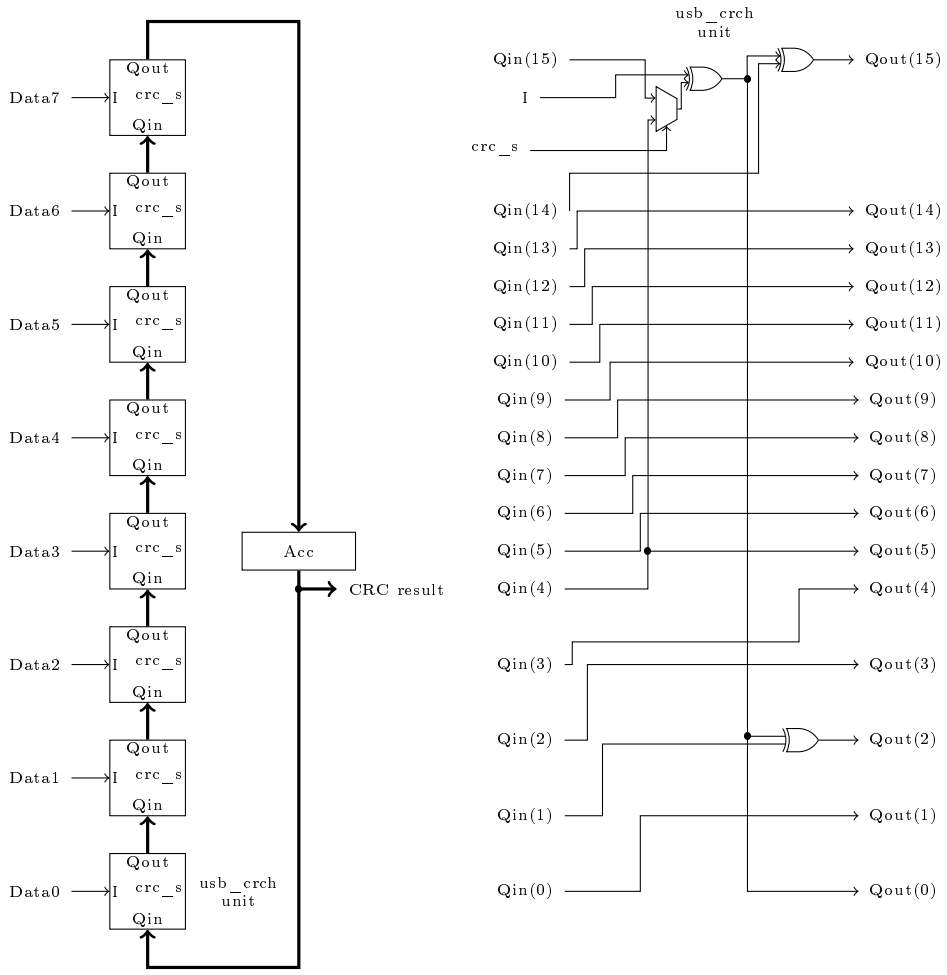


Figure 6.9: CRC Generation architecture.

algorithm defined in [12], process its input bit-wise, in contrast, the device SIE process it's data byte-wise. To overcome this issue, the SIE parallelize this process by calculating eight steps per cycle. The resulting architecture is shown in Figure 6.9.

Figure 6.10, shows the FSM for the SIE's control unit. This FSM detects valid token packets from the USB bus and responds accordingly for IN or OUT transactions. The states defined for the SIE are as follows:

- **idl**: This is the default idle state. The FSM changes state when the SIE receives data.
- **tck**: This state receives the second byte in a token packet, if there is no more data, the FSM returns to **idl**.
- **tls**: The last token data is received in this state. If there is no more data it the FSM returns to **idl**.
- **tkn**: Valid token check. IN, OUT or invalid tokens are checked in this state.

- **wer**: When there are packet errors, this state will wait until there is no more data to receive. Received data is discarded.
- **sof**: This state filters Start of frame packets or any other data, which is not detected as a token packet.
- **wdt**: This state waits for the host PC data for an OUT transaction. if the data received is not a data packet the FSM detects the error and goes to **wer**.
- **dot**: This state waits for the rest of the received data packet.
- **trn**: This state waits for the USB turn around.
- **sak**: This state sends the acknowledge to the host PC.
- **itn**: This state handles the bus turn around for USB IN transactions.
- **idt**: The SIE extracts data from the host controller and transmit it to the host PC. if the host controller does not have data, then the SIE sends a negative acknowledge or a stall packet.
- **wak**: This state waits for the acknowledge from the host after sending a data packet.

For the Host SIE unit, the FSM is similar. The token states **tck**, **tls** and **tkn** are used to extract data from the host controller. These states are also used to generate start of frame tokens on intervals of 1 ms. The host SIE then detects an IN or OUT token. Transitions of the **tkn** state in Figure 6.10 are swapped for IN and OUT conditions. The Enp_empty condition and the **wer** state are removed. The host SIE architecture is similar to the device SIE's architecture in Figure 6.8. It has small differences that can be seen in the schematic of Figure 6.21.

### 6.3.2 Endpoint 0

The purpose of the Endpoint 0 (ENP0) is to respond to control transfers from the Host PC in order to configure the USB block inside UPRC unit. Each control transfer carries a device request, which must be processed by this endpoint. Device requests needed for a successful configuration a listed in Table 6.4.

Table 6.4: USB required Device requests.

| Request | Description |
|---|---|
| Get_Descriptor | Retrieves information for the USB device. |
| Get_Configuration | Retrieves the current USB device configuration. |
| Set_Configuration | Sets a proper configuration for the USB device. |
| Set_address | Sets an address for the USB device. |

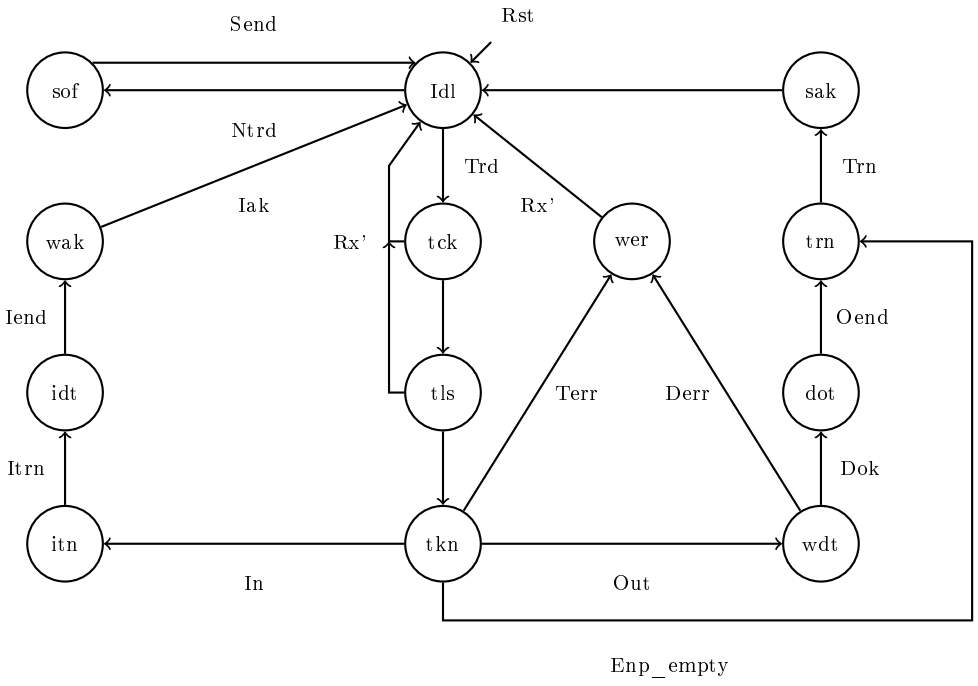


Figure 6.10: Serial Interface Engine state diagram.

This component implements both ENP0 IN and OUT (Figure 6.11). ENP0 OUT receives and process device requests. Received data is stored inside a packet buffer, then the control unit checks this data for a valid request and executes it. If the device request requires data to be sent to the host, the control inserts data inside ENP0 IN FIFO. Although the Endpoint 0 is a full featured endpoint, the UPRC only uses it to store the USB address into the register found at coordinates (3,B) of Figure 6.19. The complete architecture can be seen in Figure 6.22.

Figure 6.12 shows the FSM used to respond to control transfers. The states are the following:

- **idl**: The default idle state, the FSM transitions when a setup token is received.
- **dec**: Decode state. It wait for the Data packet that follows the setup token in order to decode the device request. On error the FSM returns to **idl**.
- **sst**: This state attends device requests that need data inserted into the ENP0 Fifo in block.
- **wst**: This state wait for the host to send the status packet to end a device request. if for some reason the hosts request more data, the FSM transitions to **zlp**.

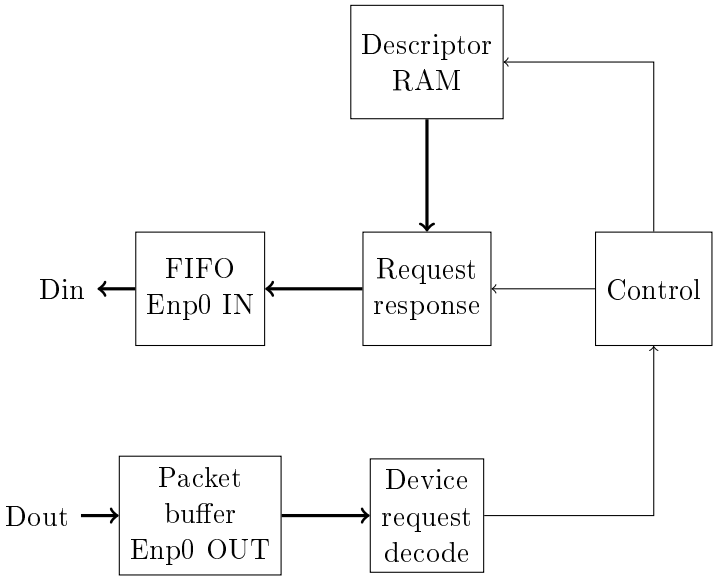


Figure 6.11: Endpoint 0 architecture.

- **zlp**: This state inserts a zero length packet inside FIFO IN, this tells the host that there is no more data to be sent. a zero length packet only has a data PID followed by the CRC bits.

- **sdt**: This state attends device requests, which don't require data to be sent to the host PC.

- **wak**: This state will wait for the device SIE to send an acknowledge to the host.

If a new setup token is received while the FSM is in states **wak** or **wst**, the FSM will immediately decode the next device request.

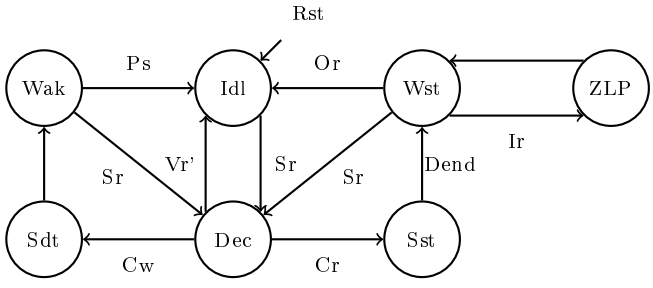


Figure 6.12: Endpoint 0 state diagram.

### 6.3.3 Host controller

Implementing a fully featured host controller is difficult and impractical for the protocol layer we need. To reduce development time and FPGA resources, we have done the following simplifications:

- Replicate packets from the host PC: A regular host controller generates token packets and in the case of USB memories, it also generates SCSI commands. Our host controller can reuse packets sent from the host PC and replicate them.
- Bus time management: A regular host controller manages bus time to attend several USB devices attached in the bus. Out host controller will only interact with a USB memory device, so there is no need to manage the bus.

The final architecture for the host controller is a USB packet replicator, Figure 6.23 shows the full schematic. This component receives all packets from the host PC and transmit them to a USB device. The device SIE sees the host controller as an Endpoint, which is accessible at every address (i.e. the host controller responds to Endpoints 0 through 15).

Figure 6.13 shows the block diagram for the host controller. Each block's function is as follows:

- **Token buffer**: This is the main data input from the device SIE. It inserts all user data inside the FIFO out or crypto unit. Its called Token buffer because the system analyzes token packets inside this buffer before insertion.
- **SCSI buffer**: This buffer captures SCSI commands from the host PC. These commands contain information, which is used by the crypto unit.
- **FIFO in**: When a USB device communicating with the host SIE sends data, it is inserted here. Every user data is decrypted first.
- **FIFO out**: All data going to the USB device must be inserted inside this FIFO.
- **Crypto unit**: All user data is processed by this block. It encrypts or decrypts data following a SCSI command.
- **Input buffer**: This buffer is used to filter the CRC portion of every packet received from the USB device.

The FSM in Figure 6.14 controls the USB packet replication process. The states are the following:

- **idl**: The default idle state. The FSM changes state when a valid token is detected in the token buffer block.
- **out**: OUT token received state. This token is inserted in the FIFO out block. when the insertion is complete, the FSM waits for the data packet.

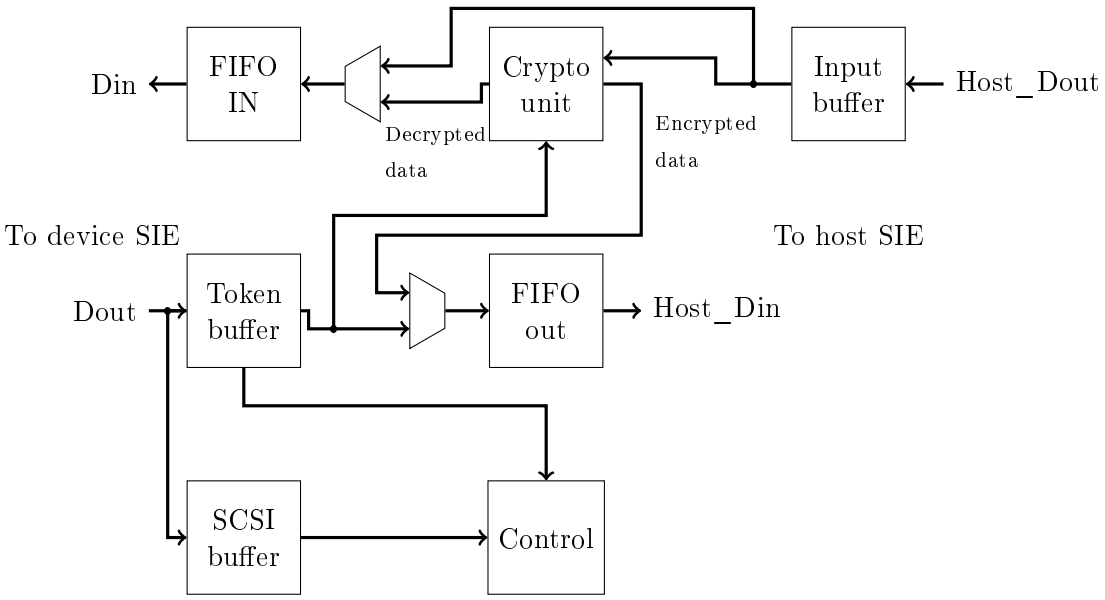


Figure 6.13: Host controller architecture.

- **wd**: Wait data state. The following data packet is inserted in the FIFO out block.
- **od**: This state waits for the token buffer to get empty.
- **in**: IN token received state. The token is inserted in the FIFO out block and then the FSM waits for the data response from the USB device.

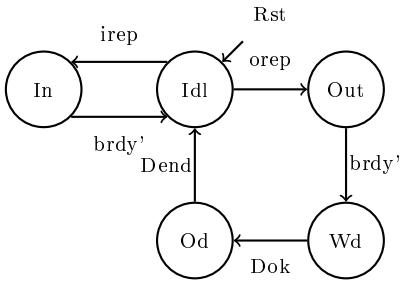


Figure 6.14: Host controller packet replicator state diagram.

The FIFO out block and the crypto unit block in Figure 6.13 use a special type of FIFO, which is called "Success-Retry FIFO". As the name implies, this FIFO has a feature to retry a previous data extraction. It functions as a normal FIFO, the difference lies in the addition of a Success and a Retry input. When the Retry input is asserted, all extracted data up to the last Success assertion is recovered. Figure 6.15 shows a diagram of it's functionality.

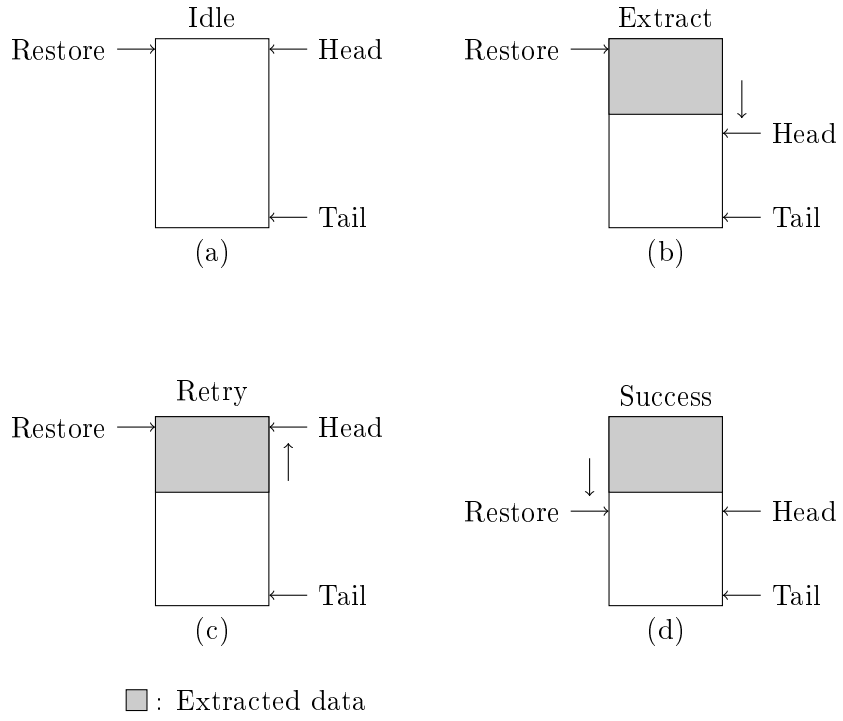


Figure 6.15: Success Retry FIFO functionality. a) FIFO before the extraction. b) Data extraction. c) Retry is asserted. d) Success is asserted.

Success-Retry FIFOs are useful for systems where the extracted data needs to be recovered when an error occurs, this is the case for the FIFO out block. When the host SIE extracts data from this block, it waits for a response from the USB device. If the external device fails to respond, then the Retry input is asserted. The crypto unit uses this FIFO to process data sequentially with an option to "seek" to the start of a sector or "free" it when the encryption is complete.

The crypto unit block is designed to capture USB data coming from the token buffer block. Figure 6.16 shows the block diagram of the crypto unit. The input register stores the sector, which is going to be encrypted or decrypted, the output register stores the result. We have implemented the STES core from [10]. This core performs disk encryption using a stream cipher and universal hash function. Figure 6.24 shows the complete implementation inside the STES unit. This unit divides the input sector in three parts, the additional buffers shown in the figure are used to swap these parts as required by the STES implementation.

In order to encrypt user data, the host controller unit implements USB bulk transfers and SCSI command detections. All USB mass storage devices use a sub set of the SCSI command [34, 5]. This command set is transmitted using bulk transfers, and stored inside the SCSI buffer in the host controller. User data is received after each SCSI Read or Write command. The SCSI command is wrapped inside a CBW structure (Chapter 3). Figure 6.17 shows the packet sequence for a SCSI command

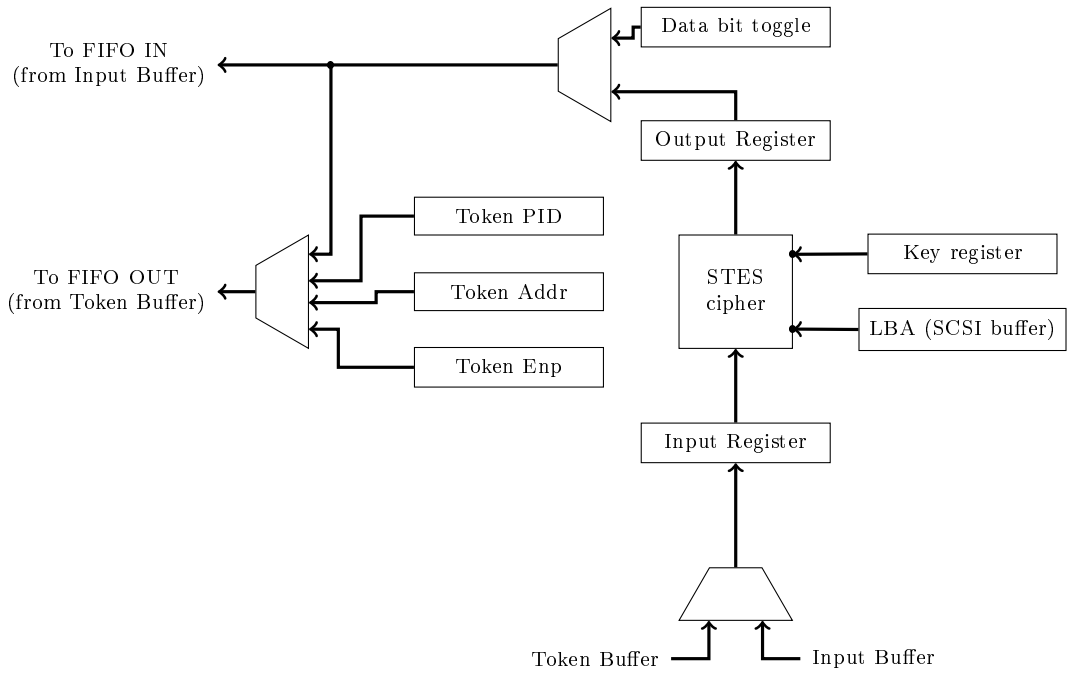


Figure 6.16: Host controller encryption architecture (Crypto unit).

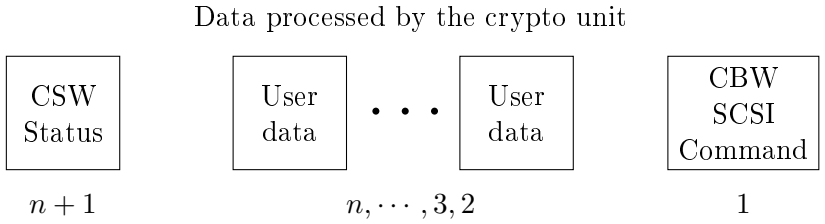


Figure 6.17: Packet sequences in a bulk transfer for a SCSI command. Each packet is prepended with an IN or OUT token.

inside a bulk transfer. The host controller detects Read(10) and Write(10) commands. The procedure for the detection and encryption of data is as follows:

- Read(10), USB Memory to host PC data transfer.
  - The USB packet containing the SCSI command is received and replicated towards the USB memory device.
  - An IN token packet is received from the host PC. This packet is replicated to the USB memory and reused until input register has a complete sector inside.
  - The STES unit will decrypt the sector inside the Input Register and store the plain text in the Output Register.

  - A DATA PID is added to the data inside the output register. If more than one packet is needed, the output register data is divided into several packets, for each packet, Data bit toggle will generate the correct DATA PID.

- Write(10), Host PC to USB Memory data transfer.
  - The USB packet containing the SCSI command is received and replicated towards the USB memory device.
  - The host controller will receive an out token packet, this packet is not replicated. The following data packet will be stored inside the Input Register. This process is repeated until the Input Register has a sector inside.
  - The STES unit process the data inside the Input Buffer Reg, the encrypted data is inserted into the Output Register.
  - The crypto unit will generate a valid token packet and insert it inside FIFO OUT. Then a DATA PID is added to the Output register and inserted into FIFO OUT. If more than one packet is needed to transfer the data, the crypto unit will generate a valid token and DATA PID for each packet.

The number of packets needed to fill the Input register depends of the USB operating speed. Full speed transmits packets up to 64 Bytes, High speed can transmit packets up to 512 Bytes.

The USB protocol uses a parity bit in order to synchronize data packets. The crypto unit stores the parity bit value of the first data packet of each SCSI command. Data bit toggle component uses this value to generate consecutive data packets.

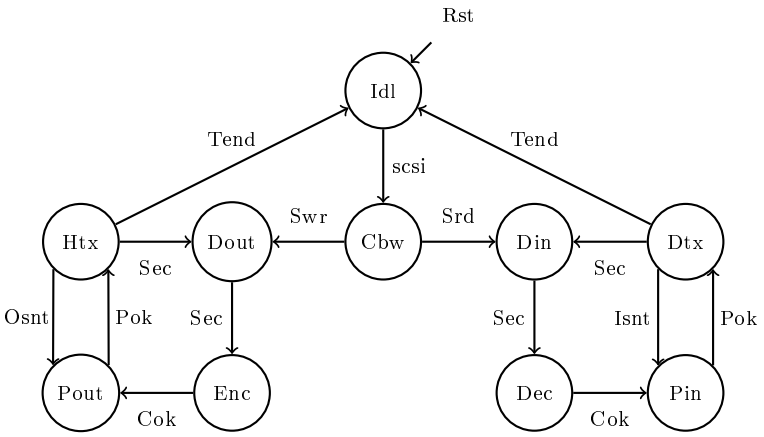


Figure 6.18: Endpoint 1 USB bulk, SCSI and Cipher state diagram.

The FSM, which controls the cryptographic part of the packet replicator, is shown in Figure 6.18. The states are described as follows:

- **idl**: The default idle state. It will transition when a command block wrapper command is detected inside the SCSI buffer.
- **cbw**: This state decodes the SCSI command inside the command block wrapper. Read(10) or Write(10) are detected here.
- **dout**: This state attends a SCSI write. It will wait for user data from the Host PC until a sector is received.
- **enc**: Encryption state.
- **pout**: This state inserts a packet with the encrypted data into the FIFO out block. This packet is sent to the USB memory.
- **htx**: Wait for the previously inserted packet to be successfully received by the USB memory. When the sector has been transmitted successfully, the FSM waits for the next sector from the host, if there is no more data then the FSM ends.
- **din**: This state attends a SCSI read. Data is read from the USB memory until a sector is received.
- **dec**: Decryption state.
- **pin**: This state inserts a packet with the decrypted data into the FIFO in block. This packet is sent to the host PC.
- **dtx**:Wait for the previously inserted packet to be successfully received by the host PC. The FSM waits for the next sector from the USB memory, if there is no more data then the FSM ends.

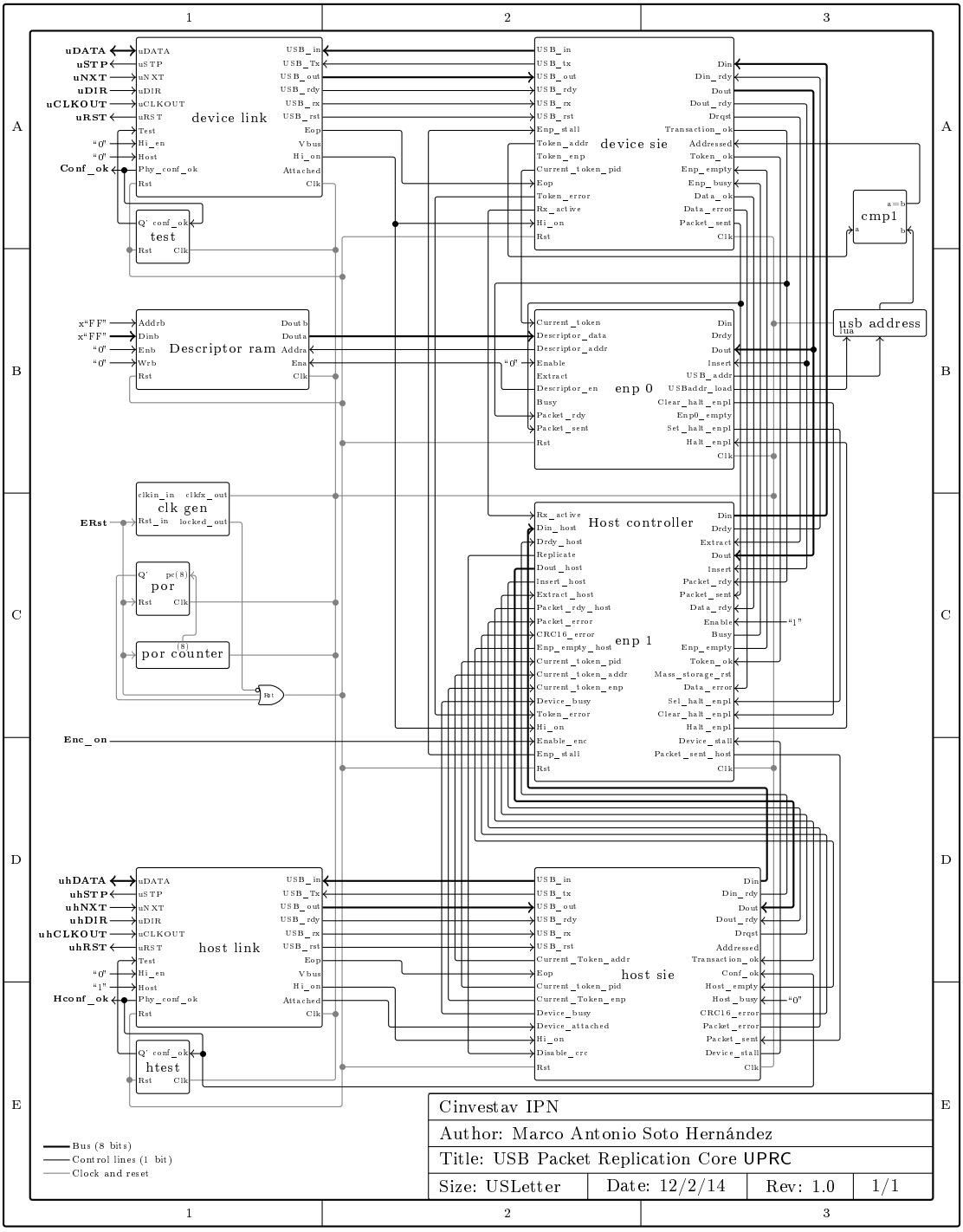


Figure 6.19: USB Packet Replication Core architecture.

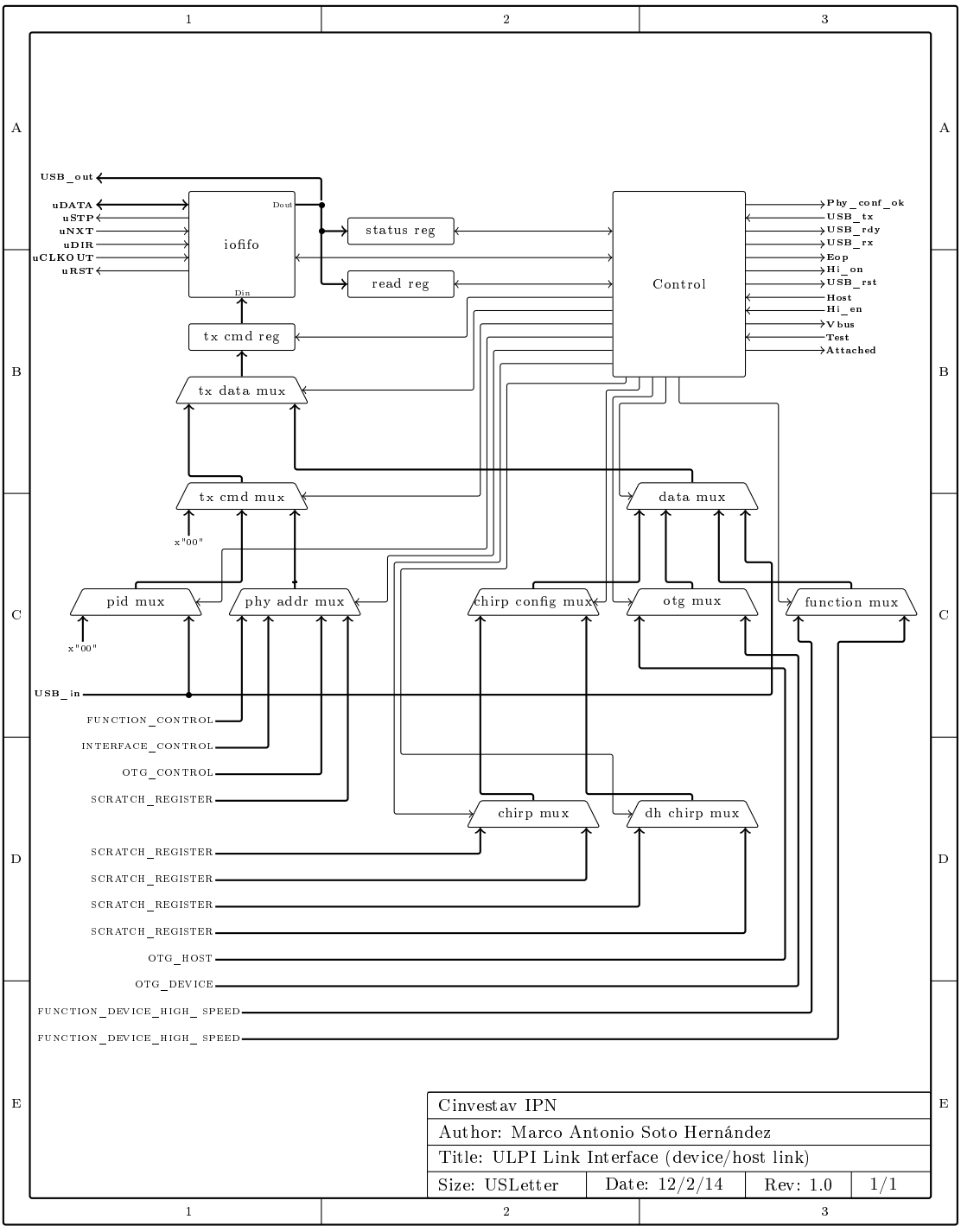


Figure 6.20: ULPI Link Layer architecture.

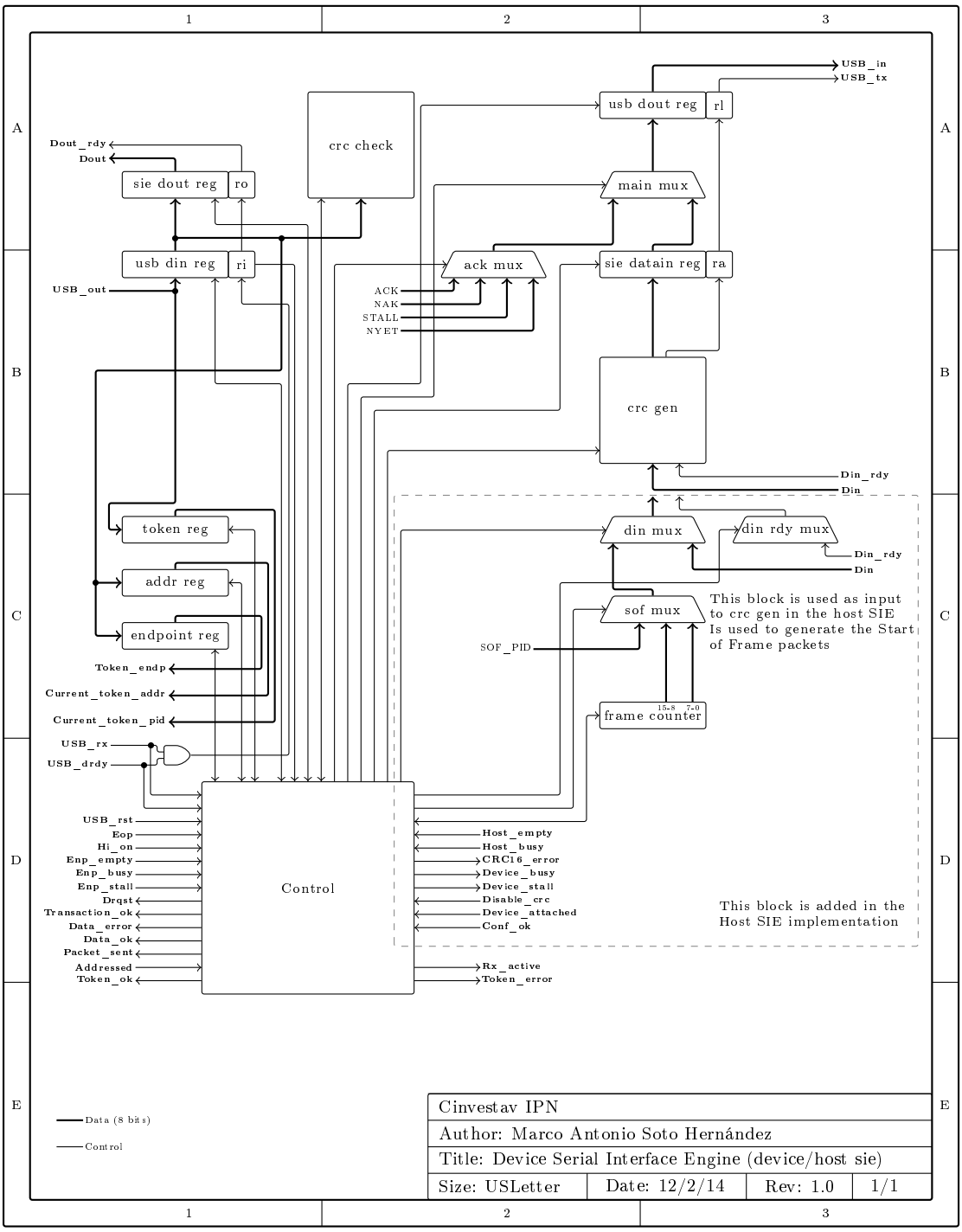


Figure 6.21: USB device and host Serial Interface Engine architecture.

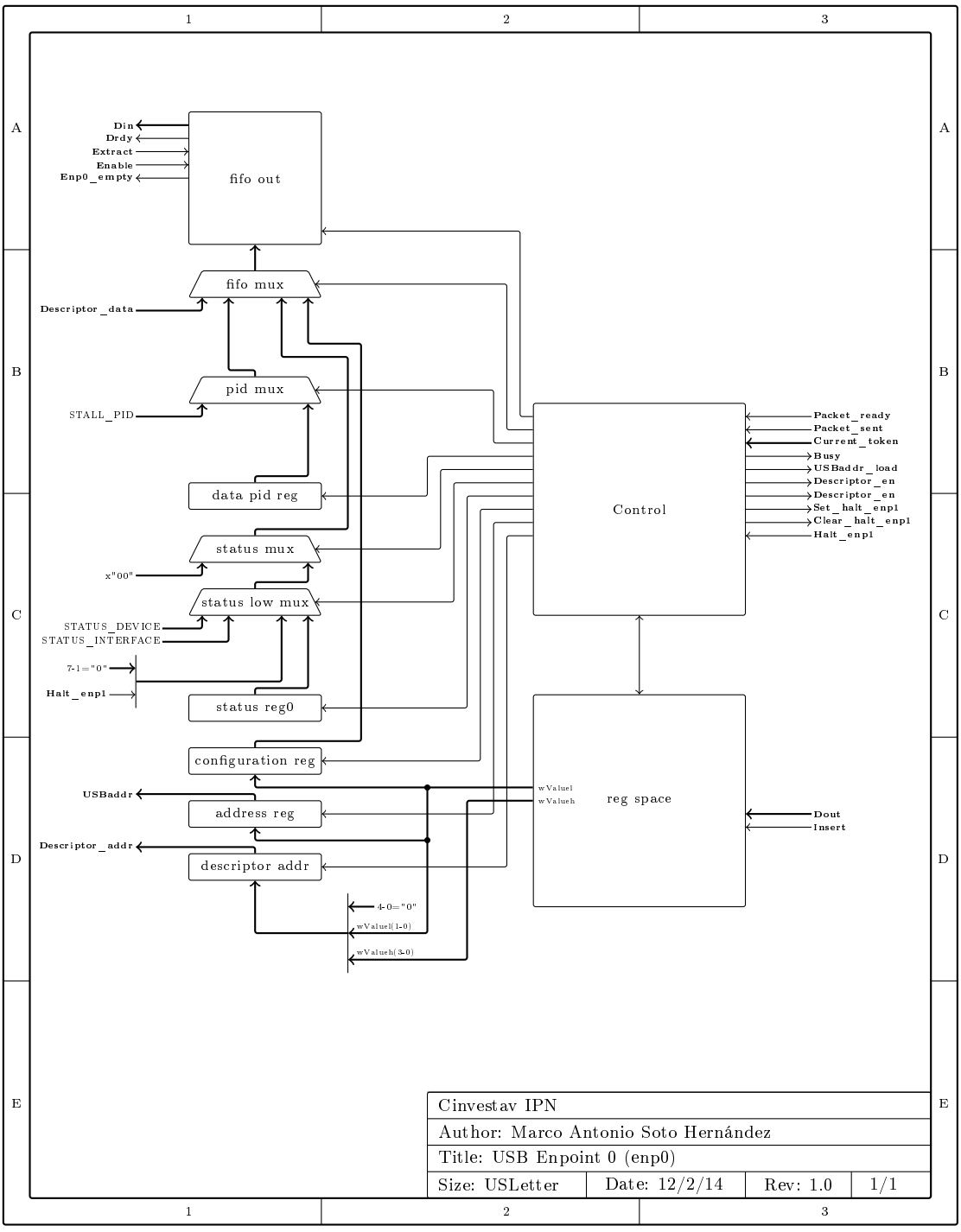


Figure 6.22: Endpoint 0 architecture.

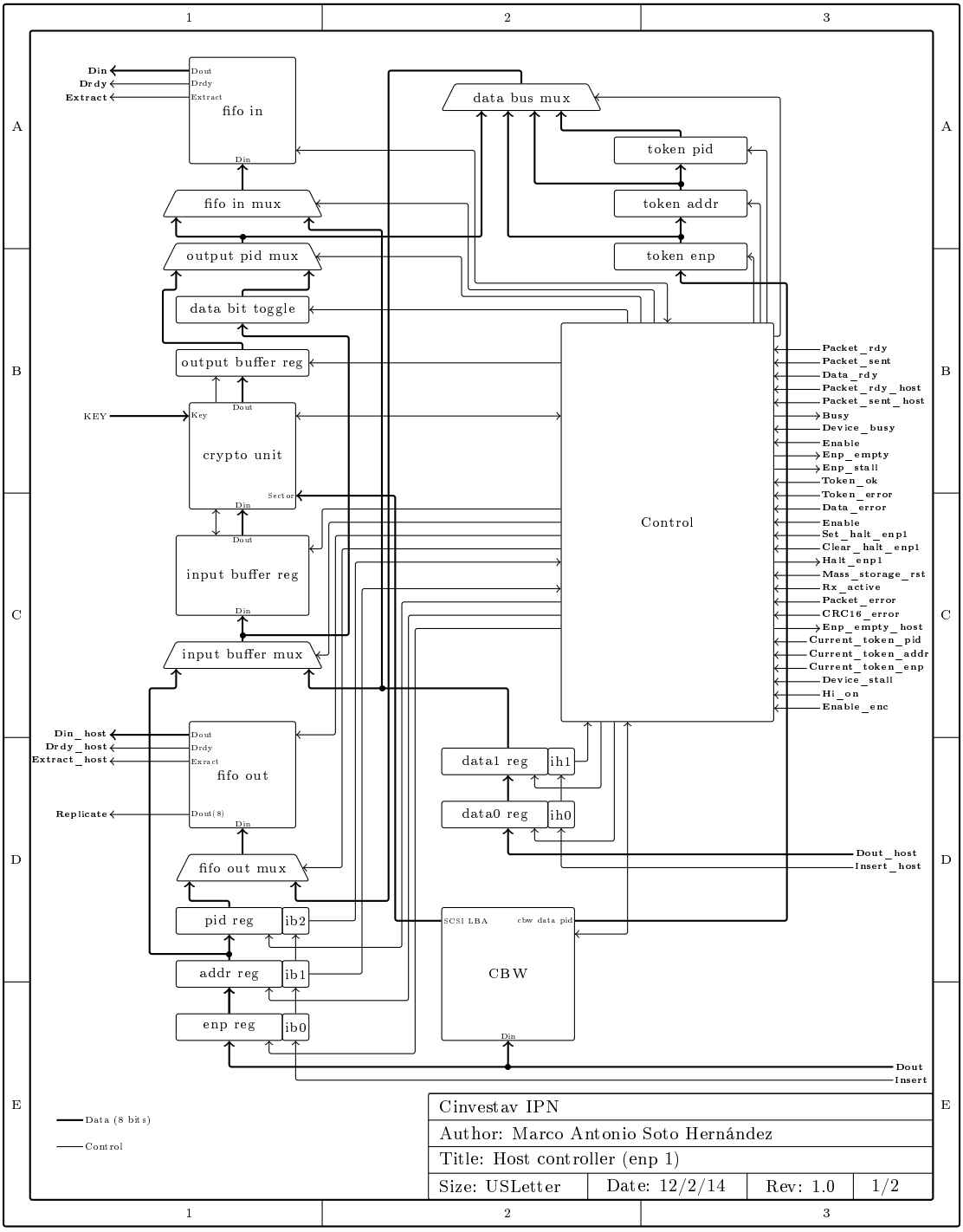


Figure 6.23: Host controller architecture.

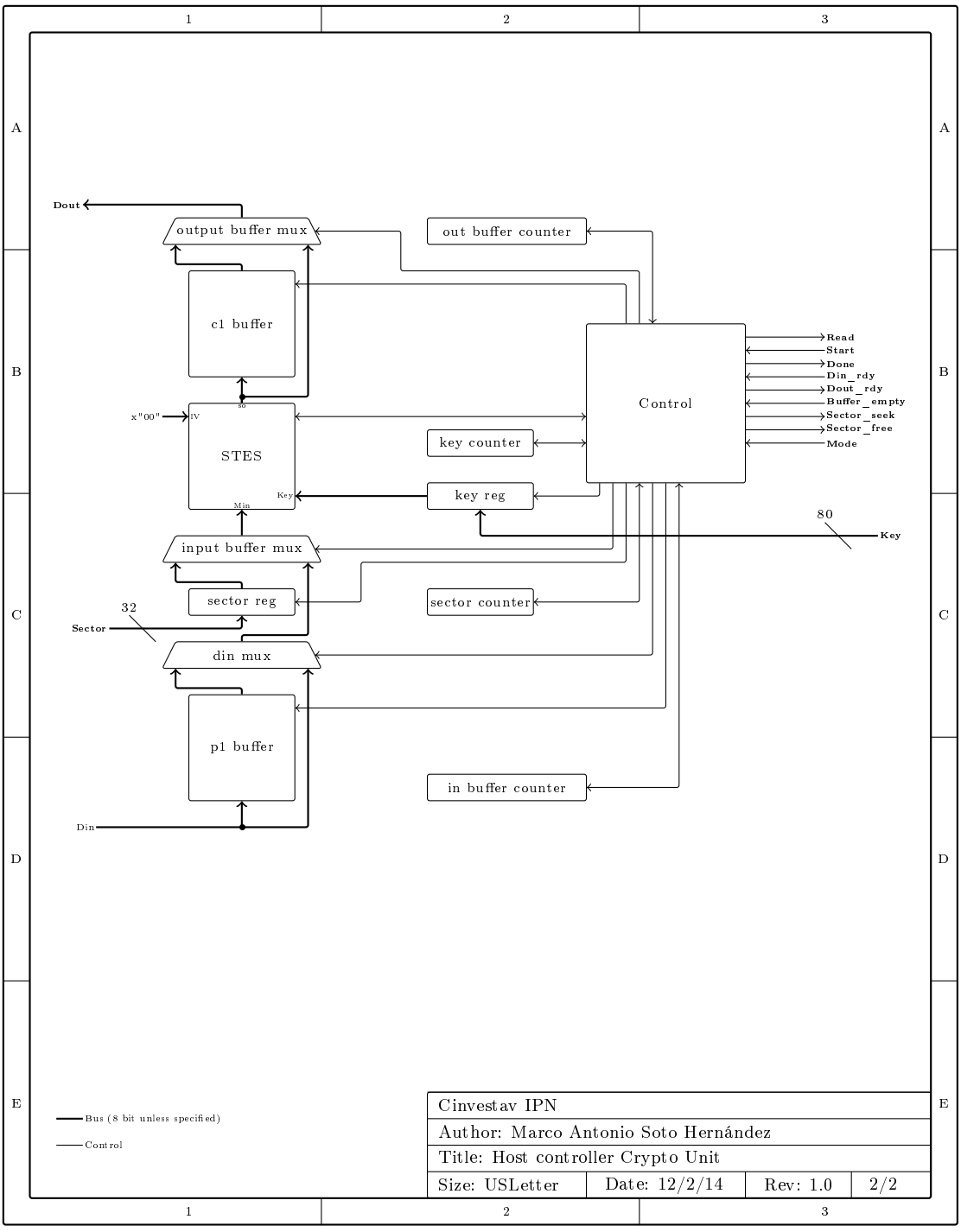


Figure 6.24: Host controller's Crypto unit architecture.

# Chapter 7

# Tests and Performance Results

In this Chapter we report some performance data of MiddleMan. First, in Section 7.1 we discuss the various tools that we required to test and debug the developed application. Then in Section 7.2 we describe a simple protocol to measure the data transfer rates within MiddleMan. Finally we present the performance data both in terms of transfer speed and area utilization in the FPGA. We also present some preliminary analysis to explain the results.

## 7.1 Test and Debug Platform

In order to debug and test the MiddleMan, we chose a collection of tools which are available in any GNU/Linux distribution. We also implemented others inside the FPGA. Here is a list of the tools we used for test and and debug.

- **usbmon**: Linux built in kernel USB monitor module.
- **Wireshark**: Network protocol analyzer for Unix and Windows.
- **RS232 Serial port**: The Spartan 3E board has a built in serial port. To use it we implemented a simple core inside the FPGA to handle this tool.
- **cutecom**: Application to communicate with serial port interfaces.
- **Xilinx ISE simulation tools**: Xilinx tool chain to synthesize and simulate reconfigurable implementations.
- **VirtualBox**: Virtualization tool, used to test USB protocol on other operating systems while analyzing traffic with **usbmon**.

We debug the USB protocol inside MiddleMan by using **usbmon** to capture USB packets. These captured packets are read using **Wireshark**, this way we can see the captured data in a more convenient way. We debug the functionality of the MiddleMan's internal components by capturing input and output data. This data is

transmitted through the FPGA serial port. We need a terminal emulator to receive the captured data inside the MiddleMan. The **minicom** terminal emulator is the most well known terminal emulator under GNU/Linux and in general it works fine. However, we chose **cutecom** because this terminal is oriented to communications with microcontrollers. This means that **cutecom** has a better hexadecimal display and dump characteristics. The serial port of the FPGA can be used to dump data from every internal unit inside the MiddleMan. We have a limitation when dumping data from the serial port, it is not fast enough to keep up with the insertion of data. This limitation was reduced by implementing a big buffer, which allows a proper dump of the monitored unit up to the maximum size of the buffer.

Host implementations differ greatly between operating systems (OS). So for testing the application in different platforms we used virtual machines for different OS inside the GNU/Linux system.

## 7.2 Performance Tests

One of the most important performance metric for MiddleMan is its transmission speed. The MiddleMan device currently works at USB full speed, theoretical speed in this mode is 12 Mbps. The bus in practice does not achieve this theoretical limit. The effective speed of the bus is reduced because of the following reasons:

- **Protocol overhead**: Transmission of data through the USB bus is performed by appending extra data and sending token packets.
- **USB host implementation**: The host schedules certain amount of packets per frame to each device connected to the bus. Bulk transfers have the lowest priority over the bus, and these transfers are used by USB memory devices, which results in a slowdown.
- **Device processing speed**: The USB memory or any other USB memory device process the USB protocol by firmware, this can be slow on some devices. Also, accessing time of a NAND memory inside a USB can take some time.

The USB host divides the bus time in frames of 1 ms each. During this time, the host can theoretically schedule up to 19 packets, each with 64 bytes of data at full speed. This amounts to an effective user data transmission rate of 9.27 Mbps. This restriction applies specifically to USB bulk transfers [12], which is the type of transfers used by storage devices.

We performed two controlled tests in order to measure the effective speed of the MiddleMan unit. The first test counts the number of packets received from the host to the FPGA, the second test counts the number of packets transmitted from the FPGA to the storage device.

To implement these tests we had to implement a small tool within the MiddleMan (i.e. in the FPGA). The tool consists of maintaining a counter, which gets incremented

every time the FPGA sends or receives an acknowledge packet, and resets when a new frame starts (i.e., a start of frame packet is received). The counter is sampled and transmitted over the serial port and captured by **cutecom**. We ensure constant use of the USB bus by copying a big file to the USB device attached to the host through the MiddleMan. We take several samples in order to calculate an average value of the number of packets received and transmitted.

In Table 7.1 we show the various parameters used for the above mentioned test and Table 7.2 shows the transfer rates.

Table 7.1: Parameters for the speed test of the MiddleMan.

| Parameter | Value | Units |
|---|---|---|
| Packet size (without USB protocol) | 64 | bytes |
| Time interval (USB frame) | 1 | ms |
| USB theoretic bus speed | 12 | Mbps |

Table 7.2: Speed results for the MiddleMan.

| Test | Packets per frame | Throughput (Mbps) |
|---|---|---|
| Host to FPGA packets | 16 | 7.8125 |
| FPGA to Device packets | 8 | 3.9062 |

Results in Table 7.2 shows that the host sends an average of 16 packets per frame[1], which means a data transmission rate of 7.8 Mbps. The average amount of packets sent by MiddleMan to the device is 8. Thus the MiddleMan slows down the performance in average by a ratio of 0.49. It is to be noted that the transfer rate of 16 packets per frame in the host device is only attained in a controlled environment where only one USB port is in use, and that port is connected to the MiddleMan. If more devices are added to the bus then the host transfer rates slows down, but this will not result in any further slowdown in the MiddleMan. In a modern computer, there exist numerous USB ports and in general they may be the part of the same hub, also in a typical usage scenario, several USB devices would be attached to the hub (say the mouse, keyboard etc.). Thus, in this scenario, the transfer rates from the host to the MiddleMan would be much lesser than what we observed in our controlled experiment.

Now, we try to analyze this result and try to find out the reason of this slowdown. The most computationally intensive operation that goes on inside the MiddleMan is the encryption through STES. The specific STES implementation that we use, uses a 8 bit data path and it takes 1705 cycles to encrypt and 1551 to decrypt a sector of 512 bytes. Thus, at an operating frequency of 60 Mhz and using a frame division of 1 ms, we can achieve a performance of 35.19 encryption operations per frame.

[1]Note that this rate is lower than the theoretical limit of 19 packets per frame which is indicated in the standard. But, this is quite normal as practically no physical host operates at the theoretical rate

Table 7.3: Spartan 3E resources used by the MiddleMan unit.

| Parameter | Value |
|---|---|
| Slices | 3215 |
| LUTs | 5073 |
| Ramblocks | 9 |
| Operating frequency | 60 Mhz |

We can see from the results that the host can transmit up to 2 sectors per frame (8 packets = 1 sector). The MiddleMan can process more than 35 sectors per frame, which is much more than the transfer rate of the host. This means that the performance drop is not due to the cryptographic process.

The real bottleneck of the system is in the communication between the MiddleMan unit and the USB storage device. In our implementation, the system can only buffer one sector. It waits for the encryption process and the transmission of data to the storage device, in order to receive more data. This is the reason of the slowdown demonstrated in our experiments. This bottleneck can be improved in the future by increasing the buffer size.

In Table 7.3 we present the amount of hardware resources utilized by MiddleMan within the FPGA. The resources used include the cores implemented to communicate with the external S3USB board, the USB protocol and the cryptographic implementation. The total implementation only uses 66% of the available space in the specific Spartan 3E device that we use. This shows that the hardware resource usage of MiddleMan is minimal.

# Chapter 8

# Conclusion and Future Work

In this Chapter we describe again the main contributions of this thesis. The main contribution of this work was to develop a detailed design and a working prototype of an USB memory encryption device. We achieved our goal to a great extent, and as a result we have a functional prototype of the MiddleMan. In Section 8.1 of this chapter we summarize our contributions and mention the main features and novelties of MiddleMan. In Section 8.2 we note down some known limitations of our design and implementation and also discuss some feasible directions to overcome the known limitations.

## 8.1 Summary of Contributions

The main contribution of this thesis is the design and implementation of a generic scheme for USB memory encryption. Our design, the MiddleMan, is different in many respects compared to available designs for storage encryption. We elaborate a bit more on this in the following paragraphs.

Disk/memory encryption was initially proposed to be implemented inside a disk/memory controller. But to the best of our knowledge this philosophy has not been still well accepted by the manufacturers of storage devices. The widely deployed systems for disk encryption are still based on software solutions, and there exist many such solutions for example Linux dm-crypt, LUKS & LVM, BestCrypt, OpenBSD softraid, Mac OS X FileVault 2 etc. These schemes does not follow the true model of a low level or in-place disk encryption. There are some commercially available secure USB memories which some what gets closer to the philosophy of in-place encryption in the sense that in these cases the encryption is a part of the memory controller. Although they are efficient, they are firmware based and still slower compared to dedicated hardware implementations.

Our design goes a step forward compared to the existing designs by the fact that we followed the true philosophy of low level encryption in a dedicated hardware with the help of minimal resources. MiddleMan is not a part of the memory controller, but it controls the USB memory itself by acting as an interface between the memory and

the host. It works in a level which is "low" enough to guarantee efficiency. Moreover, the MiddleMan being independent of both the memory and the host, provides lots of flexibility to the user in terms of repeated usage for different memory/host devices. MiddleMan can be used to secure USB memories already owned by users which does not have any encryption included in it.

Finally, the encryption algorithm implemented within the MiddleMan is a tweakable enciphering scheme. As we have repeatedly mentioned, this class of schemes are most suited both in terms of security and functionality for sector wise storage encryption. These schemes are not yet widely deployed probably because of the high costs involved in implementing them. Though there have been prototype studies which proves that many TES can achieve performance/area metrics suitable for the application, but we do not know of a real life low-level disk/memory encryption device which uses TES. Thus, our work can also be seen as a first real world implementation of a TES which performs the functionality of low level encryption in a *real* storage device.

**Features of MiddleMan:** MiddleMan has many features which are quite lucrative in terms of usability, flexibility, cost and performance. Next we discuss these novel features of the MiddleMan:

**Transparent Communication:** The design of the MiddleMan enables communications from an USB to a host and vice versa in a completely transparent manner, i.e., when in operation neither the host nor the USB is aware of its presence. This means that there is no extra overhead within the host or the memory controller for using the MiddleMan.

**Plug and Play:** There is no necessity of any driver or extra firmware for the use of MiddleMan, it can be plugged in between a host and an USB and it immediately starts functioning.

**Multi-platform Support:** As MiddleMan functions at low level, it only requires that the USB memory and the host follow the standard USB protocol. The design does not assume any other high level details of the host computing platform. A host computer with an USB port can use MiddleMan irrespective of the operating system or file system it uses.

**Reconfigurability:** The design of MiddleMan is targeted towards FPGAs. The main communication protocol and the encryption algorithm is implemented within an FPGA. This gives us options of reconfiguration. The encryption algorithm or other details can be changed or updated if required.

**Low Cost:** The hardware resources required to implement MiddleMan are low. It can be easily implemented within a low cost FPGA (like Spartan3, Lattice ICE40) with minimal additional hardware. Thus the real cost of such a device would be very low.

**The S3USB Extension:** An important part of our implementation is the design of the S3USB circuit which was used to extend the Spartan 3E board. Though we used S3USB solely for the purpose of implementing the MiddleMan, this circuit can be of independent interest as it increases the functionality of the Spartan 3E board in several respects.

The S3USB expansion provides the necessary low level USB communication. This extension was developed to overcome several issues with the current FPGA boards regarding USB support. These issues are summarized as follows:

- **Configuration only port**: Most FPGAs have a USB port only intended for for configuration of the FPGA.
- **Basic USB support**: FPGAs, which features a USB port for communications, do so by implementing basic functions. These functions are in general aimed at basic I/O devices such as keyboards, but not for storage devices which uses bulk transfers.
- **Low data transfer rates**: There are some FPGAs, in particular, made by Digilent, which has a basic USB data transfer function. This data transfer is very limited, and in order to transmit user data, a third party programs are required to be installed in the host PC.

The S3USB overcomes these issues by providing the most flexible solution possible.

We mainly used the S3USB for developing the MiddleMan. However, this expansion board can effectively be used for many more USB applications. The addition of S3USB to a Spartan 3E board converts it it into an embedded USB platform. Such a platform can be used to develop various USB based applications, we mention some of the possibilities below (the list of course is not exhaustive):

- **Embedded PC with USB support**: The Spartan 3E can function as an embedded PC with a MicroBlaze processor. We can add two full featured USB ports (using S3USB) to extend communications.
- **Two channel USB signal processing and capture device**: The Spartan 3E board features two digital to analog and analog to digital converters. We can develop a USB signal capturing device for audio or other analog signal applications using the S3USB.
- **Fast IP debugging**: We used the serial port of the Spartan 3E board in order to debug the MiddleMan. But a better debugging option for any other IP core implemented inside the FPGA can be achieved through an USB isochronous communication protocol through the S3USB. This will provide an almost real time signal debugging. Some Xilinx tools like **Chip Scope** already provide this feature. But under Linux this feature does not work well because of some driver speed issues. Thus debugging though S3USB can be a much better option.

Hence, the design and implementation of S3USB is also a contribution by itself. And it is also an open source implementation, which means that anyone is free to use it.

## 8.2 Limitations of our Implementation

The implementation that we have is prototype, and thus it has some limitations which we are aware of. Below we discuss some known limitations of MiddleMan along with the ways that we can possibly overcome them. Trying out these ways are left as future work.

**Speed**: Currently, the MiddleMan can only work at USB Full speed. High speed is possible but it has not yet been implemented. Also, it is evident from the experimental results in Section 7.2, MiddleMan does not achieve transfer speeds expected from a Full Speed device. As discussed, the implementation of MiddleMan can be improved in this respect. We discuss some possible directions of improvements in the following points:

- Upgrading MiddleMan to support High speed is not a big issue. Currently it does not support this speed because of the limitation of the USB protocol implemented within it. The protocol supporting High speed is much more complex than the one supporting only Full Speed. So, to minimize the coding efforts we chose not to incorporate this advanced USB version in our prototype. The design has no limitation on adding this functionality and the current implemented protocol can be upgraded to support high speed with just some extra amount of coding and debugging effort.

- The buffer structures and pipelining currently implemented in the host controller of MiddleMan are not optimal, this leads to delays in data transfers from the MiddleMan to the device. In intuitive terms, currently the buffer size can hold only one sector. Thus after encryption, unless the encrypted sector has been transferred completely to the device, MiddleMan does not start encrypting the next sector. This leads to delays. This can be solved by designing an optimal pipelining strategy, and we plan to work on this in the future.

- The STES implementation that we use within MiddleMan can also be made more efficient. Firstly, we use an implementation, which uses a 8-bit datapath. In [10] where STES was originally proposed, experimental results up to 40-bit datapaths are reported. Of course bigger datapaths give rise to better throughput. Additionally, the STES implementation exploits parallelism in the design to the granularity of sectors, i.e., the design considers parallelism when the data size is same as the sector. But this can be further improved by exploiting parallelism across sectors, as in MiddleMan (also in other bulk storage applications) multiple sectors are generally read from and written into a storage device. Considering issues of such parallelization a more improved and efficient design of

STES is possible. Though this is an interesting work which we plan to attempt in the future, it is worth mentioning that the specific design of STES that we use is efficient enough for the application. The reason why MiddleMan does not achieve the speeds of a full speed device has more to do with the sub optimal pipelining and buffering strategy which we adopted in the implementation of the host controller than with the inefficiency of the encryption algorithm.

**Key Management**: Securing user data requires the input of a key which must be provided by the user. At this point we haven't implemented any key management functionality within MiddleMan. Currently the key is hardcoded inside the device, and for key changes the circuit needs to be reconfigured. This is of course not a viable option both from usability and security perspectives. However, there can be various ways in which a proper key management module can be added on to MiddleMan. We discuss some of these possibilities, which we plan to add to our prototype in future.

1. **Keypad password insertion**: A simple keypad can be provided as a part of MiddleMan. With this an user can directly feed in the key in the device whenever it is to be used. Moreover in this paradigm, options for changing keys can also be provided.

2. **Fingerprint scan**: A fingerprint scanner can be attached to the MiddleMan. There exists techniques to convert biometric information like fingerprints to cryptographic keys. Chips implementing fingerprint scanners are commercially available. Adding such a chip to MiddleMan would not involve significant complexity.

3. **RFID Key Device**: We can also think of a scheme, where the key itself is a hardware, say a RFID tag. We can implement a RFID reader within the MiddleMan and when the RFID tag containing the key is brought in physical contact to (or near) the MiddleMan the key is read.

4. **Software Based key management**: In this option, the key is inserted by a software support. The USB specification allows the developer to insert custom device requests. We can define some of these requests in order to generate and set a key inside MiddleMan.

It seems to us that the above discussed options are all viable to add a functional key management module to MiddleMan. But we need to do a systematic security and feasibility analysis for these options. We plan to do this in the near future.
**Power issues**: MiddleMan is currently implemented in a Spartan 3E board. This requires to be powered up through an external power source. But if we make MiddleMan into an independent device, then issues of power consumption would become important. Feeding power from an external source for a device like MiddleMan does not seem to be satisfactory. It would be the best if the power requirements of the

MiddleMan can be fed by the host USB port. Though we have not made any extensive power measurements, it seems that the power consumption of Middleman is low enough to support such a design.

**Other Minor Issues:** The current implementation still have some other minor issues mainly in the circuit board of S3USB which are required to be fixed. For completeness we list them:

1. The Link layer inside the MiddleMan has yet to be improved to better handle the USB ports. The host link can handle well when a USB memory is attached at the port. But detachments are yet to be implemented.

2. The device section of the S3USB board has the D+ and D- lines swapped, this was fixed by building a cable with the lines swapped. When we print the circuit board for S3USB again, we would fix this issue.

3. The connection between the USB ports and the USB3300 must have a differential impedance close to 45 Ω. In the current circuit this impedance is much more than that. The device still works fine in Full Speed. But improving the impedance would improve the error rate at High speed, and this change is necessary if we upgrade MiddleMan to support High speed.

# Appendix A

# USB Suplementary Information

## A.1 USB Enumeration Process

The USB enumeration is the process used by a host in order to configure a USB device. This process gathers information needed in order to select a proper configuration, the process steps are as follows:

- The host detects a new device attached to the bus.
- The host sends a reset signal to the device.
- After reset, information is gathered from the device.
- After gathering enough information, the host assigns an address and configuration to the device.

This process is performed by sending packets to the device with a structure called device request. We present the structure in table A.1, for a deep description of each field refer to [12, P. 293]. We can see the minimum device requests in Table A.2. These are the device requests needed to configure a USB device. Every device request is transmitted to a device by performing a control transfer. These transfers are always addressed at endpoint 0. A control transfer carrying a device request has a packet ID for a setup transaction as seen on table 3.3 page 22.

Information requested by GET_DESCRIPTOR is stored inside a USB device in a data structured called descriptor [12]. There are several type of descriptors, they store information about the number of endpoints the device supports, vendor code, device ID and device type like storage or capture.

## A.2 USB CRC Generation

To protect data transmitted in the bus, the USB protocol implements Cyclic Redundancy Codes (CRC). These codes protect non PID fields inside a token or data packet

Table A.1: Device requests data structure. This table was taken from the USB 2.0 specification.

| Offset | Field | Size | Value | Description |
|---|---|---|---|---|
| 0 | bmRequestType | 1 | bitmap | Characteristics of Request<br>D7: Data transfer direction<br>0 = Host-to-device<br>1 = Device-to-host<br><br>D6...5: Type<br>0 = Standard<br>1 = Class<br>2 = Vendor<br>3 = Reserved<br><br>D4...0 Recipient<br>0 = Device<br>1 = Interface<br>2 = Endpoint<br>3 = Other<br>4...31 = Reserved |
| 1 | bRequest | 1 | Value | Specific request see table A.2 and USB specification [12, p. 250] |
| 2 | wValue | 2 | Value | Word size field that varies according to request |
| 4 | wIndex | 2 | Index or offset | Word size field that varies according to request; typically used to pass an index or offset |
| 6 | wLength | 2 | Count | Number of bytes to transfer if there is a data stage |

Table A.2: Device requests needed to perform the enumeration process.

| Request | Description |
|---|---|
| GET_DESCRIPTOR | The device must send the requested information to the host. |
| SET_ADDRESS | The device must set the address value contained inside the request as it's own device address |
| SET_CONFIGURATION | Every device must support at least one configuration, this command tell the device to set a specific configuration |

and are appended at the end of the transmitted data. The size of a CRC field for a token packet is 5 bits, for a data packet is 16 bits.

We can intuitively obtain a CRC by computing $I \mod G_c(x)$, where $I \in \{0,1\}^*$ is the input data, $G_5(x) = x^5 + x^2 + 1$ is a generator polynomial for $\mathbb{F}_{2^5}$ and $G_{16}(x) = x^{16} + x^{15} + x^2 + 1$ is a generator for $\mathbb{F}_{2^{16}}$. We use the $\mathbb{F}_{2^5}$ field to calculate CRC for token packets, and $\mathbb{F}_{2^{16}}$ for data packets. To calculate the modulo operation we take the binary representation of $G_c(x)$. For both fields we have $G_5(x) = 100101$ and $G_{16} = 11000000000000101$.

For USB, calculation of CRC varies slightly. The USB implementation of CRC requires to initialize a remainder accumulator with all bits set to one. The next input bit is XOR'ed with the most significant bit of the accumulator. Then this accumulator is shifted to the left, a '0' is inserted in the least significant bit. If the result of the XOR is '1', then the generator polynomial is subtracted from the accumulator. When the division is complete, we must invert the accumulator to obtain the CRC. Algorithm 1 shows the CRC calculation procedure, the architecture is in Figure A.1. The algorithm and implementation presented here can compute CRC5 and CRC16. Examples of CRC calculation can be consulted in [13].

**Algorithm 1** Procedure to calculate a CRC5 or CRC16 code for a USB packet.

**CRC** $(I, n, c)$ ▷ $I \in \{0,1\}^n, n \in \{0, \cdots, 4096\}, c \in \{5, 16\}$
1: $Acc \leftarrow \{1\}^c$
2: **if** $c = 16$ **then**
3: $X_c \leftarrow$ "11000000000000101" ▷ $X_c \leftarrow x^{16} + x^{15} + x^2 + 1$.
4: **else**
5: $X_c \leftarrow$ "100101" ▷ $X_c \leftarrow x^5 + x^2 + 1$
6: **end if**
7: **for** $i = 0; i \leq n; i \leftarrow i + 1$ **do** ▷ $Acc$ shift and accumulate cycle
8: $Acc[0] \leftarrow I[i] \oplus Acc[c-1]$
9: **for** $j = 1; j < c; j \leftarrow j + 1$ **do**
10: **if** $X_c[j] = 1$ **then**
11: $Acc[j] \leftarrow Acc[j-1] \oplus Acc[0]$
12: **else**
13: $Acc[j] \leftarrow Acc[j-1]$
14: **end if**
15: **end for**
16: **end for**
**return** $\sim Acc$

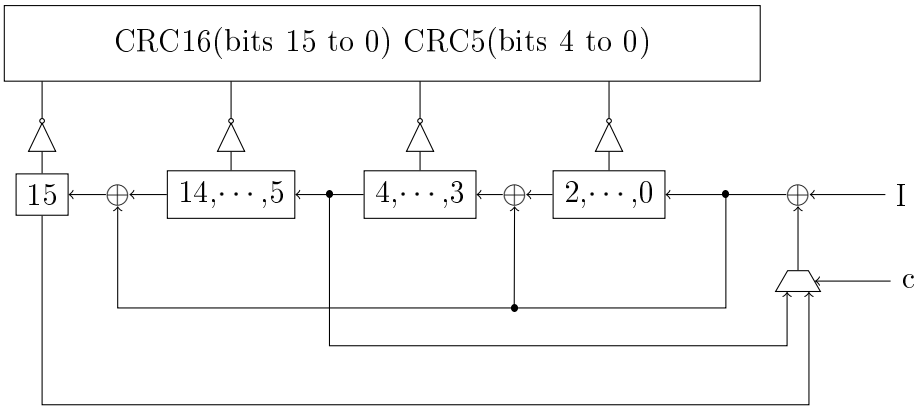


Figure A.1: Architecture of the CRC calculator.

# Bibliography


[1] Kingston, secure USB drives. http://www.kingston.com/en/usb/encrypted_security/. Consulted on January $22^{th}$ 2014.

[2] Public comments on the XTS-AES mode. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/XTS/collected_XTS_comments.pdf.

[3] IEEE Std 1619-2007: IEEE standard for cryptographic protection of data on block-oriented storage devices. IEEE Computer Society, April 2008. http://standards.ieee.org/findstds/standard/1619-2007.html.

[4] IEEE Std 1619.2-2010: IEEE standard for wide-block encryption for shared storage media. IEEE Computer Society, March 2011. http://standards.ieee.org/findstds/standard/1619.2-2010.html.

[5] Jan Axelson. *USB Mass Storage: Designing and Programming Devices and Embedded Hosts*. Lakeview Research, 2006.

[6] Jan Axelson. *USB Complete: The developers guide*. Lakeview Research, 4th edition, 2009.

[7] D. Chakraborty, C. Mancillas-Lopez, F. Rodriguez-Henriquez, and P. Sarkar. Efficient hardware implementations of brw polynomials and tweakable enciphering schemes. *IEEE Transactions on Computers*, PP(99):1, 2011.

[8] D. Chakraborty and P. Sarkar. HCH: A new tweakable enciphering scheme using the hash-counter-hash approach. *IEEE Transactions on Information Theory*, 54(4):1683 –1699, april 2008.

[9] Debrup Chakraborty, Vicente Hernandez-Jimenez, and Palash Sarkar. Another look at xcb. Cryptology ePrint Archive, Report 2013/823, 2013. http://eprint.iacr.org/.

[10] Debrup Chakraborty, Cuauhtemoc Mancillas-Lopez, and Palash Sarkar. STES: A stream cipher based low cost scheme for securing stored data. Cryptology ePrint Archive, Report 2013/347, 2013. http://eprint.iacr.org/.

[11] Debrup Chakraborty and Palash Sarkar. A new mode of encryption providing a tweakable strong pseudo-random permutation. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 293–309. Springer, 2006.

[12] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. *Universal Serial Bus Revision 2.0 specification*, 2.0 edition, april 2000.

[13] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. *USB CRC Description*, 2.0 edition, april 2000.

[14] Altera corporation. High-speed board layout guidelines. `www.altera.com/literature/an/an224.pdf`, 2003.

[15] Cypress Semiconductor Corporation. EZ-USB SX2TM high speed usb interface device. `http://www.cypress.com/?docID=47166`, 2003.

[16] Fox Electronics. Quartz crystal design notes. `www.foxonline.com/pdfs/xtaldesignnotes.pdf`, 2004.

[17] Nallely Itzel Guadalupe Trejo García. Efficient software implementations of disk encryption schemes using AES-NI support. Masters Thesis, Computer Science Department, CINVESTAV-IPN, Mexico City, Mexico, 2012. `http://www.cs.cinvestav.mx/TesisGraduados/2012/TesisNallelyTrejo.pdf`.

[18] Shai Halevi. EME*: Extending EME to handle arbitrary-length messages with associated data. In Anne Canteaut and Kapaleeswaran Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004*, volume 3348 of *Lecture Notes in Computer Science*, pages 445–462. Springer Berlin / Heidelberg, 2005.

[19] Shai Halevi. Invertible universal hashing and the TET encryption mode. In Alfred Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 412–429. Springer, 2007.

[20] Shai Halevi and Phillip Rogaway. *A Tweakable Enciphering Mode*, volume 2729 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003.

[21] Shai Halevi and Phillip Rogaway. A parallelizable enciphering mode. In Tatsuaki Okamoto, editor, *Topics in Cryptology – CT-RSA 2004*, volume 2964 of *Lecture Notes in Computer Science*, pages 1995–1995. Springer Berlin / Heidelberg, 2004.

[22] Vicente Hernandez-Jimenez. Some iussues on security of tweakable enciphering schemes. Masters Thesis, Computer Science Department, CINVESTAV-IPN, Mexico City, Mexico, 2013. `http://www.cs.cinvestav.mx/TesisGraduados/2013/TesisVicenteHernandez.pdf`.

[23] ECS Inc. CSM-7X-DU SMD crystal. `http://www.ecsxtal.com/store/pdf/CSM_7X_DU.pdf`.

[24] Maxim Integrated. Dual USB switch with fault blanking and autoreset. `http://datasheets.maximintegrated.com/en/ds/MAX1823-MAX1823H.pdf`, 2003.

[25] Jon L. Jacobi. The best encrypted flash drives. `http://www.pcworld.com/article/254816/the_best_encrypted_flash_drives.html`, May 2012.

[26] Howard W. Johnson and Martin Graham. *High-speed digital design : a handbook of black magic*. Englewood Cliffs, N.J. Prentice Hall, 1993.

[27] F.A. Jolfaei, N. Mohammadizadeh, M.S. Sadri, and F. FaniSani. High speed USB 2.0 interface for fpga based embedded systems. In *Embedded and Multimedia Computing, 2009. EM-Com 2009. 4th International Conference on*, pages 1 –6, dec. 2009.

[28] Cuauhtemoc Mancillas López. Studies in disk encryption. PhD Thesis, Computer Science Department, CINVESTAV-IPN, Mexico City, Mexico, 2013. `http://www.cs.cinvestav.mx/TesisGraduados/2013/TesisCuauhtemocMancillas.pdf`.

[29] C. Mancillas-Lopez, D. Chakraborty, and F. Rodriguez Henriquez. Reconfigurable hardware implementations of tweakable enciphering schemes. *IEEE Transactions on Computers*, 59(11):1547 –1561, nov. 2010.

[30] L. Martin. XTS: A mode of AES for encrypting hard disks. *Security Privacy, IEEE*, 8(3):68–69, 2010.

[31] D. McGrew and S. Fluhrer. The security of the extended codebook (XCB) mode of operation. In *Selected Areas in Cryptography*, pages 311–327. Springer, 2007.

[32] Palash Sarkar. Efficient tweakable enciphering schemes from (block-wise) universal hash functions. *IEEE Transactions on Information Theory*, 55(10):4749–4760, 2009.

[33] Palash Sarkar. Efficient tweakable enciphering schemes from (block-wise) universal hash functions. *IEEE Transactions on Information Theory*, 55(10):4749–4760, 2009.

[34] Seagate. *SCSI Commands Reference Manual*. `www.seagate.com/staticfiles/support/disc/manuals/scsi/100293068a.pdf`, 2006.

[35] SMSC. Hi-speed USB host, device or OTG PHY with ULPI Low Pin Interface. `http://www.smsc.com/Downloads/SMSC/Downloads_Public/Data_Sheets/3300.pdf`, 2013.

[36] T. Tiong Hong, S. Zeeshan, and M.Z. Abdullah. A power efficient USB 2.0 device controller architecture and its implementation. In *Consumer Electronics (ISCE), 2011 IEEE 15th International Symposium on*, pages 388 –392, june 2011.

[37] Peng Wang, Dengguo Feng, and Wenling Wu. HCTR: A variable-input-length enciphering mode. In Dengguo Feng, Dongdai Lin, and Moti Yung, editors, *CISC*, volume 3822 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2005.

[38] ULPI Working Group. *UTMI+ Low Pin Interface*. `http://www.ulpi.org/ULPI_v1_1.zip`, 2004.

[39] ULPI Working Group. *UTMI+ White paper*. `http://www.ulpi.org/utmiplus_whitepaper.pdf`, 2004.

[40] Xilinx. *Spartan-3E FPGA Starter Kit Board User Guide*. `www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf`, 2011.