TECHNICAL UNIVERSITY OF CRETE

BACHELOR OF SCIENCE THESIS

IMPLEMENTING A THIN CLIENT FOR CLOUD CONNECTION USING FIELD PROGRAMMABLE GATE ARRAY [FPGA]

MARCH 2015

TITLE

Implementing a Thin Client for Cloud connection using

Field Programmable Gate Array [FPGA]

AUTHOR:

KARATZAFERIS EFSTATHIOS ALEXANDROS

SUPERVISOR:

ASSOCIATE PROFESSOR, YANNIS PAPAEFSTATHIOU

COMMITTEE:

PROFESSOR, APOSTOLOS DOLLAS

PROFESSOR, MICHALIS ZERVAKIS

ABSTRACT

Technology, nowadays, dictates the use of high energy consumption devices in order to run applications that demand a large amount of computing and network resources. Video streaming, video calls, gaming, fancy multifunctional HTML5 webpages, web-based applications and media editing software are a few examples of everyday usage of devices, almost reaching their computing potential.

Portable devices crave for a reliable and portable power source while Desktop Computers demand more and more computing power and higher processing speed. A common solution is applied to address both of these issues, and that is Cloud Computing. Servers on the Internet are assigned to execute a big chunk of the workload while Clients are only responsible for data representation.

Our goal is to exploit the advantages of this scheme and we achieve that by introducing a new level of abstraction which manages every I/O event that takes places between the user and the machine. By isolating the data representation from the data processing procedures, we eliminate the need of the CPU's / GPU's / HDD's physical presence inside the modern devices structure, therefore lowering the energy consumption and computing needs.

Our proposed system consists of two parts. A Thin Cloud Client designed in a Field-Programmable Gate Array device which allows the user to remotely connect to a PC of his choice, and a Cloud Server which is a C# application running on the connected PC. Computer and FPGA communicate through the Internet and exchange every I/O event that is generated from/for the peripheral devices. Our system is connected with the following peripherals: mouse, keyboard, screen, speakers, and manages the data stream of all four. The FPGA user can remotely access and fully use a PC running the Server application.

KEYWORDS

FPGA, REMOTE CONTROL, REMOTE ACCESS, STREAM, CLOUD, CLIENT, SERVER

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Yannis Papaefstathiou for his help and guidance, for giving me the opportunity to work on my thesis, and also for giving me the freedom to implement it the way I imagined I would!

I would also like to express my sincere gratitude to Joel Williams, who helped me and advised me in the design process of the network interface, without even knowing who I am or what my thesis is about! I am also thankful to Giota Mofori, for proofreading this thesis!

Finally, I would like to thank my family and friends for their support through all these years!

ABBREVIATIONS

APP	-	**App**lication
ARP	-	**A**ddress **R**esolution **P**rotocol
CPU	-	**C**entral **P**rocessing **U**nit
DLL	-	**D**ynamically **L**inked **L**ibrary
DVI	-	**D**igital **V**isual **I**nterface
FPGA	-	**F**ield-**P**rogrammable **G**ate **A**rray
FSM	-	**F**inite **S**tate **M**achine
FPS	-	**F**rame **P**er **S**econd
GPU	-	**G**raphics **P**rocessing **U**nit
GUI	-	**G**raphical **U**ser **I**nterface
HDD	-	**H**ard **D**isk **D**rive
IP	-	**I**nternet **P**rotocol
I/O	-	**I**nput / **O**utput
MAC	-	**M**edia **A**ccess **C**ontrol
MIG	-	**M**emory **I**nterface **G**enerator
OS	-	**O**perating **S**ystem
PC	-	**P**ersonal **C**omputer
UI	-	**U**ser **I**nterface
VGA	-	**V**ideo **G**raphics **A**rray
VNC		**V**irtual **N**etwork **C**omputing

Contents

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION AND BASIC CONTRIBUTION

1.1 INTRODUCTION

Nowadays, technology dictates the frequent use of portable computing devices, such as Laptops, Smartphones, Tablets etc. These are mostly used to surf the Internet, playback media (videos, music), make phone and video calls and to get through some lightweight work, such as responding to e-mails. Video streaming, access to multifunctional HTML5 webpages, media playback, flash storage and access to web applications are only a few features that every design must be equipped with.

All these devices require a portable power source and batteries are used to fulfill these needs. Even though we had some major improvements in this field, batteries cannot provide energy to the modern devices for more than ~35 hours, and if features like WIFI and 4G are used to access the Internet, or the CPU operates at full speed, the batteries durability is approximately 20 hours. Comparing these numbers with the evolution of today's CPUs and GPUs we can safely assume that a more elegant solution is much needed.

Besides the portable devices dominance, many users acquire a PC to work on or a video game console to entertain themselves while playing games. Hardware designers, web designers, media editors, graphics designers, architectures to a name a few, are using computers with great computing power and enormous storage capacity. The applications designed or the games played, crave for more and more computing power themselves, which forces users to update their PC with better and faster components.

Researchers came up with a new scheme, Cloud Computing, which tries to address some of the issues we just mentioned. Computing workload is transferred through the Internet and is assigned to powerful servers which just return the results of the calculations. This model saves up a lot of computing time since many computers can cooperate to finish a difficult task. The same concept is used in order to save digital files to the Cloud, eliminating the need for large HDDs.

This is the scheme is we are trying to exploit by assigning a server (Cloud Server) to carry out all the computing tasks for a Cloud Client, leaving it with one and only duty, the data representation. Specifically, we have managed to redirect all I/O peripheral stream from/to the FPGA through the Internet, thus enabling it to remotely control the Cloud Server and use it like an ordinary PC.

1.2 GOAL & MOTIVATION

Our main motive is to drastically lower the energy consumption levels of modern devices, while lightening their computing workload by designing a new genre of machines. Users will theoretically be equipped with more powerful and less power hungry personal computers. PC's which are more time enduring. Achieving that required to reach a number of goals which will be now listed.

- ✓ Isolate the user peripherals from the CPUs / GPUs in order to eliminate the physical presence of the second. We achieve that by introducing a new level of abstraction which will manage all I/O events that take place between a user and a Personal Computer.

- ✓ Design devices (Clients) without a processing unit that are interfacing with the standard peripherals (mouse, keyboard, speakers and screen) only. The lack of a CPU / GPU drastically reduces the energy consumption of the device. A closer look at the results of a paper called "*An Analysis of Power Consumption in a Smartphone*" can validate our hypothesis.

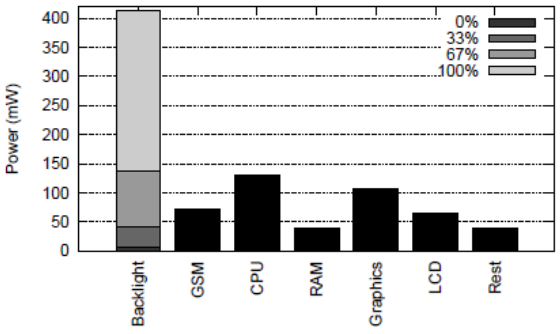


Figure 9: Video playback power breakdown. Aggregate power excluding backlight is 453.5 mW.

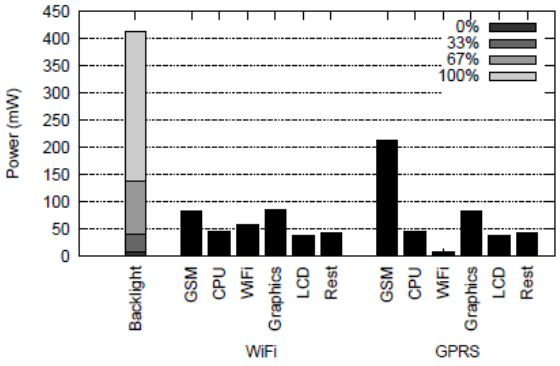


Figure 13: Web browsing average power over WiFi and GPRS. Aggregate power consumption is 352.8 mW for WiFi, and 429.0 mW for GPRS, excluding backlight.

Figure 1: Energy Consumption on Smartphones

- ✓ Simulate the communication of Cloud Clients with Cloud Servers, which will be responsible to transfer all the representation data to the peripherals. The processing power of the connected Clients would be the same as the Cloud Servers.

Our proposed system is a Thin Client designed in a Field - Programmable Gate Array device which allows the user to remotely connect to a PC of his choice, and a C# application that enables this functionality. Computer and FPGA communicate through the Internet and exchange every I/O event that is generated from/for the peripheral devices. Our device is connected with the following peripherals: mouse, keyboard, screen, speakers, and manages the data stream of all four. The FPGA user can remotely access and fully use a PC on the Cloud.

1.3 CONTRIBUTION

Even though we managed to design an operating device that reaches its goals, the whole concept is open for research. We still have to examine whether it is really worth adapting this scheme. Reaching to a positive conclusion will bring up a few new topics which our thesis can contribute to and open up new possibilities for further improvement.

First of all, it explores a new area of technology. Devices with such features do not exist nowadays and it will be interesting to see how the manufactures will adapt to those changes. Also, we should consider new OS for our Cloud Servers, systems which will not be responsible to communicate with peripherals since the I/O will be simulated and transferred through Internet. Finally, that leads to the idea of reestablishing a new network scheme.

1.4 STRUCTURE OF THESIS

[**Chapter** / Description / *Keywords*]

Chapter 1 is an introduction to this thesis. Here are stated the goals of our design along with the contributions it offers.

Introduction, goal, contribution

Chapter 2 examines some background research that is related with our Thesis.

Background, related work

Chapter 3 describes what our design must do. A simple breakdown of our design's functionality alongside with some early insights on how things work.

Simple, brief, functionality, top level

Chapter 4 is where all the details of our work are listed and the algorithms behind every implementation are introduced and explained.

Detail, algorithm, architecture

Chapter 5 is a collection of measurements and diagrams outlining the performance and cost of our design.

Cost, measurement, diagrams table

Chapter 6 sums up every solution we came up to in order to overcome some critical problems we encountered.

Problem, challenge, solution

Chapter 7 reaches a conclusion for our thesis and suggests possible improvements or topics that are open for research.

Conclusion, future work, improvement

Chapter 8 is a simple user guide for everyone who is trying to download our design to an FPGA and see how it actually works.

User guide, download design

2. RELATED WORK

2.1 BACKGROUND

2.1.1 REMOTE CONTROL

Remote control is a term that when used brings to mind the controllers we all use when we are watching TV. The correspondence between the 2 examples is:

	TV	Thesis
Server	TV Device	C# app
Communication Medium	Infrared	Internet
Client	Remote Controller	FPGA

Table 1: A Thesis - TV Analogy

The TV remote controller is used the same way we are using the FPGA, for controlling a distant machine through a medium. In our case we are using the Internet in order to transfer data, while the remote controller communicates with the TV via infrared.

The current technologies used in order to connect two distant PCs, letting the one control the other, but presupposing the use of two PCs. Our aim is to replicate the example above where the TV Remote is not a Television itself! That is why the remote control term is closer to our case than the Remote Desktop term, or Screen Sharing which is used to describe the current technologies.

The term remote desktop refers to a software or operating system feature that allows a personal computer's desktop environment to be run remotely on one system (usually a PC, but the concept applies equally to a server), while being displayed on a separate client device. Remote desktop applications have varying features. Some allow attaching to an existing user's session (i.e., a running desktop) and "remote controlling", either displaying the remote control session or blanking the screen. Taking over a desktop remotely is a form of remote administration.

Remote desktop sharing is accomplished through a common client/server model. The client, or VNC viewer, is installed on a local computer and then connects to the network via a server component, which is installed on a remote computer. In a typical VNC session, all keystrokes and mouse clicks are registered as if the client were actually performing tasks on the end-user machine.

It is clear that the model above fits our needs with one exception. The client device we are planning to use will not be a computing system. It will be a simple controller, a Thin Client, responsible for data representation and I/O management. Other than that, the same concept will be used in our effort to design the software that will allow the remote access to the PC that runs it (VNC). Windows have already designed an application for Remote Access that requires 2 PC's running this OS. Commercial apps have also be designed and are widely used by companies in order to access and repairs distant computers.

2.1.2 STREAMING MEDIA

Streaming media is multimedia that is constantly received by and presented to an end-user while being delivered by a provider. The verb "to stream" refers to the process of delivering media in this manner; the term refers to the delivery method of the medium rather than the medium itself.

Live streaming, which refers to content delivered live over the Internet, requires a form of source media (e.g. a video camera, an audio interface, screen capture software), an encoder to digitize the content, a media publisher, and a content delivery network to distribute and deliver the content.

As I have already mentioned in Chapter 2.1.1, Remote Desktop is also known as Screen Sharing, meaning that the two connected devices provide the same visual data to their users. Since our FPGA will not be a computing system the only way for it to provide the visual data needed is for the Cloud Server to live stream them through the Internet. The content delivered will be the server's frame data, the source media, the server, the encoder and media publisher our VNC software, and finally the content delivery network the Internet.

Many problems arise in the design of a network that supports live streaming and additional research is required in order to examine the efficiency of our proposed design when using the Internet. Leaving the transfer medium aside, we focus on encoding and publishing of the media we are trying to stream.

Commercial and open source applications have already been developed. That, enables users to stream / share their screen and audio content through the Internet. Most of them use Direct X to capture the frame data and encode them using H264 (x264) (AAC for the audio). They can also handle Real Time Streaming constraints while adjusting the encoding and the data rate of the stream. Studying the way these applications operate, was crucial in our effort to design a simple and thin version of this type of software.

2.1.3 FPGA-BASED APPS

The FPGA we are using is equipped with many media ports, such as PS/2 ports, DVI port, audio ports (jacks) and Ethernet port. In order to interface with each of them we needed and thankfully we were able to locate some related example designs from the manufacturer, and some well conducted user guides.

XPS Thin Film Transistor (TFT) Controller was the design that helped us to interface with the DVI port. The component is able to display up to 256k colors but we changed that and now we can properly use the 24-bit RGB channel and get 16,777,216 different combinations. We decided not to use the PLB Bus that was included in order to keep the design cost as low as possible. The most important aid from this component, was the Chrontel CH-7301 Chip initiation.

XAPP58, High Performance DDR2 SDRAM Interface was the example design we used for the sake of interfacing with a RAM Device. All of the initial parameters where explained and that helped us to avoid some pitfalls in the design process. Also, the Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC User Guide was a very long Guide with every detail we could possibly need to get the Ethernet Core working.

Some external sources were taken into consideration while interfacing with the AC97 chip, the Chrontel Chip and the PS/2 ports. LM4550 has a very informative data sheet as the CH-7301 DVI chip does. Finally, credits should be given to *Piotr Foltyn* for the PS/2 core that was used to communicate with the mouse and the keyboard. It was obtained from *OpenCores.org*.

2.2 RELATED DESIGNS

A small research was conducted before the actual design of our device began. We studied many applications and designs to get some tips and hints on how things actually work, besides the theoretical background we had already examined. Some examples are listed below:

i. Hardware

a. The project "*Implementing a Real Time Music Pedal in FPGA*" was further studied. Audio samples are read and processed in real time in this design. We adapted this algorithm along with the Audio Chip Drivers (AC97). In our case the sample source was the Cloud Server and the signal processing part was replaced with a simple buffer that fed the output with the incoming audio stream. Eventually, we found every technical information needed in order to redirect the audio stream from the Ethernet, where it was received, to the output, the AC97 chip. The original driver was found at *javiervalcarce.eu*.

b. FPGA NES System, by Brian Bennett, was examined to better understand the usage of the TFT controller and send visual data to the DVI screen. This design revealed the actual way the DVI Chip should be fed with data and helped us design the way video data were temporarily stored in the FPGA.

c. Implementation Scenario for Teaching Partial Reconfiguration of FPGA was another Paper we studied and actually helped us understand and visualize the way our video stream works. In this case, the FPGA is fed with video data from a file located in a PC, and then sends them to screen using a DVI controller.

d. A user design in Spartan-3E Starter Board was explored while designing the Mouse/Keyboard interface. Useful technical information about the PS/2 protocol were found and were paired with some theoretical background from Computer-Engineering.com.

ii. Software

Fortunately, the software design had fewer pitfalls and obstacles, and it was pretty clear from the beginning. Besides some software for C# components, such as Sockets, Queues etc., one software category was enough for us to acquire all the theoretical background needed.

Open Broadcast Software, which is a free and open source application for recording and live streaming, is a typical example and member of this category. We adapted some of the concepts it uses, though none of the encoding methods.

3. DESCRIPTION OF APPLICATION

3.1 INTRODUCTION

The purpose of this chapter is pretty simple. We are attempting to deconstruct the structure of our design and outline what our proposed system should do. Our system consists of 2 basic elements, the Cloud Server (software) and the Cloud Client (hardware). Our software is responsible to capture the video stream that is produced from the screen and the speakers of the PC that runs on. It also delivers the stream to the Client through the network. Also, it receives all I/O actions from the FPGA and translates them to mouse movement and keyboard strokes. The hardware part is responsible for capturing every mouse and keyboard event that is produced from these peripherals, encodes them and sends them through Internet to the software to simulate them. It also receives the video stream, decodes the data and drives the DVI screen and the native speakers. The following chapter contains the basic functionality of the system.

3.2 SOFTWARE

3.2.1 GUI

The Graphical User Interface was designed using a Metro-Style pattern which is encountered on Windows 8 Applications. It is a single window application, all settings and functionalities scattered across 5 tabs. Upon execution, the user is greeted with the starting screen of the application. Here, the user can launch the application and the software initiates all the components needed.

After that, the main window appears with its five tabs. The first one contains network information about the application. Start Streaming button initiates a polling process using a socket. The software is on standby mode waiting for the Client to send the starting packet. This packet contains identity information about the Clients MAC address, the IP address, and the ports used. These information are used to launch the virtual devices of our Cloud Server. Finally, the Stop Button suspends every software process, clears away any junk data left and moves into standby mode again.

The last tab is called Usage Statistics. Here we calculate some useful informatics that give an insight to the resources consumed by our application. Ending the application, leads to the disposal of all C# object used in our application, and any kind of streaming / communication with / to the Cloud Client is terminated.

3.2.2 PERIPHERAL CONTROL

Our GUI includes 3 more tabs that concern the virtual devices of our Cloud. Let's examine how these devices are controlled in order to use their I/O as the media stream that is transferred to the client.

First, we create a virtual keyboard device as a C# object. The commands issued to this device simulate actual keystrokes with the use of a Windows DLL library, thus enabling us to use this object as a physical keyboard. The virtual mouse operates the same way, only its input is bit more complicated since it is responsible for the cursor placement on the screen. Each of these two devices is controlled by a different module that feeds them with commands received by the Client. Upon the commands arrival to our application, they are placed in a Queue in order to make sure that they are executed in the same order that they arrive. More information about the algorithms used to control these devices are found in Chapter 4.

Moving on, a virtual device for the PC screen is also needed in order to capture its output and stream it to the connected Cloud Client. Our application includes two different controllers that are fed from the virtual screen with data in two different formats. Bitmap or DX Surface. If a Client is connected to our application, these data are divided into packets and then sent through the network. The algorithm which is used to determine the data needed to be captured and transferred, is similar to the MPEG.

Finally, we create another virtual device to control the speakers. The NAudio library is used to capture the audio stream and provides our device with a wave audio stream. Again, we divide the stream into packets and deliver them to the connected Client.

3.2.3 STREAM

A big part of the Server - Client communication is the network management. Although the virtual screen and speakers can handle the packet creation and use the sockets to send data to the Client, mouse and keyboard incoming events need to be decrypted from the UDP packets before being placed in the Queue. Also, a special C# object is used to establish the connection between the two sides, which also includes the needed UDP packet decryption before the application can handle them.

3.3 HARDWARE

3.3.1 HARDWARE NEEDED

The Cloud Client needs to simulate all the I/O of a Personal Computer. An FPGA device is programmed in order to communicate with the Cloud Server, receive the media stream and send all peripheral events to be simulated. The FPGA screen port is DVI and a compatible screen is needed. A set of speakers with a Jack port is also needed, alongside with a PS/2 keyboard device and a PS/2 mouse device. We choose the ML505 Virtex 5 lx110t board for our device which includes all these connectivity ports and comes with a big enough FPGA to fit our hardware design files.

3.3.2 USER INTERFACE

Since our goal is to create a really thin Client we chose not to include a complex user interface. An interface with mouse and keyboard connectivity and visual representation would require many precious resources from our FPGA. To overcome this issue, we hardcoded just the starting screen to indicate that our system is ready to operate. Also, we are using the FPGA push buttons to trigger specific actions such as Reset or Cloud Connection. All connection information (IP address, MAC address, port number) are hardcoded too and need to be changed when the FPGA is connected to a different PC.

3.3.3 FUNCTIONALITY

The hardware design is divided into two main categories. The driver interfaces, which includes deterministic interfaces to the FPGA chips that drive the connectivity ports. The other category includes design modules that manipulate or store the data we need to drive the peripheral devices. A simple list of the available operations would be like this:

- Capture data from the PS/2 ports and compose data packets for each peripheral event.
- Create a frame buffer and store incoming frame data.
- Read frame data from RAM and feed the screen device.
- Create an audio buffer and store incoming audio data.
- Read audio data and feed the speakers device.
- Transfer all outgoing network data to the Cloud Server using a MAC core.
- Receive all ingoing network data and notify the corresponding modules.

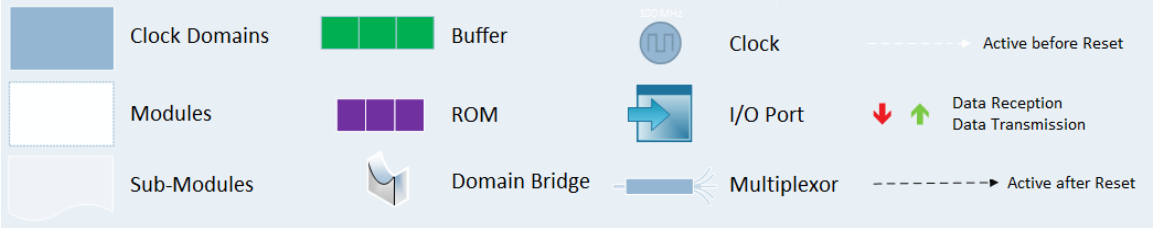
4. DESCRIPTION OF SYSTEM

4.1 HIGH-LEVEL ARCHITECTURE

In this Chapter we present the High-Level data path alongside with High-Level Architecture diagrams. The data path is extensive and includes every data transformation from the moment we capture them to the moment we deliver them and finally simulate them or represent them. The diagrams will be expanded more into later chapters, but they provide a sufficient representation of the module connectivity.

4.1.1 FPGA DESIGN (DATA PATHS)

Top Level Diagram Annotations



Data Paths Annotations

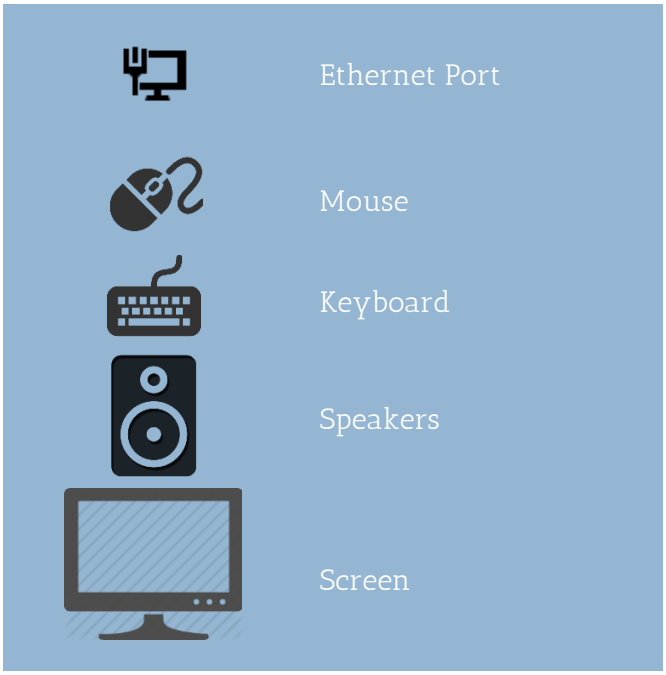


Figure 2: Top Level Hardware Diagram

Figure 3: FPGA Connectivity Data Path

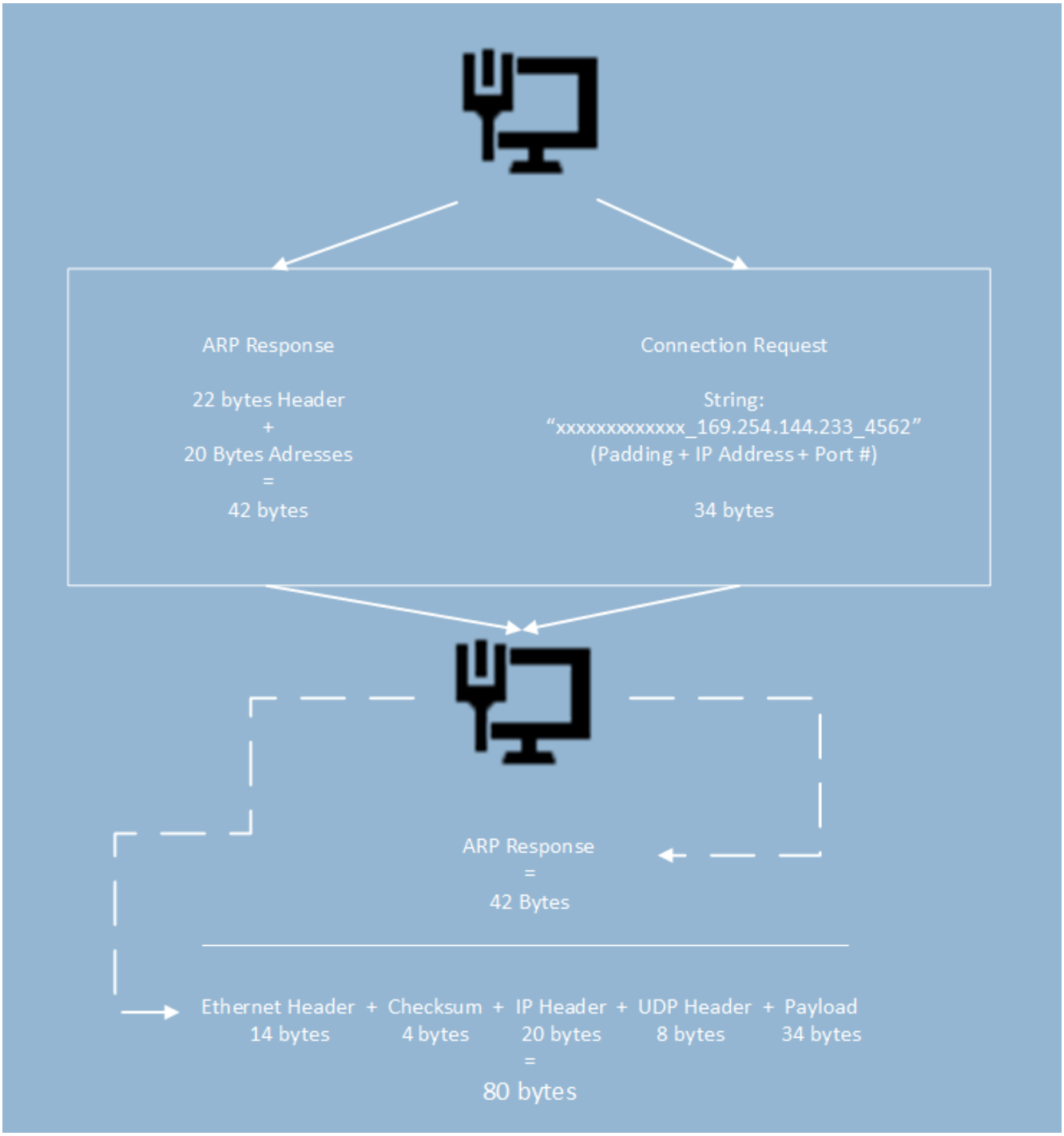


Figure 4: Keyboard (Physical) Data Path

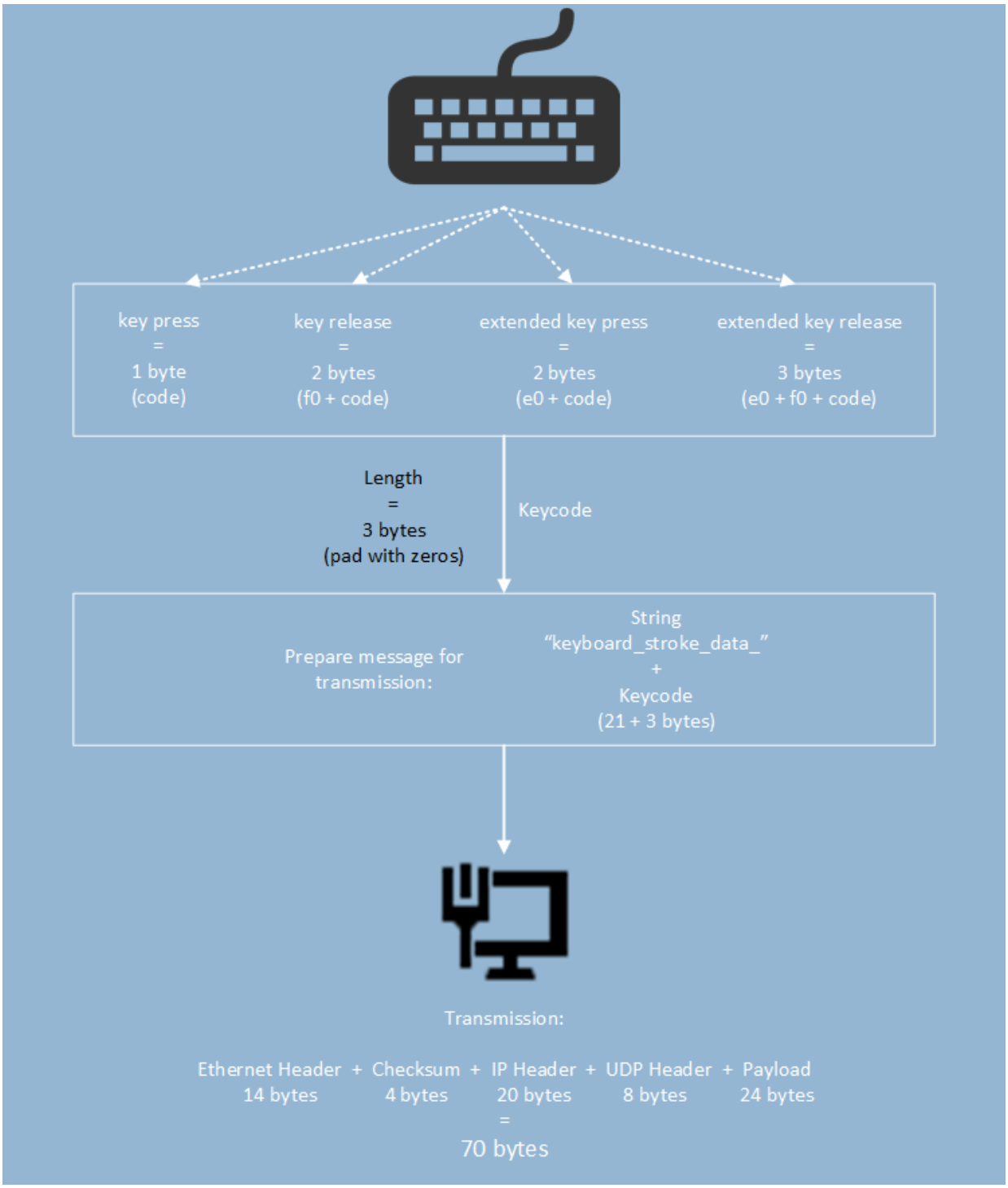


Figure 5: Mouse (Physical) Data Path

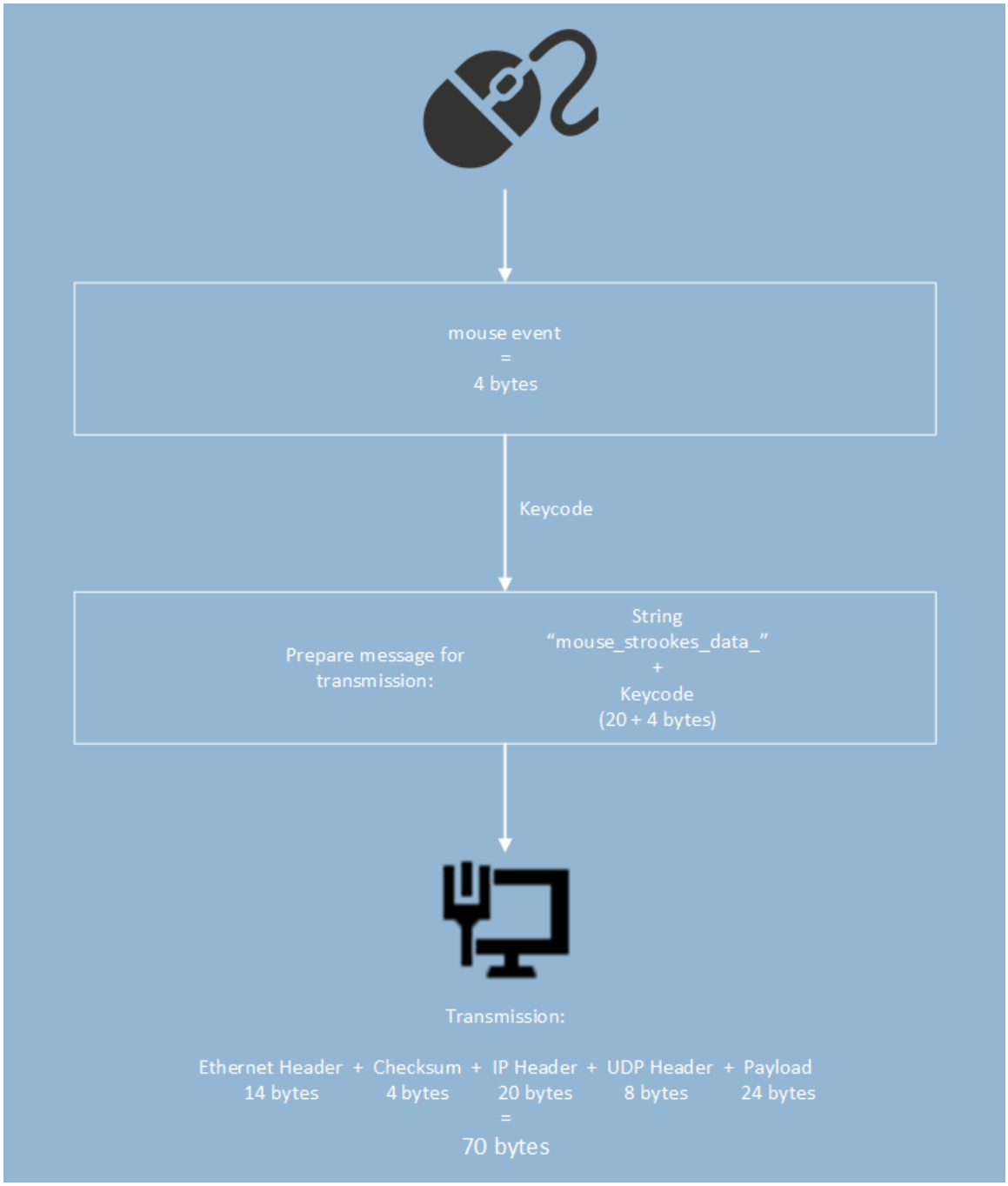


Figure 6: Audio Data Path (FPGA)

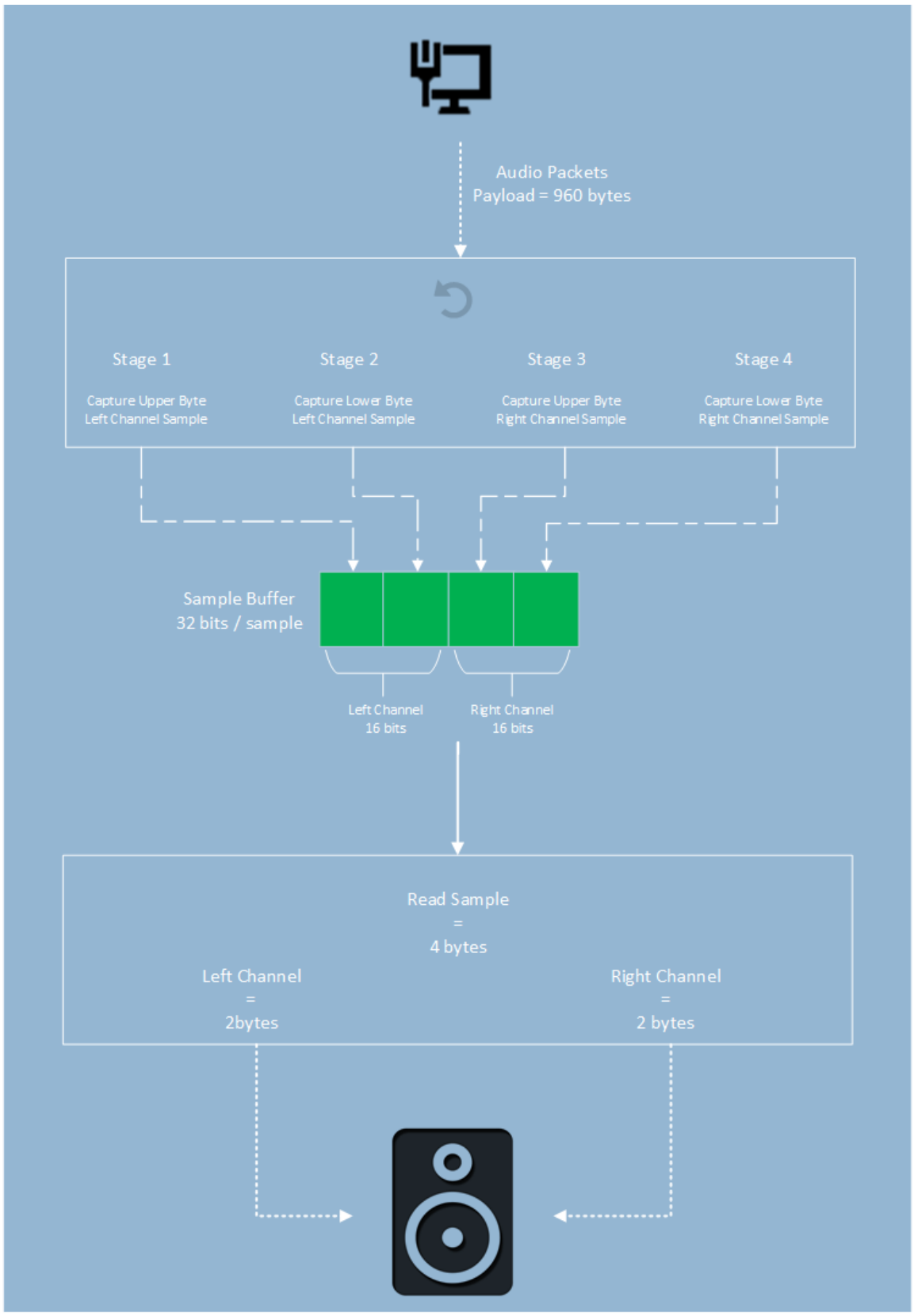
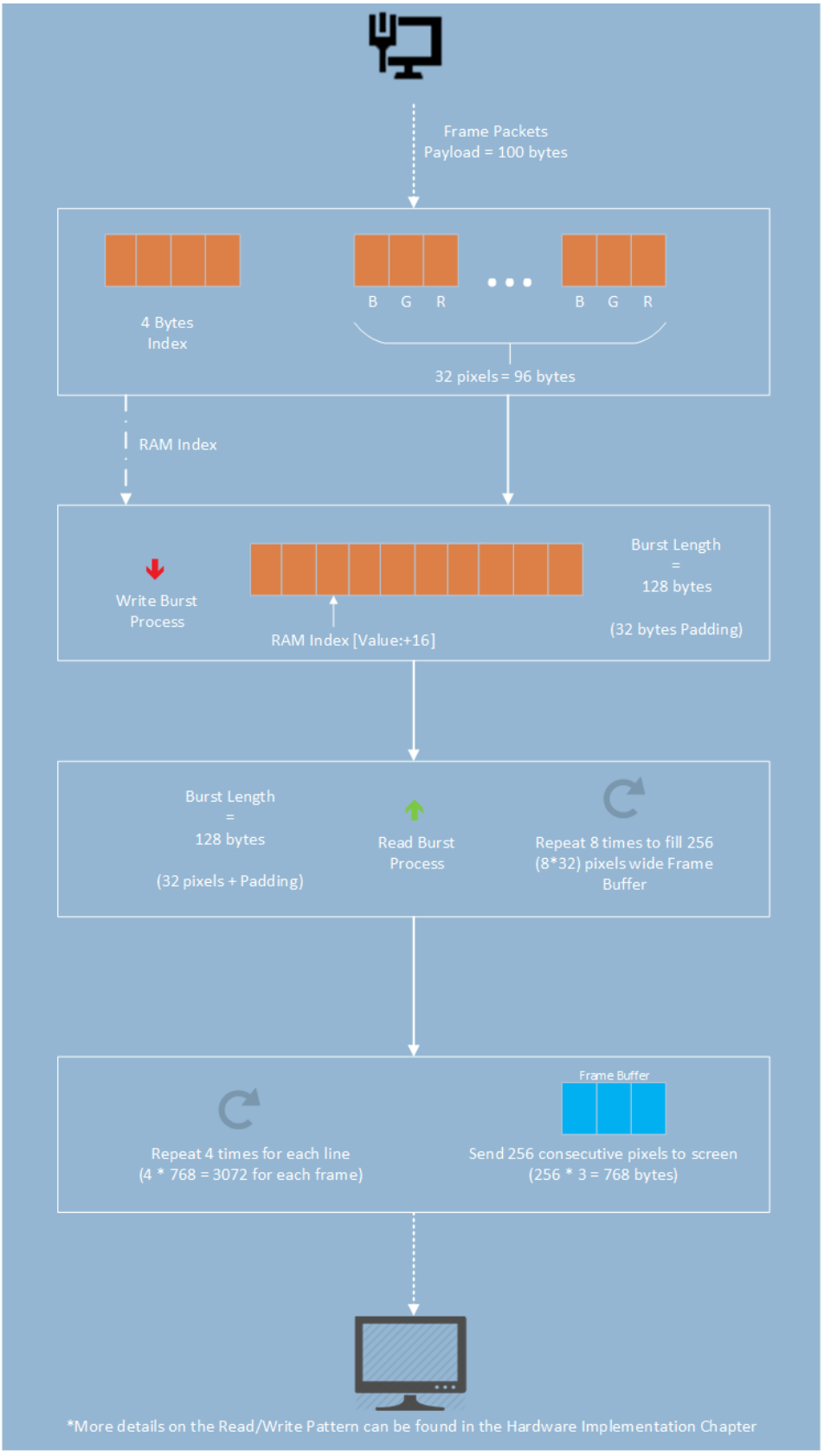


Figure 7: Frame Data Path (FPGA)



4.1.2 C# DESIGN

Figure 8: Audio Data Path (C#)

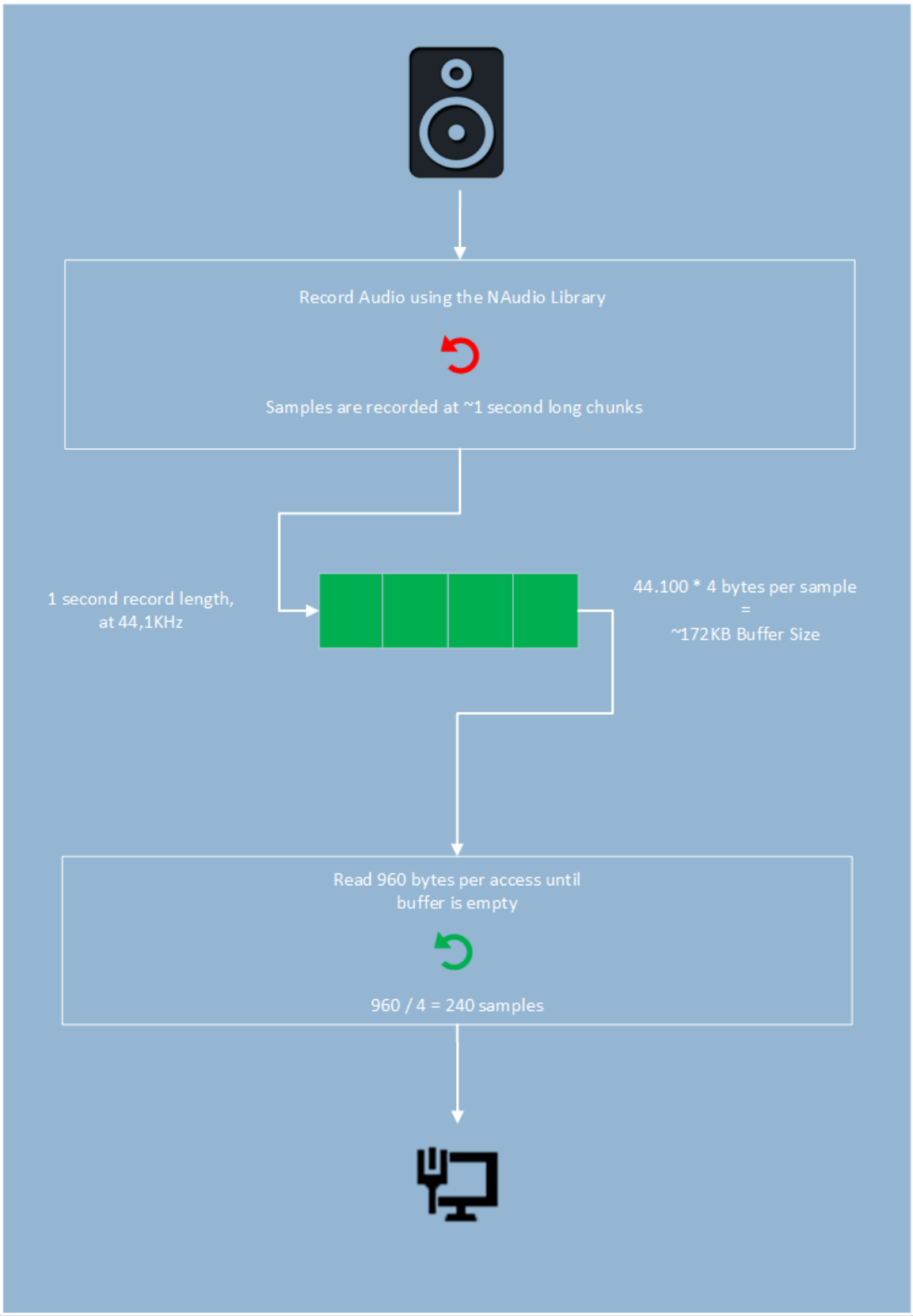
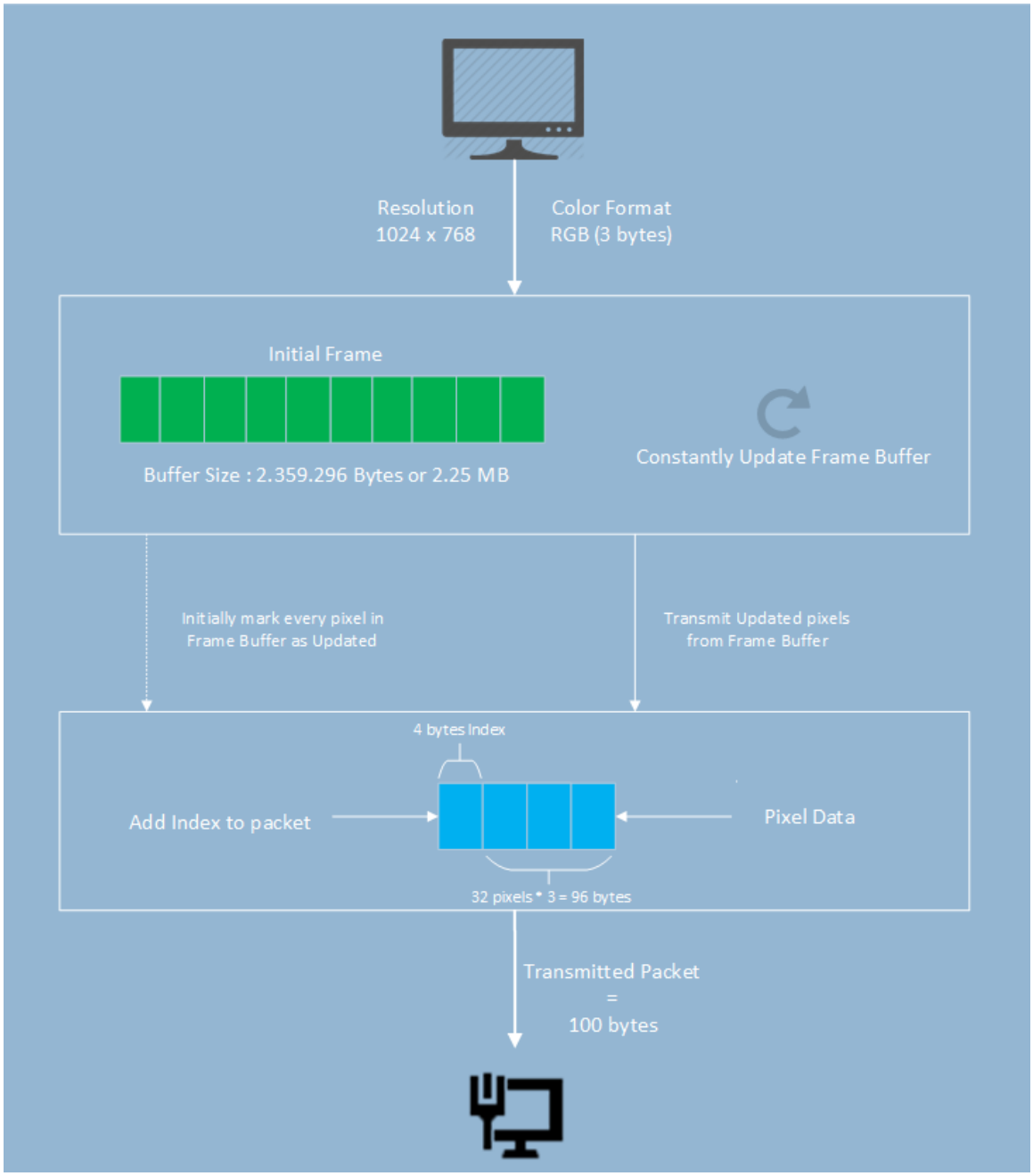
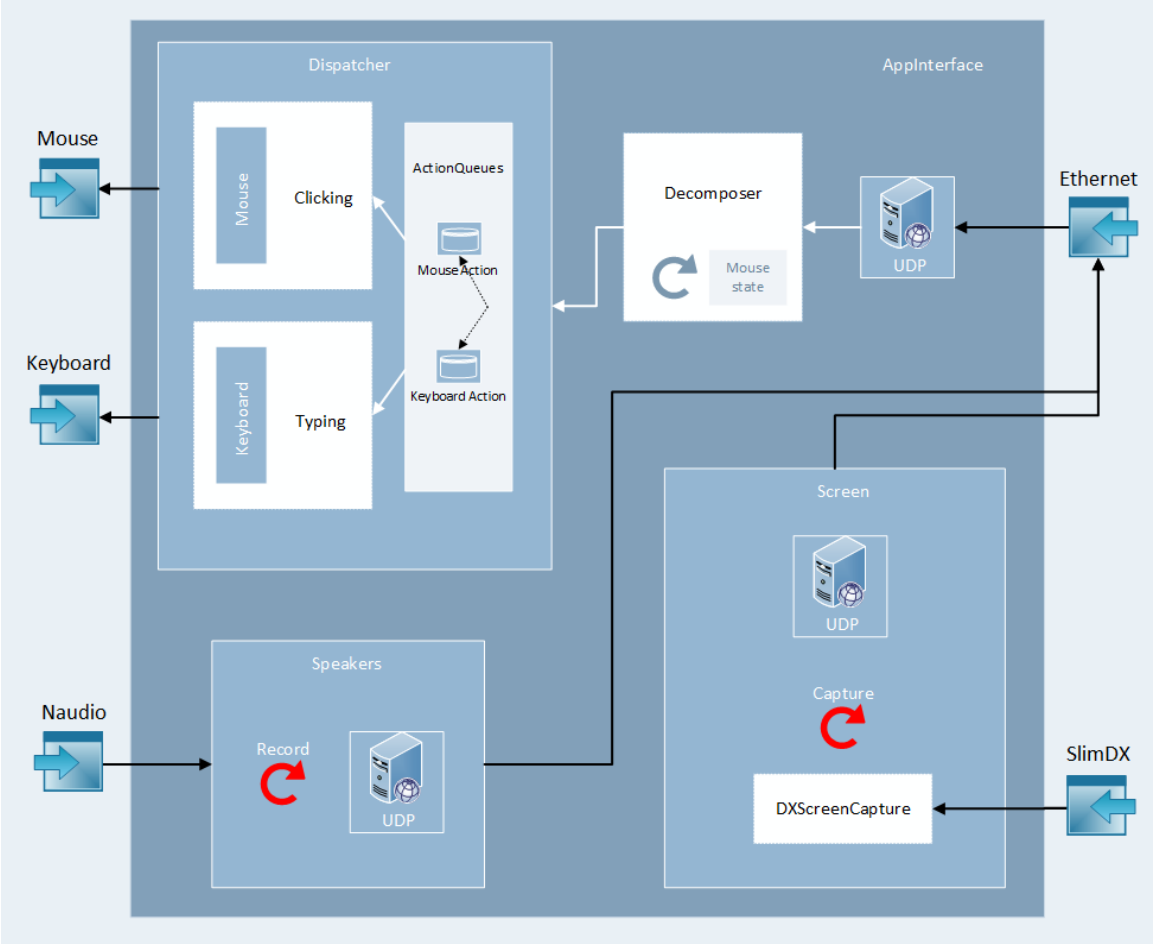


Figure 9: Frame Data Path (C#)

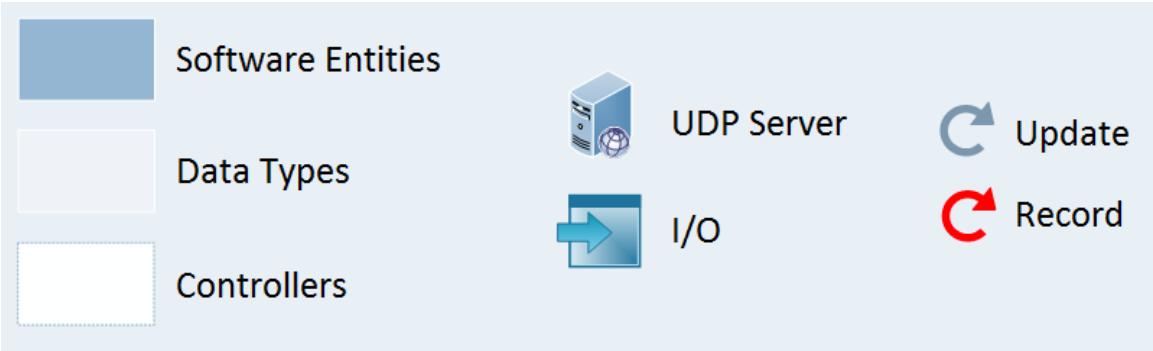


It is clearly shown in the Top Level Diagram that follows up, how the incoming packets from the FPGA are received from the Ethernet port and led through the Decomposer, to the virtual Devices. This Data Path has minor data manipulation (only String tokenization), so no diagrams were made for these cases. The Software Implementation Chapter contains a detailed description of the decoding process that transforms the incoming packets to commands for our virtual devices.

Figure 10: Top Level Software Diagram



Annotations



4.1.3 EXHAUSTIVE FRAME STREAMING DATA PATH

1. Windows 8 Frame Format

a. Bitmap ARGB (1 Byte Alpha Channel + RGB Color).
b. Dimension **1024x768** pixels.
c. So, total # of pixels is 786.432
d. Total # of Bytes for ARGB Format is 3.145.728.
e. Total # of Bytes for RGB Format is **2.359.296.**

2. Capture Screenshot

a. All pixels are captured and stored in a Byte Array [length = **2.359.296**].
b. We completely ignore Alpha channel, and save only RGB Color.
c. The pattern is

Array Position	X	X + 1	X + 2	X + 3	X + 4 …
Color Byte	B	G	R	B	G …

d. The first Byte we read is the Blue Color value at the position (0, 0) which is bottom left.
e. We continue reading the whole line until (0, 1023).
f. Full format of the captured frame

Array Position	0	1	2	3	4	5	6	7	8
Color	B	G	R	B	G	R	B	G	R

g. **2-D** Representation: (Only Showing Blue Color Values)

ByteArray[2356223]	…	…	…	ByteArray[2359293]
…	…	…	…	…
ByteArray[3073]	…	…	…	…
ByteArray[0]	ByteArray[3]	…	….	ByteArray[3070]

Table 2: 2-D Representation of Base Frame (Software)

3. **Packet Creation For Base Frame**

   a. Packet is **100** Bytes long. **4** Bytes are reserved for indexing and **96** Bytes are Pixel/Color data.
   b. **96** / **3** Bytes per pixel for colorization = **32** pixels per packet!
   c. Indexing (first **4** Bytes of the packet)

      Pattern: 0 , 16 , 32 , 48 , 64 , 80 , 96 , 112 …

      Maximum Index Calculation:

      i. We send a total # of 1024 * 768 = 786.432 pixels, in 32 pixels long packets.
      ii. The total # of packets is 24.576.
      iii. Every packet needs 16 indexes in RAM, so we will be using 393.216 indexes.
      iv. The index span is [0:393215] so the Max Index we will be sending is 393.200.

4. **Packet Creation for Streaming**

   a. Read **32** pixels (32*4 Bytes = **128** Bytes) from the freshly captured frame (contains alpha channel).
   b. Compare Color value of these pixels to the **Base Frame**.
   c. If there is a **data mismatch**, transmits the packet as described in *Step 3*.

Software Side Summary

i. Sends 100 Bytes long packets.
ii. Each packet contains Color Values for 32 consecutive pixels.
iii. First 4 bytes of the packet are reserved for RAM Indexing.
iv. Next 96 Bytes are pure information.
v. Representation of the packet:

Byte#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Data	RAM Index				B	G	R	B	G	R	B	G	R	B	G	R	B	G
Pixel					Pixel 0			Pixel 1			Pixel 2			Pixel 3			…	

Table 3: Representation of Transmitted Frame Packet

5. Hardware: Receive Data from Ethernet Port

If **valid** data are being received:

a. Store incoming index from the first **4** received Bytes:

RAM Write Index **[Bits]**	31	30	…	24	23	22	…	16	15	14	…	8	7	6	…	0
Received Index **[Bytes]**	[1st Byte]				[2nd Byte]				[3rd Byte]				[4th Byte]			

b. Store incoming pixel values:

Data **[Bits]**	767	..	760	759	…	752	751	…	744	…	23	…	16	15	…	8	7	…	0
Pixel Color	B			G			R				B			G			R		
Pixel #	0			0			0				31			31			31		

6. RAM Multiplexing

a. Received data bypass the *RAM Initiator Module* and move on to the **Arbitrator**

b. The Arbitrator decides when RAM is available to be written and only then sends the Write Burst Command.

7. Write Burst to RAM

a. Since we are writing **1024** Bits in a single Write Burst, and every word is **64** Bits long, the length of the Burst in words, is **16**. So, we will be needing **16 RAM indexes**.

Write Burst Input	256 Bits	256 Bits	256 Bits	256 Bits
Index	Received	Received + 4	Received + 8	Received + 12
Bit Aligning	[767:640] [639:576 & **Pad**]	[575:448] [447:384 & **Pad**]	[383:256] [255:192 & **Pad**]	[191:64] [63:0 & **Pad**]
Byte Aligning	[16 Bytes] [8 Bytes][**Pad**]	[16 Bytes] [8 Bytes][**Pad**]	[16 Bytes] [8 Bytes][**Pad**]	[16 Bytes] [8 Bytes][**Pad**]

**Pad = 64 Zero Bits*

b. Write Burst receives **4** words per cycle. Let's translate these words from **Byte Aligning** to **Pixel Count**.

Pixel #	1	2	3	4	5	6	7	8	
[16][8][Pad]	[BGR] [BGR] [BGR] [BGR] [BGR] B					GR [BGR] [BGR]			0

c. We can see that **4** words contain **8** pixel values. Since every RAM Write Burst is **16** words long, we will be storing **32** pixels (or one received packet) in total.

Store Frames Summary

i. Every received packet is stored in RAM with a single Write Burst.

ii. Every Write Burst needs **16** RAM indexes, just as many as our software counts for every packet it transmits.

iii. Packets contain **32** pixels, or **96** Bytes, or **768** Bits. Since every RAM Write Burst is **1024** Bits long, we use the Zero Padding technique.

iv. 2-D Representation of the frame as it is stored in the FPGA RAM:

Table 4: Index Distribution of a Frame inside RAM

392.704	…	…	…	…	393.200
…	…	…	…	…	…
512	…	…	…	…	…
0	16	32	48	…	496

Table 5: Frame Coordinates (x,y) inside RAM

(767,0 – 767,31)	…	…	…	(767,992 – 767,1024)
…	…	…	…	…
(1,0 – 1,31)	…	…	…	…
(0,0 – 0,31)	(0,32 – 0,63)	(0,64 – 0,95)	…	(0,992 – 0,1023)

8. DVI Core

a. Receives 24 Bits Color vectors as input.

[23 … 16]	[15 … 8]	[7 … 0]
BLUE	GREEN	RED

b. Pixel drawing pattern in 2-D. **Notice that rows are not read in the same order that our software stores them!**

Y	X					
0	0	1	2	3	…	1023
1	0	1	2	3	…	1023
…	0	1	2	3	…	1023
767	0	1	2	3	…	1023

Table 6: Pixel Drawing Pattern in the FPGA

9. RAM DVI Synchronization

a. Feeds DVI Core with RGB pixels.
b. Signals Frame Reader one Row before the first visible one is being drawn.
c. Asks Frame Reader for new pixel values preemptively (when buffer is ~97%).
d. Uses two **6.144** Bits long buffers.
e. Buffer Refill Methodology:

    i. Each buffer is **6.144** Bits long
    ii. Equals to **768** Bytes.
    iii. Equals to **256** pixel values!
    iv. Since every row has **1.024** pixels, each buffer should be refiled twice!

f. The first pixel we extract from the buffer is the one placed at [**6143**:**6120**] and we move until [**23**:**0**]. **Note:** The order of the Bytes is BGR.

10. Frame Reader

a. After the Base Frame has been built in RAM, the *New Frame* signal from RAM DVI Synch triggers the functionality of this module.
b. From now on, every time the Synching module is asking for pixels, Frame Reader is responsible to deliver them.
c. Frame Reader acquires pixel values from the RAM, by issuing a **RAM Read Burst** command to the Arbitrator.
d. Since we are reading **32** pixels every time we are accessing RAM, we need to repeat this process **8** times in order to deliver **256** pixels to the Synching module.
e. Let's review some time constraints:

    i. **75** MHz domain: The frequency that pixel values are being sent to DVI Core.
    ii. **125** MHz domain: The frequency that RAM operates.

Frequency	Period	Task	Constraint
75 MHz	13.3 ns	Request Pixels from Ram	256 cycles * 13,3 = **~3413 ns**
125 MHz	8 ns	RAM Access	8 cycles * 8 = 64 ns
		Read Latency	18 cycles * 8 = 144 ns
		Max Read Latency	26 cycles = 208 ns
		8 Accesses	8 * 64 = 512 ns
		8 Accesses + Latency	512 + (8*144) = **1664 ns**
		8 Accesses + Max Lat.	512 + (8*208) = **2176 ns**

Table 7: RAM Reading Constraints

f. Index Calculation System

i. The Frame Reader module starts serving the DVI Core as long as it receives the *New Frame* signal.
ii. From that point on, it **keeps track** of the pixels that have already been sent, so it is able to feed the DVI Core correctly.
iii. We have already mentioned that the DVI Core requests pixels to be read in different way than they are stored. So, we need to calculate indexes in a different way in order to access RAM properly.
iv. Index Calculation:

1. We begin with **top row** (**767**, **0**) and read pixels all the way across the line. Since that is the last row we received and stored inside the RAM, our starting index will not be **0**, but **392.704** *(See Table 3)*.
2. We index increment is still **16**.
3. We subtract **512** to move to the next row (e.g. (**767**, **0**) to (**766**, **0**)).

11. Read Burst to RAM

a. Read Burst commands retrieve **1024** Bits or **16** words from the RAM.
b. We have already been storing **32** pixels using **16** words in RAM, so Read Burst command responds with **32** pixels or **1** received packet!
c. Read pattern:

Index	Index + 4	Index + 8	Index + 12
256 bits	256 bits	256 bits	256 bits

d. We know that some of these bits are intentionally filled with Zeros, so every 3 words we read from the RAM, one is ignored. That leaves us with 12 words * 8 Bytes = 96 Bytes or 32 pixels! *(see Section 7)*
e. Decoding pattern: *(How to fill 32 pixels long buffer)*

Index (Bits)	Bits Count	Word Count
[767 : 640]	128	2
[639 : 576]	64	1
Ignore	64	1
[575 : 448]	128	2
[447 : 384]	64	1
Ignore	64	1
[383 : 256]	128	2
[255 : 192]	64	1
Ignore	64	1
[191 : 64]	128	2
[63 : 0]	64	1
Ignore	64	1

Table 8: Reading pixels from RAM

Read Frames Summary

i. We maintain two **256** pixels long buffers, in order to feed the DVI Core.

ii. Frame Reade sends **4** Read Burst commands to fill one buffer.

iii. Frame Reader is responsible to inverse the row reading order by calculating the read index as explained in *Section 10.f*.

iv. Frame Reader filters the unwanted zero's we filled the RAM with in order to align data, and returns only true pixels values.

Dx = Word [] = Valid {} = Ignored			
RAM Access	RAM Access	RAM Access	RAM Access
[D0D1][D2]{D3}	[D4D5][D6]{D7}	[D8D9][D10]{D11}	[D12D13][D14]{D15}

4.2 LOW-LEVEL MODULES

4.2.1 HARDWARE DRIVERS

a. ETHERNET INTERFACE

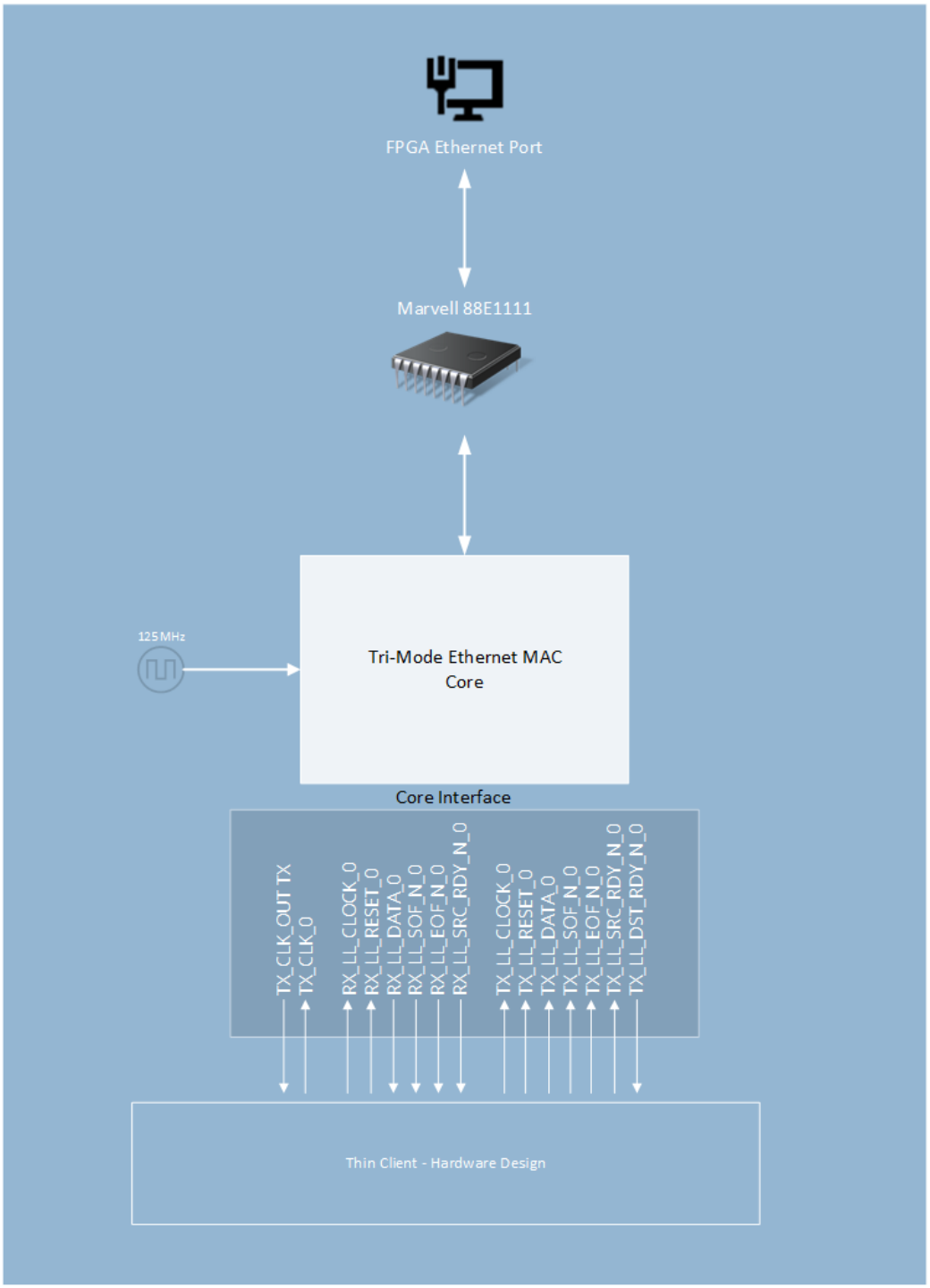


Figure 11: Ethernet Core Interface

The Ethernet module was one of the most important parts of our design since it is the bridge between the two communicating entities, Server and Client. The FGPA board we used, Virtex 5 XUP lx110t came with a Marvell 88E1111 chip embedded, able to provide us a Gigabit Ethernet Interface. This kind of bandwidth is more than enough to transmit the stream from the Cloud to the Client and vice versa.

We are using the Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC Core to interface with the board's chip. Xilinx supports this core pretty well and has published an extensive user guide. The core is able to handle two Ethernet interfaces simultaneously, each of them operating at speed of 10/100/1000 Mb/s. We instantiated only one interface for our design. Other than choosing the desired operating speed, we only made one change on the core's settings:

PHYINITAUTONEG_ENABLE = TRUE

Auto negotiation is an Ethernet procedure by which two connected devices choose common transmission parameters, such as speed, duplex mode, and flow control. In this process, the connected devices first share their capabilities regarding these parameters and then choose the highest performance transmission mode they both support. We chose to enable this functionality in our core in order to avoid manual configuration of settings.

We also provided the core with a 125MHz reference clock which is the maximum clock rate of the cat5 Ethernet cables. The core also provided a lot of interface signals, like frame drop, statistics, delays, pauses etc. but we chose not to use any of them. Using this interface to send simple UDP packets was still too complex, so we implemented two more modules in order to interface with the core in a higher abstraction level.

Core Signal Interface (TX – transmit, RX – receive)	
TX_CLK_OUT	TX Clock output
TX_CLK_0	TX Clock input
RX_LL_CLOCK_0	Reference Clock (125MHz)
RX_LL_RESET_0	Reset RX
RX_LL_DATA_0	RX Data (8 bits)
RX_LL_SOF_N_0	Start of RX Frame
RX_LL_EOF_N_0	End of RX Frame
RX_LL_SRC_RDY_N_0	Source Ready (Core has received a packet)
TX_LL_CLOCK_0	Reference Clock (125MHz)
TX_LL_RESET_0	Reset TX
TX_LL_DATA_0	TX Data (8 bits)
TX_LL_SOF_N_0	Start of TX Frame
TX_LL_EOF_N_0	End of TX Frame
TX_LL_SRC_RDY_N_0	Source Ready
TX_LL_DST_RDY_N_0	Destination Ready (Core is ready to transmit)

Physical pinout interface is not mentioned (the auto generated UCF from the IP Core Generator takes care of that issue). Also, the unused signal are left unmentioned. Finally, it is important to use a buffer (BUFG) in the reference clock, although the generated test bench we are using took care of that too.

b. PS/2 INTERFACE

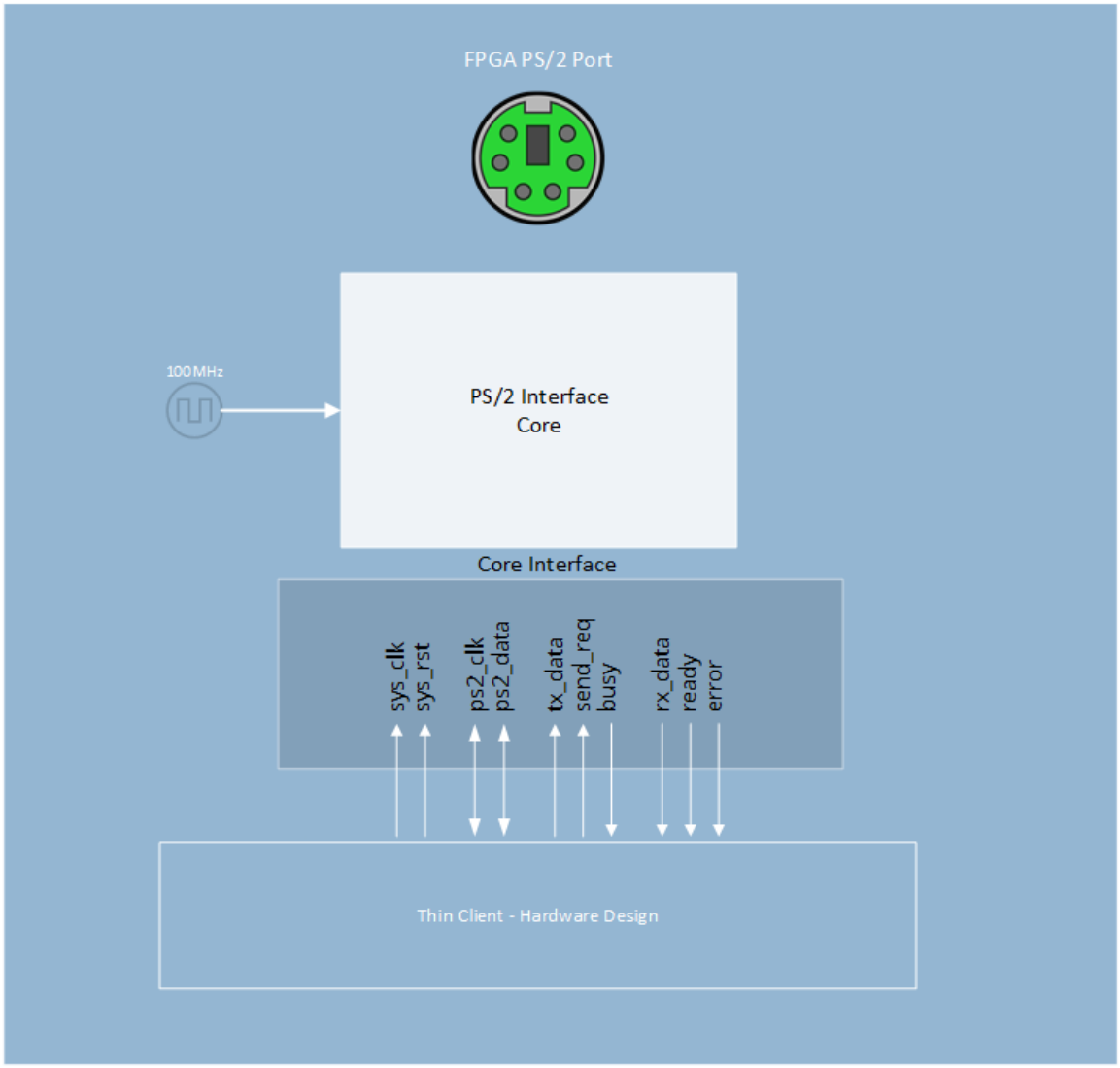


Figure 12: PS/2 Interface

The PS/2 connector is a 6-pin mini-DIN connector used for connecting some keyboards and mice to a PC compatible computer system. Our Virtex-5 Board is equipped with two PS/2 ports, one for keyboard connection and one for mouse. The protocol is old and simple, so we were able to find a module interface pretty easily.

Core Signal Interface	
sys_clk	System Clock (100 MHz onboard clock)
sys_rst	Reset
ps2_clk	Bidirectional PS/2 Clock
ps2_data	Bidirectional PS/2 Data
tx_data	TX Data (8 bits)
send_req	Request Transmission
busy	Core is busy
rx_data	RX Data (8 bits)
ready	RX ready
error	Error in data reception

c. DVI (DIGITAL VIDEO INTERFACE)

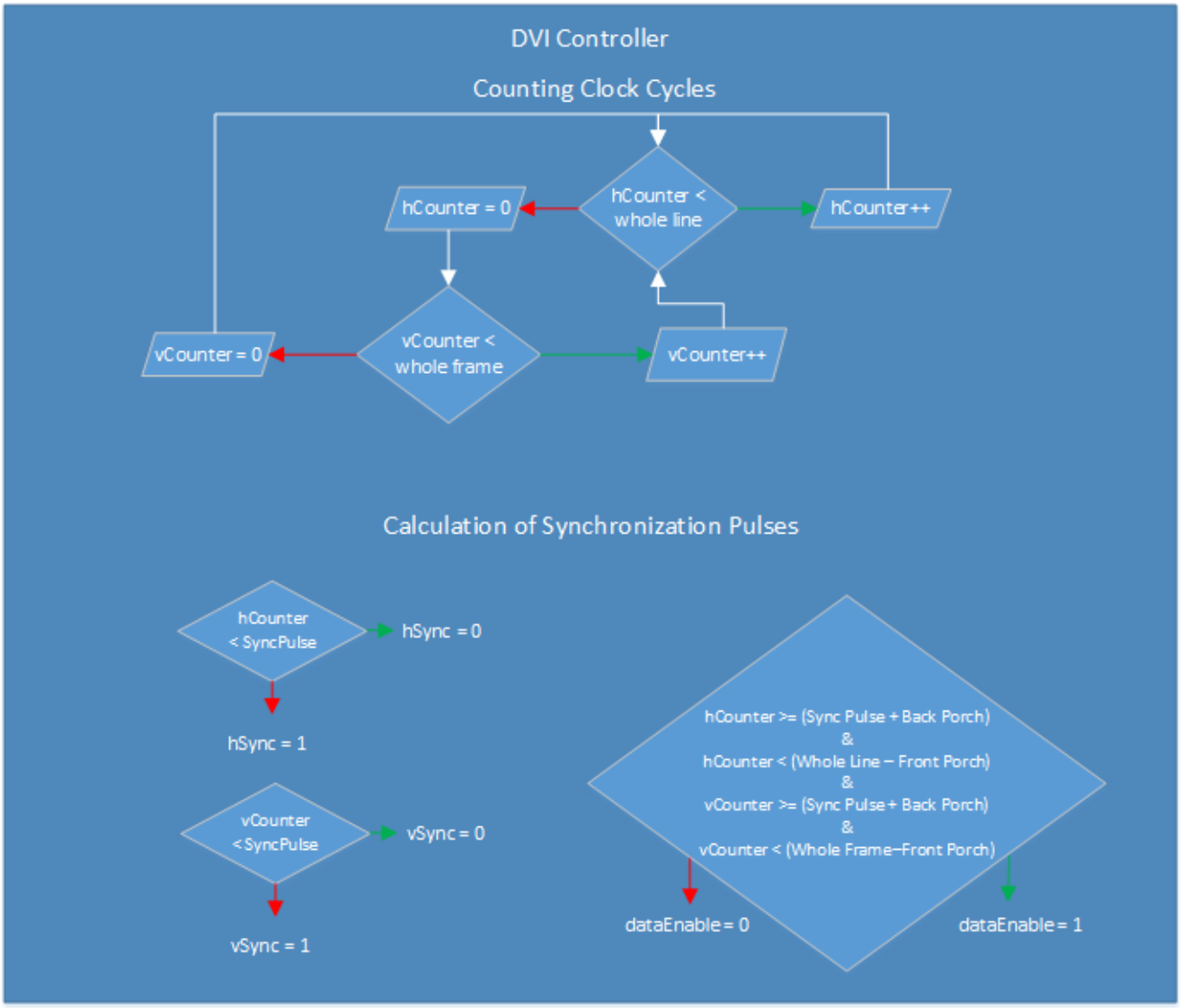


Figure 13: DVI Controller Synchronization

Our Virtex-5 board comes with a DVI port for monitor connection. Digital Visual Interface (DVI) is a video display interface used to connect a video source, such as a display controller to a display device, such as a computer monitor. We use Xilinx's XPS Thin Film Transistor (TFT) Controller in order to interface with the DVI port.

The Chrontel CH-7301 chip is controlling the DVI port on our board so it is the first thing we should communicate with. The FPGA connects with the chip using the **I²C** protocol. **I²C** (Inter-Integrated Circuit), pronounced I-squared-C, is a multi-master, multi-slave, single-ended, serial computer bus, used for attaching low-speed peripherals to computer motherboards and embedded systems.

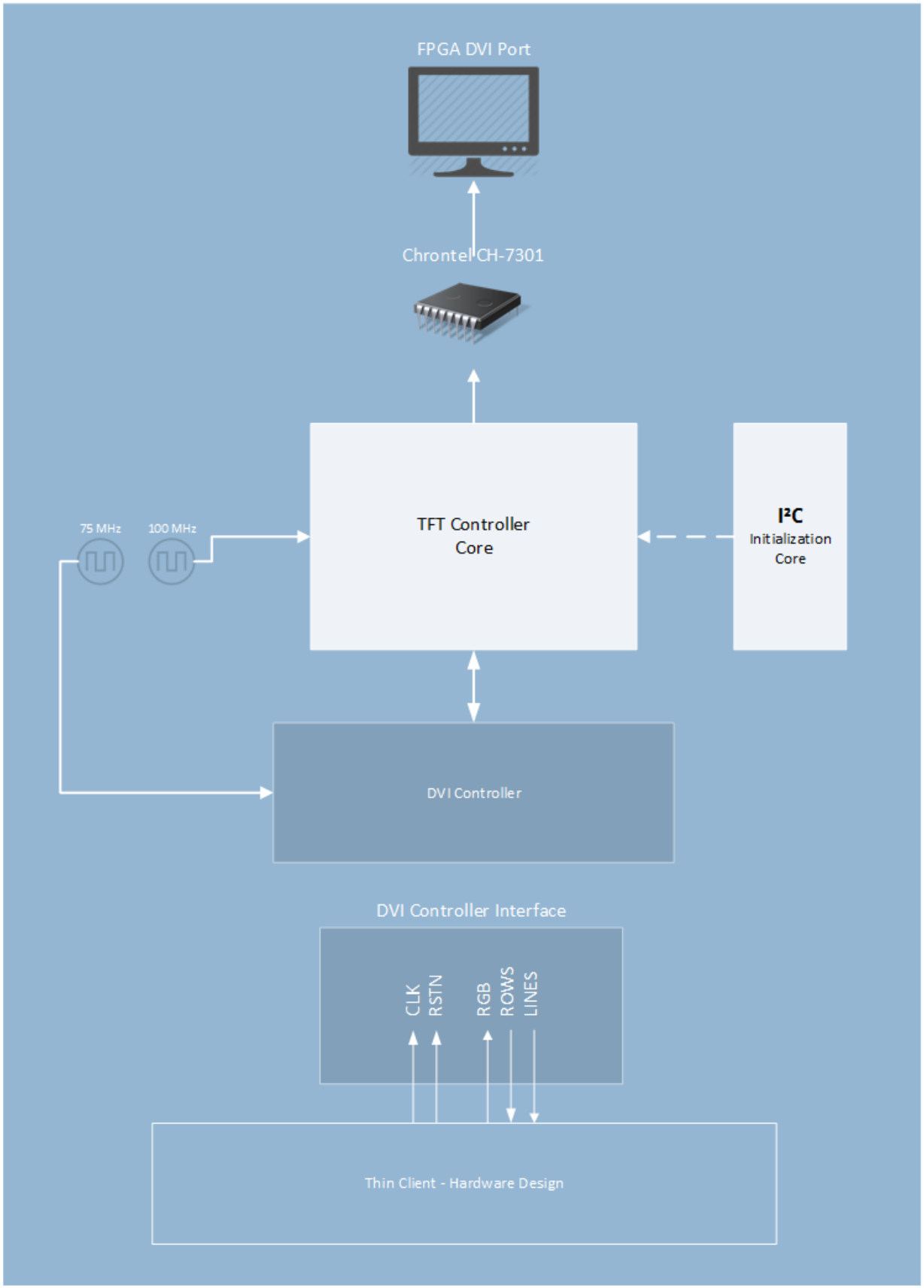


Figure 14: DVI Interface

The XAPP we used included a demo module which could take care of the Chrontel chip settings. First of all, we had to alter the slave address (found on the Chrontel manual, its 7'h76) and the clock rate parameter (100 MHz). Moving on, an embedded FSM is writing the correct register values to the chip in order to initiate its operation. These are:

Register Number (hex)	Value (hex)	Description
49	c0	Power Management, Set DVIP and DVIL to '11'
21	09	DAC Control Register Enable DAC Bypass and HSYNC/VSYNC output
33	06	DVI PLL Charge Pump Control Register Scale Divider
34	26	DVI PLL Supply Control Register
36	a0	DVI PLL Filter Register

Table 9: Chrontel Register Settings

Even though we are using a 75 MHz clock to generate the HSYNC / VSYNC signals, we have been noticing some glitches in the core's operation whilst setting up the PLL register according to the table below. We experimented by alternating the initial settings, once using the ones with <65 MHz and again with >65 MHz. More on Chapter 6.

Register / Frequency	<=65 MHz	>65MHz
33h TPCP	08h	06h
34h TPD	16h	26h
36h TPF	60h	a0h

Moving on, we had to alter the core in order to display 24-bit color. Its previous configuration was for 18-bit color so we changed the pin assignment for the 12-bit DVI Data signals that the core was using to this:

assign dvi_data_a	[0]	GREEN[4]
	[1]	GREEN[5]
	[2]	GREEN[6]
	[3]	GREEN[7]
	[4]	RED[0]
	[5]	RED[1]
	[6]	RED[2]
	[7]	RED[3]
	[8]	RED[4]
	[9]	RED[5]
	[10]	RED[6]
	[11]	RED[7]
assign dvi_data_b	[0]	BLUE[0]
	[1]	BLUE[1]
	[2]	BLUE[2]
	[3]	BLUE[3]
	[4]	BLUE[4]
	[5]	BLUE[5]
	[6]	BLUE[6]
	[7]	BLUE[7]
	[8]	GREEN[0]
	[9]	GREEN[1]
	[10]	GREEN[2]
	[11]	GREEN[3]

Table 10: 18 to 24 bit Color Conversion

More details on table 4 of the Chrontel Manual.

The physical pinout to the Chrontel chip was pretty straightforward. Our board's Master UCF file had all the information needed. Finally, we implemented one more module which took care of the connection of the TFT module with the FPGA and it also generated some much needed DVI signals. Its operation is closely related with the chip interfacing that is why it is included in this chapter.

The system's supported resolution is **1024x768** with **70** Hz refresh rate.

General Timing		
Screen Refresh Rate	70 Hz	
Vertical Refresh Rate	56.475903614458 kHz	
Pixel Frequency	75 MHz	
Horizontal Timing (Line)		
Polarity of horizontal sync pulse is negative.		
Scanline Part	Pixels	Time[µs]
Visible Area	1024	13.653333333333
Front Porch	24	0.32
Sync Pulse	136	1.8133333333333
Back Porch	144	1.92
Whole Line	1328	17.706666666667
Vertical Timing (Frame)		
Polarity of vertical sync pulse is negative.		
Frame Part	Lines	Time[ms]
Visible Area	768	13.59872
Front Porch	3	0.05312
Sync Pulse	6	0.10624
Back Porch	29	0.51349333333333
Whole Line	806	14.271573333333

Table 11: Screen Timings

We created the much needed HSYNC and VSYNC with two simple VHDL counters with a range of **0** to **1327** for horizontal synchronization and **0** to **805** for vertical. Also, we had to create the Data Enable DVI signal according to the timings of the table above. It is important to remember that both the horizontal and vertical conditions need to be met in order to drive the Data Enable signal high.

a. HSYNC Conditions

i. The horizontal pulse is repeated every **1328** clock cycles.
ii. The sync pulse is **136** cycles long.
iii. So the HSYNC is high whenever the horizontal counter is less than **136** and greater than **0**.
iv. The front porch is **24** cycles long so the first Data Enable condition is that the horizontal counter is less than **1328-24=1304**.
v. The back porch is **144** cycles long so the second Data Enable condition is that the horizontal counter is greater than **136+144=280** cycles.

b. VSYNC Conditions

i. The vertical pulse is repeated every **806** clock cycles.
ii. The sync pulse is **6** cycles long.
iii. So the VSYNC is high whenever the vertical counter is less than **6** and greater than **0**.
iv. The front porch is **3** cycles long so the first Data Enable condition is that the vertical counter is less than **806 - 3 = 803**.
v. The back porch is **29** cycles long so the second Data Enable condition is that the vertical counter is greater than **6 + 29 = 35** cycles.

Core Signal Interface (Physical pinout not included)	
CLK	Reference clock or pixel clock (75 MHz)
RSTN	Reset
RGB	24-bit Color Vector for every pixel
ROWS	Row Number (11-bit)
LINES	Line Number (11-bit)

d. AC97 (AUDIO INTERFACE)

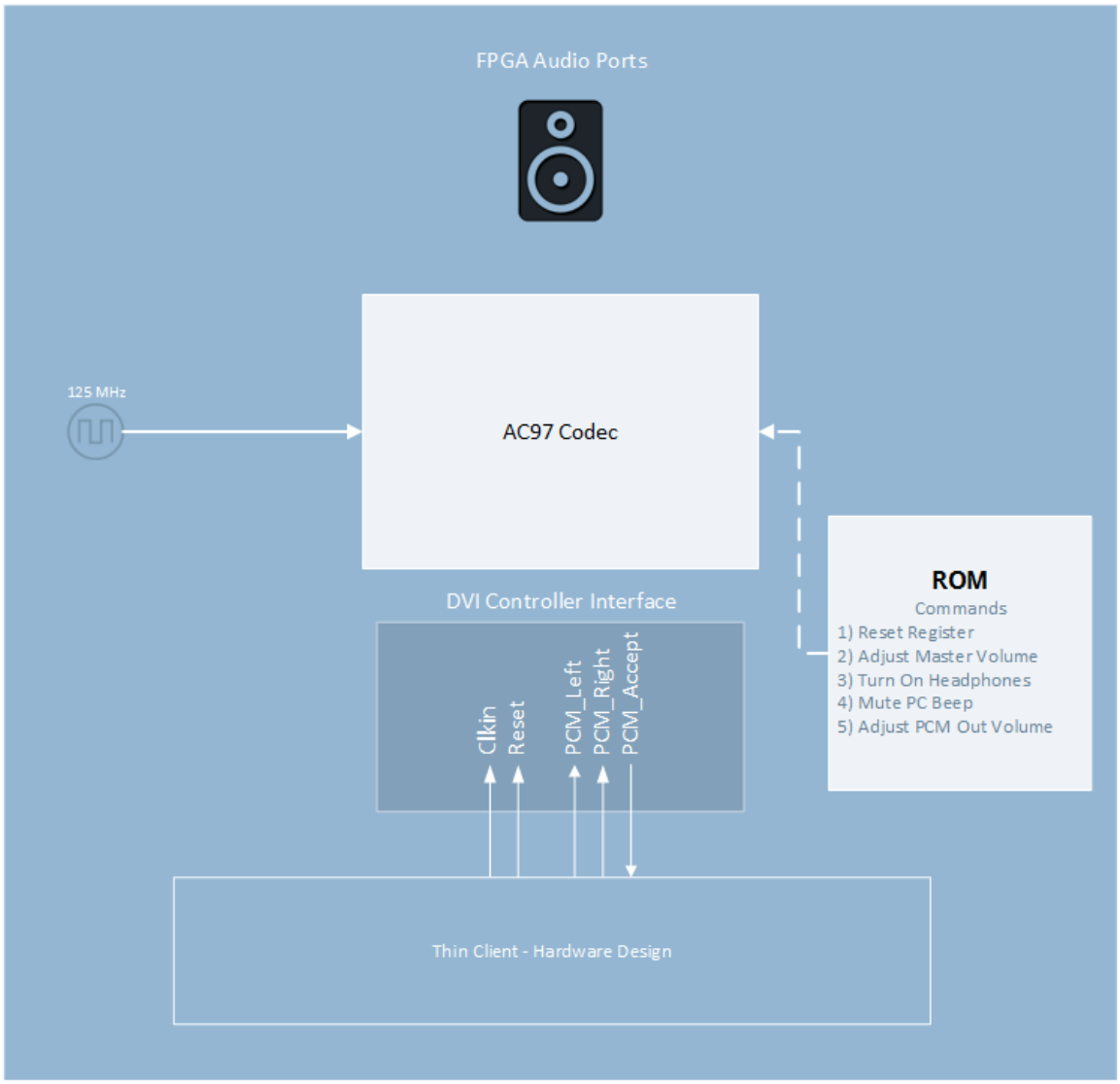


Figure 15: AC97 Interface

AC'97 (Audio Codec '97; also MC'97 for Modem Codec '97) is an audio codec standard developed by Intel Architecture Labs in 1997. The standard was used in motherboards, modems, and sound cards. AC'97 defines a high-quality, 16- or 20-bit audio architecture with surround sound support for the PC. AC'97 supports a 96 kHz sampling rate at 20-bit stereo resolution and a 48 kHz sampling rate at 20-bit stereo resolution for multichannel recording and playback. AC97 defines a maximum of 6 channels of analog audio output. Our board comes with a LM4550 AC'97 Audio Codec with I/O ports. We are using the Line Out port to output the audio stream from the Cloud Server.

We are using the AC'97 Core interface from Mike Wirthlin. First, we need to initiate the core by writing on the chip's registers. The following commands are sent:

X"000000	Write 0x0 to 0x0 (Reset Registers)
X"020808	Write 0x0808 to 0x2 (Master Volume 0db gain)
X"040808	Write 0x0808 to 0x4 (Headphone Volume)
X"0a8000	Write 0x8000 to 0xa (Mute PC Beep)
X"180808	Write 0x0808 to 0x18 PCM Out Volume (Amp out line)
X"208000	Write 0x8000 to 0x20 (General purpose PCM Out Path = 1, 3D bypassed)

Table 12: AC97 Initiation Commands

Core Signal Interface (Physical pinout not included)	
Clkin	125 MHz Reference Clock
Reset	Reset
PCM_Playback_Left	16-bit Left Channel
PCM_Playback_Right	16-bit Right Channel
PCM_Playback_Accept	Core is ready to read the 2 channel sample

The physical pinout was easily assigned with the help of Virtex-5 Master UCF File.

e. RAM (MEMORY INTERFACE)

Xilinx's Memory Interface Generator (MIG) is the tool we are using to interface the board's DDR2 SDRAM. DDR2 SDRAM interfaces are source-synchronous and double data rate. They transfer data on both edges of the clock cycle.

The MIG tool offers the option to implement more than one controllers for the SDRAM but that would make the design too complex for no reason, so we chose to implement only one. Since the RAM will be fed with data from the Ethernet MAC modules it was convenient to choose the same operating frequency for both. That's why we are choosing the reference clock to be set at **125** MHz frequency, or 8000ps period.

The memory type of our RAM chip is SODIMM, so we chose that on the settings along with the specific memory part, in our case the MT4HTF326HY-53E. The data width is **64** bits. Also, we disabled the error correction code (ECC) and the Data Masking option to make the component as much lightweight as possible. The burst length is **4** and the burst type sequential, which are the default settings. We also chose the default settings for the CAS latency, output drive strength, RTT – ODT and Additive latency.

Finally, we disabled the Debug signal and the PLL option. This is important because we had to provide the clocks to the design, so we were obliged to design them too. No other change was made to the Generator Guide. The following table describes the signal interface.

Core Signal Interface (Physical pinout not included)	
clk0	125 MHz Reference Clock
clk90	90 Degrees Phase Shift
clkdiv0	Clock Divided
clk200	200 MHz Clock
sys_rst_n	Negative Reset Input
phy_init_done	RAM Initialization Completed
clk0_tb	Clock Output for the Design
rst0_tb	Reset Output for the Design
app_wdf_afull	Write Data FIFO Almost Full
app_af_afull	Address FIFO Almost Full
rd_data_valid	Read Data Valid
app_wdf_wren	Write Data FIFO Write Enable
app_af_wren	Address FIFO Write Enable
app_af_addr	Address
app_af_cmd	Command
rd_data_fifo_out	Read Data
app_wdf_data	Write Data FIFO Input

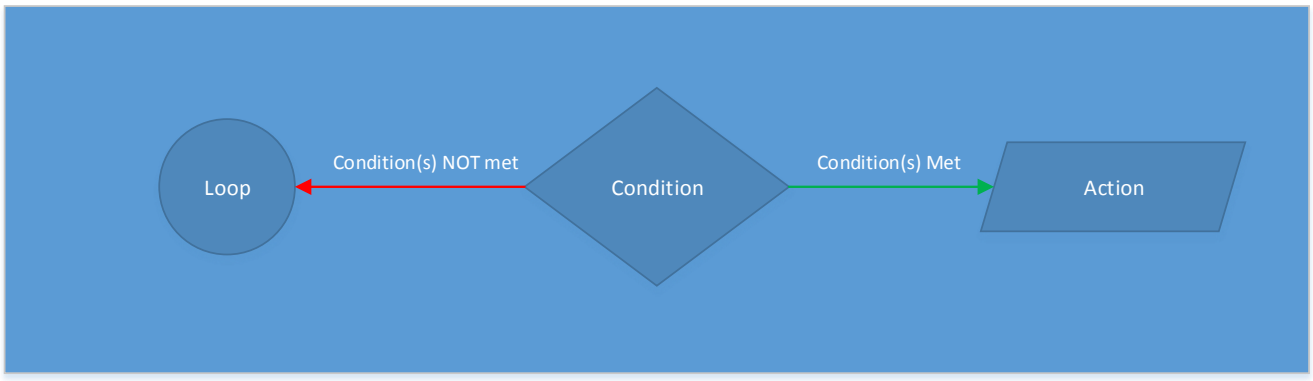
4.2.2 HARDWARE IMPLEMENTATION

This chapter will be extensive, since we will be breaking down the entire design to examine how our proposed system is working and what algorithms we are using to overcome our problems.

We could say that the main purpose of this hardware design is to connect all the interfaces we described in the above chapter and let the data flow and transform through them until they feed, or be created by, the peripheral devices.

The Ethernet port is the only way for the Cloud's streaming data to flow in and out of the design (peripherals included), so we will begin by describing the modules that help us interact with the Ethernet core.

Annotations for the following Flow Charts



- TX DATA

The purpose of this module is to provide us a simple interface to communicate with the Ethernet Core and feed it with data to be transmitted. These data can be either in the form of UDP packet, or ARP packets.

If one of the peripherals triggers this module, then its data are transferred with the help of the Ethernet Core to the Cloud Server. Before these data are delivered to the Core, the module attaches all the information needed in order the packet to be recognized as a UDP. All these information are hard-coded to the module, alongside with the destination port number, target IP address, target MAC address, source port number, source IP address and source MAC address.

We mentioned before that this module is also used to transmit ARP packets. When a node on the network graph tries to communicate with a specific address (IP or MAC) it checks its ARP tables to find an IP-to-MAC address assignment. If it fails to locate such an assignment, it broadcasts an ARP packet to all the nodes of the network. The node that bares the IP or MAC address that the ARP packet is targeting, has to respond to this packet in order to be recognized by the others. When our Cloud Server tries to send the first UDP packet to the Client, our FPGA is not listed in its ARP tables, so it broadcasts an ARP packet. The RX Data module of our design "catches" these packets, and notifies the TX Data module to respond to this ARP request.

To sum up, this module has an ARP interface which can be triggered by the RX Data module. In this event, it sends an ARP response packet to the original sender containing the IP and MAC address information. The second Flow Chart explains how this process is encapsulated in the main FSM, which comes first.

[An embedded module is responsible for the IP Header Checksum. This module was written by Joel Williams, and takes 5 32-bit words as input. The generated checksum appears on the 16-bit output port 2 cycles after the last word was inserted].

Figure 16: TX Data Flow Chart

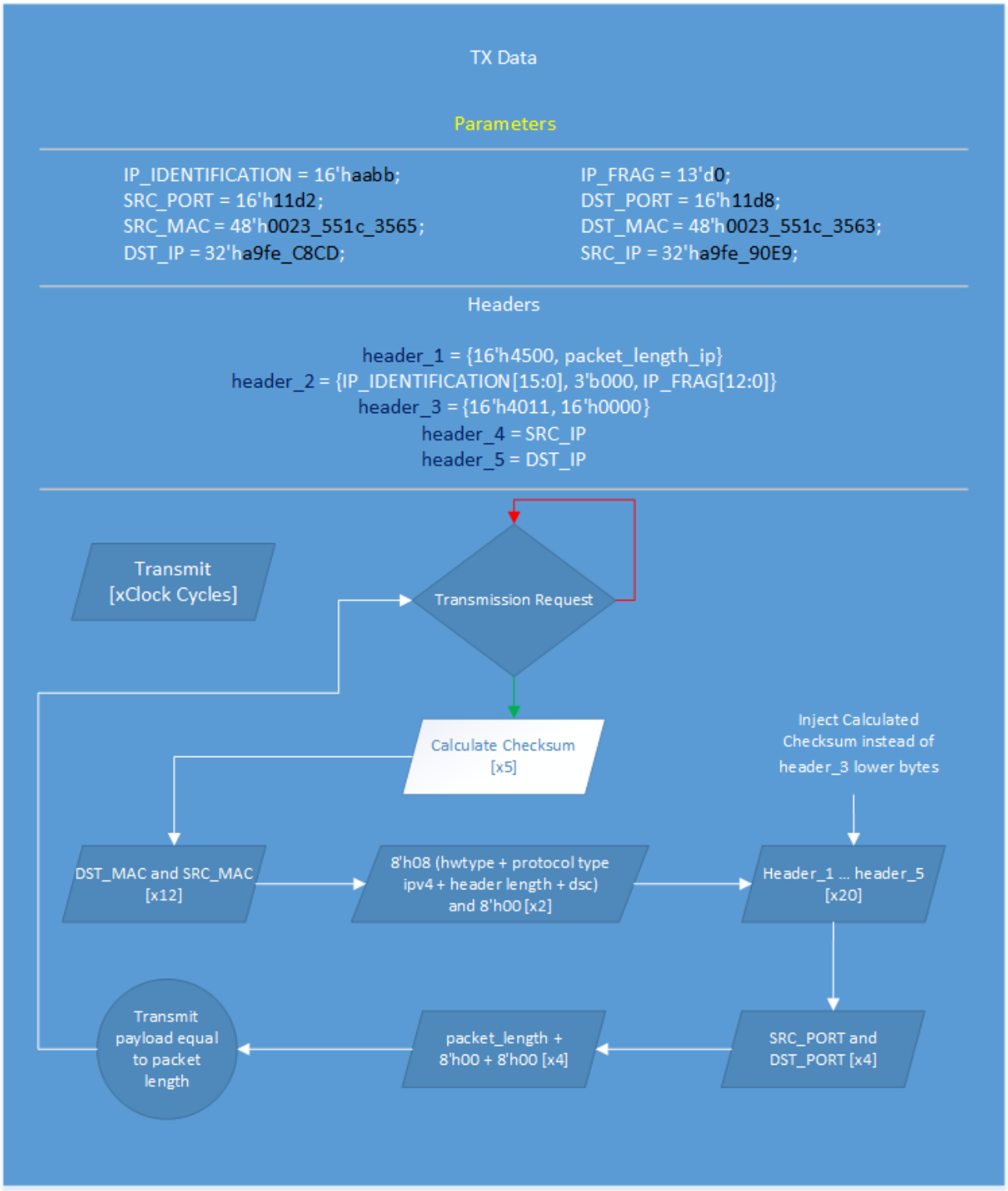
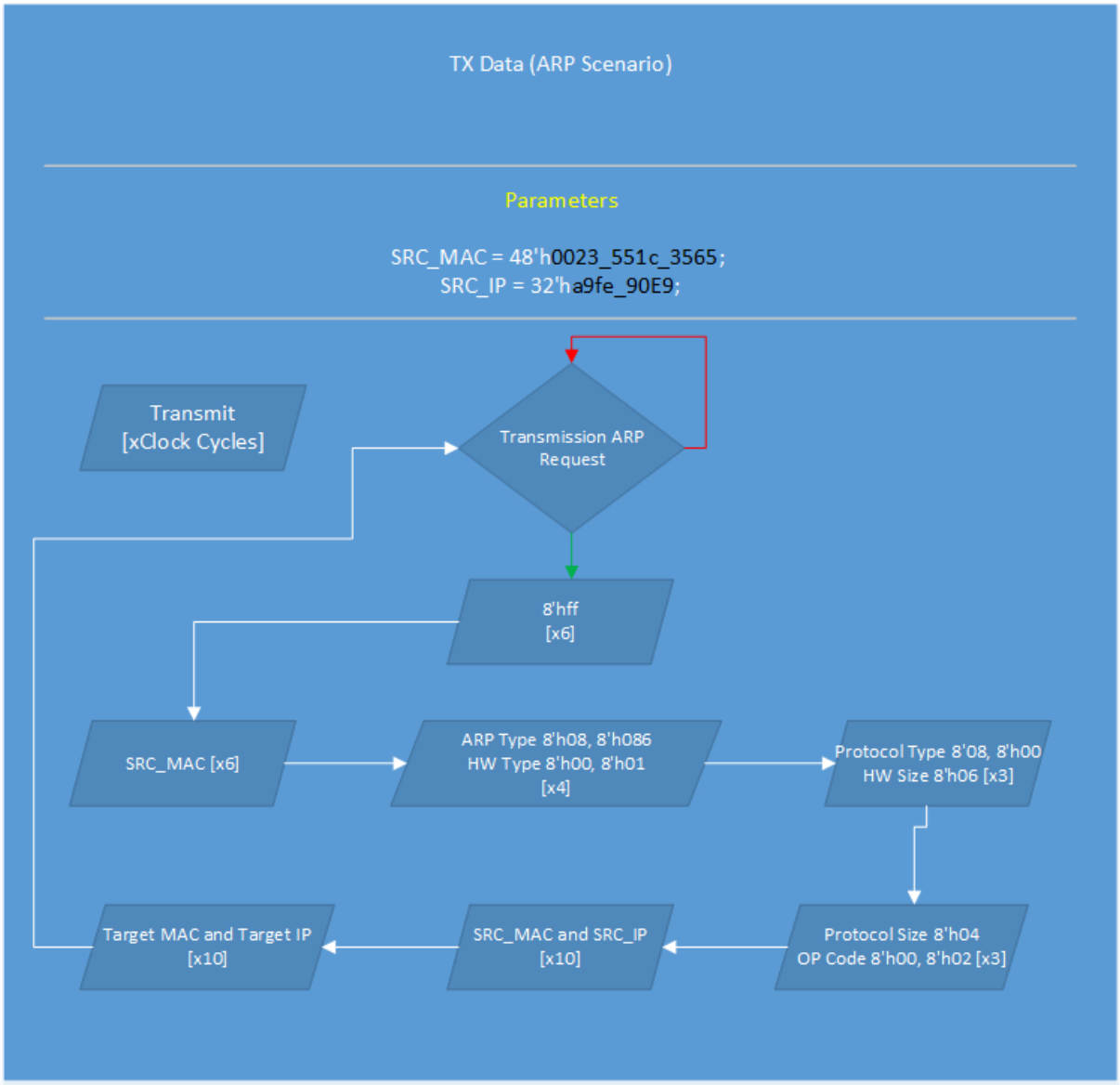


Figure 17: TX Data (ARP Scenario)



- RX DATA

This module monitors the Ethernet Core RX interface for incoming packets. It filters all incoming traffic and saves only the incoming packets with same IP and MAC address that we have hardcoded inside. There are two types of packets that this module can receive, ARP packets and stream packets.

ARP packets are explained in the TX Data module. The only thing we need to add here, is that we dispose all the information in these packets except for the address we need to respond to. When those are fully received, we pass them to the transmitting module and notify it to begin the ARP response.

Stream packets are categorized into frame and audio packets. We have made a convention to use 2 different port numbers in order to identify the nature of the incoming data. In the process of receiving the UDP packet it is easy for us to identify the port number and notify the corresponding module that incoming data are going to arrive to the Client.

To sum up, this module just filters the incoming traffic and notifies other modules to handle the incoming data. Neither it saves data nor does it perform any kind of calculations. The following Flow Chart explains its functionality.

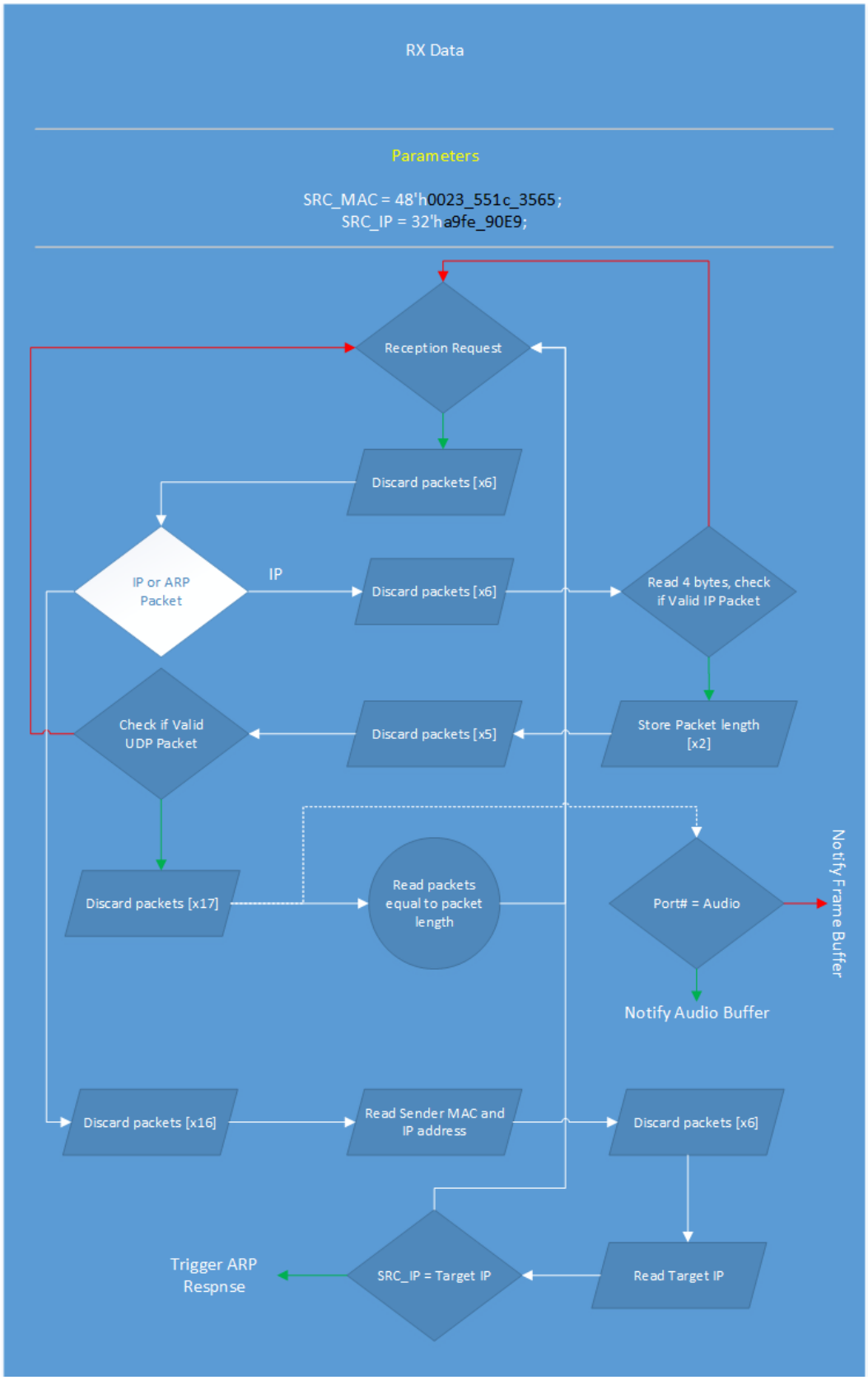


Figure 18: RX Data Flow Chart

- DATA TX

In our effort to reduce the overall complexity of the modules, we decided to divide the packet transmission workload into two different modules. This modules acts as an intermediate between the peripheral modules and the TX Data. Modules that wish to transmit data to the Cloud Server, trigger this module which adds some special prefixes to the data before it passes them to the transmission module.

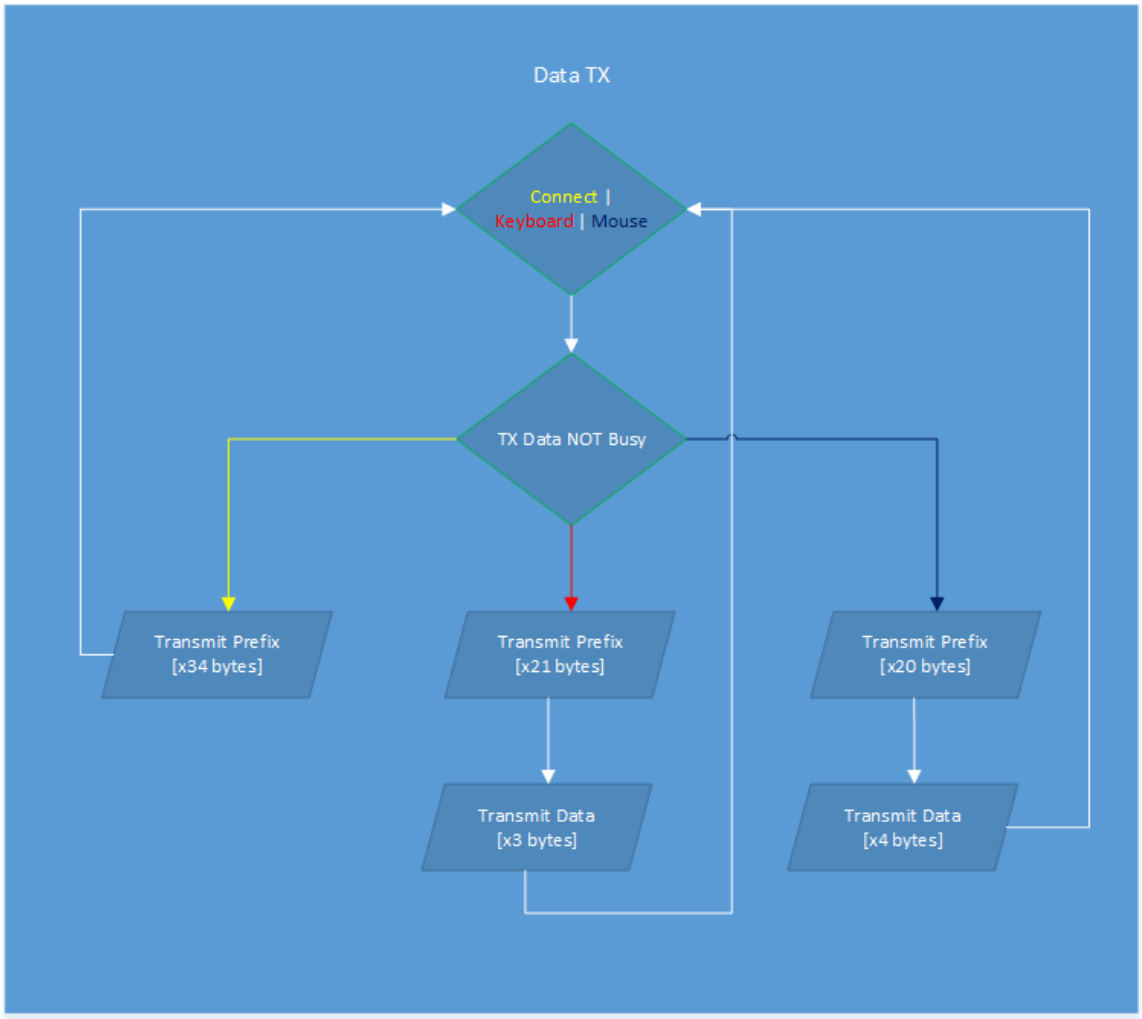


Figure 19: Data TX Flow Chart

This module is used in 3 cases:

Cause	Prefix
Transmit Keyboard events	karajizzferis_169.254.144.233_4562
Transmit Mouse events	mouse_strookes_data_
Transmit Connection Packet	keyboard_stroke_data_

This module withholds any activity until the connection packet is successfully transmitted to the Cloud Server. Also, UDP packets have a minimum data length because the IP packet will additionally be wrapped in a MAC packet (14 byte header + 4 byte CRC) which will be embedded in an Ethernet frame (8 byte preamble sequence). This adds 26 bytes of data to the IP packet. Sending the peripheral data without the prefixes causes the core to fill the packet with "trash data" in order to reach the minimum packet length.

Moving on, we describe the modules responsible for the data flow from the peripherals, mouse and keyboard, to the data transmission module. We will briefly explain the way we capture the peripheral events through the PS/2 ports and the way we encode them before the transmission.

So far, we have described the PS/2 interface that we are using. An instance of this "driver" is used in the modules of keyboard and mouse, alongside with a controller, simulating a virtual device in our design.

- KEYBOARD

This module is the virtual keyboard of our design. It includes a PS/2 interface to read/write data from the port, a controller that eventually acts as the virtual device, and finally a component to monitor this interface and notify the controller for incoming events.

The monitor component is a simple edge detector, so we continue our breakdown with the controller. Every action from the physical keyboard is transmitted through the PS/2 port to our interface, and then out of the component to the transmission module. Our controller's job is to reset the keyboard and monitor the incoming traffic for specific actions.

To reset the keyboard, the controller sends the reset command through the PS/2 port. After that, it filters all incoming traffic in search for 3 specific key codes. Caps Lock, Num Lock and Scroll Lock. After either of these codes is received, we have to send a command to the keyboard to light up the indication LED. See the diagram below for details:

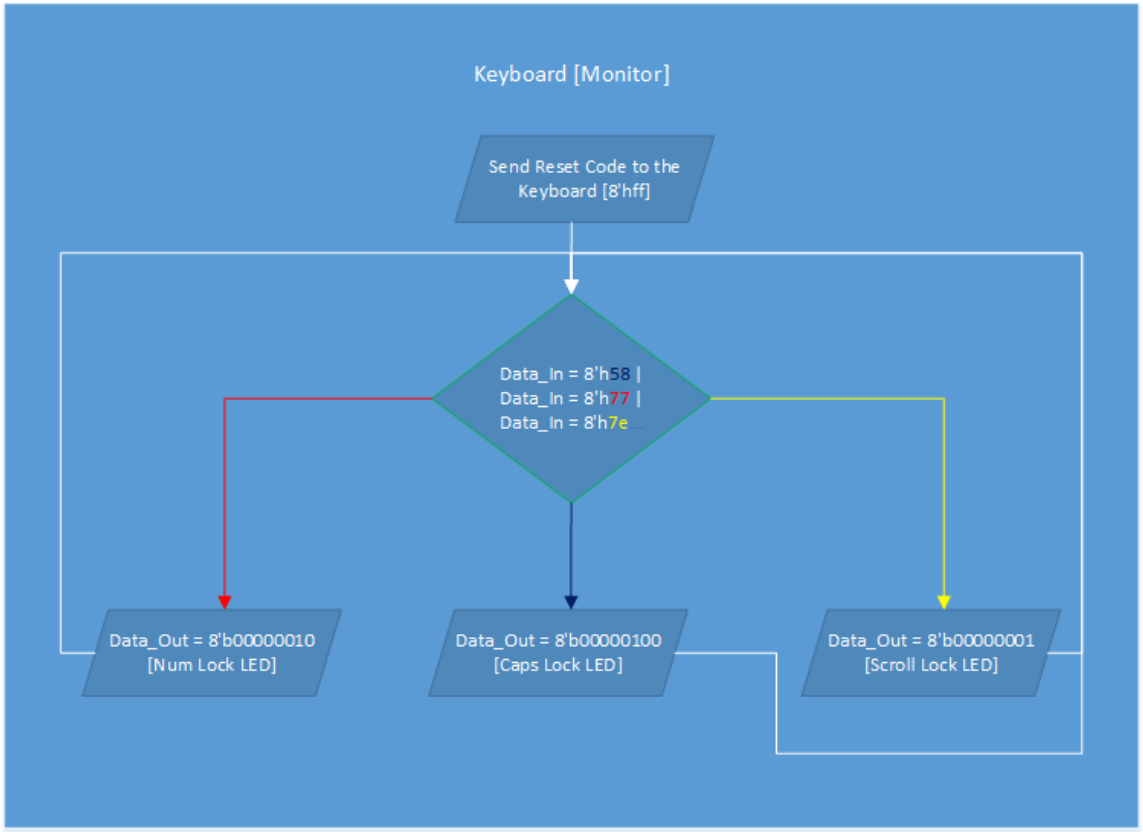


Figure 20: Keyboard Flow Chart

- MOUSE

This module is the virtual mouse of our proposed system. It also includes a PS/2 interface, a controller and an edge detector just like the keyboard module. We also bypass every transmitted action from the physical device to the transmission module, so the only task of our controller is to reset the device.

This task is trickier than the single reset command we send to the keyboard. We have to wait the device to pass the self-test, read the device ID and finally write a number of registers to get the mouse working. The diagram below explains the FSM we use to achieve this.

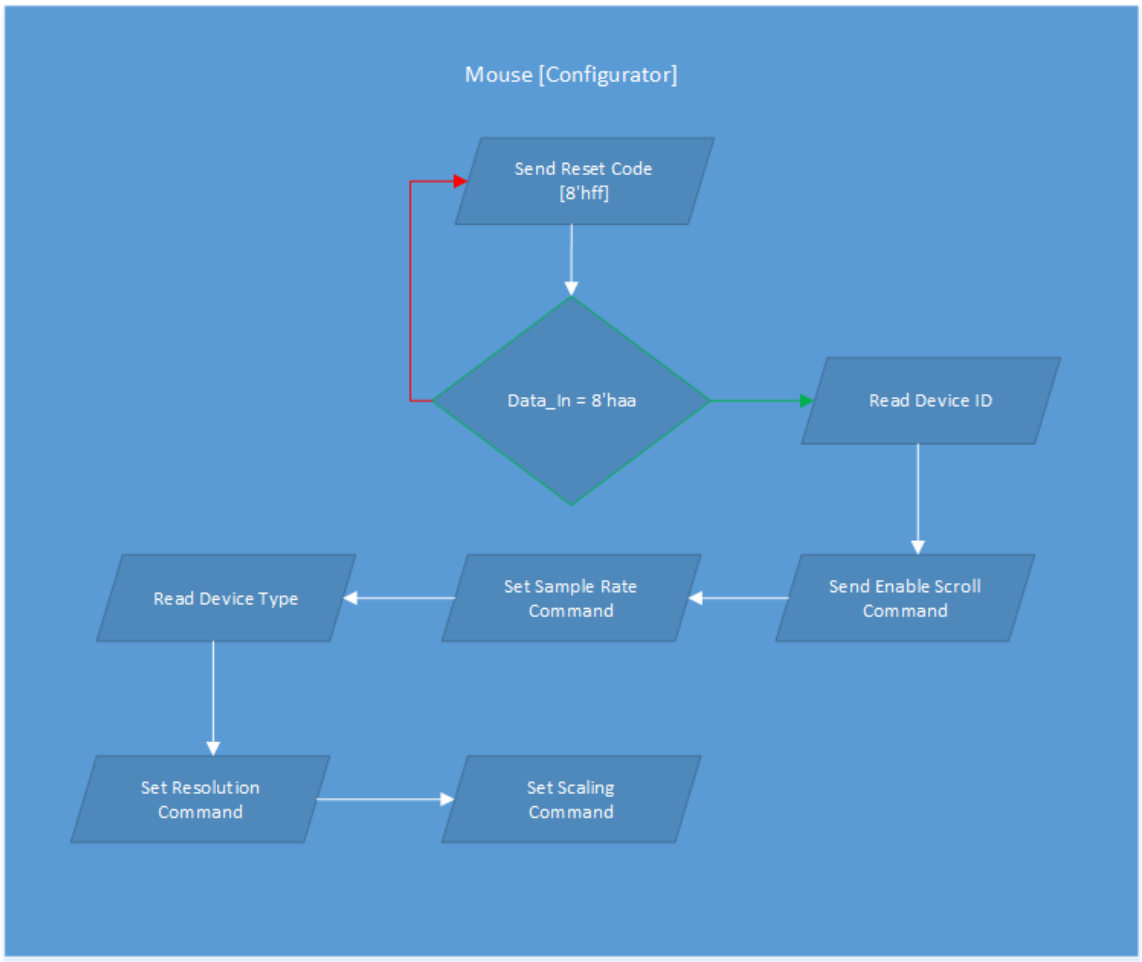


Figure 21: Mouse Flow Chart

- ## CROSS DOMAIN CONNECTOR

The PS/2 interface operates with a 100 MHz clock which we provide from the onboard clock. But the transmission modules operates with a 125 MHz clock in order to achieve Gigabit speed. To bridge the gap between these two domains, we coded a simple component that reads data from the 100 MHz domain and sends them to the 125 MHz domain in order to avoid any irregularities in the data flow.

The way it works is pretty simple. A trigger notifies the module to buff the data from the 100 MHz domain. The module closes the trigger itself and then proceeds to send the data to the 125 MHz domain, avoiding any error that may occur.

- ## PERIPHERAL MONITOR

So far, we have mentioned that the data flow from the devices to the transmission is unfiltered. This is the point where some order is restored and we adjust the data flow in order to make sense to the receiver, the Cloud Server.

The case concerning the mouse data flow was easy to be handled, since this device sends data with a standard length, which is **4** bytes. Every time we monitor a transmission from the mouse, we buff the four incoming bytes and proceed to notify the Ethernet transmission module that we are ready to send some data to the Cloud Server. After the prefix is sent to the core, the transmission triggers this module to send the rest of the payload to the Ethernet core.

Now, we have to adjust the keyboard data flow. The keyboard device transmits codes of a different length depending on the key we pressed and the nature of the stroke (press or release). For example, some codes will be 2 bytes long with a byte prefix of **0xf0** which indicates that the next byte we will receive, will point to a key that was released. Also, some keys have a constant **0xe0** prefix, which makes the release sequence of these keys: **0xf0 0xe0** + key code, which is 3 bytes long!

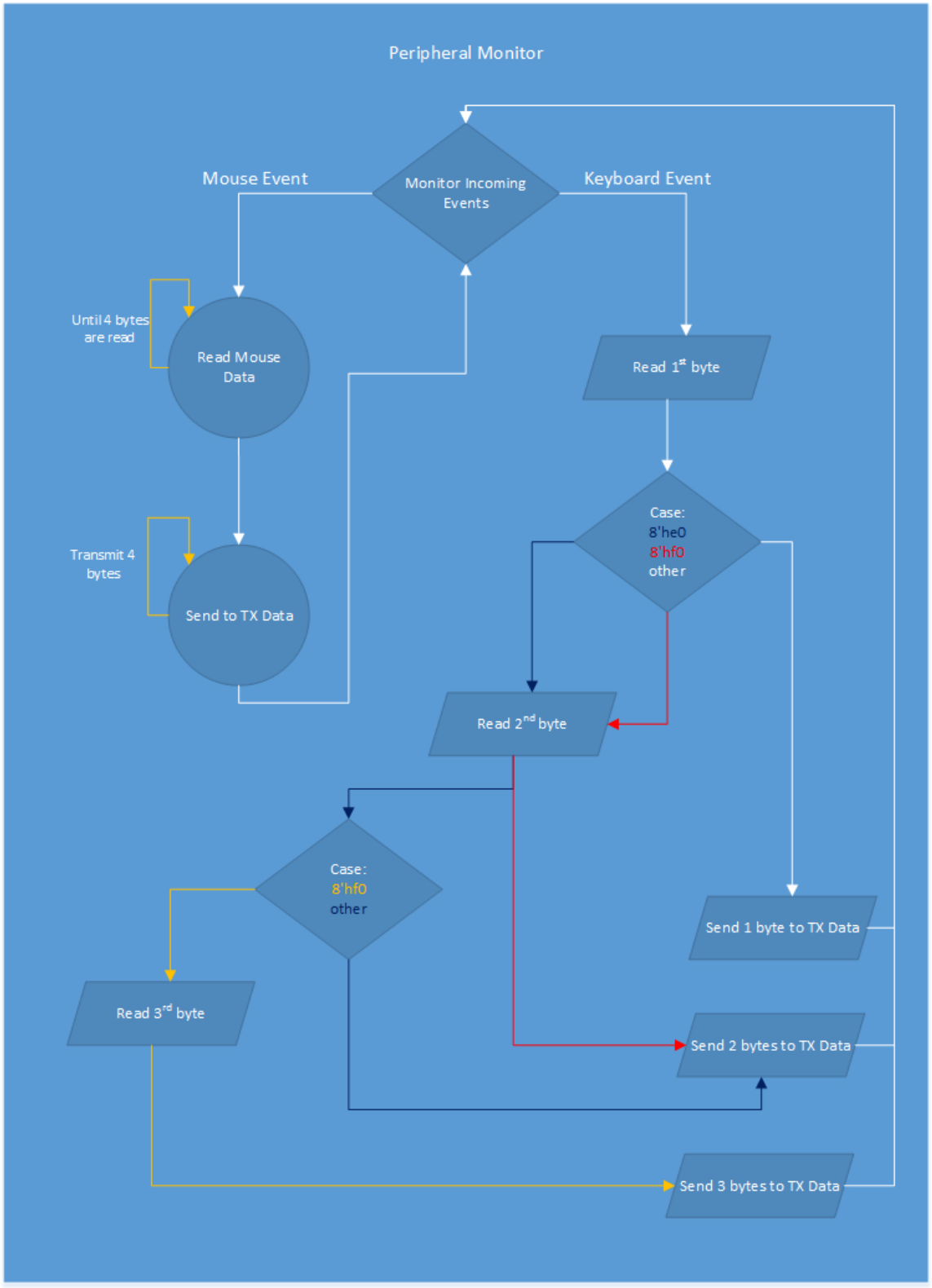


Figure 22: Peripheral Monitor Flow Chart

- AUDIO BUFFERING

This module handles the incoming flow of the audio stream and it is also responsible for the sample output to the AC97 codec we are using for the audio playback. We will examine these operations separately and then the Flow Chart at the end of this section we clear out how these two are combined inside the module.

Let's start with the storing process. The Cloud Server delivers the audio samples to our Client through the Internet in small chunks of information, the packets. In Chapter 4.1.1, you can actually see how many bytes long these packets are and how many samples they contain. The module's internal buffer is 4 bytes wide and 48000 indexes long. Since every audio sample is 4 bytes long, 2 bytes per channel or 16 bit, and our application on the Cloud Server samples the audio stream at 44.100 KHz, we can see that our module is able to buff a bit more than one 1 second of the audio stream, enabling us to experience a lossless playback.

The FSM that stores the incoming values from the RX Data module filters the incoming packets and decides to keep only the ones that are delivered to the port we have already reserved for the audio stream. The port number is hard coded into the RX Data module as we have already mentioned. In a few words, our FSM gathers up 4 bytes at the time in order to make a full audio sample and then attempts to write it to the internal buffer. It also keeps track of the next empty index we can send the sample to.

A big part of this module is the playback process. As explained in Chapter 4.2.1, the audio driver requires a full audio sample to be ready in its incoming port at the rate of 44.1 KHz. This module reads samples from the internal buffer and delivers them to the audio driver with a very simple and elegant way. We could describe this process as two software pointers filling in and out the buffer, but the whole concept is better explained in the following Flow Chart.

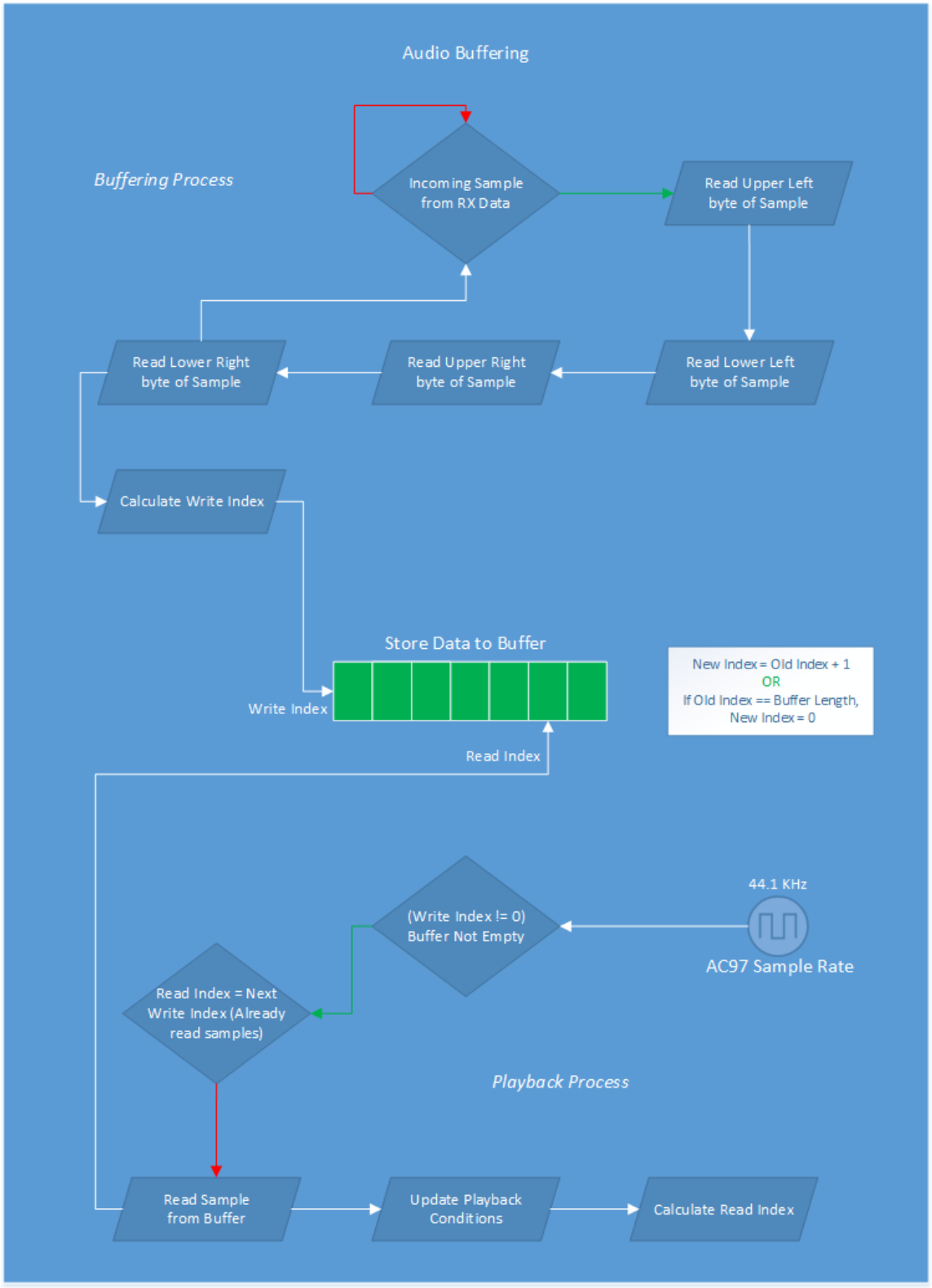


Figure 23: Audio Buffering Flow Chart

▪ FRAME BUFFERING

Here we are introduced to the frame stream process. This module is subdivided into three other modules. The first one is the MIG interface and it is thoroughly explained in the according section. That leaves us with a module that handles the incoming data stream from the Ethernet MAC Core, and another one that diverges the data flow from/to the RAM module.

o FRAME DATA FROM ETHERNET TO RAM MULTIPLEXOR

So far we have described the process of capturing incoming stream data from the RX Data module. Again, we are filtering the incoming stream for frame stream data only and when a packet arrives we buff its contents. We make sure to deliver these packets to the RAM only when we have captured enough data to perform a Write Burst command to the RAM. More on that issue can be found at Chapter 4.1.3. Every outgoing packet from this module starts with the RAM index that it will be written to, and the rest of the pixel data follow.

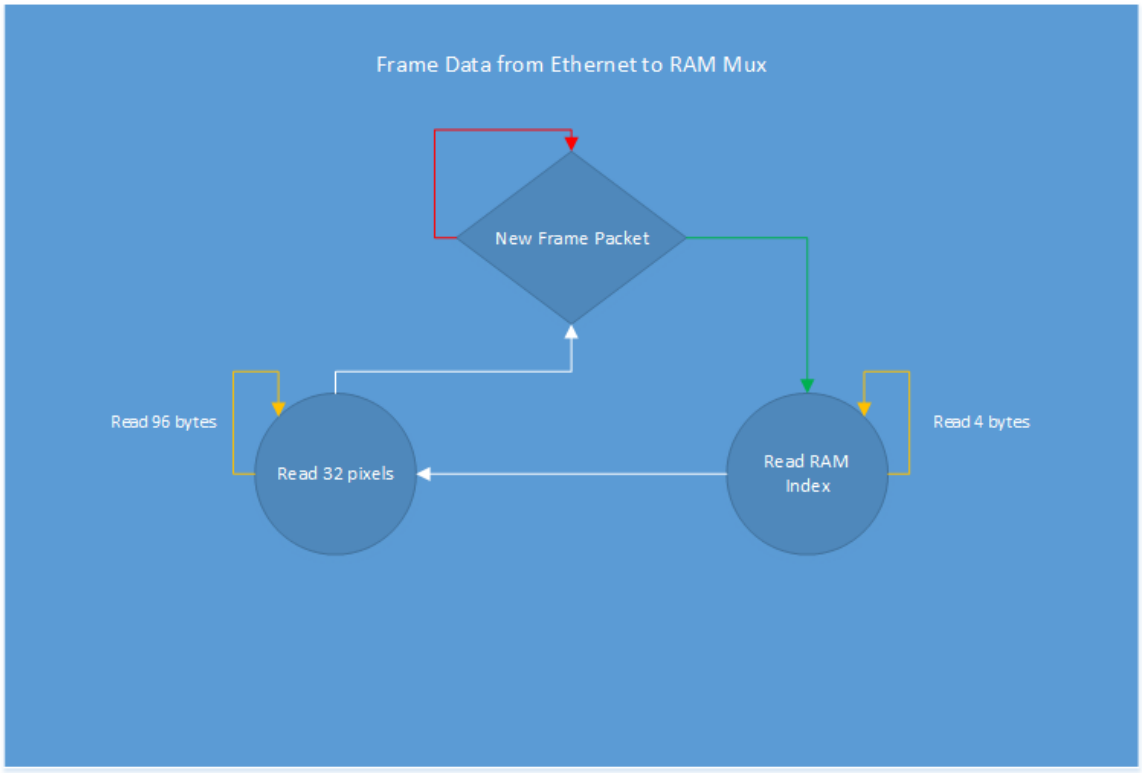


Figure 24: Frame Data from Ethernet to RAM Mux Flow Chart

- RAM MULTIPLEXOR

This is the most important and complex part of our design. In a few words, it is responsible to create and maintain the reference frame of the design and also make it available for access from the RAM-DVI Sync module. Again, the module is subdivided into 5 others that execute specific operations.

- RAM INITIATOR

After the reset signal has been asserted to the device, a reference frame should be created and stored in the RAM device in order for the screen to have something to display initially. That is the main task of this module, to send specific pixel data to the RAM in order to slowly create the reference frame of the design.

This is accomplished with a simple FSM described in the Flow Chart below. After is has ran its course and the reference frame has been sent to the RAM, this module's output is multiplexed with the incoming stream from the Ethernet MAC module. So, after it has sent the initial reset data (reference frame) it enables the Frame Data from Ethernet to RAM Mux outgoing data to bypass this module and reach the RAM. That way, we ensure that the incoming frame stream data are saved to the RAM only after the initialization has finished.

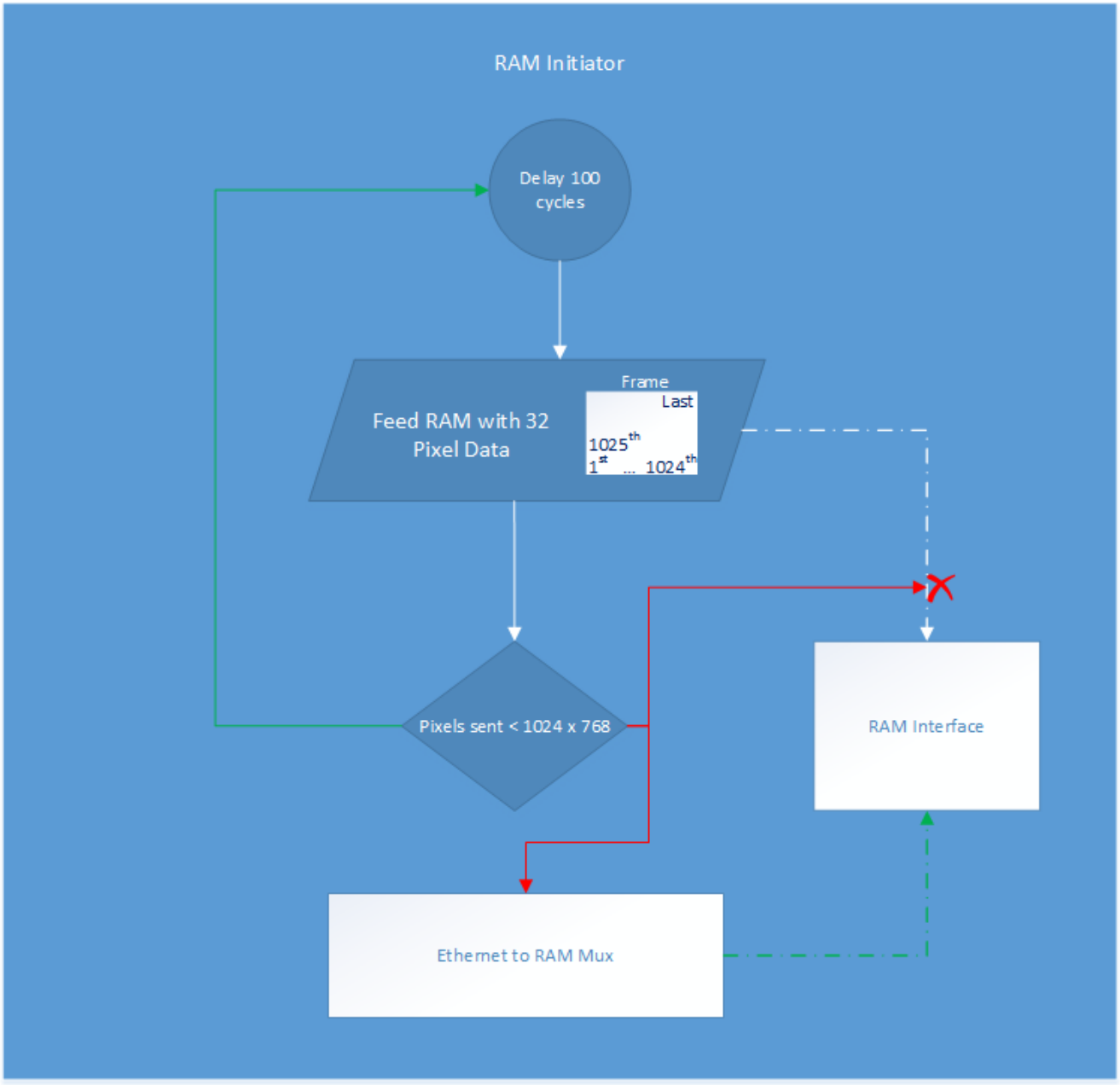


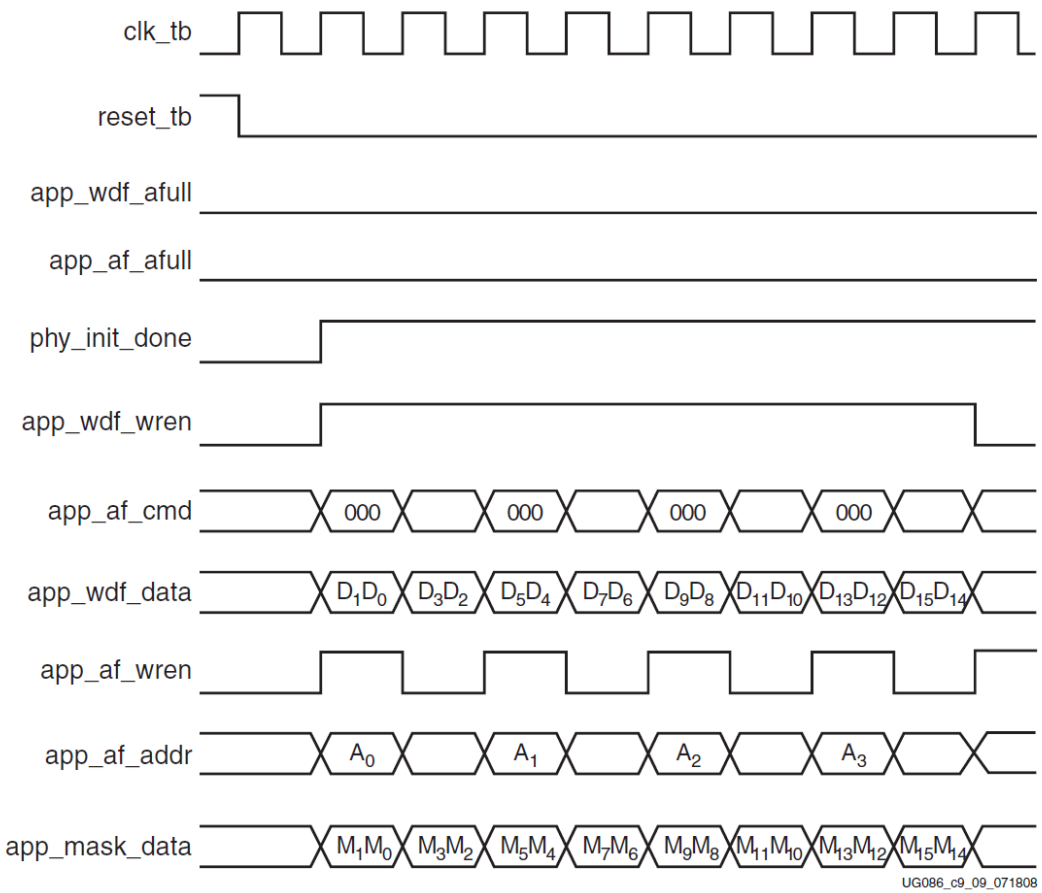
Figure 25: RAM Initiator Flow Chart

- WRITE BURST / READ BURST

As we explained in Chapter 4.1.3, the RAM handles packets that are a lot bigger than a single pixel. Not only that, but our controller is reading/writing data to/from the RAM in a bursting manner. The process to read or write a burst of data to the RAM is thoroughly explained the MIG manual. In these modules we implement this process, thus making the communication between the sub-modules much easier.

The following diagrams taken from the Xilinx MIG User Guide describe these processes. Notice in Chapter 4.1.3, how we skip a writing/reading cycle every now and then in order to achieve the proper data aligning.

Figure 26: Write Burst



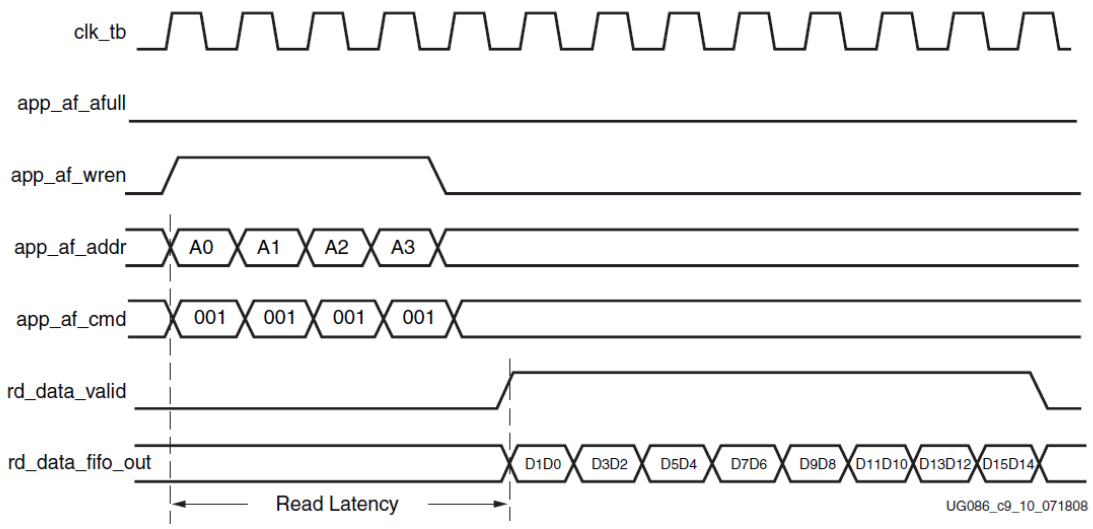


Figure 27: Read Burst

- ARBITRATOR

Since our DDR2 SDRAM device is being used by two different modules, the Frame Reader and the Frame Data from Ethernet to RAM Mux module, we need some kind of arbitration in order to avoid data collision.

This module handles read requests from the Frame Reader and write requests from the Frame Data from Ethernet to RAM Mux module. Some constraints need to be met before these requests are resolved. In case one of these requests needs to be stalled, the arbitrator keeps track of the request and serves it a bit later when it is sure that no data collision will occur. The timing constraints of Chapter 4.1.3 make our job easier since they eliminate a lot of faulty possibilities. The following Flow Chart describes the arbitration algorithm.

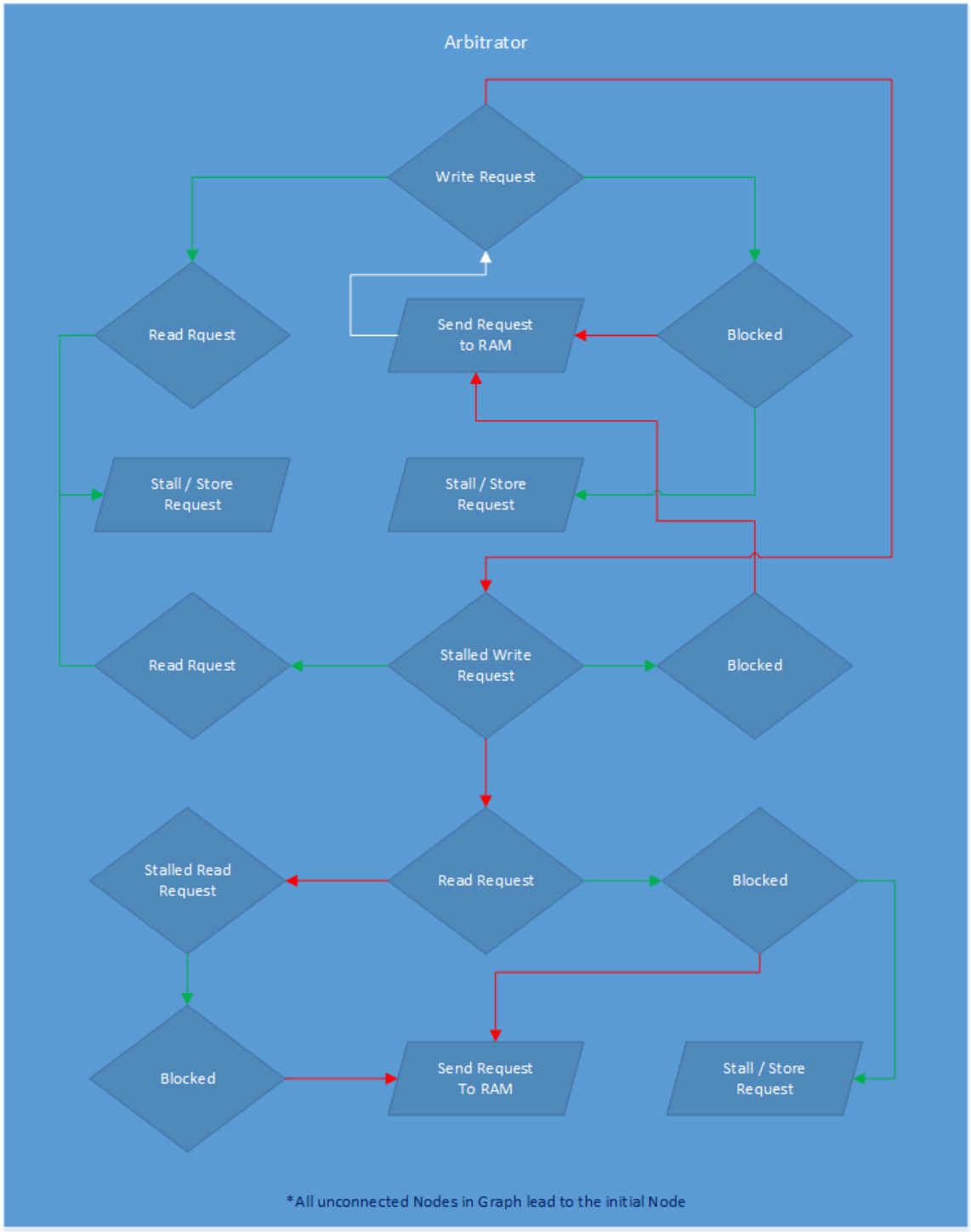


Figure 28: Arbitrator Flow Chart

- FRAME READER

The connection between the DVI and the DDR2 SDRAM is made through this module. Every time the DVI module needs data (pixels) in order to feed the screen, it signals this module preemptively. The Frame Reader proceeds to send multiple read requests to the RAM in order to buff a bigger chunk of information (see 4.1.3) and then delivers it to the RAM-DVI Sync module. The read indexes are generated here since the DVI is only asking for more pixels and not specific data. The following Flow Chart explains the simple process that takes place inside this module.

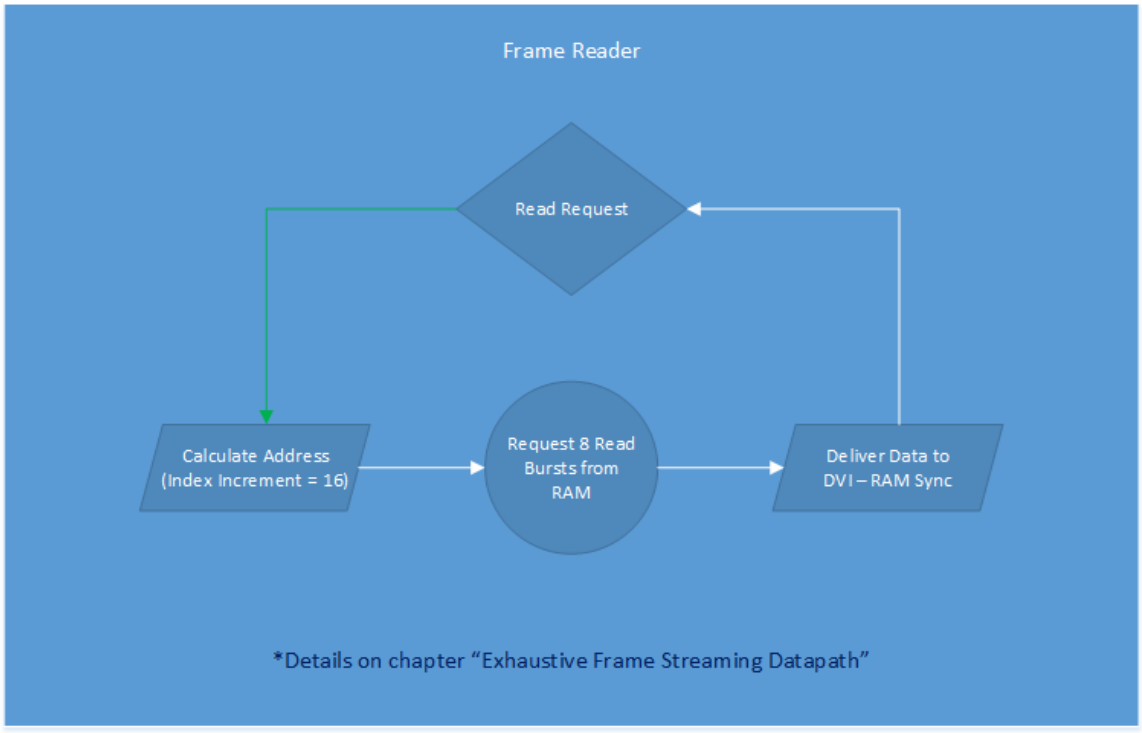


Figure 29: Frame Reader Flow Chart

- RAM-DVI SYNC

The last connecting link between the Cloud Server's frame stream and the DVI driver is this module. The timing constraints which are described in Chapter 4.1.3 are justifying the use of two main buffers in this module. In a few words, every time we are filling one buffer with data, we read data from the other and vice versa. That way, the data flow is continuous with no delays or missing pixels. This module signals the Frame Reader at some critical points in order to fill the already used buffer preemptively. The following Flow Chart describes the tasks of this module and the way they are accomplished.

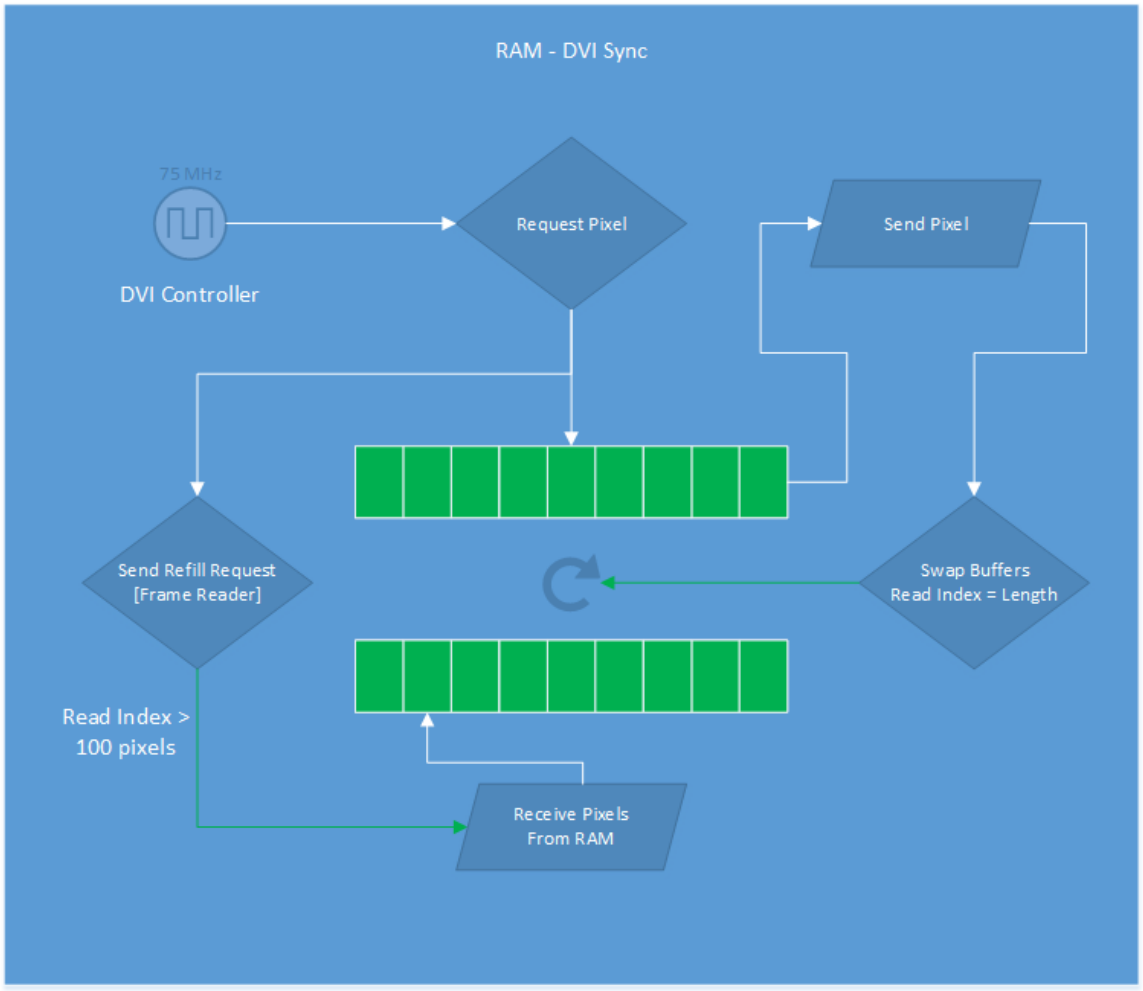
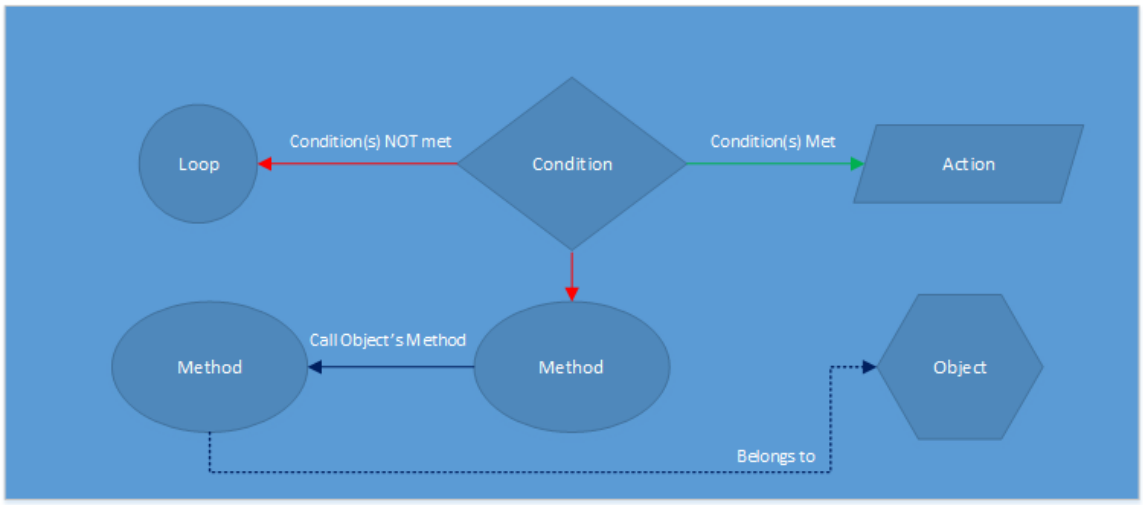


Figure 30: RAM-DVI Sync Flow Chart

4.2.3 SOFTWARE LIBRARIES

A fair amount of the workload needed while implementing the software-based Cloud Server could be found in ready-to-execute libraries. A few thousand program code lines were spared because of three precompiled libraries in our project, the NAudio library for the audio recording, the Slim-DX for the screenshots, and Metro Framework for the GUI. The following chapter outlines the way we are utilizing these libraries for our profit. Before we move on, here a useful diagram:

Annotations



- NAUDIO

NAudio is an open source audio API for .NET written in C# by Mark Heath, with contributions from many other developers. It is intended to provide a comprehensive set of useful utility classes from which we can construct our own audio application. It includes features like: record wave stream, playback, edit etc.

We chose NAudio simply because it is open source, light-weight, easy to use and covers our design needs. Its ability to manipulate the audio stream as a signal, was also a critical point on our decision. Let's see in detail how this library is being used in our software.

First of all, we need a way to capture the server's audio stream, in order to stream it to the Client, which will handle the playback. Every possible audio source needs to be captured and the only way to do that, is through our operating system, in our case Windows 8.1. Windows come with an embedded API, the Windows Session API or WASAPI. NAudio provides an interface to this API, called WasapiLoopBackCapture, which is using the Windows API to capture the audio stream to a wave object.

Moving to a signal domain while capturing the audio stream, proved to be useful since we can apply transformations to the signal, such as the volume modification. We can also change the signal precision or the sampling frequency but we chose not to, in order to maintain a certain level of quality to our application.

After we instantiate out interface to the Wasapi, we have to override the callback method of the library, which will be executed every time the API has captured a number of samples. In this method we choose to repeatedly extract a few samples from the buffer until it is empty, while also creating small packets to be delivered to our client through a UDP server. More details on the following Flow Chart.

Finally, we designed our own simple interface for starting / pausing / stopping the recording process, using some NAudio functions like StartRecording and StopRecording. Chapter 5 holds all the statistical data concerning this library's usage.

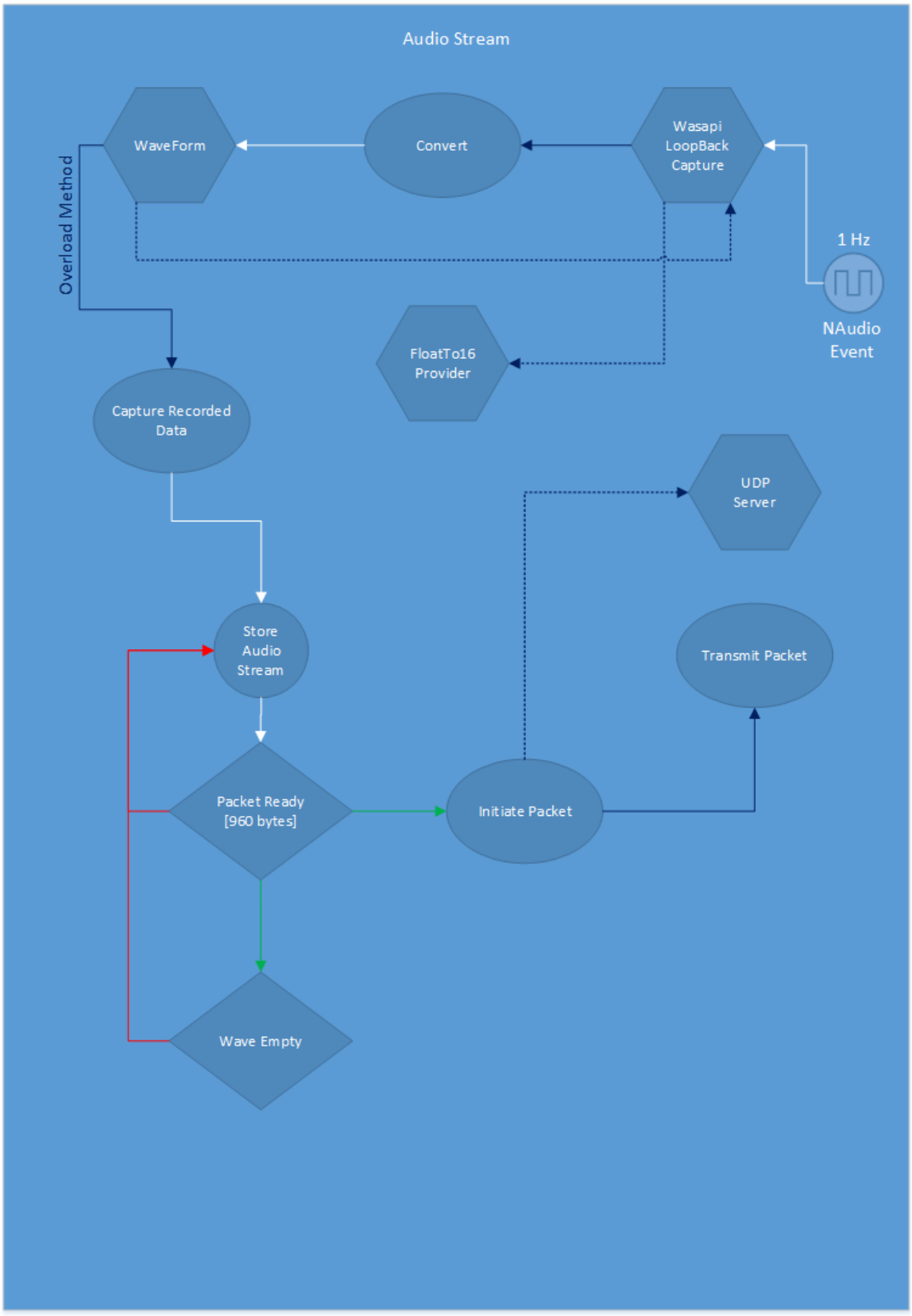


Figure 31: Audio Recording Flow Chart

- SLIM-DX

Slim-DX is a free open source framework that enables developers to easily build DirectX applications using .NET technologies such as C#. It is designed to be an efficient, simple, and lean wrapper that fully encompasses all of Microsoft's gaming and multimedia technologies and exposes them to managed code. All of the code is under the MIT/X11 license, and all content is under the Creative Commons Attribution-Share Alike license.

Before moving on to the details of the way we are utilizing this library, we have to write down a few things. Direct access to the Direct X using a C# application, seems to be impossible in the Windows 8 environment, especially in our case, where the OS recognizes the embedded Intel Graphics Card as the main graphics card of our PC. Thus, we tried to import the User32.dll Library in order to capture the screenshots for our frame stream using the Graphical Device Interface (GDI). That method proved to be ineffective, but more on that issue later.

So, we end up using the Slim-DX library which is a fairly lightweight interface compared to the Direct X. The huge advantage over the GDI method, is the capture format. This library gave us the option to receive the data as an array of elements, whilst GDI method returns a Bitmap. It is way easier to extract data from an array than from an image file structure. Not to mention the time wasted on creating this unused structure at the first place.

We implemented a plain interface to utilize this library, which includes a single method, the CaptureScreen method. First of all, we need to initiate a virtual screen and we do that with the help of the Slim-DX library. We are given the option to create a software-based or a hardware-based device, and of course we chose the second one. After the initialization, our capturing method uses a Surface object, the size of our computer's screen, to capture all the pixels that are being displayed at that point. This surface object holds an array buffer with all data needed, and that' is the object we are returning to the caller.

- METRO FRAMEWORK

The computer we used to develop our design comes with a Windows 8 OS. After our decision to implement a Graphical User Interface for the software that our Cloud Server will be running, it was only logical to choose a library that will give our application a Windows 8 feel. Metro Framework does that fairly easy, since our only task was to extend our GUI class to a MetroForm class. All objects, like buttons and panels, were overwritten by this library and eventually redesigned to match today's representation trends. Any feature provided by this library and not the standard C# classes, will be noted.

4.2.4 SOFTWARE IMPLEMENTATION

Our whole software design is composed of a few thousand lines of code, so quoting them here will not be a smart choice. Instead, we drew a class diagram (Chapter 4.12). Let's start with a more generic categorization of the classes used.

- **Controllers**

  In this category are included classes which are either responsible to interact with an external library, such as Slim-DX, or they're assigned with a managing task, such as simulating mouse clicks or decomposing incoming UDP messages.

- **Data Types**

  In this category are included classes which can also be described as the widely known C structures. No methods are included other than accessing methods to the encapsulated data. These structures can be simple data "boxes" (MouseAction), or more complex like a Queue (ActionQueues).

- **Devices**

  In this category are included classes which simulate one the four peripheral devices we are interacting with in this project. The finals processing steps of the I/O communication take place inside these classes.

- **Network**

  In this category are included only two classes, one responsible for creating and maintaining UDP servers, and another one which initiate the network interface. Since a valid network connection is a prerequisite, every critical object is instantiated step by step through this class.

Every software class we designed will be explained using the same pattern as in the Hardware section, a few words followed by a Flow Chart. The suffix indicates the category it belongs to.

- **ACTION DISPATCHER**
  Controllers

This class manages two of the most basic controllers of our application, clicking and typing. The workload is assigned into two different threads, one responsible for the mouse actions, and one for the keyboard actions. The two threads run constantly but synchronization is achieved by the Queue class. Specifically, the thread execution is protected by a lock/unlock/monitor pattern which forces the two threads to wait until a monitor pulse has arrived, then proceed to pop a task from the Queue and finally execute it using the controllers.

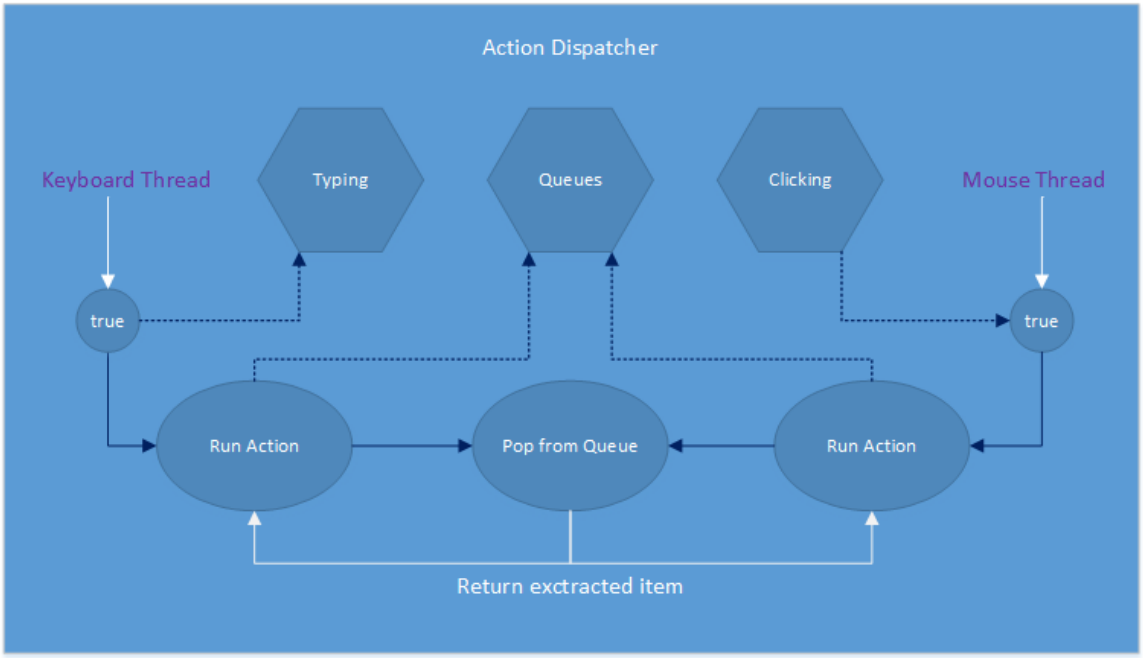


Figure 32: Action Dispatcher Flow Chart

- **CLICKING**
  Controllers

As we have already mentioned, this class is managed and instantiated by the class ActionDispatcher. We could best describe this class as a wrapper for the virtual device, Mouse. Every time ActionDispatcher pops a task from the Queue, this class is responsible to issue the command to our virtual device.

- **DECOMPOSER**
  Controllers

This class holds responsibility for decomposing incoming packets and extracting key elements from them. In Chapter 4.1.1, we can see that every incoming data packet from the Client is encoded in a plain, wet unknown to our application, format. Here, we are trying to "tokenize" the incoming messages and according to the information we extract from them, return key codes to the caller. In total, five different decomposing methods have been implemented to extract the key codes needs, and each of them reveals a different chunk of information, like mouse position, click type, keyboard stroke etc. In sections Mouse and Keyboard you can find a translate table for every aspect of the key codes used in our application.

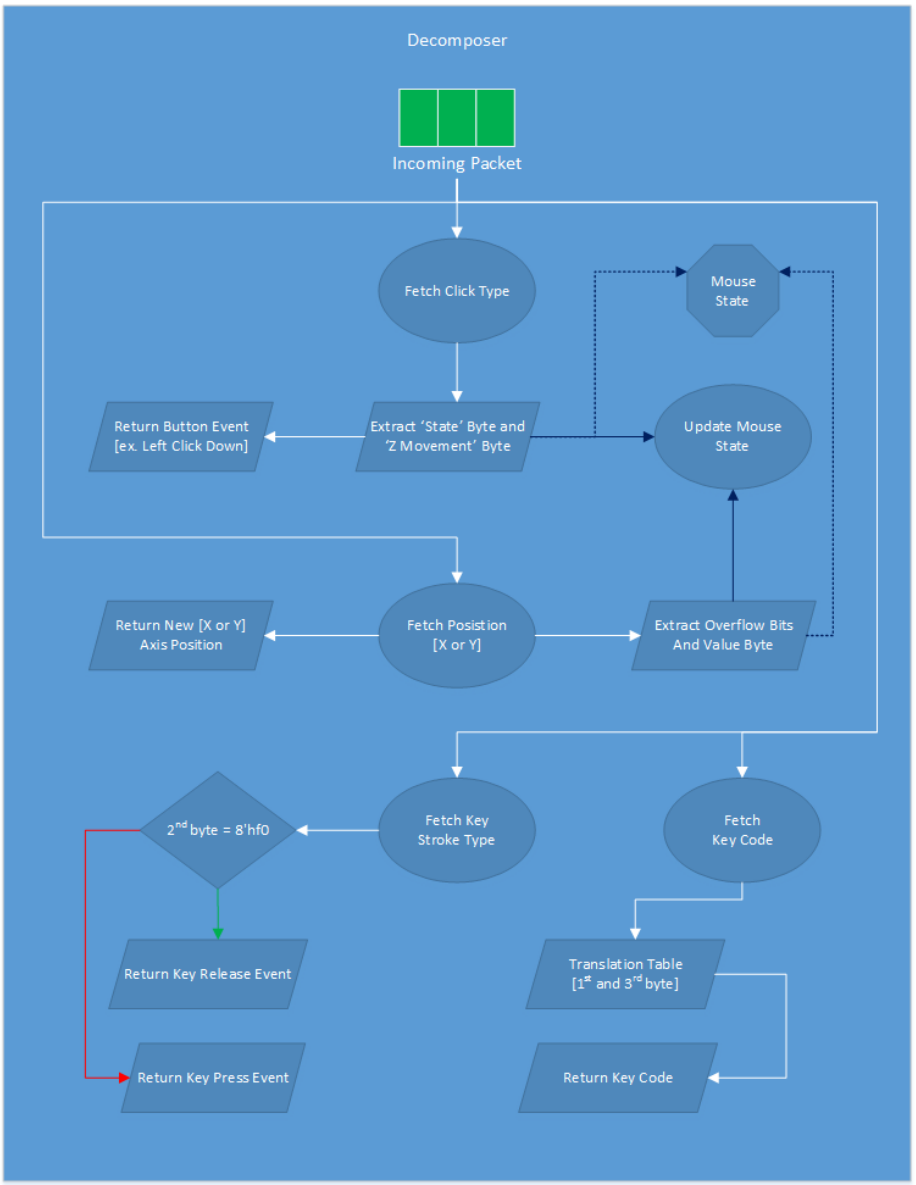


Figure 33: Decomposer Flow Chart

- **DX SCREEN CAPTURE**
  Controllers

  Read Chapter 4.2.1

- **SCREENSHOT**
  Controllers

This class was designed to capture a screenshot and return the data to the caller in a BMP format. The GDI method that is being used, requires some methods and structures from the User32.dll Library, so we start of by importing this library to our project. Before we continue on describing the main task of this class, we have to point out that this class is able to locate the mouse pointer information from the OS and use them to draw it on the screenshot. This feature is not available on the DxScreenCapture class that is our default capturing class, but if we take into account issues like speed and throughput it is clear that the GDI is ineffective. Moving on, in order to collect the frame data, we instantiate a Graphics object which is similar to the Surface object of the Slim-DX library. Inside this object, we are copying pixel values from the whole screen area. The mouse pointer is also drawn onto the image, before returning the bitmap file to the caller.

- **TYPING**
  Controllers

This class is designed as a mirror to the Clicking class. Again we have designed a wrapper for our virtual device (Keyboard) and proceed to pass the incoming tasks to the actual Keyboard class. The only minor change, is the introduction of fixed interval between two consecutive incoming tasks. More on that issue at Chapter 6.2.

- ACTION QUEUES

  Data Types

This class contains two action queues, one filled with mouse actions and one with keyboard actions. Access to the queues is granted through methods which are protected with synchronization blocks. Add and Remove methods are mutually excluded by locking the entire queue every time an action is performed (push or pop). Finally, we have taken into consideration the possibility of attempting to read from an empty queue.

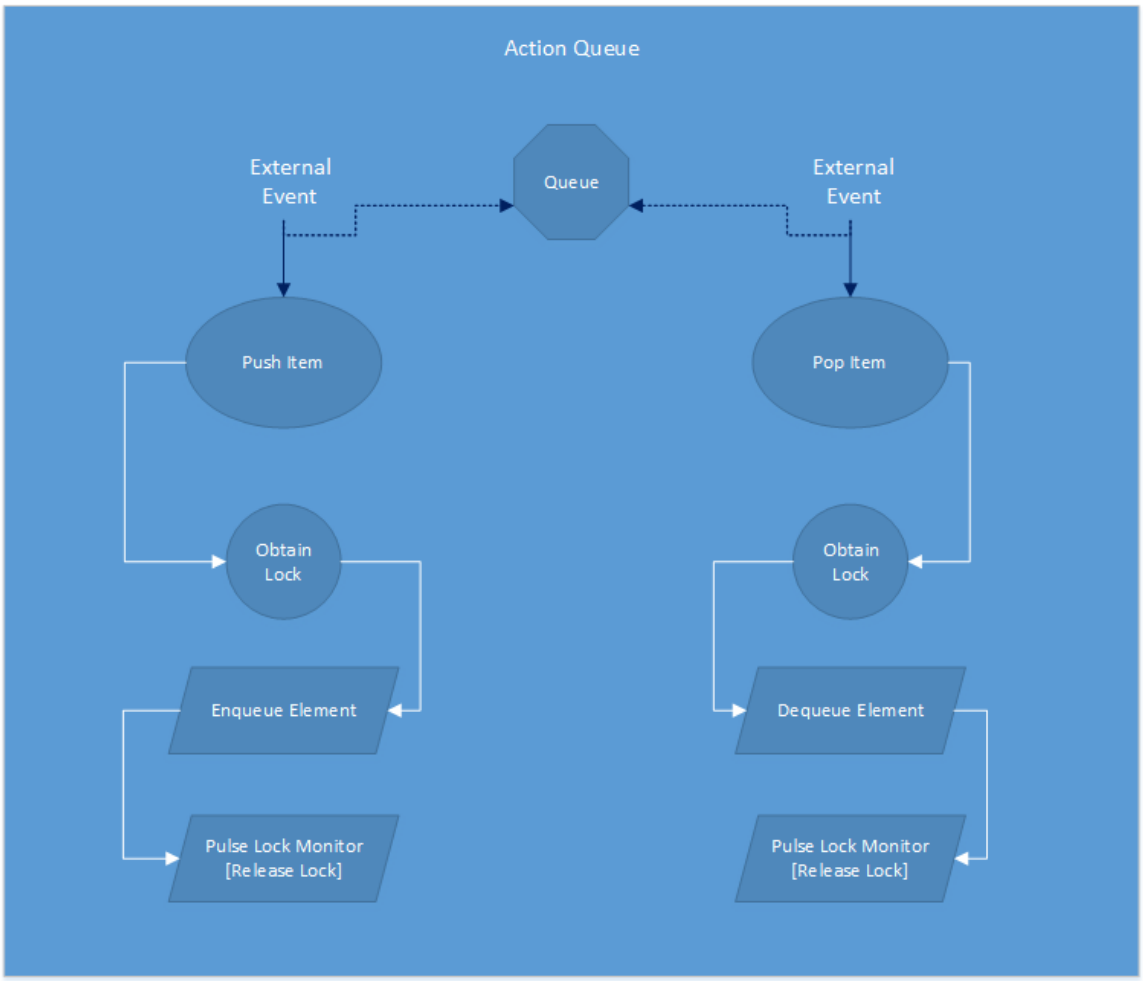


Figure 34: Action Queues Flow Chart

- KEYBOARD ACTION
  Data Types

This class could be described as a C structure that comes with a set of setters and getters for the embedded elements. Here, we have anticipated a maximum number of three simultaneous key strokes, but in reality our Cloud Client sends one key stoke at a time. So, we only need an integer key code and key hit descriptor (press key or release key) in order to fully describe a keyboard action.

- MOUSE ACTION
  Data Types

This class is designed as a mirror to the KeyboardAction class and contains all the information needed to describe an occurred mouse event, which are: X coordinate, Y coordinate and a click descriptor.

- MOUSE STATE
  Data Types

According to the PS/2 protocol, the mouse device does not transfer the current position of the mouse pointer, instead it calculates the movement of the device and transfers that value. While drawing the mouse pointer onto the screen using only the horizontal and vertical movement values, we need to know the prior position of the pointer in order to calculate the new one. This is the purpose of this class, to memoir the current mouse pointer state and enable us to calculate the new one using the information we receive from the Client.

Maintaining the right x-y coordinates proved to be a tricky task, since we had to be careful not to overflow the mouse position value and assign an out of bounds position to the pointer. Also, we had to simulate the constant press of a mouse click, in case of a drag-n-drop event and that is why because the Client keeps sending "press left click down" messages according to the PS/2 protocol, which should be ignored from us in order to avoid sending multiple left click actions to our Server.

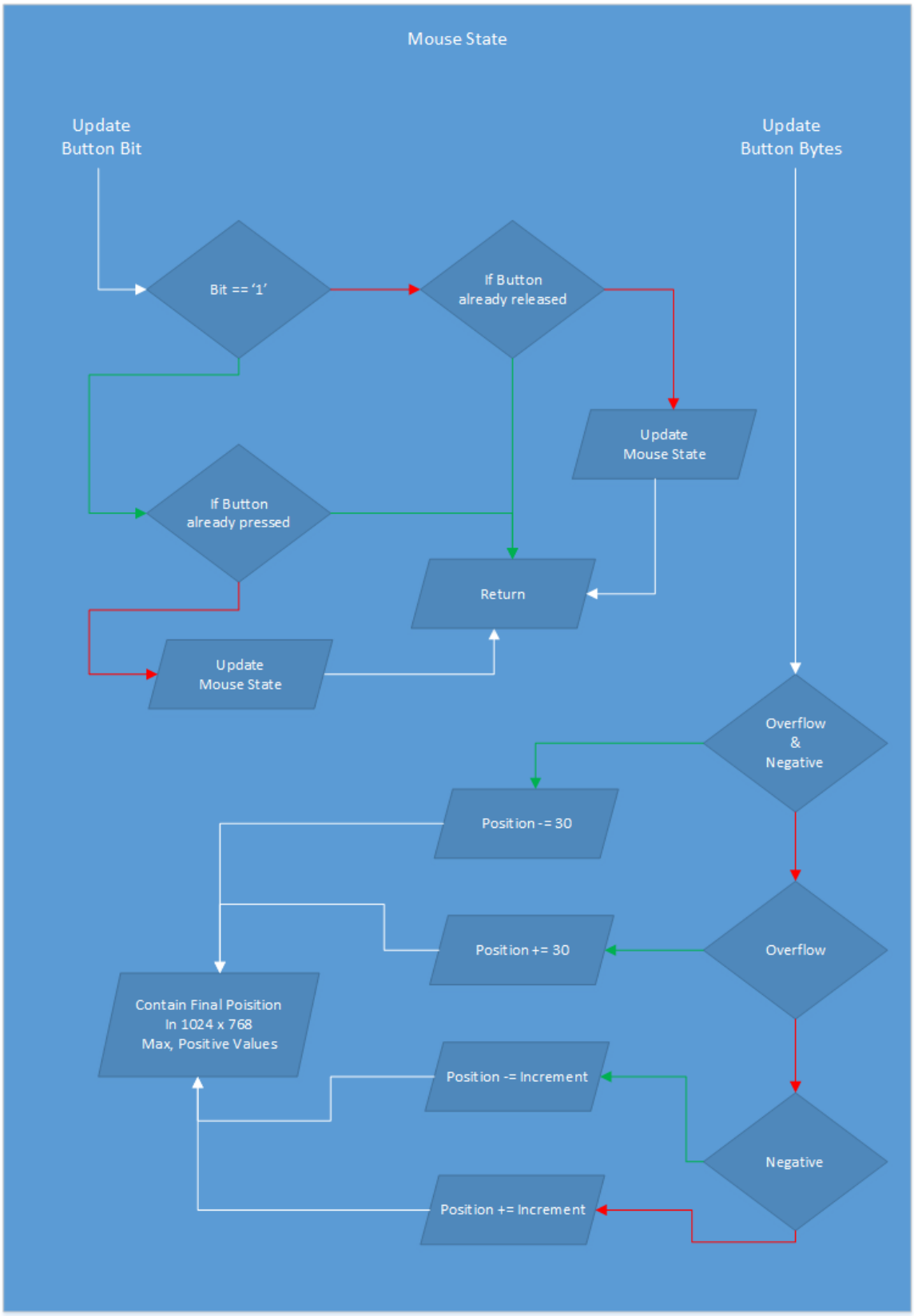


Figure 35: Mouse State Flow Chart

- **KEYBOARD**
  Devices

This class simulates a virtual keyboard device. Again, we are importing the User32.dll library in order to retrieve the critical keybd_event function. Class Keyboard Action contains all the necessary information to call this function and finally simulate the keyboard event we received from the Client.

Up to three simultaneous key hits are supported, but not used. Finally, a big translation function was implemented in order to decode the key codes to software codes.

Table 13: Keyboard Translation Table

Received Key Code	Translation Stg 1	Virtual Key Name	Virtual Key Code
0x76	9	VK_ESCAPE	0x1B
0x16	23	VK_1	0x31
0x1e	24	VK_2	0x32
0x26	25	VK_3	0x33
0x25	26	VK_4	0x34
0x2e	27	VK_5	0x35
0x36	28	VK_6	0x36
0x3d	29	VK_7	0x37
0x3e	30	VK_8	0x38
0x46	31	VK_9	0x39
0x45	22	VK_0	0x30
0x4e	107	VK_OEM_MINUS	0xBD
0x55	105	VK_OEM_PLUS	0xBB
0x66	1	VK_BACK	0x08
0x0d	2	VK_TAB	0x09
0x15	48	VK_Q	0x51
0x1d	54	VK_W	0x57
0x24	36	VK_E	0x45
0x2d	49	VK_R	0x52
0x2c	51	VK_T	0x54
0x35	56	VK_Y	0x59
0x3c	52	VK_U	0x55
0x43	40	VK_I	0x49
0x44	46	VK_O	0x4F
0x4d	47	VK_P	0x50
0x54	111	VK_OEM_4	0xDB
0x5b	113	VK_OEM_6	0xDD
0x5a	4	VK_RETURN	0x0D
0xe0 + 0x14	94	VK_RCONTROL	0xA3
0x14	93	VK_LCONTROL	0xA2
0x1c	32	VK_A	0x41
0x1b	50	VK_S	0x53

0x23	35	VK_D	0x44
0x2b	37	VK_F	0x46
0x34	38	VK_G	0x47
0x33	39	VK_H	0x48
0x3b	41	VK_J	0x4A
0x42	42	VK_K	0x4B
0x4b	43	VK_L	0x4C
0x4c	104	VK_OEM_1	0xBA
0x52	114	VK_OEM_7	0xDE
0x0e	110	VK_OEM_3	0xC0
0x12	91	VK_RSHIFT	0xA1
0x5d	112	VK_OEM_5	0xDC
0x1a	57	VK_Z	0x58
0x22	55	VK_X	0x5A
0x21	34	VK_C	0x43
0x2a	53	VK_V	0x56
0x32	33	VK_B	0x42
0x31	45	VK_N	0x4E
0x3a	44	VK_M	0x4D
0x41	106	VK_OEM_COMMA	0xBC
0x49	108	VK_OEM_PERIOD	0xBE
0x4a	109	VK_OEM_2	0xBF
0x59	92	VK_LSHIFT	0xA0
0xe0 + 0x11	96	VK_MENU	0x12
0x11	95	VK_MENU	0x12
0x29	10	VK_SPACE	0x20
0x58	8	VK_CAPITAL	0x14
0x05	77	VK_F1	0x70
0x06	78	VK_F2	0x71
0x04	79	VK_F3	0x72
0x0c	80	VK_F4	0x73
0x03	81	VK_F5	0x74
0x0b	82	VK_F6	0x75
0x83	83	VK_F7	0x76
0x0a	84	VK_F8	0x77
0x01	85	VK_F9	0x78
0x09	86	VK_F10	0x79
0x78	87	VK_F11	0x7A
0x07	88	VK_F12	0x7B
0x75	16	VK_UP	0x26
0x6b	15	VK_LEFT	0x25
0x74	17	VK_RIGHT	0x27
0x72	72	VK_ADD	0x6B
0xe0 + 0x1f	58	VK_LWIN	0x5B
0x1f	0	VK_OEM_8	0xDF
Default	0	VK_OEM_8	0xDF

- **MOUSE**
  Devices

This class is a mirror to the Keyboard class. The Mouse device is simulated, but in this case we are importing two functions from the User32.dll library in order to carry out the mouse event. The mouse_event (handles the press/release of mouse buttons) and SetCursorPosition.

Virtual Event	Key Code
MOUSEEVENTF_MOVE	0x0001
MOUSEEVENTF_LEFTDOWN	0x0002
MOUSEEVENTF_LEFTUP	0x0004
MOUSEEVENTF_RIGHTDOWN	0x0008
MOUSEEVENTF_RIGHTUP	0x0010
MOUSEEVENTF_MIDDLEDOWN	0x0020
MOUSEEVENTF_MIDDLEUP	0x0040
MOUSEEVENTF_XDOWN	0x0080
MOUSEEVENTF_XUP	0x0100
MOUSEEVENTF_WHEEL	0x0800
MOUSEEVENTF_VIRTUALDESK	0x4000
MOUSEEVENTF_ABSOLUTE	0x8000
WHEEL_DELTA	0x70

Table 14: Mouse Translation Table

- **SCREEN**
  Devices

This class simulates a virtual screen device. A capturing controller is included alongside with a UDP Server instance. In our implementation we are using this device with the Slim-DX controller, but the user is given the option to change the controller from the GUI.

A number of standard methods have been designed for this class that allow us to start, restart, stop and initiate the device. Also, we view usage statistics, save the current screen state to a Bitmap file and finally shut down entirely the device.

Into more technical details, the frame data are stored to a huge array containing all the information needed to represent 1024 x 768 pixels with 24bit color resolution. Before we start using this device in order to stream data to the Cloud Client, we need to build the first frame by filling up this array with data. This is achieved by calling the screenshot controller, retrieving the returned object which is a Surface object and step by step extract the data buffer containing the pixel values. We have to ignore the alpha channel while filling up the buffer because the DVI Driver of the Cloud Client has no need of it, so we would be sending junk data.

After the base screen is ready we continue by transmitting the whole buffer to the Client. Some data may be lost, but that is not a big problem because the streaming algorithm we are using will replace them at some point. This algorithm works similarly to the MPEG since it chooses to transmit only the differences between two consecutive frames. In order to achieve a good FPS rate, we are forced to take screenshots as fast as we can, compare the newly captured data with the base frame and finally send the pixels that differ from the original image. In the Hardware Implementation Chapter (4.2.2) we described how the DVI Driver reads data in chunks and not pixel by pixel, so we are trying to speed up this process by comparing data the same way. So, every time a difference is spotted in a pixel of two consecutive frames we are sending the entire chunk of pixels to the Client and not just the changed one.

The following Flow Chart outlines the algorithm used while streaming the Cloud's frame stream. In a higher abstraction level, the true difference between the GDI method and the Slim-DX method, is only the object we're reading from in order to fill our array (Surface vs Bitmap).

Figure 36: Frame Streaming Flow Chart

- SPEAKERS
  Devices

This class simulates the speaker device. Compared to other classes, it is autonomous since it does not interact with other controllers or devices other than the application's GUI. The NAudio library is imported in order to instantiate the WasapiLoopbackCapture object which will eventually provide us with the needed wave stream. In Chapter 4.2.1 we have already described how this library is utilized.

- APP INTERFACE
  Network

This class is multi-threaded and responsible for the application's network interface. Since we need to establish a valid connection to the Client in order for the application to operate, this class is actually our main class.

First of all, five threads are constructed and initiated though this class. The actionServerThread handles the UDP server, whilst the pollingThread's task is to poll the listening socket for incoming packets that may concern our application. Frame thread and Audio thread instantiate the corresponding virtual devices and continue on working on these data streams. Finally, connectionThread's only task is to poll the listening port and establish a valid connection with the Client. After that, it is disposed.

All controllers and devices are instantiated inside this class in order to have a valid reference to them in case we need to exchange data. The execution thread of this class is outlined in the GUI section, so we will just explain what these method calls actually do.

NetworkStart starts the action server thread, which construct the listening UDP server. This server will be used later on to establish a connection to the Client and receive incoming packets. NetworkShutDown terminates the controllers and device operations. EstablishConnection blocks the calling thread as it is waiting to receive a connection request from the Client. When it is successfully received, the initiateDevices method is called, which handles the start of the other thread's execution. PollPort method is called while initiating the polling thread and it also blocks the calling thread. Its task is to receive incoming packets, decode them, and finally decide if they concern our application. If they do, all information is extracted from the packet and with the help of set of decomposing function, the right controller is called to carry out the task.

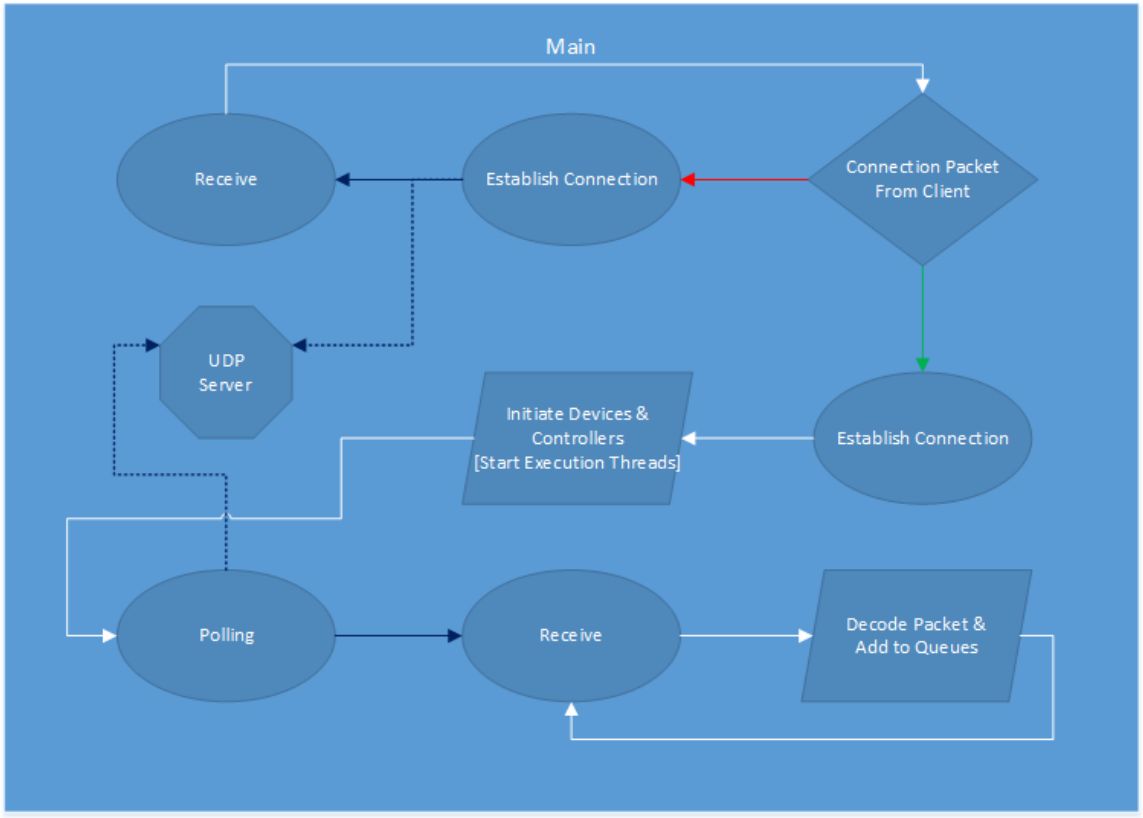


Figure 37: App Interface Flow Chart

- **UDP SERVER**
  Network

This class acts as a UDP packet server, responsible to receive and transmit packets from a specific port number and IP address. The UDP listening port is 4567 by default. While initiating the server, we have to provide and IPEndPoint, derived from the specific port we will be using, and IPHostEntry. Also, we initiate two different sockets for data transmission, one for the audio stream and another for the frame stream.

Besides the close, receive (it is important to remember that is a blocking method) and send methods which are pretty standard, we implemented some more, in case we need to readjust some features of the server, like the listening port number. Finally, this class holds statistics which can be retrieved from a public method, alongside with other data, like Local IP and Global IP.

- GUI

Inside this class, we are drawing the Graphical User Interface while initiating the entire application. The initiation simply concerns the management of the App Interface class. The execution thread is pretty straightforward, we are just running the Start method of the App Interface class, whilst setting the Local IP for the UDP Server.

While initiating the application, we need to handle some visual elements to transition from the starting screen to the main screen. Background execution is assigned with these tasks in order to avoid interrupting the constant redrawing of our GUI.

Finally, we developed a series of methods to update, maintain and handle events that occur from the GUI. An extensive guide at Chapter 8 is enough to give the user an idea of what every button does in our GUI, so we will be merely mentioning the methods in order to avoid redundancy.

5. MEASUREMENTS, HARDWARE COST AND PERFORMANCE

5.1 MEASUREMENTS

We downloaded our design to the FPGA and then proceed to remotely control the connected PC. We carried out a few tasks in a time span of ~19 minutes. Before presenting the measuring diagrams, here is a table of the activities we ran in the Cloud Server:

Time Span (min:sec)	Activity
00:01 – 02:16	Idle
02:17 – 03:53	Music
03:54 – 06:13	Idle
06:14 – 08:46	Music
08:47 – 09:28	Office
09:29 – 09:48	YouTube
09:49 – 11:15	Reading Email / Browsing Imgur
11:16 – 12:01	YouTube
12:02 – 12:07	Idle
12:08 – 13:05	Browsing Reddit with Music On
13:06 – 13:28	Browsing File Explorer
13:29 – 15:26	Watching a movie (1080p)
15:27 – 15:50	Muting sound on movie
15:51 – 16:48	Unmute – Keep watching
16:49 – 17:03	Launching Premiere
17:04 – 18:18	Windows 8 Game Application
18:19 – 18:30	Idle

**Idle state may contain a few mouse movements and Windows swapping. Also, applications were launched during the previous time span.*

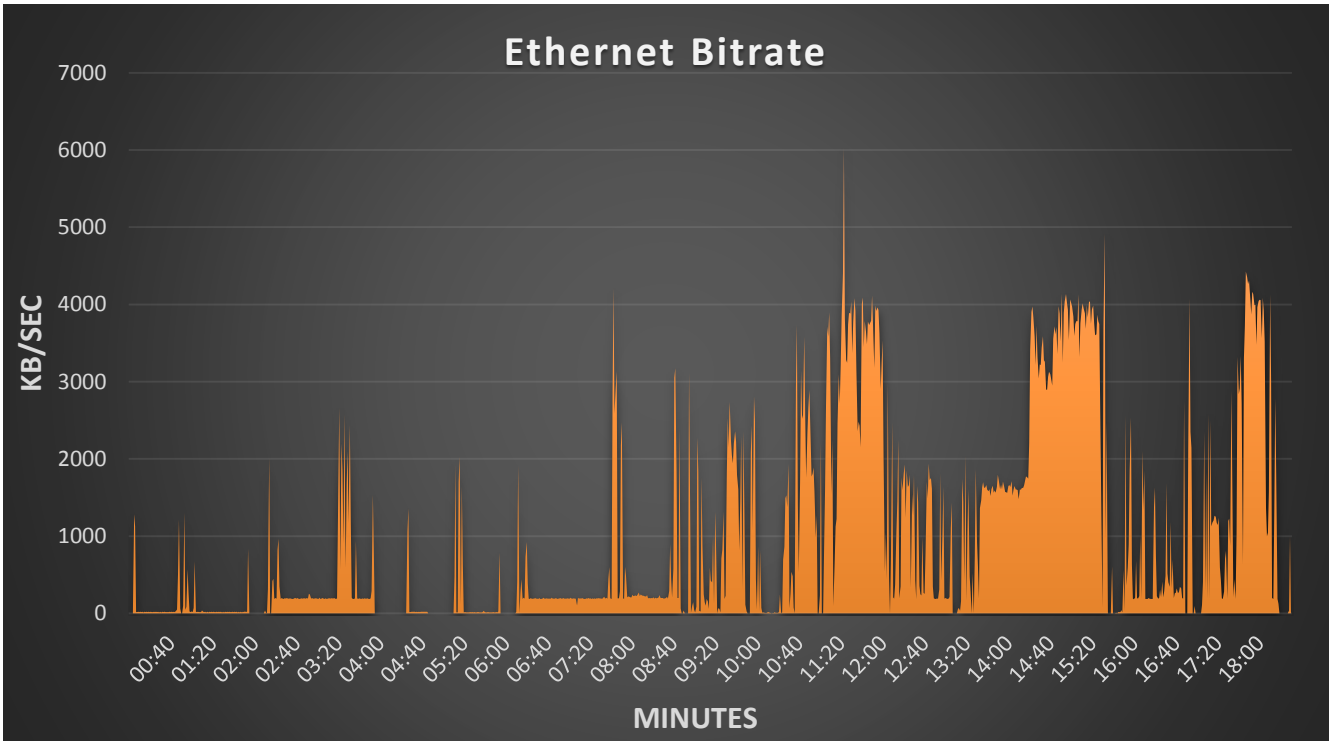


Figure 38: Overall Ethernet Bitrate Diagram

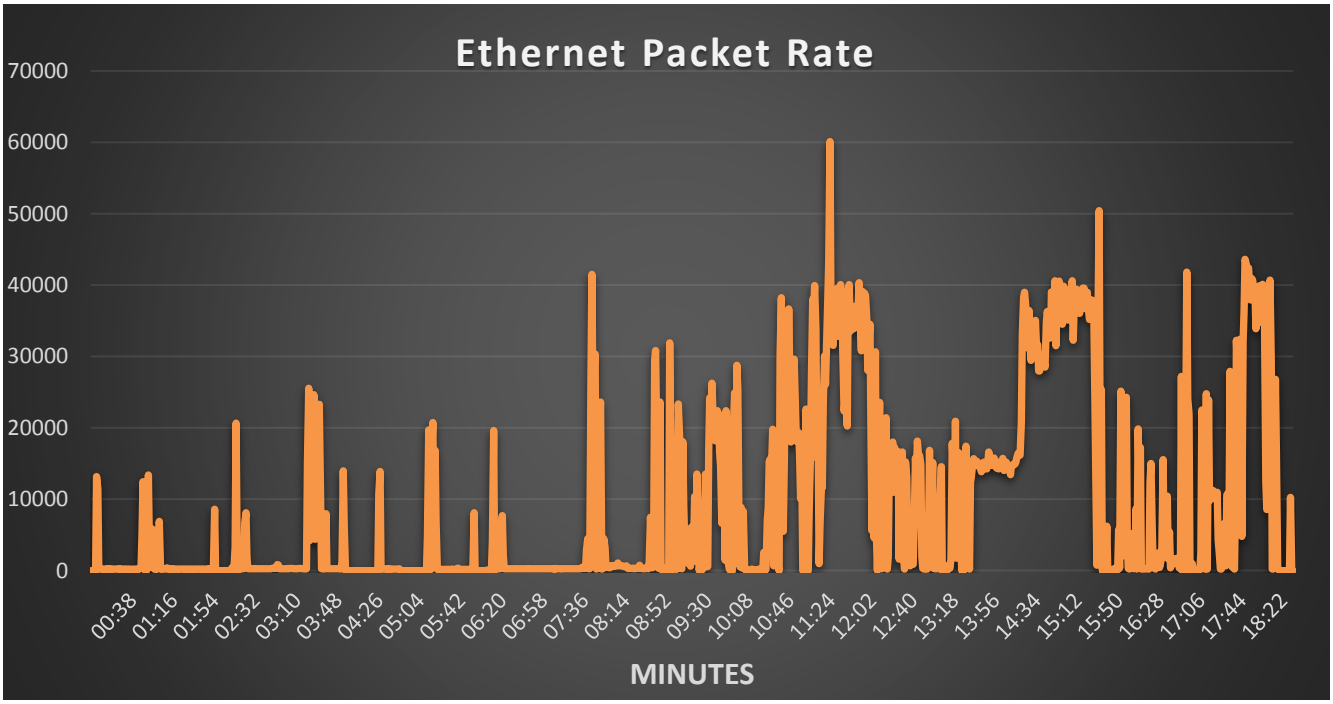


Figure 39: Overall Ethernet Packet Rate Diagram

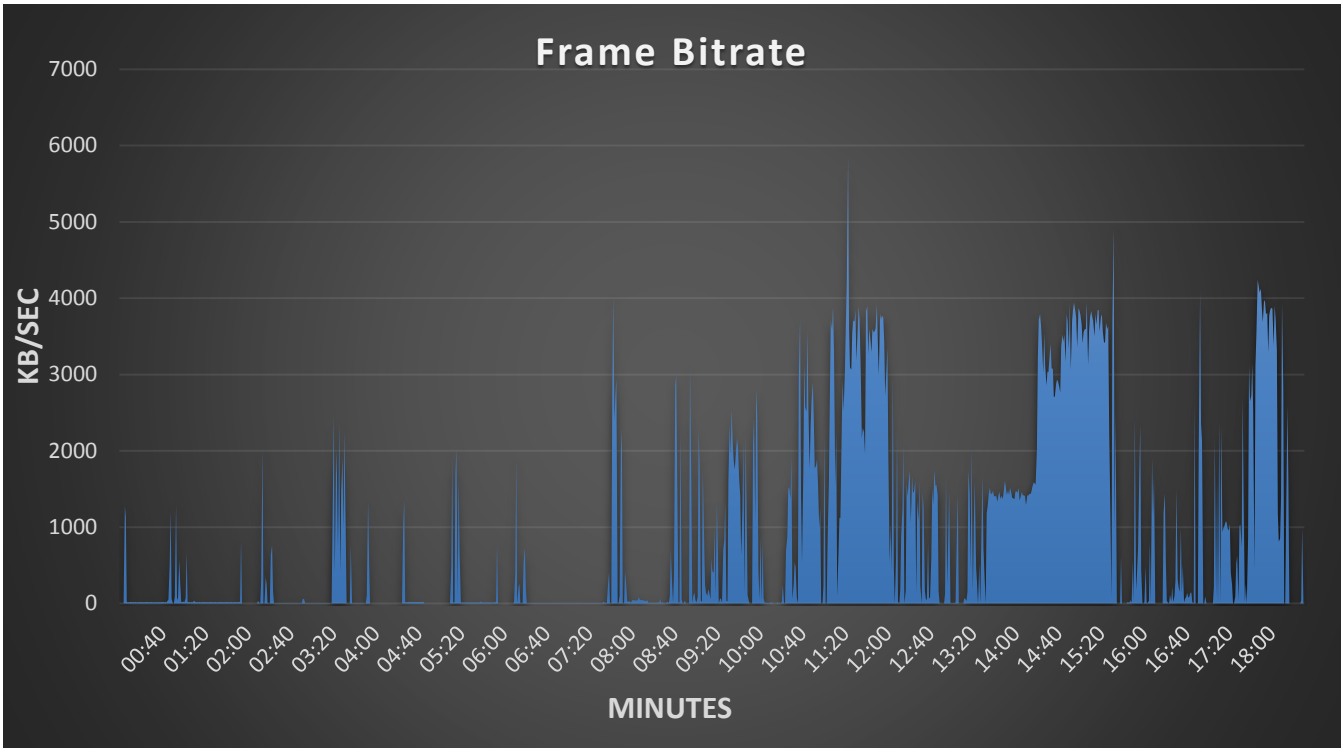


Figure 40: Overall Frame Bitrate Diagram

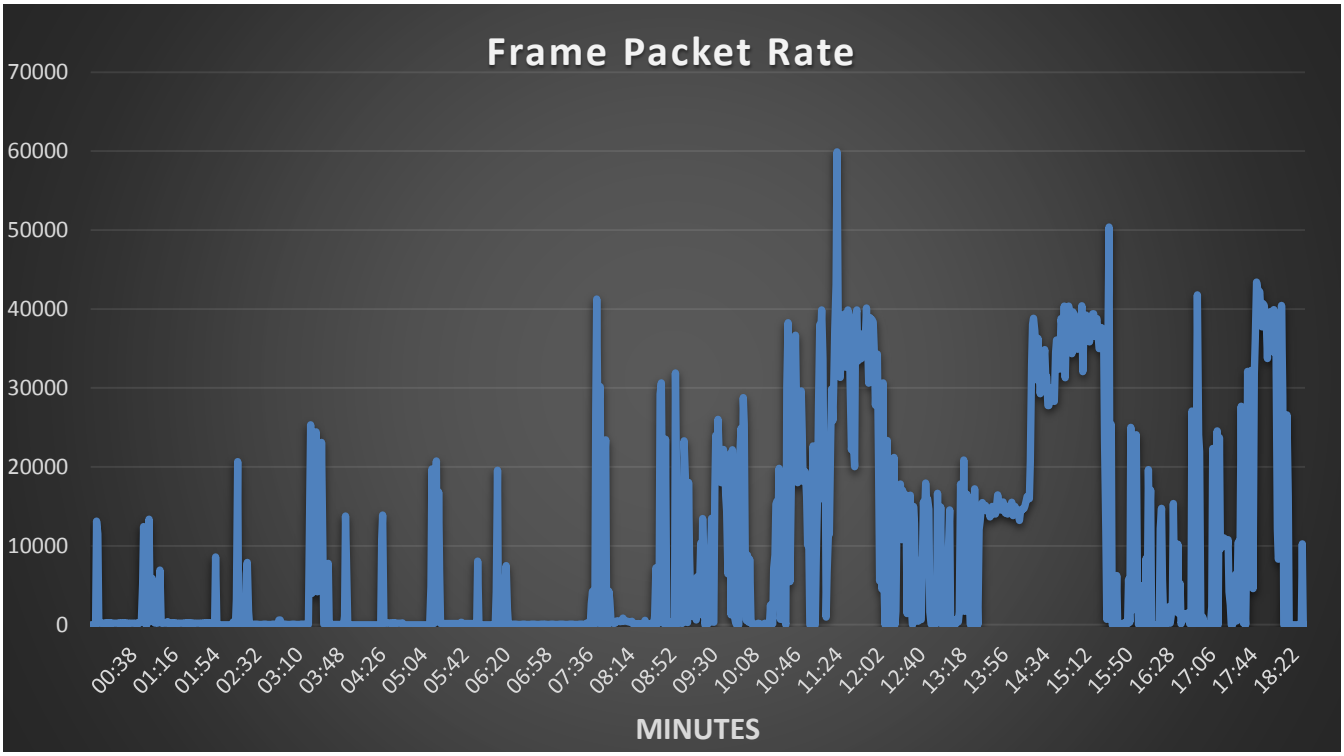


Figure 41: Overall Frame Packet Rate Diagram

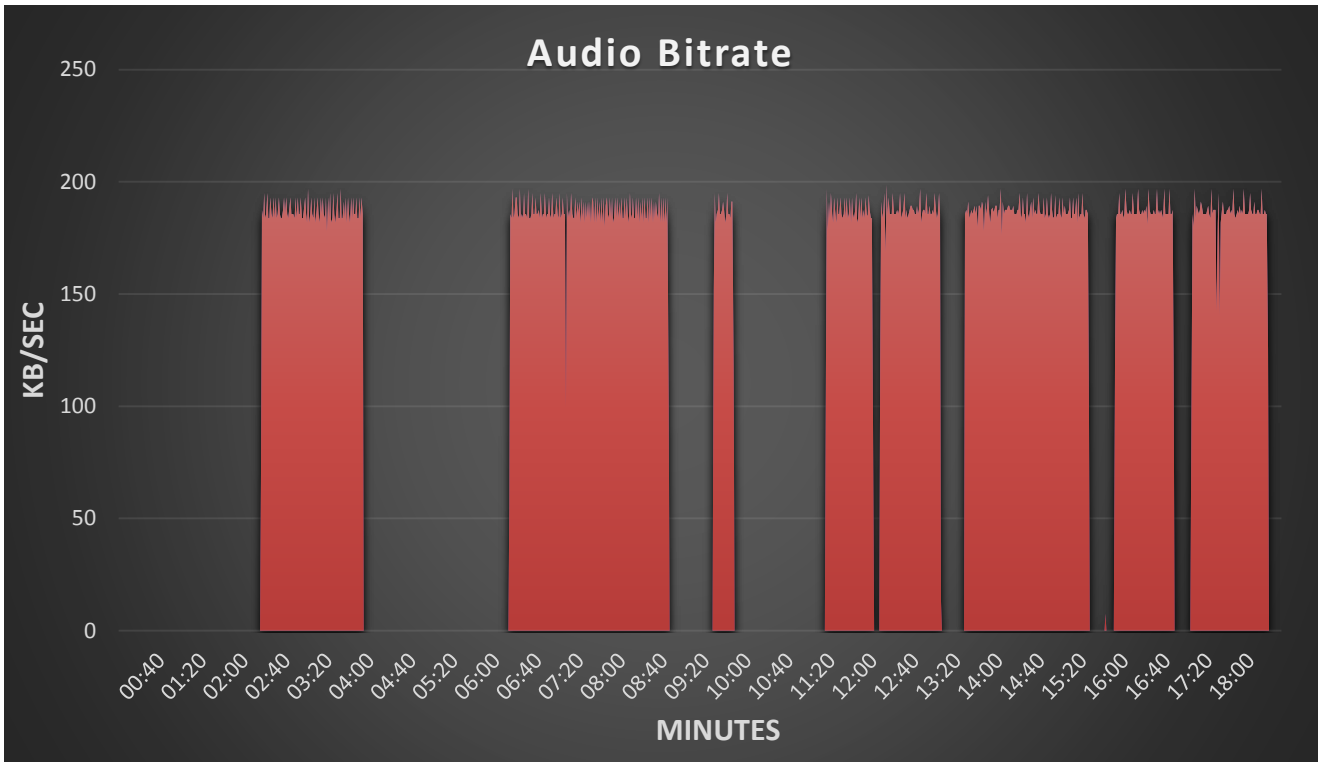


Figure 42: Overall Audio Bitrate Diagram

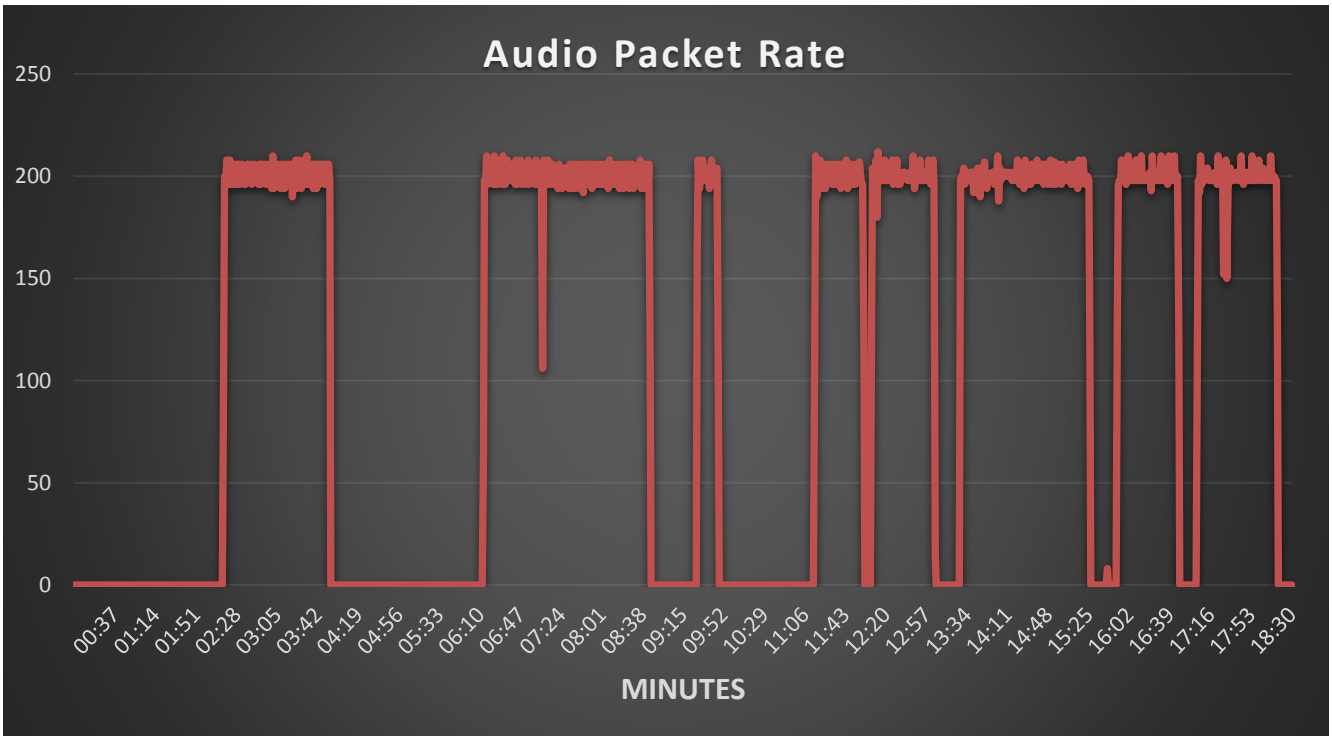


Figure 43: Overall Audio Packet Rate Diagram

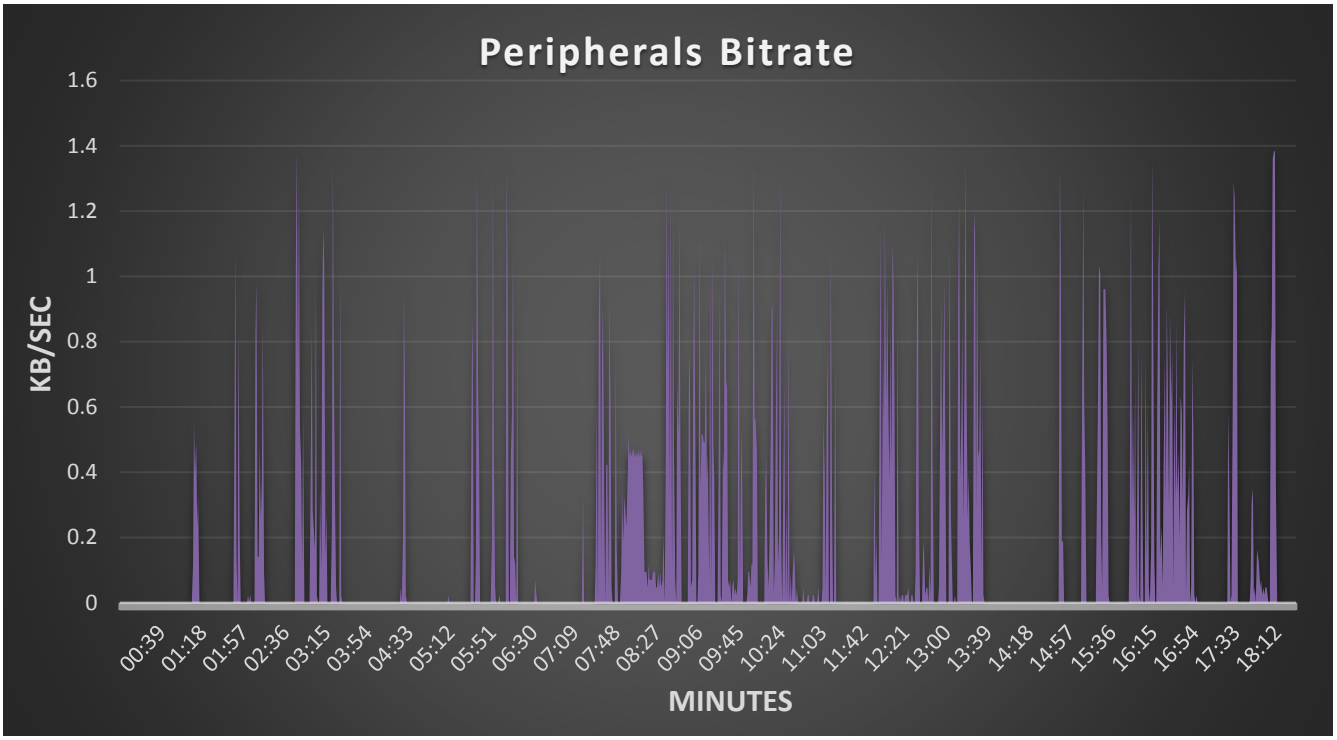


Figure 44: Overall Peripheral Bitrate Diagram

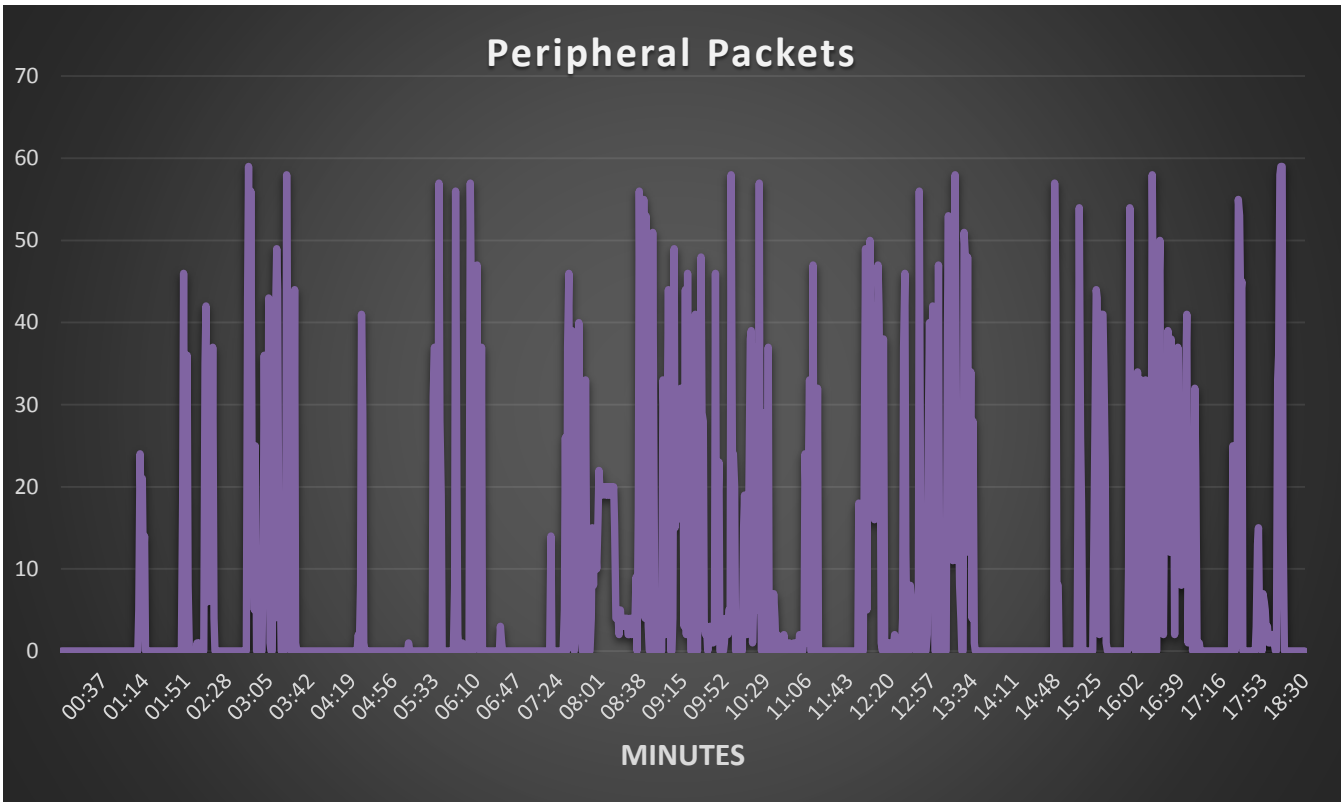


Figure 45: Overall Peripheral Packet Rate Diagram

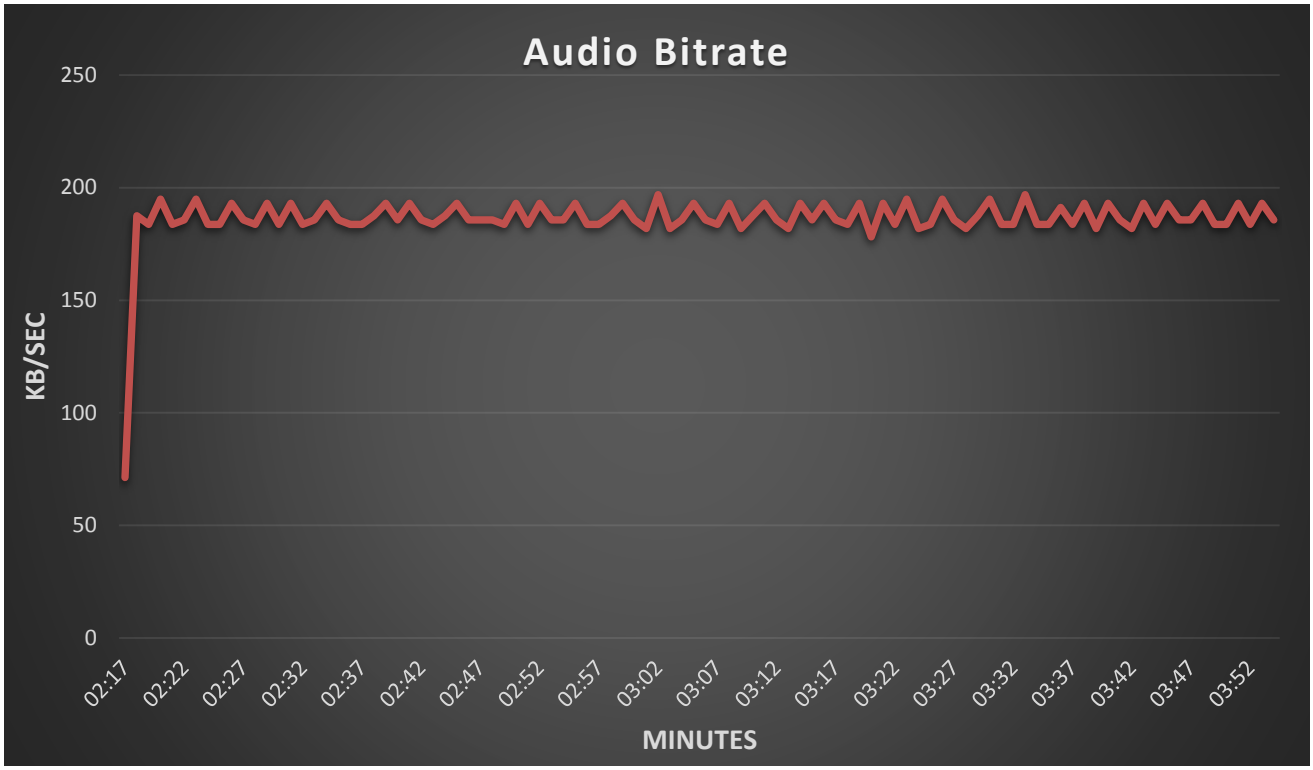


Figure 46: Audio Bitrate Diagram while listening to Music

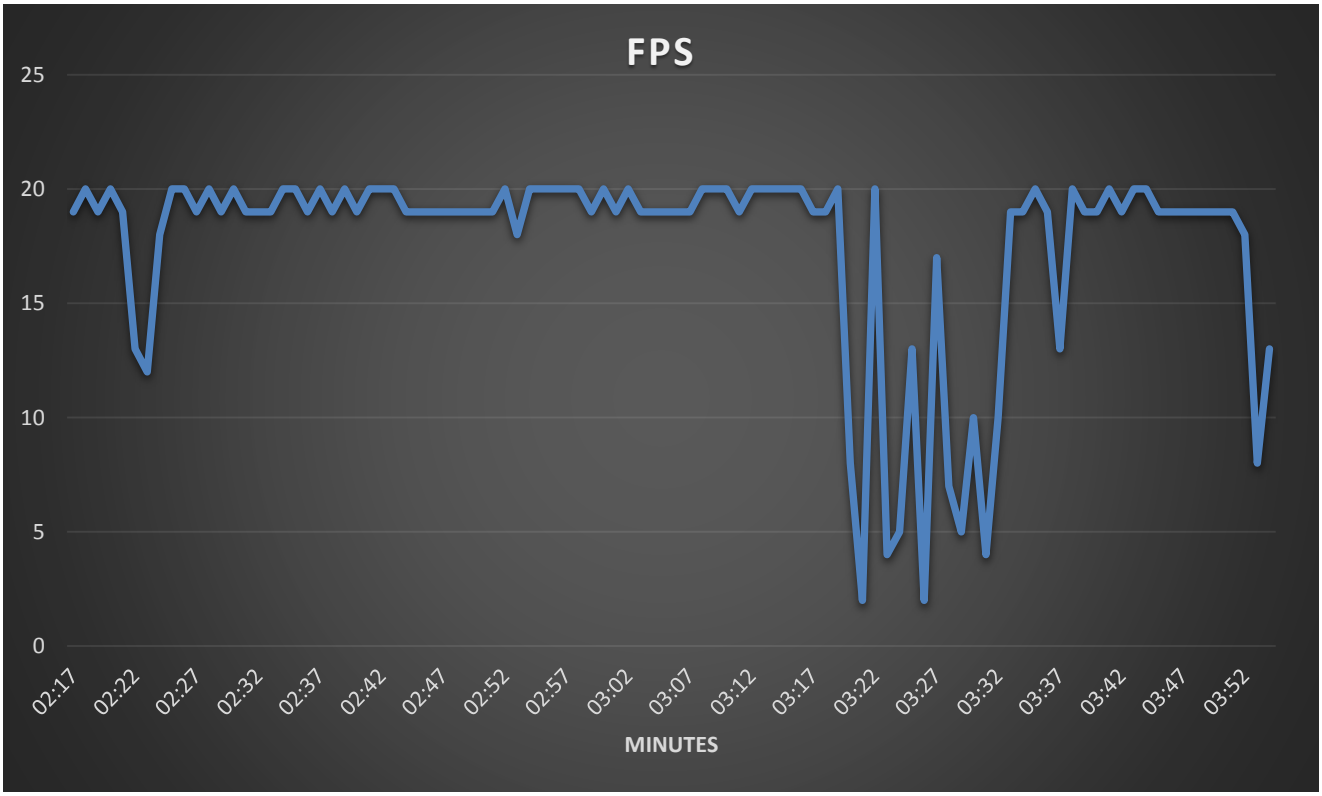


Figure 47: Frames Per Second (FPS) Diagram while listening to Music

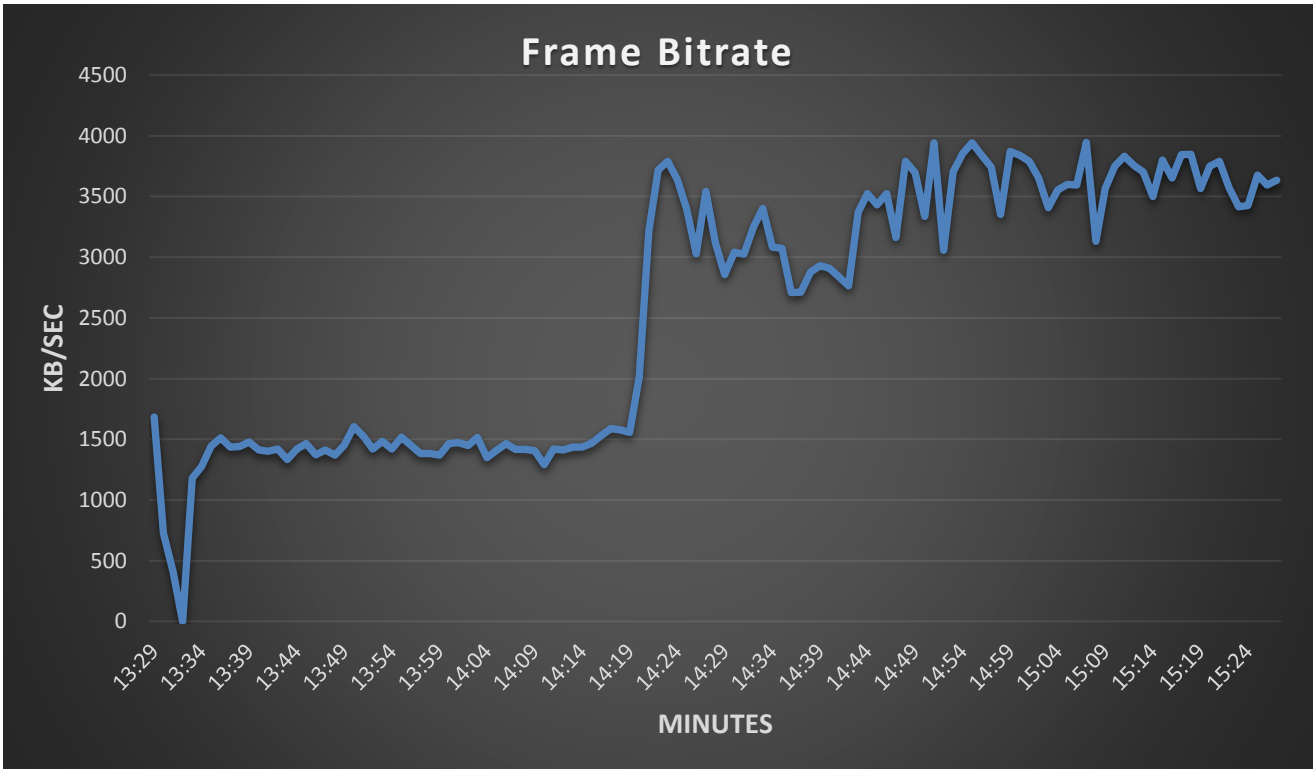


Figure 48: Frame Bitrate Diagram while watching a Movie

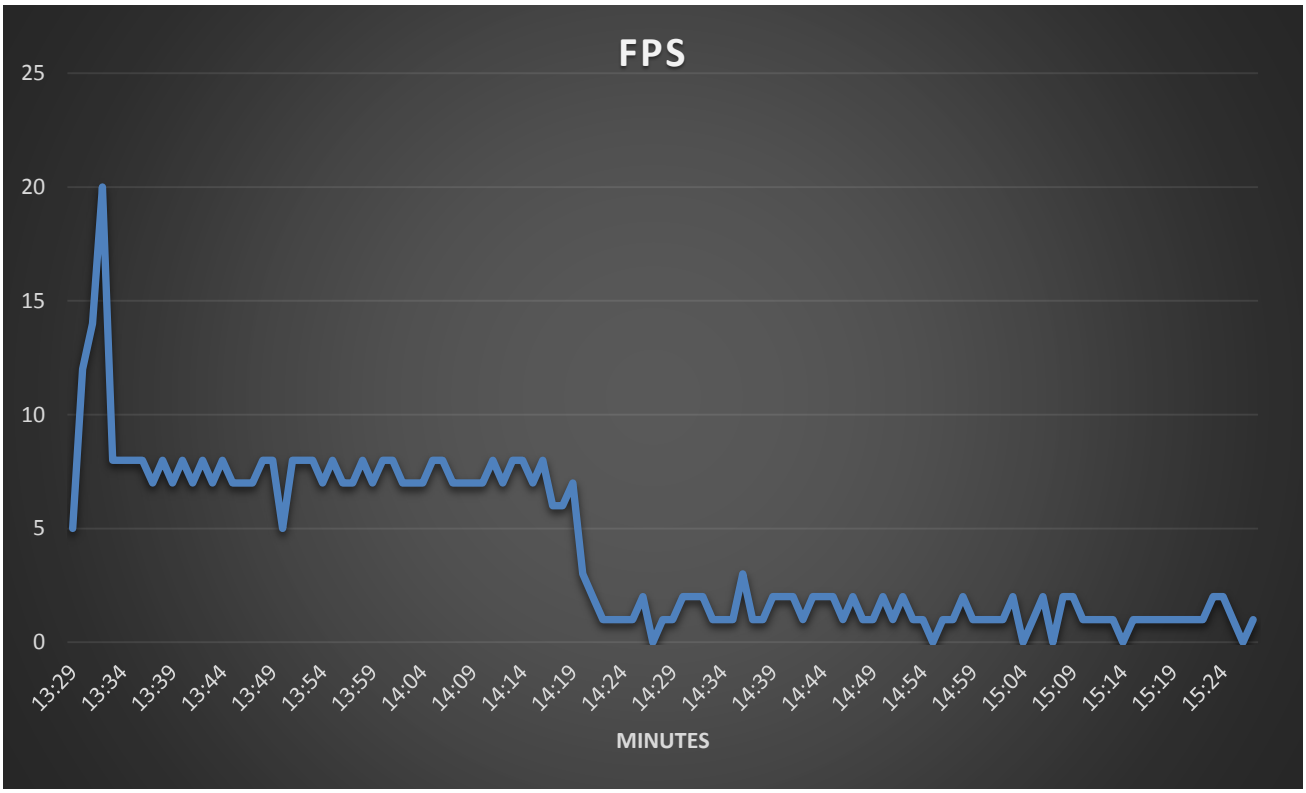


Figure 49: Frames per Second (FPS) Diagram while watching a Movie

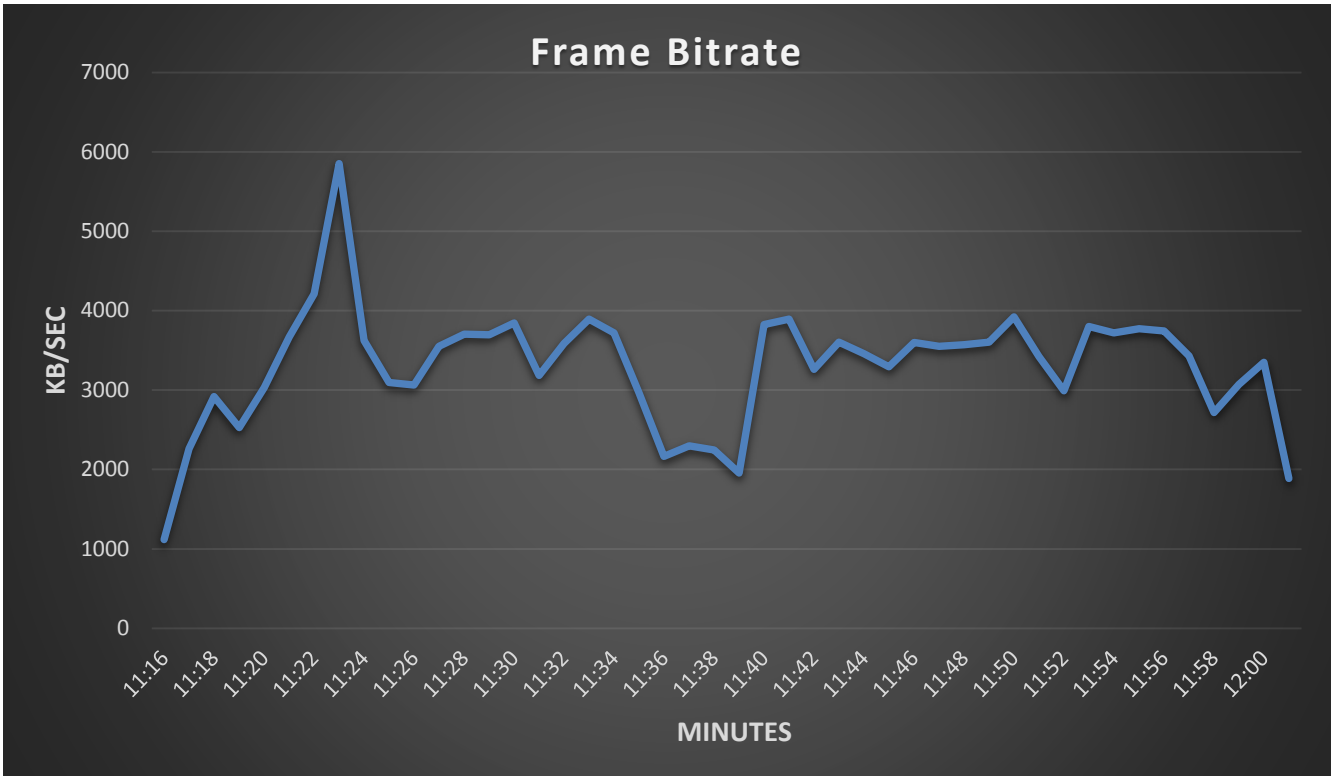


Figure 50: Frame Bitrate Diagram while watching a YouTube Video

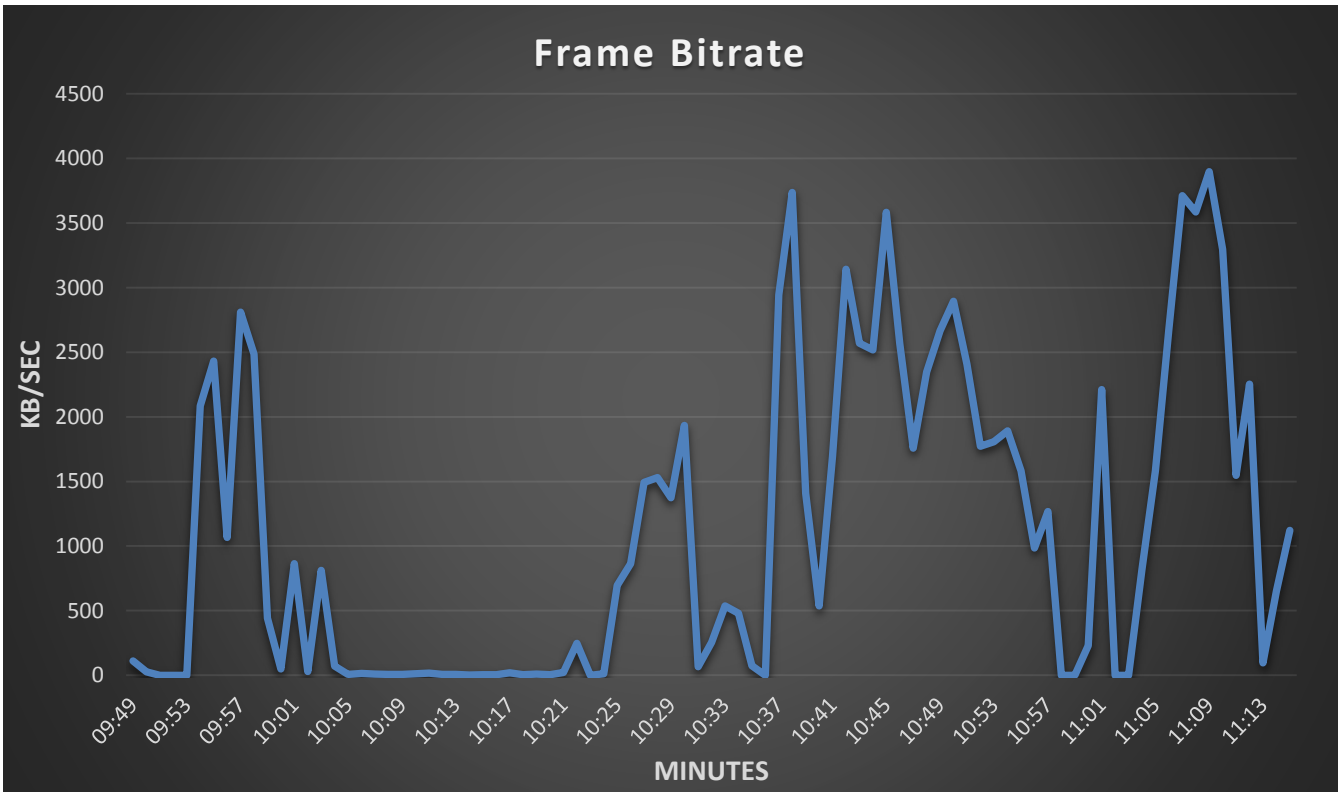


Figure 51: Frame Bitrate Diagram while Browsing the Internet

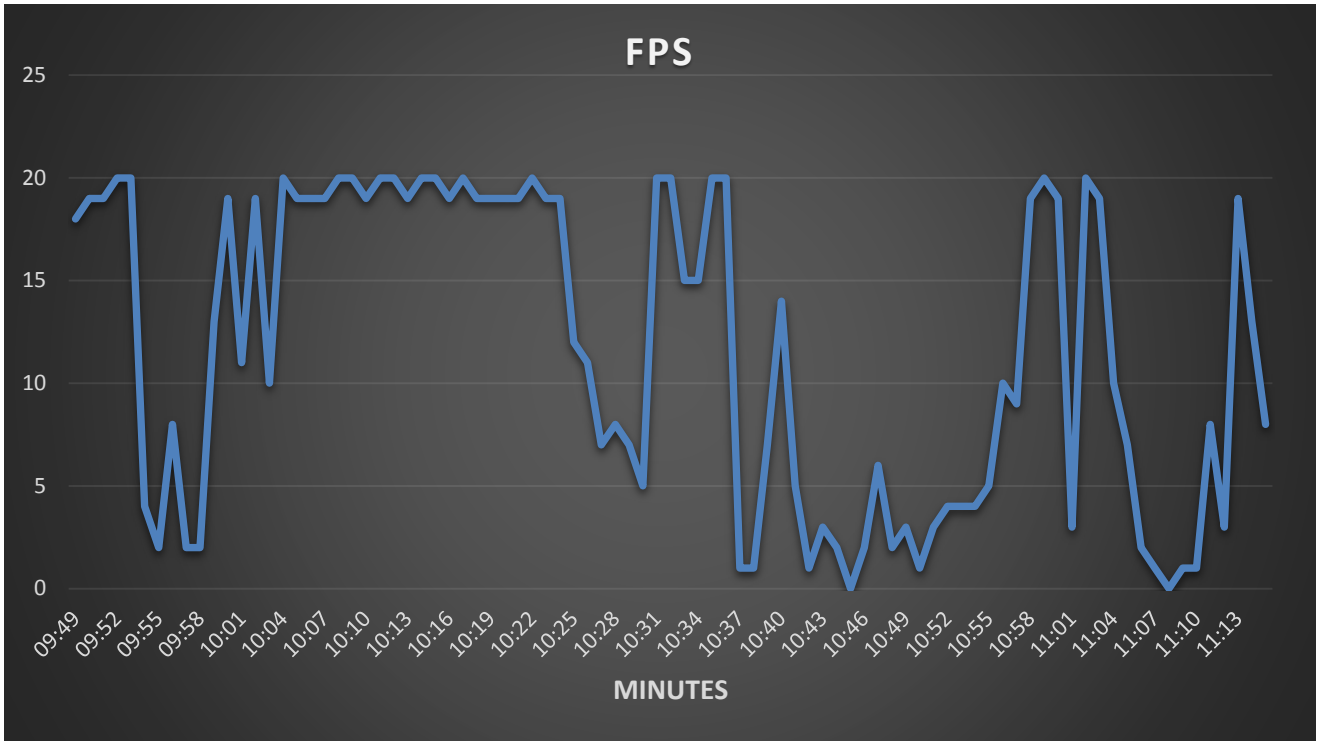


Figure 52: Frames per Second (FPS) Diagram while Browsing the Internet

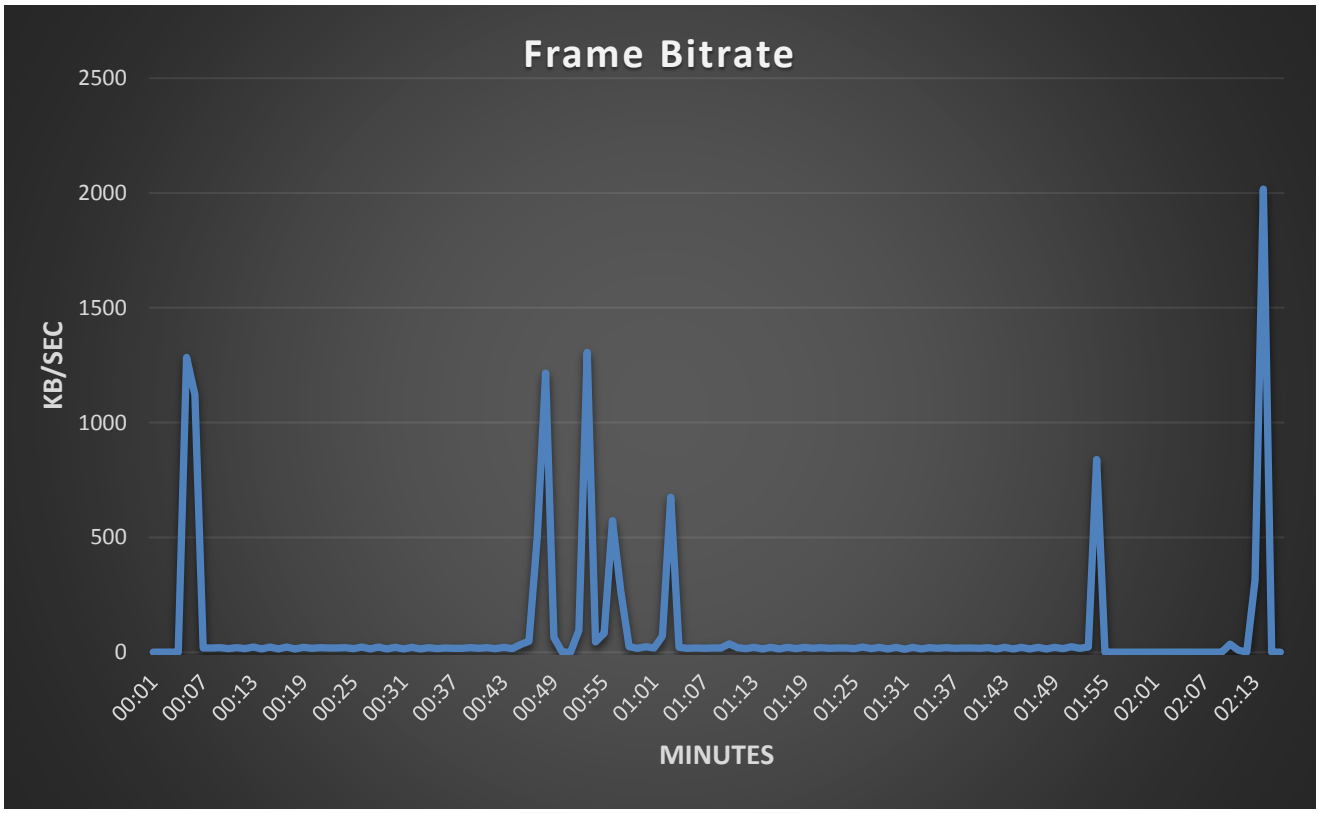


Figure 53: Frame Bitrate Diagram while Cloud Server is Idle

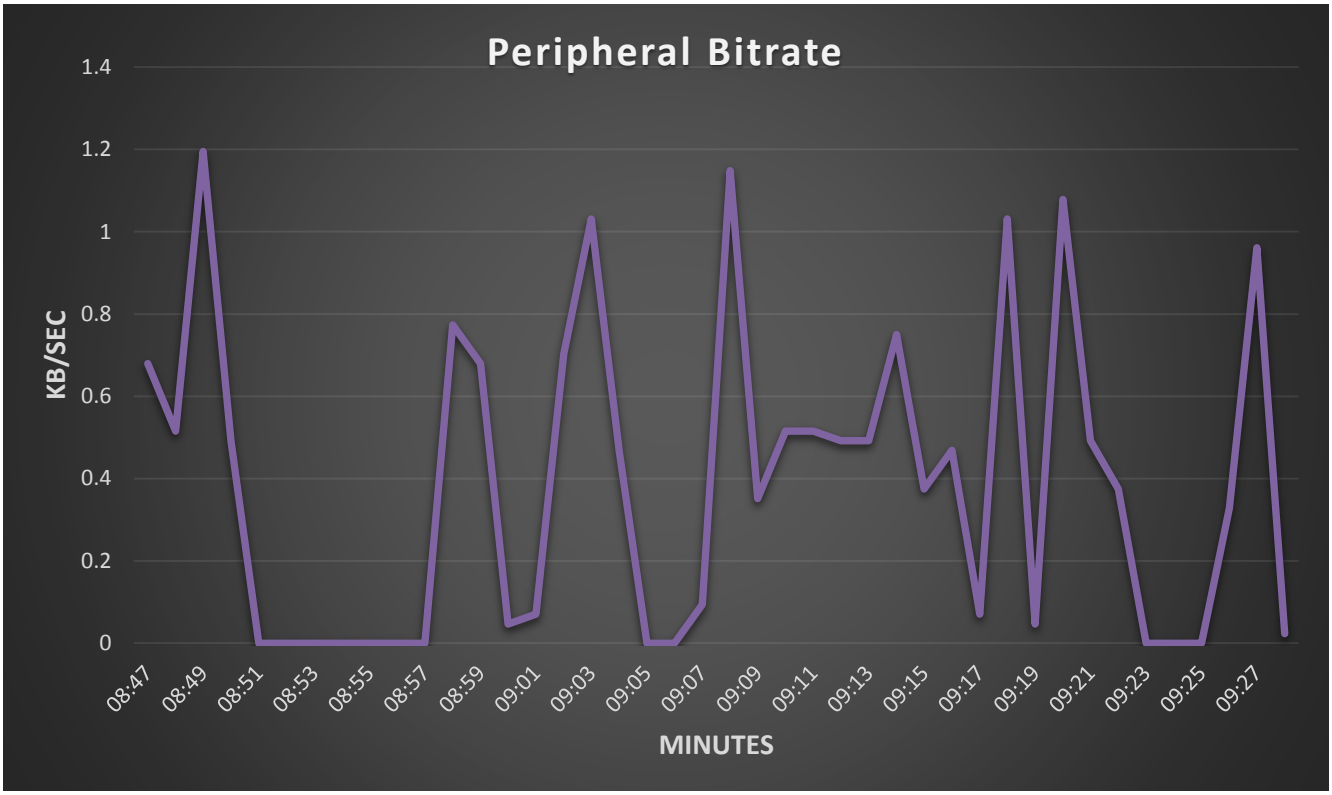


Figure 54: Peripheral Bitrate Diagram while using Office

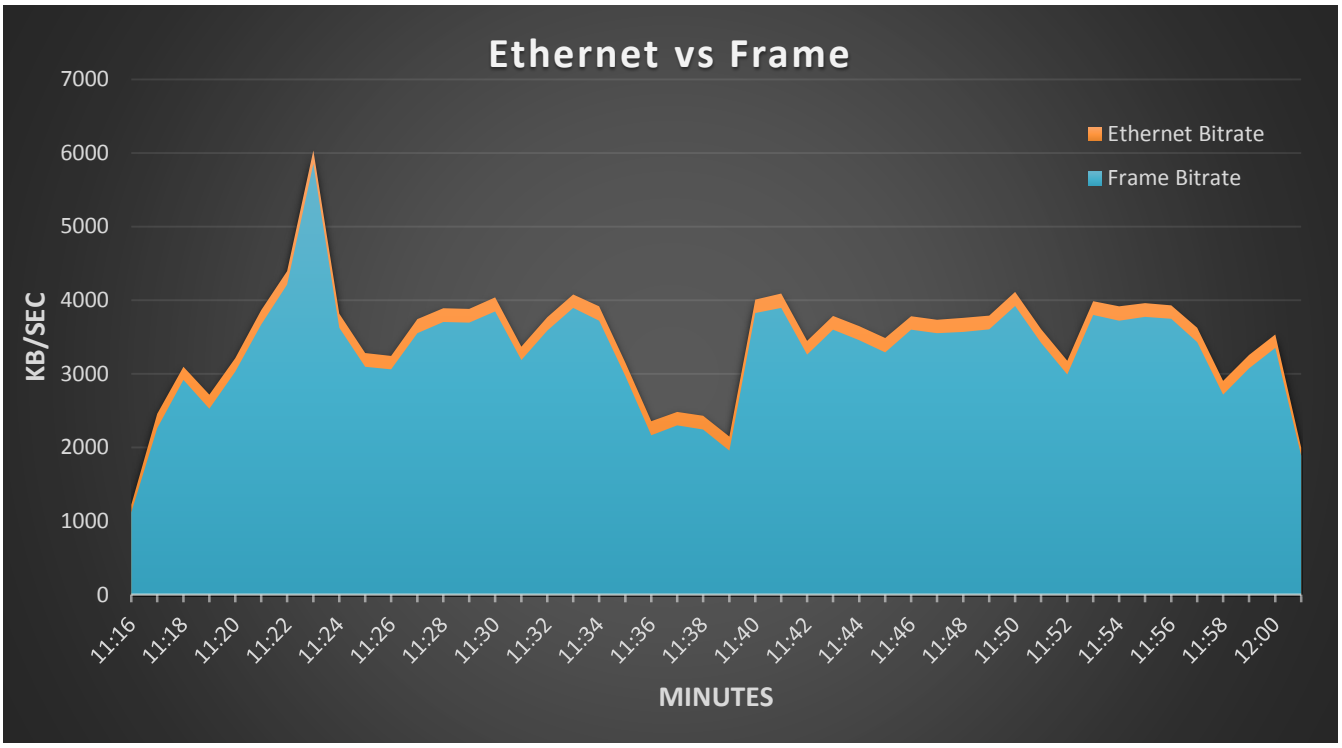


Figure 55: Ethernet vs Frame Bitrate while watching a YouTube Video

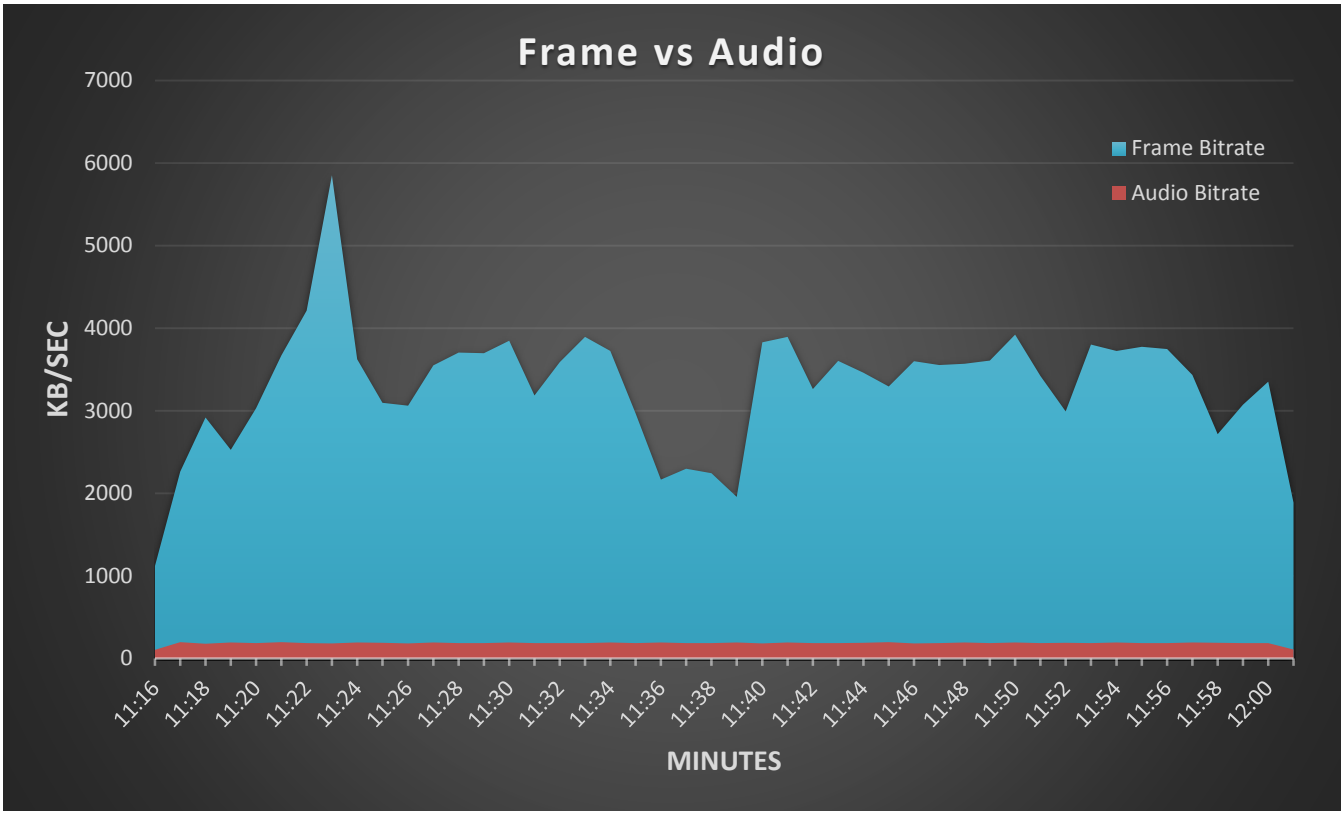


Figure 56: Frame vs Audio Bitrate while watching a YouTube Video

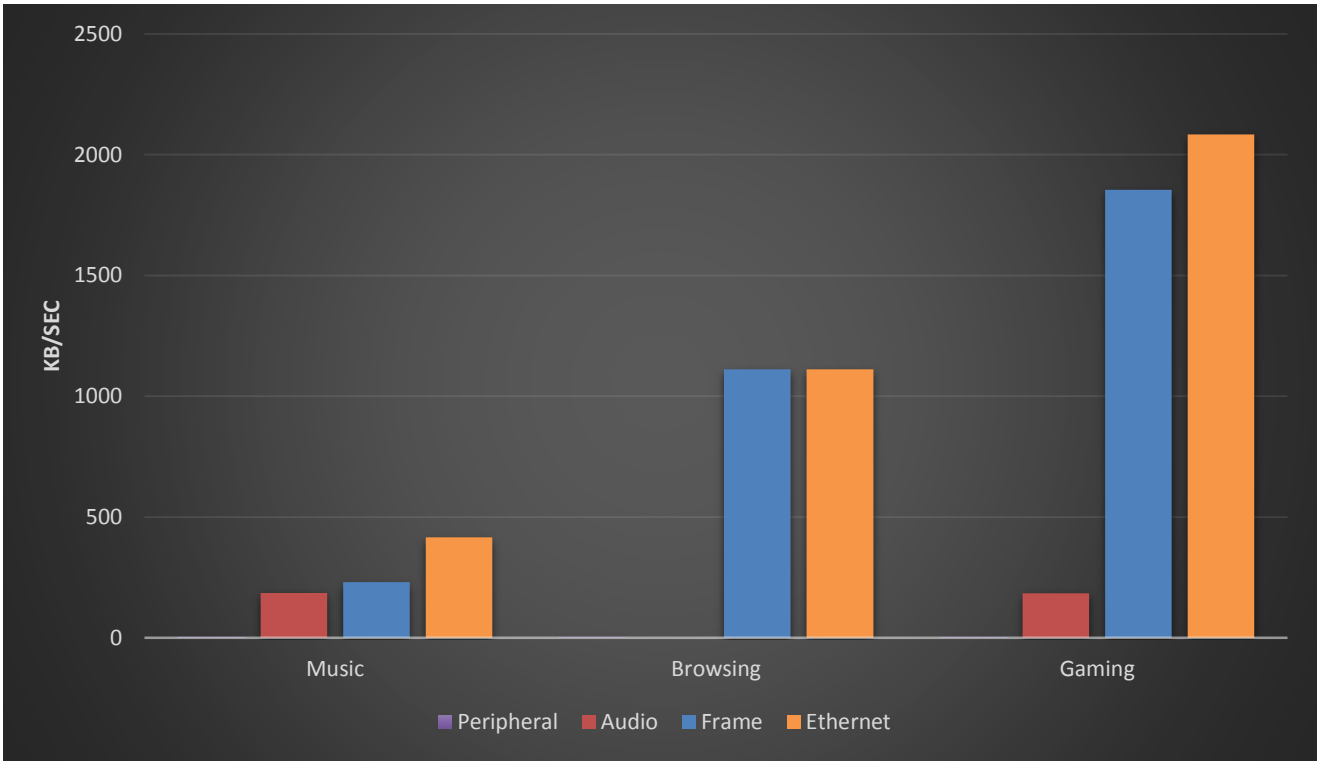


Figure 57: Usage while executing different Tasks

Studying the above diagrams, we are able to extract a large amount of useful information. First of all, we can distinguish the heavier and high demanding tasks from the light ones and in the future, redesign our application to cover our needs. Secondly, we identify which parts of the application are high demanding because we already know what task they are trying to accomplish at a given point. Finally, it is the simplest yet effective way for someone else to understand the impact that our application will have whilst running on a PC.

Some random thoughts:

- The Overall Ethernet/Frame Bitrate diagrams are pretty much the same, since the frame streaming process is the one needing the most bandwidth.
- When we are streaming Audio the Bitrate is constant because the source produces data at a standard rate.
- Peripheral Packets are received with a burst rate, because that is the nature of their creation. Also, since every event can be represented with a few Bytes, the Bitrate is never too high.
- While listening to Music, the FPS reached satisfying values. At the end of the diagram, we observed a drop which is explained by some Window swapping that are Frame-Costly.
- While watching a movie we observed higher Bitrate in action scenes where more different frames are drawn.
- Also, the FPS values change the opposite way than the Frame Bitrate.
- The Frame Bitrate was constantly low while we were Browsing and reading an HTML page, and high when we were scrolling down the page (Facebook).
- The Frame Bitrate is very low when the Cloud Server is idle.
- In the Audio-Frame comparison diagram we can verify again the constant Audio Bitrate, while Frame Bitrate depends on many things (frames drawn from the Cloud Server, algorithm and system workload).
- While carrying out a simple task, like listening to music, the average usage of Frame and Audio is pretty much the same.
- That is not the case on more complex task like Browsing, where even though there's no audio streaming, we can see that the Frame streaming would occupy the biggest part of the bandwidth anyways.

As it was expected, the "heavier" tasks proved to be the ones that produce new frames at high rate. These were the video playback, gaming and browsing which occasionally includes video playback. Interestingly, applications that were high demanding in computing resources, like Adobe Premiere, did not limit our application's effectiveness that much even though they acquired many workload quantums from the CPU task scheduler. Also, audio streaming seems to be working 100% no matter what. Keeping in mind that the audio stream has a constant Bitrate, it automatically makes this feature the most effective and reliable! Finally, tasks like typing a text or listening to music, even browsing text-based or image-based websites, produces a really satisfying ~20 FPS.

Unfortunately, in the following diagrams we can clearly see the large amount of computing resources that out application demands when operating with maximum demand. The CPU percentage usage increases drastically as our software is constantly taking screenshots and creates data packet out of the captured pixels. That's not good news for our application because high CPU usage also means high energy consumption and eventually it wears out of the chip. Also, considering the CPU chip we are actually using, an Intel i5 with a strong computation capability, it may be difficult to execute our application effectively in a weaker PC. Finally, we observed high Network usage, but that is only expected since we are striving for a top quality stream without using a sophisticated stream compression algorithm.

On the other hand, our application occupies a really small amount of Memory in the RAM, which is encouraging, considering how demanding the newly developed applications are. RAM usage is also a good indicator for a rough estimation of the simultaneous programs that a PC can run. Occupying that small amount of memory almost places our software in the Background Services category, which is good for the role our application should play.

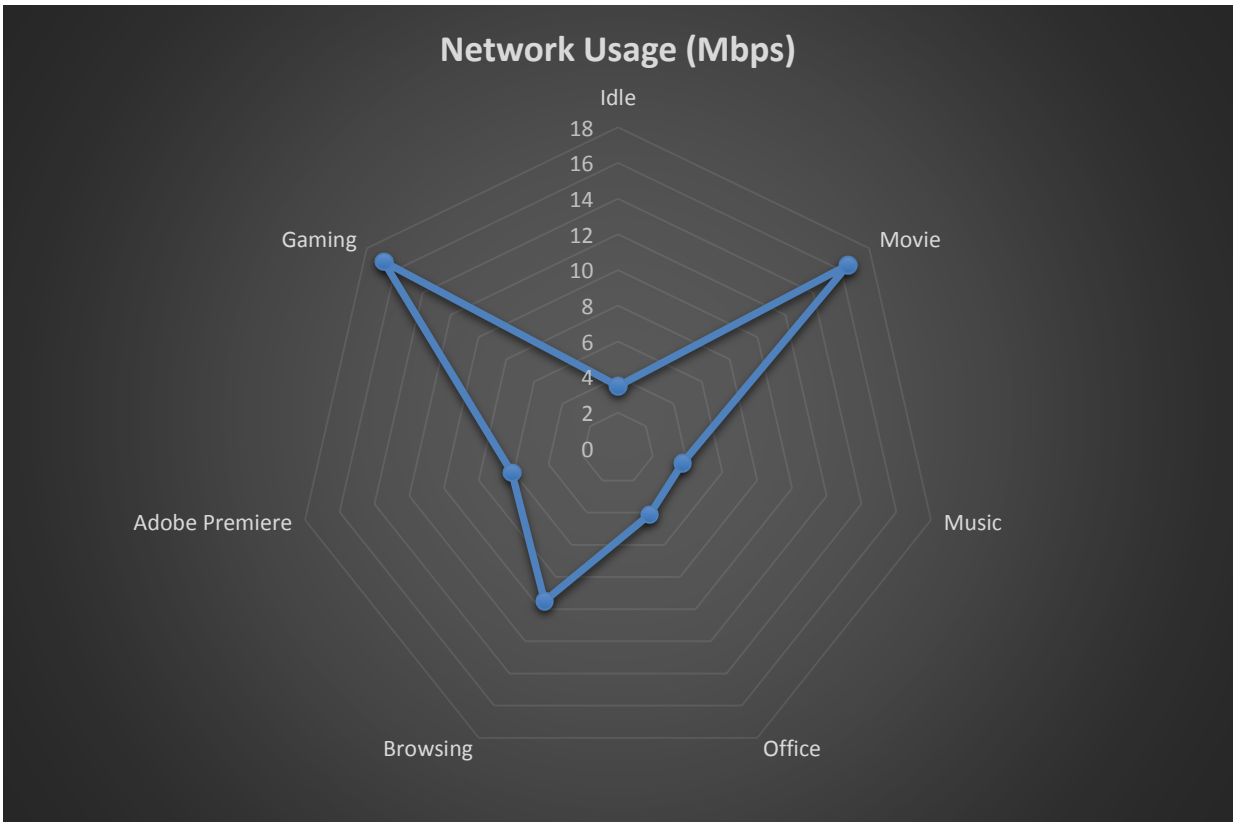


Figure 58: Server's Network Usage

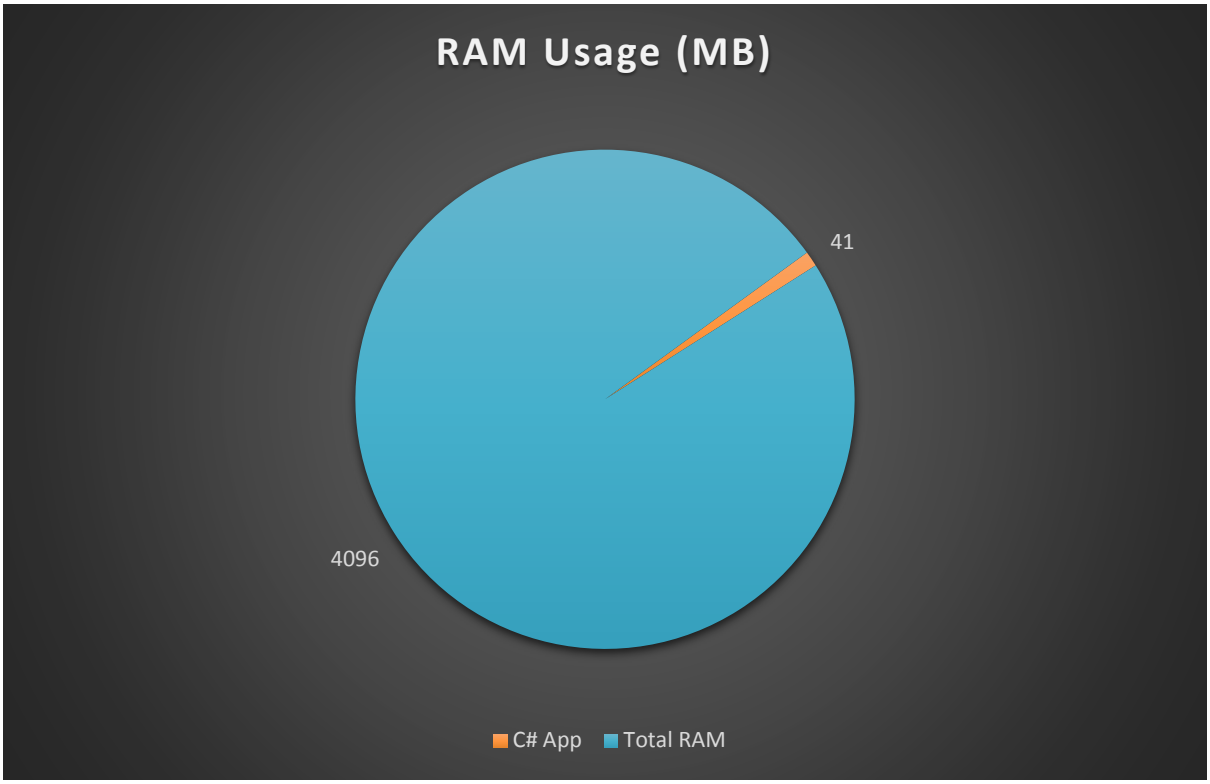


Figure 59: Server's RAM Usage

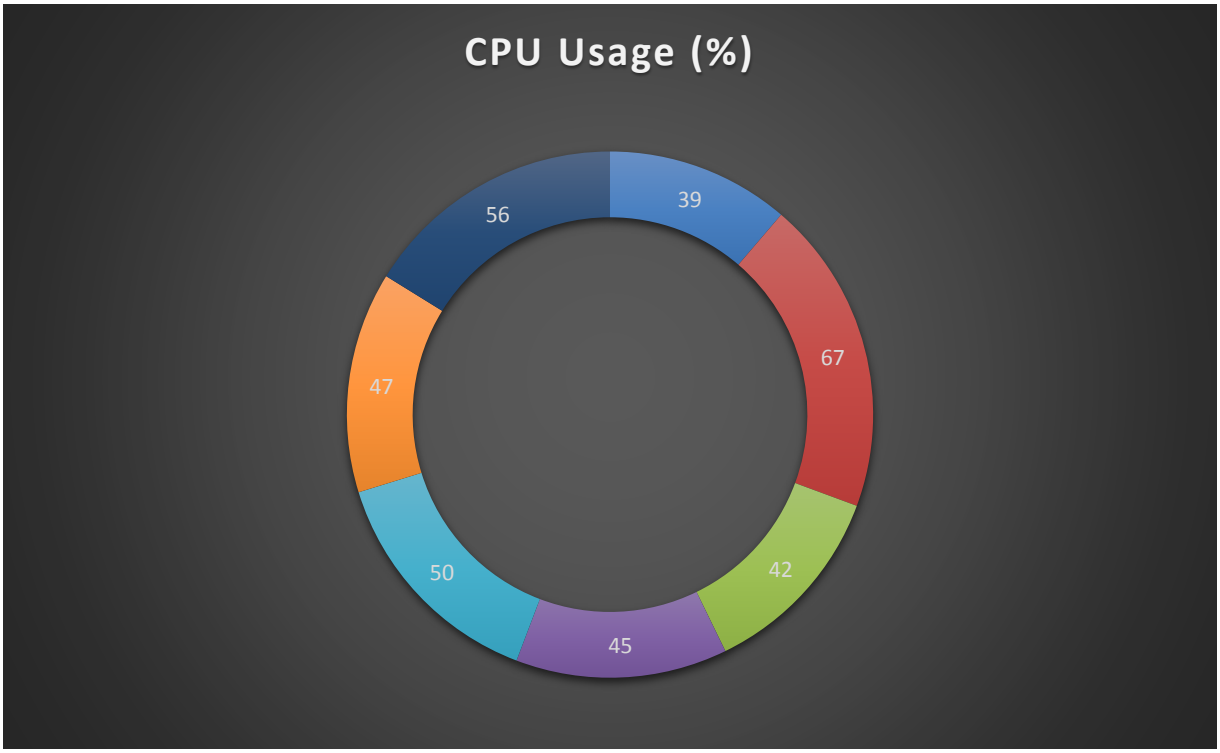


Figure 60: Server's CPU Usage

5.2 HARDWARE COST

Synthesis Report Summary

Source Parameters		
Input File Name : "example_design.prj"	Input Format : mixed	Ignore Synthesis Constraint File : NO
Target Parameters		
Output File Name : "example_design"	Output Format : NGC	Target Device : xc5vlx110t-1-ff1136
Source Options		
Top Module Name : example_design	Automatic FSM Extraction : YES	FSM Encoding Algorithm : Auto
Safe Implementation : No	FSM Style : LUT	RAM Extraction : Yes
RAM Style : Auto	ROM Extraction : Yes	Mux Style : Auto
Decoder Extraction : YES	Priority Encoder Extraction : Yes	Shift Register Extraction : YES
Logical Shifter Extraction : YES	XOR Collapsing : YES	ROM Style : Auto
Mux Extraction : Yes	Resource Sharing : YES	Asynchronous To Synchronous : NO
Use DSP Block : Auto	Automatic Register Balancing : No	
Target Options		
LUT Combining : Auto	Reduce Control Sets : Auto	Add IO Buffers : YES
Global Maximum Fanout : 100000	Add Generic Clock Buffer(BUFG) : 32	Register Duplication : YES
Slice Packing : YES	Optimize Instantiated Primitives : NO	Use Clock Enable : Auto
Use Synchronous Set : Auto	Use Synchronous Reset : Auto	Pack IO Registers into IOBs : Auto
	Equivalent register Removal : YES	
General Options		
Optimization Goal : Speed	Optimization Effort : 1	Power Reduction : NO
Keep Hierarchy : No	Netlist Hierarchy : As_Optimized	RTL Output : Yes
Global Optimization : AllClockNets	Read Cores : YES	Write Timing Constraints : NO
Cross Clock Analysis : NO	Hierarchy Separator : /	Bus Delimiter : <>
Case Specifier : Maintain	Slice Utilization Ratio : 100	BRAM Utilization Ratio : 100
DSP48 Utilization Ratio : 100	Verilog 2001 : YES	Auto BRAM Packing : NO
	Slice Utilization Ratio Delta : 5	

HDL Synthesis Report

Macro Statistics

# ROMs : 1				
		16x25-bit ROM : 1		
# Multipliers : 2				
		3x3-bit multiplier : 2		
# Adders/Subtractors : 47				
12-bit adder : 2	12-bit subtractor : 4	16-bit adder : 3	16-bit adder car/out:2	16-bit subtractor : 1
17-bit adder car/out :1	2-bit adder : 2	3-bit adder : 4	32-bit adder : 3	5-bit adder : 2
5-bit subtractor : 2	6-bit adder : 4	6-bit adder car/out :4	6-bit subtractor : 3	7-bit adder : 3
7-bit addsub : 1	8-bit adder : 2	8-bit adder car/out :2	8-bit subtractor : 1	9-bit adder : 1
# Counters : 130				
8-bit up counter : 1	10-bit up counter : 1	11-bit up counter : 1	12-bit up counter : 4	14-bit down counter :2
15-bit down counter :2	16-bit up counter : 2	2-bit down counter : 1	2-bit up counter : 2	3-bit down counter : 3
3-bit up counter : 4	32-bit up counter : 3	4-bit down counter : 2	4-bit up counter : 8	5-bit down counter : 5
5-bit up counter : 1	6-bit up counter : 1	6-bit updown cntr:82	7-bit up counter : 1	8-bit down counter : 2
		9-bit updown cntr:2		
# Accumulators : 4				
12bit updown accmltr:1		2bit up accmltr: 1		32-bit up accmltr: 2
# Registers : 2786				
1-bit register : 2616	2-bit register : 11	12-bit register : 18	128-bit register : 2	15-bit register : 1
16-bit register : 14	20-bit register : 2	24-bit register : 1	25-bit register : 4	28-bit register : 1
3-bit register : 13	31-bit register : 6	32-bit register : 9	4-bit register : 5	48-bit register : 1
5-bit register : 6	6-bit register : 9	6144-bit register : 2	64-bit register : 8	7-bit register : 4
	8-bit register : 42		768-bit register : 11	

# Comparators : 54				
11bit comparator >= : 5	11bit comparator < :4	13-bit comparator !=: 4	15-bit comparator = : 1	16-bit comparator = :2
17-bit comparator != :1	2-bit comparator = : 5	2-bit comparator >= : 1	2-bit comparator < : 2	3-bit comparator >= :2
3-bit comparator <= :3	32bit comparator >= :4	32bit comparator less :5	4bit comparator >= : 1	4-bit comparator <= :1
5-bit comparator = : 1	5-bit comparator >= : 1	5-bit comparator < : 1	5-bit comparator <= : 5	6-bit comparator < : 1
7-bit comparator !=: 1		8-bit comparator = : 2		8bit comparator <= : 1
# Multiplexers : 15				
1-bit 40-to-1 multiplexer : 5		1-bit 64-to-1 multiplexer : 2		1-bit 8-to-1 multiplexer : 8
# Tristates : 4				
		1-bit tristate buffer : 4		
# Xors : 43				
1-bit xor2 : 39		1-bit xor8 : 2		1-bit xor9 : 2

Advanced HDL Synthesis Report

Macro Statistics

# FSMs : 28				
# ROMs : 1				
		16x25-bit ROM : 1		
# Multipliers : 2				
	3x3-bit registered multiplier : 2			
# Adders/Subtractors : 45				
12-bit adder : 2	12-bit subtractor : 4	16-bit adder : 3	16bit adder car/out : 2	16-bit subtractor : 1
17bit adder car/out : 1	3-bit adder : 4	32-bit adder : 3	5-bit adder : 2	5-bit subtractor : 2
6-bit adder : 4	6-bit adder car/out : 4	6-bit subtractor : 3	7-bit adder : 3	7-bit addsub : 1
8-bit adder : 2	8-bit adder car/out : 2	8-bit subtractor : 1	9-bit adder : 1	
# Counters : 128				
10-bit up counter : 1	11-bit up counter : 1	12-bit up counter : 4	14-bit down counter :2	15-bit down counter :2
16-bit up counter : 2	2-bit down counter : 1	2-bit up counter : 2	3-bit down counter : 3	3-bit up counter : 4
32-bit up counter : 3	4-bit down counter : 1	4-bit up counter : 8	5-bit down counter : 4	5-bit up counter : 1
6-bit up counter : 1	6-bit updown cntr :82	7-bit up counter : 1	8-bit down counter : 2	8-bit up counter : 1
		9-bit updown cntr :2		
# Accumulators : 4				
12bit updown accmltr:1		2-bit up accmltr: 1		32-bit up accmltr: 2
# Registers : 26233				
		Flip-Flops : 26233		
# Comparators : 54				
11-bit cmprator >= :5	11-bit comparator < : 4	13-bit cmprator!= : 4	15-bit comparator = : 1	16-bit cmprator = : 2
17-bit cmprator!= : 1	2-bit comparator = : 5	2-bit comparator >= : 1	2-bit comparator < : 2	3-bit cmprator >= : 2
3-bit cmprator <= : 3	32-bit cmprator >= : 4	32-bit comparator < : 5	4-bit cmprator >= : 1	4-bit cmprator <= : 1
5-bit comparator = : 1	5-bit comparator >= : 1	5-bit comparator < : 1	5-bit comparator <= : 5	6-bit comparator < : 1
7-bit comparator!=: 1		8-bit comparator = : 2		8-bit cmparator <= : 1
# Multiplexers : 15				
1-bit 40-to-1 multiplexer : 5		1-bit 64-to-1 multiplexer : 2		1-bit 8-to-1 multiplexer : 8
# Xors : 43				
1-bit xor2 : 39		1-bit xor8 : 2		1-bit xor9 : 2

Final Register Report

Macro Statistics

# Registers : 26084				
		Flip-Flops : 26084		
# Shift Registers : 26				
10-bit shift register : 1	12-bit shift register : 1	15-bit shift register : 3	16-bit shift register : 1	2-bit shift register : 10
	24-bit shift register : 1		3-bit shift register : 9	

Final Report

Final Results

RTL Top Level Output File Name : example_design.ngr				
Top Level Output File Name : example_design				
Output Format : NGC	Optimization Goal : Speed		Keep Hierarchy : No	
Design Statistics				
	# IOs : 220			
Cell Usage :				
# BELS : 13570	# GND : 2	# INV : 183	# FDCPE_1 : 8	# FDE : 6808
# LUT1 : 389	# LUT2 : 792	# LUT3 : 1258	# FDP : 431	# FDPE : 34
# LUT4 : 1924	# LUT5 : 1375	# LUT6 : 6272	# FDR : 937	# FDRE : 16190
# MUXCY : 665	# MUXF7 : 76	# VCC : 2	# FDRS : 93	# FDRSE : 379
# XORCY : 632	# FlipFlops/Latches : 26367	# FD : 947	# FDRSE_1 : 128	# FDS : 60
# FD_1 : 9	# FDC : 59	# FDCE : 15	# FDS_1 : 2	# FDSE : 29
# FDCPE : 21	# IDDR_2CLK : 64	# ODDR : 153	# RAMS : 49	# RAMB16 : 4
# RAMB18 : 4	# RAMB36_EXP : 41	# Shift Registers : 43	# SRLC16E : 25	# SRLC32E : 18
# Clock Buffers : 13	# BUFG : 12	# BUFGP : 1	# IO Buffers : 209	# IBUF : 39
IBUFG : 1	## IOBUF : 70	# IOBUFDS : 8	# OBUF : 89	# OBUFDS : 2
# DCM_ADVs : 2	# DCM_ADV : 2	# Others : 107	# BUFIO : 8	# FIFO36_72_EXP : 2
# FIFO36_EXP : 1	# IDELAYCTRL : 2	# IODELAY : 91	# PLL_ADV : 2	# TEMAC : 1

Device utilization summary

Selected Device: 5vlx110tff1136-1

Slice Logic Utilization	
Number of Slice Registers	26357 out of 69120 **38%**
Number of Slice LUTs	12236 out of 69120 **17%**
Number used as Logic	12193 out of 69120 **17%**
Number used as Memory	43 out of 17920 **0%**
Number used as SRL	43
Slice Logic Distribution	
Number of LUT Flip Flop pairs used	34418
Number with an unused Flip Flop	8061 out of 34418 **23%**
Number with an unused LUT	22182 out of 34418 **64%**
Number of fully used LUT-FF pairs	4175 out of 34418 **12%**
Number of unique control sets	749
IO Utilization	
Number of IOs	220
Number of bonded IOBs	220 out of 640 **34%**
IOB Flip Flops/Latches	10
Specific Feature Utilization	
Number of Block RAM/FIFO	45 out of 148 **30%**
Number using Block RAM only	45
Number of BUFG/BUFGCTRLs	13 out of 32 **40%**
Number of DCM_ADVs	2 out of 12 **16%**

Number of PLL_ADVs	2 out of 6 **33%**
Timing Summary	
Speed Grade	-1
Minimum period	7.677ns (Maximum Frequency: 130.264MHz)
Minimum input arrival time before clock	3.089ns
Maximum output required time after clock	5.595ns
Maximum combinational path delay	4.526ns
Timing Detail	
Total REAL time to Xst completion	798.00 sec
Total CPU time to Xst completion	797.84 sec
Total memory usage is 1052380 kilobytes	

:

Utilization by Hierarchy

Module Name	Partition	Slices	Slice Reg	LUTs	LUTRAM	BRAM/FIFO	DSP48E	BUFG	BUFIO	BUFR	DCM_ADV	PLL_ADV
emac_example_design		11/10787	27/26016	9/11916	0/36	0/48	0/0	5/13	0/8	0/0	0/2	0/
HDMIController		34/71	64/146	119/215	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
interface		4/37	3/82	1/96	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
gen_dvi_if.iic_init		33/33	79/79	95/95	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
ac97_codec		10/54	25/82	21/91	0/1	0/0	0/0	0/0	0/0	0/0	0/0	0/
ROM		6/6	8/8	5/5	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
ac97_core_I		31/38	38/49	52/65	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/
clk25_gen		0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0	1/
clock_gen		0/0	0/0	0/0	0/0	0/0	0/0	3/3	0/0	0/0	1/1	0/
connect_button		3/3	3/3	2/2	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/
cross_domain_connector		16/16	36/36	4/4	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
data_tx		46/46	53/53	105/105	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
dev_reset		3/3	3/3	2/2	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/
devices		0/119	0/208	0/202	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
keyboard		0/53	0/92	0/89	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
configure		13/13	17/17	26/26	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
edger		4/4	10/10	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
ps2_interface		1/36	0/65	0/62	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
mouse		0/66	0/116	0/113	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
configure		28/28	41/41	51/51	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
edger		4/4	10/10	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
ps2_interface		1/34	0/65	0/61	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
frame_buffer		0/4764	0/12116	0/4007	0/22	0/3	0/0	0/4	0/8	0/0	0/1	0/
MIG		0/1284	0/2417	0/1512	0/22	0/3	0/0	0/0	0/8	0/0	0/0	0/
clk200gen		0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	0/0	1/
ethernet_to_ram		278/278	810/810	117/117	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
ram_clk_gen		0/0	0/0	0/0	0/0	0/0	0/0	3/3	0/0	0/0	1/1	0/
ram_mux		23/3202	0/8889	34/2378	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
arbitrator		653/653	1659/1659	842/842	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
frame_reader		1555/1555	6181/6181	55/55	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
ram_initiator		355/355	62/62	923/923	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
read_burst		350/350	815/815	70/70	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
write_burst		266/266	172/172	454/454	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
peripheral_monitor		32/32	74/74	48/48	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
ram_dvi_sync		4943/4943	12328/12328	5994/5994	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
rx_data_mac		198/198	247/247	239/239	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
sound_buffering		53/161	109/115	118/296	0/0	0/43	0/0	0/0	0/0	0/0	0/0	0/
buffer		0/108	0/6	0/178	0/0	0/43	0/0	0/0	0/0	0/0	0/0	0/
tx_data_mac		169/191	138/173	316/398	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
ip_header_checksum		22/22	35/35	82/82	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/
v5_emac_ll		8/175	18/405	0/304	0/11	0/2	0/0	0/0	0/0	0/0	0/0	0/

Device Utilization Summary:	
Number of BUFGs	13 out of 32 **40%**
Number of BUFIOs	8 out of 80 **10%**
Number of DCM_ADVs	2 out of 12 **16%**
Number of FIFO36_72_EXPs	2 out of 148 **1%**
Number of FIFO36_EXPs	1 out of 148 **1%**
Number of IDELAYCTRLs	5 out of 22 **22%**
Number of ILOGICs	82 out of 800 **10%**
Number of LOCed ILOGICs	8 out of 82 **9%**
Number of External IOBs	220 out of 640 **34%**
Number of LOCed IOBs	181 out of 220 **82%**
Number of IODELAYs	91 out of 800 **11%**
Number of LOCed IODELAYs	8 out of 91 **8%**
Number of OLOGICs	132 out of 800 **16%**
Number of PLL_ADVs	2 out of 6 **33%**
Number of RAMB18X2s	4 out of 148 **2%**
Number of RAMB36_EXPs	41 out of 148 **27%**
Number of TEMACs	1 out of 2 **50%**
Number of Slices	9438 out of 17280 **54%**
Number of Slice Registers	26016 out of 69120 **37%**
Number used as Flip Flops	26015
Number used as Latches	0
Number used as LatchThrus	1
Number of Slice LUTS	11923 out of 69120 **17%**
Number of Slice LUT-Flip Flop pairs	30267 out of 69120 **43%**
Overall effort level (-ol): High	
Router effort level (-rl): High	
Router Info	
Phase 1 : 106772 unrouted; REAL time: 45 secs	
INFO: Route: 538 - One or more MIG cores have been detected in your design and have been successfully placed and routed. These MIG core(s) have critical skew and delay requirements that are independent of the user (UCF) timing constraints. These MIG-related timing constraints have been successfully met in this design. However, the user must verify separately that all timing constraints specified in the UCF file have been met.	
Phase 2 : 95314 unrouted; REAL time: 51 secs	
Phase 3 : 39787 unrouted; REAL time: 1 mins 39 secs	
Phase 4 : 39806 unrouted; (Setup:0, Hold:872, Component Switching Limit:0) REAL time: 1 mins 49 secs	
Updating file: example_design.ncd with current fully routed design	
Phase 5 : 0 unrouted; (Setup:0, Hold:865, Component Switching Limit:0) REAL time: 2 mins 25 secs	
Phase 6 : 0 unrouted; (Setup:0, Hold:865, Component Switching Limit:0) REAL time: 2 mins 25 secs	
Phase 7 : 0 unrouted; (Setup:0, Hold:865, Component Switching Limit:0) REAL time: 2 mins 25 secs	
Phase 8 : 0 unrouted; (Setup:0, Hold:865, Component Switching Limit:0) REAL time: 2 mins 25 secs	
Phase 9 : 0 unrouted; (Setup:0, Hold:0, Component Switching Limit:0) REAL time: 2 mins 27 secs	
Phase 10 : 0 unrouted; (Setup:0, Hold:0, Component Switching Limit:0) REAL time: 2 mins 31 secs	
Total REAL time to Router completion: 2 mins 31 secs	
Total CPU time to Router completion: 2 mins 31 secs	

Clock Report

Clock Net	Resource	Locked	Fanout	Net Skew(ns)	Max Delay(ns)
clk75	BUFGCTRL_X0Y25	No	3123	0.588	2.120
frame_buffer/clk0	BUFGCTRL_X0Y26	No	3127	0.579	2.108
tx_clk_0	BUFGCTRL_X0Y12	No	879	0.544	2.126
frame_buffer/clkdiv0	BUFGCTRL_X0Y28	No	476	0.542	2.075
AC97_clk_BUFGP	BUFGCTRL_X0Y4	No	31	0.189	1.783
CLK_100_IBUFG	BUFGCTRL_X0Y0	No	123	0.348	2.055
frame_buffer/clk90	BUFGCTRL_X0Y27	No	121	0.327	2.111
rx_clk_0_i	BUFGCTRL_X0Y29	No	50	0.427	2.090
refclk_bufg_i	BUFGCTRL_X0Y10	No	5	0.155	1.723
frame_buffer/clk200	BUFGCTRL_X0Y5	No	10	0.278	1.896
o/delayed_dqs<0>	IO Clk	No	17	0.095	0.419
o/delayed_dqs<1>	IO Clk	No	17	0.083	0.380
o/delayed_dqs<2>	IO Clk	No	17	0.101	0.425
o/delayed_dqs<3>	IO Clk	No	17	0.107	0.404
o/delayed_dqs<4>	IO Clk	No	17	0.101	0.425
o/delayed_dqs<5>	IO Clk	No	17	0.101	0.425
o/delayed_dqs<6>	IO Clk	No	17	0.096	0.393
o/delayed_dqs<7>	IO Clk	No	17	0.101	0.425

* Net Skew is the difference between the minimum and maximum routing only delays for the net. Note this is different from Clock Skew which is reported in TRCE timing report. Clock Skew is the difference between the minimum and maximum path delays which includes logic delays.

* The fanout is the number of component pins not the individual BEL loads, for example SLICE loads not FF loads.

Timing Score	0 (Setup: 0, Hold: 0, Component Switching Limit: 0)
Peak Memory Usage	939 MB
Placer	Placement generated during map
Routing	Completed - No errors found
Timing	Completed - No errors found
Number of error messages	0
Number of warning messages	0
Number of info messages	2

5.3 PERFORMANCE

Before we begin commenting and eventually concluding on the hardware performance we ought to clear out a few things first. The system we developed in our FPGA board operates exclusively the way we have designed it to. There are no side processes to be taken care of or other entities consuming the board's computing resources. So, a critical point on the hardware performance is whether it is able to handle and process any incoming events.

The "data consuming" peripherals such as the monitor or the speakers, are connected to interfaces that feed them with a standard amount of information no matter the input on our system might be. So, as long as our system is working, the performance of these devices is guaranteed to be stable and satisfactory, leaving these interfaces out of the performance question. Specifically, monitor operates at 70Hz with 1024 x 768 resolution or ~157MB/s bandwidth. Speakers operate at 48 KHz with 4 bytes sample resolution or ~187.5KB/s bandwidth.

Moving on, we have to conclude on the other interfaces keeping in mind that they are handling spontaneous events, such as the creation of mouse events or a change on the audio stream which we are receiving from the Cloud. Every test so far gave us good results, which means that the Ethernet, RAM and PS/2 interfaces are able to handle the amount of incoming information and transform them into a stable stream of information to the "data consuming" peripherals or through the internet to the Cloud.

Judging by the incoming stream of data in our system, we can assume that our Thin Client is able to handle the tasks we have assigned to it, whilst occupying a small amount space on the FPGA board, which is whole point of our Thesis.

The main goal of the software application we designed was to carry out the task of managing the incoming stream of events from the Client and produce and deliver the "feedback" stream. As we can see on the measurements Chapter 5.1, our application is consuming a fair amount of computing resources while operating under the most difficult conditions. Even though, our multi-thread approach gave us ability to handle perfectly aspects such as the incoming peripheral simulation events. The thing that our application struggled the most to handle, was clearly the FPS.

6. LESSONS LEARNT

6.1 CRITICAL PROBLEMS AND ISSUES

6.1.1 CHIP INTERFACING

In this Chapter we will be making a sort list containing all kind of difficulties and obstacles we have faced in our effort to interface our proposed hardware system with the board's physical ports. Some problems occurred because of our unfamiliarity with the proper way that some devices operate. That was the case with the AC97 interface and the Ethernet's Marvell interface, and that is the reason we will not be mentioning any issues concerning these two interfaces.

- DVI

Let's begin with the DVI interface. Digital Visual Interface is not a popular interface in the FPGA world since most boards come with a VGA port. The driver support on the DVI port is almost nonexistent and we had to use the interface from another Verilog project we found on the Internet. Still, changes needed to be made in order to come up with a working monitor interface.

Our board is equipped with a Chrontel 7301C chip which is responsible to feed the physical DVI port with the pixel values needed. Our first task was to turn on this chip and instruct it to initialize the communication between our system and the DVI port. The Chrontel chip accepts commands through an I^2C bus which operates in each own protocol. Learning this protocol and sending the right initialization commands was the very first challenge we faced.

According to the XPS Thin Film Transistor (TFT) Controller, we had to send a series of commands to the chip's registers concerning the Power Management, the Digital to Analog (DAC) register and the PLL adjustment. We chose the settings that best fit our design's prerequisites but came up to "dead end". We end up having two different set of initialization commands that both appear to have normal results. However, after a certain amount of operation time, glitches begin to appear on our monitor that eventually mess up with the screen drawing timings. The problem persists even when we reset the device. But, if we synthesize the design from scratch using the other set of initialization commands, the problem disappears. This weird behavior leads us to the conclusion that either the Chrontel chip suffers from a malfunction, either we are unable to initialize the chip correctly.

The second problem we faced with this interface had to do with the color resolution. This driver was designed to draw pixels using a 6-bit color model, which is not acceptable in our case. Our Cloud server sends frames with 24-bit pixel resolution, so we had to find a way to feed the chip with 8-bit color. The challenge was, that even though the Chrontel Manual explicitly mentioned the right timings to send each color's bits, our driver's manual was not very clear on that issue. We were forced to take an educated guess with the color's bits order, but finally we found the proper way to feed the chip with data.

- PS/2 - USB

Starting the design process, our intention was to use the board's USB port to interface with the peripheral devices, mouse and keyboard. Luckily enough, our research prevented us from actually trying to implement the USB interface, and here is why.

We had to choose an FPGA board to design our system in, but every available option was equipped with only one USB port. It was only logical then, to seek a combo device that could handle both peripherals, or a wireless USB receiver. The problem was that both these type of devices used a USB Hub model to exchange data with the system they're connected to. We were left two options: either design a USB Hub interface from scratch, or choose another port/protocol to interface with our peripherals.

Keeping in mind that in a future expansion of our system, we may need a USB port to connect portable flash drives or other types of peripherals, we chose to use the board's PS/2 ports. This protocol may be outdated but in our effort to get the best out of our board, it was great choice.

- RAM

The biggest challenge we faced was to find adequate memory inside our board to store one frame from the Cloud Server. Since the image resolution is 1024x768, with 24-bit color resolution, we needed ~2.25 MB of memory. Our miscalculation of the FPGA's max memory capacity led us to believe that we will not need an external source of memory. To correct this mistake we began searching for an alternative way to store and access data, fast enough.

Random Access Memory was the device of our choice. Xilinx has developed the Memory Interface Generator (MIG) in order to implement the correct memory interface for our board. The first major problem was the luck of support concerning the UCF files. The example designs were assuming a specific design pattern, which in our case did not match our own. With the help of the Xilinx Community we managed to find the correct setting and finally enable the RAM in our system.

The final problem we solved had to do with a slight in MIG manual. While embedding the MIG interface in our design, we chose to not use Data Masking for reasons of simplicity. The auto generated Verilog file does not contain a connection port for the Data Mask bits of the device, and nowhere in the manual is mentioned in what logic level we have to tie the Data Mask pins in our board, high or low. Again, the Xilinx Community solved this issue for us and we were able to test our design without risking the integrity of our device.

6.1.2 PROGRAMMING IN WINDOWS 8.1

Developing and designing a system on Windows 8.1 OS was a challenge on its own. The problem does not lie only on the OS, but also on the support that companies provide on their designing tools.

First of all, the tool we are using to develop the hardware part of system is Xilinx ISE 14.7. This platform is not yet supported in the new Windows OS so while working on it, several bugs appeared. The most important ones, were the fact that we were unable to open a new window dialog in case it was needed (e.g. Open New File), and we were also unable to synthesize our design. Eventually we were forced to replace some software libraries with older version of the platform.

Our choice for the software designing tool was Microsoft's Visual Studio, since we are programming in the C# language. Our need for access to the rendering process of the computer screen and our need to apply graphical changes to the OS, is something that Windows 8.1 do not easily cover. In order to develop an application with that kind of authority over the OS, we had to authenticate the project before actually running it.

Specifically, when simulating keyboard strokes we noticed that the Alt + Tab command was not executed, whilst the Alt + Esc command was working fine. After several hours of research, we found out that Windows 8 do not permit a user developed program to apply graphical changes to the OS. In our case, a little panel is drawn on top of the screen every time the Alt + Tab combination is pressed, that contains all the active windows of the OS. And that was the problem. The simulated command was issuing a draw command to the OS which our program was not permitted to do so. Eventually, we compromised with the use of the Alt + Esc combination only.

The same denial of access to the OS drawing process, combined with the fact that our laptop recognized the embedded Intel Graphics Card as the primary one, was the problem with our effort to directly access the Direct X API. However, we bypassed this problem too, by importing the Slim-DX library. Not only it is a light-weight library but it saved us the trouble from authenticating our application.

6.2 OTHER CHALLENGES

Here we will be discussing some random challenges we faced throughout the design process, that did not really belong to the previous chapters, but might be useful to anyone that might want to migrate our design to different board or run our software on different PC or OS.

- Gigabit Ethernet seems to be a common communication port nowadays, but that is a misconception. Neither our laptop, nor our router were equipped with a Gigabit Ethernet port, so we had to improvise. The only port with an equal or bigger bandwidth than the Gigabit Ethernet on our laptop, was a USB 3.0 port. A brief internet search led us to an Ethernet to USB adapter that simply transformed our USB port to an Ethernet one!

- We started designing our application with the false notion that running a software was a deterministic process. That is not always the case though. To send UDP packets through the Ethernet port, using the Ethernet protocol, we instantiated software sockets. We falsely assumed that if the socket accepts the few thousand packets we are sending to it, they will eventually be delivered to the network. Whether they will reach their destination is a completely different story.

  The fact is, that if an error occurs during the socket's try to send these packets, our application will never have any indication. So, we were forced to assume an upper limit to the socket's incoming data bandwidth in order to manually prevent the data overflow that may occur. And we were right, since the Ethernet protocol does not have the same throughput as a software application. It was hard to readjust the data flow in our application and even harder to redesign the whole hardware system in order to increase its error tolerance.

- Another "network" issue was the UDP's minimum length. When we first captured the signals from the peripheral devices, we attempted to transfer these information through Ethernet frames. These frame's minimum length is 64 bytes (14 bytes header, 4 checksum, 46 data section). These 46 bytes will hold the IP header which is 20 bytes long, plus the UDP header which is 8 bytes long. That leaves us with a 46 – 28 = 18 bytes long data packet at minimum. These data bytes have to be filled, otherwise they will be padded with zeroes. Going back to our captured signals, the information we attempted to transfer was less than 18 bytes, and that was quite confusing while crosschecking the results via Wireshark. Eventually, we added a few extra bytes as a header of our own, in order to fulfill the minimum length requirement. We also took care not to surpass the MTU (Maximum Transmission Unit) which is 1500 in the Ethernet Frame Protocol.

- As explained in section Mouse of the Chapter 4.2.2, the process of resetting and initializing the mouse device is bit more complex than just issuing a single reset command. We were able to enable the data transmission from the mouse to the FPGA board through a series of commands, but we also experienced an inconsistency. Specifically, there are times where the mouse initialization commands have no effect in the mouse's state and we need to resend them by resetting the whole system. It is possible though, that the problem lies on the physical device we are using as a mouse, so there's not much we can do in this case.

- The keyboard device also offered some challenges. Every keyboard device is equipped with three indication lights concerning the state of the Caps Lock, Scroll Lock and Num Lock buttons. We assumed that pressing on of these buttons would trigger an inner mechanism of the keyboard that would light up the corresponding LED light. But that is not the case, since we have to monitor the entire data flow and locate the events that suggest the pressing of one these three keys. Then, it is our responsibility to send the right commands in order to instruct the keyboard to light up the right LED light.

  Another keyboard issue was the possibility of simulating numerous keyboard events in a tiny time gap. The internet network does not guarantee the on schedule delivery of the packets, so it is possible that some packets may be delayed. Our application stores all incoming packets in a Queue structure before handling them. In some scenarios, the polling thread may receive two consecutive packets with keyboard events and place them on the Queue. When the serving thread begins its execution in its own quantum of time, two events will be popped from the Queue and will be served with a tiny time difference. That may cause trouble since all keyboards have a standard character repeat rate that we're exceeding by sending keystrokes too fast, leading to possible loss of information.

- The AC97 interface we are using in our proposed system accepts 16-bit resolution samples for each channel. The library we are using in our software application records audio at 32-bit float resolution, so the mismatch is obvious. We had to transform the captured format to a new waveform with 16-bit resolution samples, but the compatibility was questionable. Finally, through the NAudio community we found a way to apply the transformation without losing too much information.

- Finally, we had to choose between two different screen capturing methods, the GDI method and the Slim-DX method. The main advantage of the GDI method was the ability to draw the mouse pointer on top of the image, something that Slim-DX is not able to do. We tried to find a workaround but the resources needed and the delay that was introduced in the streaming process, led to pick the Slim-DX even if our system does not support mouse pointer for the time being.

7. CONCLUSION AND FUTURE WORK

7.1 CONCLUSION

The goal of the thesis was pretty simple. Even though devices with the same theoretical concept already exist (terminals) we wanted to create a Thin Client that is not limited to a local network, and also supports the new features of I/O interaction between user and machine.

And we achieved that. Our design is fully functional and ready to remotely access every PC with a static IP address in its possession. Peripheral simulation runs smoothly and flawlessly as does the sound and video stream. We encountered some FPS issues but that is an obstacle that can be surpassed.

Specifically, every mouse event, clicks and movement, is successfully transmitted to the Cloud Server in 24 byte long packets. Cloud Server decodes them and simulates the peripheral commands it receives. The recorded audio that is streamed has a quality of 48 KHz, better than studio CDs. The video stream reaches 19-20 FPS in mostly inactive states, but drops to 2-3 FPS at the weakest point of our encoding algorithm.

Our Thin Client operates as expected, but that does not mean there is no room for improvement. As explained in the following chapters, some concepts of the theoretical background need to be reevaluated, such as the way our current networks are constructed, and some practical issues can be addressed and rise the user experience to a new level.

Many challenges were faced and many problems were solved, especially regarding the reuse of many hardware interfaces. Modules and concepts that were being used so far to interface devices with the FPGA, turns out that they were being designed mainly for integrating them to systems equipped with a processor but that was not our case. Simple modifications were made in order for them to operate the way we wanted. This process was extremely educating for us because it brought up many issues and limitations that our initial design had. Overcoming these, actually shed more light on our solution and helped us readjust critical parts of our design which finally led to a fully functional Thin Client implemented in the FPGA.

All in all, the most important thing to learn from this thesis, is to think outside the box. The whole concept of this thesis has no point considering how most of our computing devices work nowadays. But it brought up some unused space, a gap ready to be utilized and be exploited in order to handle computing and energy issues from a different perspective.

7.2 FUTURE WORK

In this chapter we are trying to list any possible future work that may be related to our design. As it was mentioned before, we are dealing with a physical design and not a theoretical concept and that leaves a lot of room for research. Now that we established that is actually possible to introduce a new level of abstraction in the user - machine interaction scheme, a lot of new topics are open for debate.

First of all, it is the new global network topology. The Internet is a global system of interconnected computer networks that use the standard TCP/IP protocol to link several billion devices worldwide. It is a network of networks that consists of millions of private, public, academic, business, and government networks of local to global scope, linked by a broad array of electronic, wireless, and optical networking technologies.

The first major change concerns the sum of active Internet connections. By transforming all clients to simple data representation devices, the only ones in need of accessing the Internet data are the Servers, members of the Cloud. The data path of every webpage starts at the Web Server and reaches only the Cloud Servers, which are now responsible for delivering these data to the real clients. A strongly established connection between Web and Cloud Servers leads to simpler, faster and more stable Internet connection for the Clients.

Now, let's dig into some traits of the Cloud-Clients connection. Since most of the data Clients are sending through the Internet is produced from the peripheral devices, for example mouse actions, chat text, video calls, nothing is expected to change from this point of view. But we will have to exclude file sharing from this model. Files will be physically stored to the Cloud Servers so the Clients will only be sending commands for the file transfer which will take place inside the Cloud! Regarding the incoming data rate, as discussed before, the Clients are merely data representation devices. With the implementation of new frame capturing algorithms, the required data rate can be pretty much standard since the stream is active all the time. Burst data transfers will not be that common in our model, so the connections that are established between the routers in the network should be reconsidered.

Moving on, since our main form of communication is a media stream, the encoding should be taken into account. Our application takes for granted that the network bandwidth will always be enough, but that is not the case most of the times. New encoding / decoding algorithm may be needed to be developed to tackle this issue, while keeping in mind that the workload should not be devastating for the Cloud Servers, and that our design does not have the required computing power to decode complex video or sound streams.

That brings up another question, whether it will be viable to incorporate a graphics card in our design and instead of streaming the whole video data, rendering commands will be sent and the frame data will be drawn locally. And since the OS of the Cloud Servers will not have to draw the pixel screen, it makes screen support an unnecessary feature for them. That along with some safety issues, regarding the remote connection, are some of the topics to be examined in the Cloud OS functionality.

7.3 SUGGESTIONS AND IMPROVEMENT

In the process of implementing our design, we noticed a few things that could be improved and touch up the experience of remotely connecting to our PC. Every concept is examined separately in an attempt to add more depth in our design.

- Features like the mouse and keyboard simulation are working flawlessly on the FPGA side. In our app, we faced some difficulties programming for the Windows 8 OS. Windows 8 prevent user applications from making graphical changes by issuing keyboard commands. That means that sending the Alt + Tab command, which draws a little window to screen with all running applications to choose one and bring it to the foreground, is not allowed. Authenticating this application in order to be considered a trusted one, may solve this problem.

- While the AC 97 chip plays the audio stream perfectly on the FPGA and the NAudio library records all audio events in the sound card flawlessly, it would be nice to include some features in our design that alter the recording and playback experience. Right now, the sound stream can be turned on/off from the application but it would be convenient to have this option to the FPGA too. Finally, users may desire to choose a different recording encoding, a new samples per second rate or the number of channels recorded (Stereo - Mono).

- Finally, there is a lot of room for improvement regarding the frame stream. Although our encoding algorithm is a simplified version of MPEG, the use of a new one could increase the FPS which is struggling a lot when data need to be delivered to the FPGA. On top of that, the mouse pointer is a must-have feature in modern PC's and a new way of drawing it on the screenshots should be implemented. This task that can be assigned to the FPGA and not our C# application.

Continuing, a list of suggestions that can expand the functionality and usability of our design is proposed. These suggestions aim towards concentrating enough workload to complete another thesis, and hopefully address or examine some of the issues mentioned before.

- First on the list, is the improvement of the theoretical background. Encoding algorithms need to be reexamined along with an overall optimization of the system. Especially in the software, bottlenecks should be located and resolved while in the hardware a better clocking distribution should be considered. Reducing the slices occupied in the FPGA will be a big deal, alongside with a lightweight software application that allows user to fully take advantage of the computing power of the Cloud Server he is connected to.

- Last but not least, Ethernet packet distribution in the software side should be monitored and adjusted all the time. Using Gigabit Ethernet, every packet or frame has a maximum physical size of 1538 Octets. That leads to a Max Frame Rate of 81274 Ethernet FPS, since we are not using Jumbo Frames. It is possible that our application may produce data in a faster rate, which leads to packet loss inside the network since sockets cannot handle the amount of incoming data. More on that issue on the Chapter 6. Displaying more statistics about the this issue and the data flow inside the application in general will allow the user to make the required tweaks in order to ensure the steady operation of our design.

- Moving down the list, we encounter some more practical suggestions. USB was designed to standardize the connection of computer peripherals and it has become a commonplace on almost every device. It is only logical that USB support should be included in our design. First, USB Hub should be an option since most of the mouse and keyboard devices come in combo (a single USB receiver is used to communicate with both peripherals). Second, USB peripherals like web cameras, printers and external hard disk drives are used quite frequently in our everyday life and it is essential for the user to be able to use them in this new scheme we are introducing. The basic idea will be the same and all communication data will be streamed from/to the Cloud according to the USB protocol.

- At the bottom of the list are sitting the screen upgrades/suggestions but their importance is fundamental in our effort to improve the user experience. HDMI is the new video/audio interface for monitors and screens, so it should be included in our design. It also has the benefit that it can transfer sound as well alongside with the video data. More screen resolutions should also be available to the user to choose from. Finally, a very simple starting menu (no reason to increase the complexity of our Thin Client) should be included in our design with a few specific options for the user, such as: Insert target IP address, target port number, username and password. In case the FPGA has no LED indicators, some statistics about the driver state (Ethernet, RAM, DVI, AC97) would be a useful feature.

8. USER GUIDE

8.1 SOFTWARE INSTALLATION

Currently there is no installer for the Windows 8 App that's used in this thesis. There are some critical DLL files that need to be in the same location as the executable file. These libraries are:

Slim-DX DLL's	Used from Slim-DX in order to build the video stream.
NAudio DLL's	Used in order to record sound & build the sound stream.
MetroFramework DLL's	Used in the application's GUI.

The peripheral actions simulations require a Windows Library, User32.dll. In most systems, linking to this library won't be an issue.

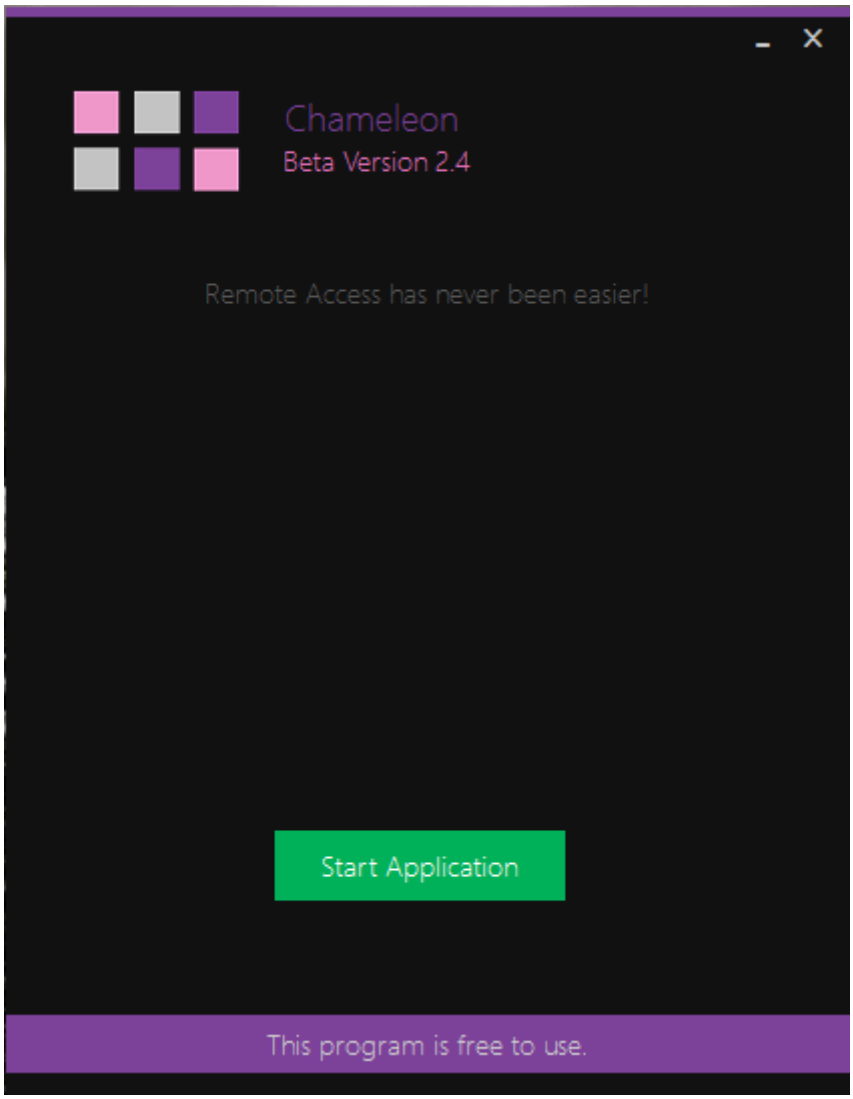
Here is a complete list of the libraries needed:

Assimp64.dll	NAudio.dll
AssimpNet.dll	NAudio.WindowsMediaFormat.dll
MetroFramework.dll	SharpDX.dll
MetroFramework.Fonts.dll	SlimDX.Dircet3D9.dll
MetroFramework.Design.dll	User32.dll

The application was designed, debugged and tested using Visual Studio 2012 in the Windows 8 environment. Its behavior may be unpredicted if run in other operating systems. Other than that, the DLL libraries and the .exe file are the only prerequisites needed to launch the application.

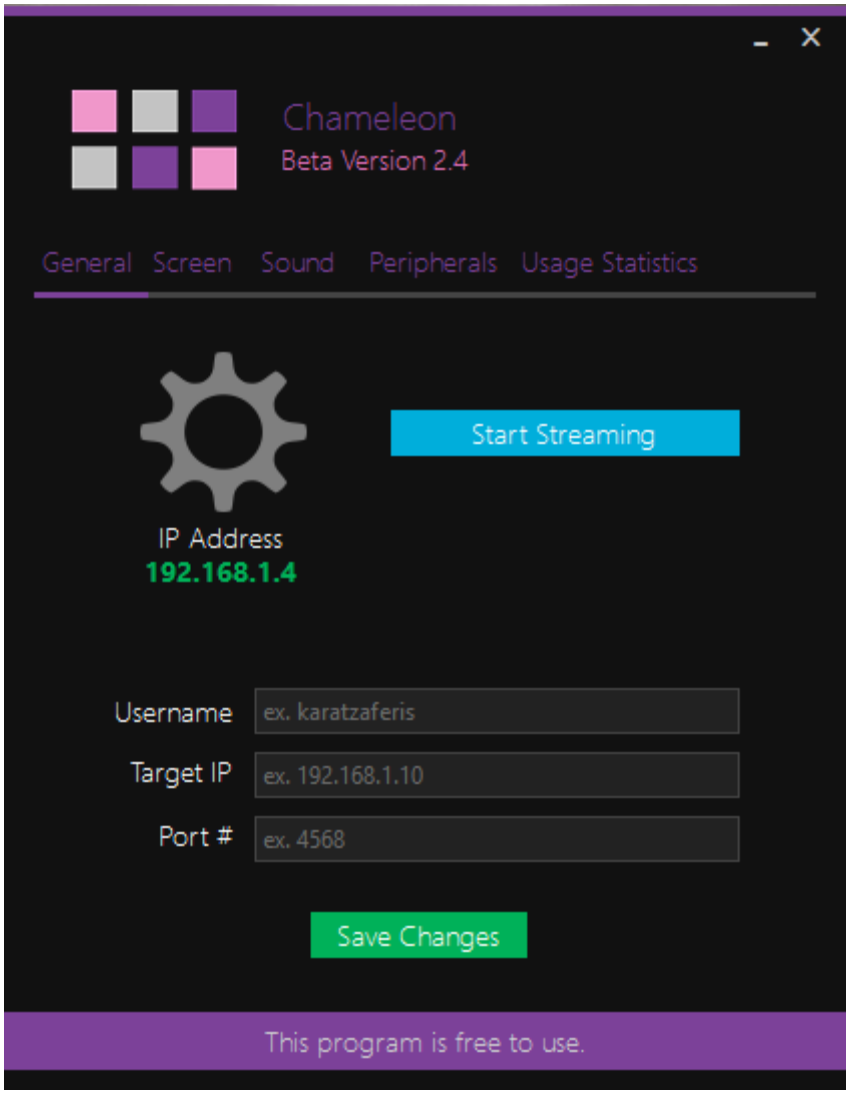
8.2 SOFTWARE MANUAL

Upon launch, user is greeted in the starting screen where the current version of the application is found, along with a "Start Application" button. After the Start command is issued, the application is loading some assets and the GUI is initialized. The functionality of the app is categorized in five tabs. Let's examine every possible action the user can take:



8.2.1 GENERAL SETTINGS

It is advised to check on your system's current IP address by issuing the ipconfig command in the Windows CMD, and crosscheck with the IP that is displayed on this tab. Remember that this is the IP that the FPGA needs in order to remotely connect to this PC.

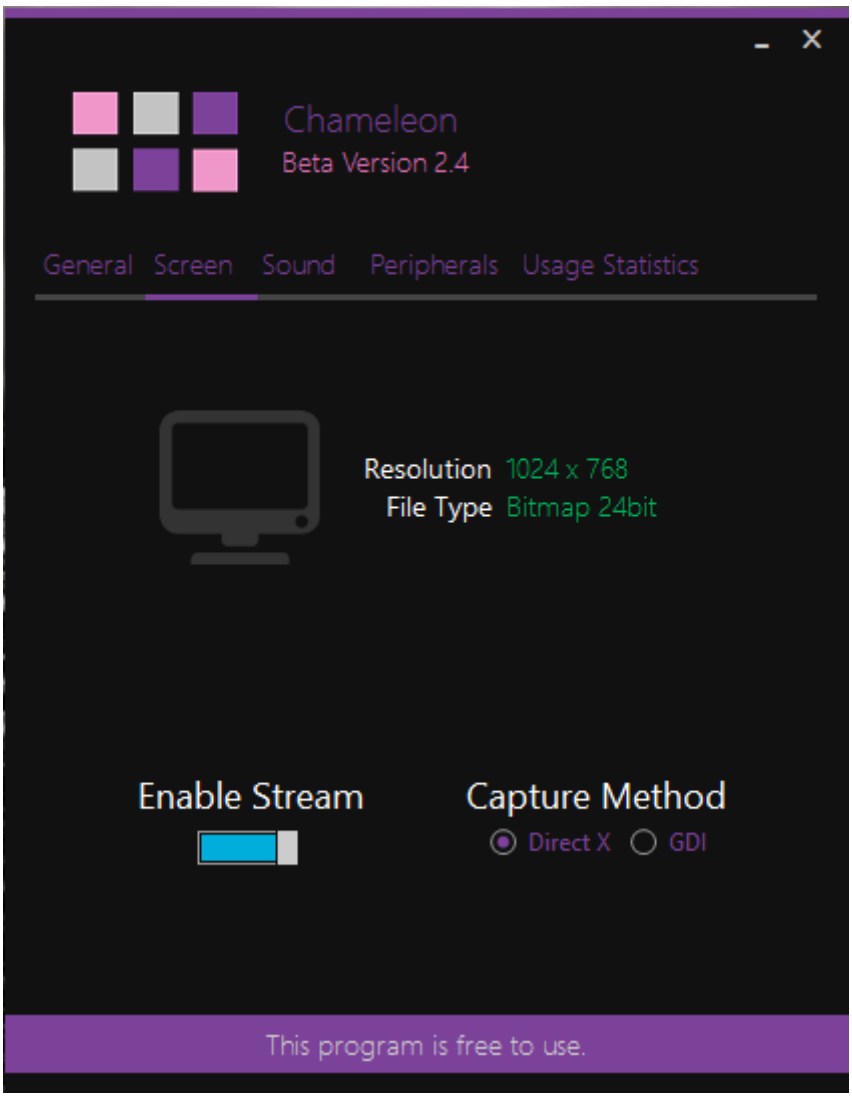


The user is given the option to complete the following forms, and update some connection data:

- **Username**: The FPGA device sends some authorization data in order to connect with the app, and one of these is a username. Changing the default value will restrain the FPGA from interacting with this app.
- **Target IP**: This is the IP address that the app's sockets send the data to.
- **Port #**: This is the port number that the app's sockets send the data to.

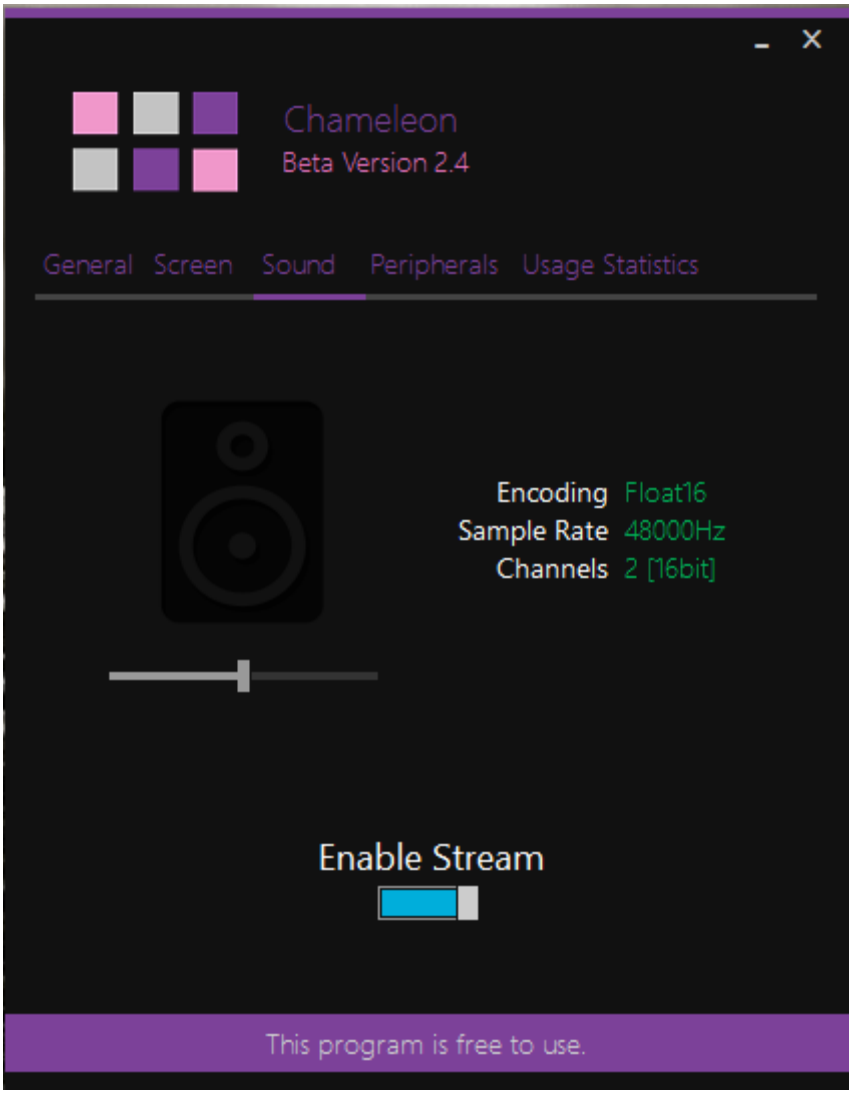
- **Save Changes** button will make any alterations permanent. The button's color will change to notify the user that the action was completed.
- **Start Streaming** enables the server behavior of the app where the connection between PC and FPGA is established. Upon receiving the right authorization, the app starts streaming data through the Internet.
- **Stop Streaming** is visible once the Start button has been pressed. User is given the option to terminate the stream.

8.2.2 SCREEN



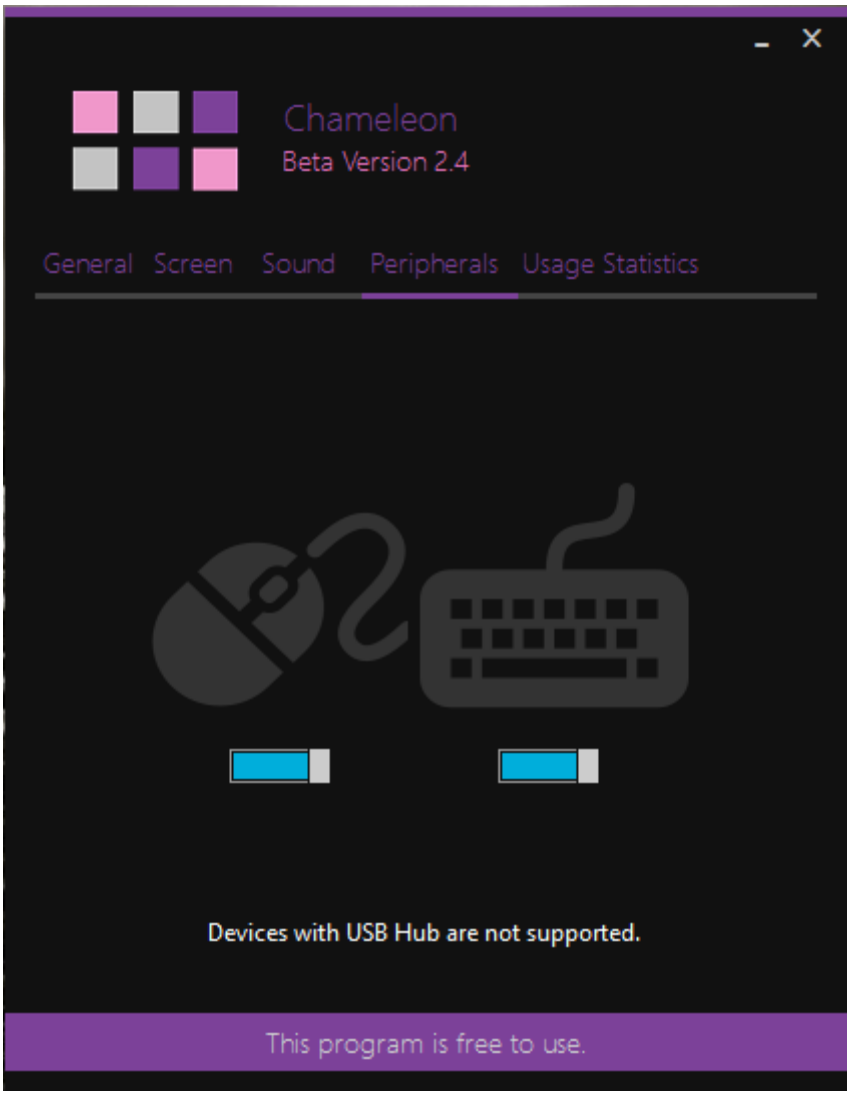
- **Screen Resolution** The only resolution supported is 1024x768. Any other value indication may lead to malfunctions.
- **File Type** and **Bits per Color.**
- User is given the option to turn on/off the video stream.
- User can choose between two frame capturing methods. DirectX uses the Slim-DX Library methods, while GDI uses Windows Graphic Device Interface. GDI encapsulates the mouse drawing on the screenshots. Warning: This option reduces performance drastically.

8.2.3 SOUND



- **Sound Recording Encoding** The NAudio Library records sound in 16 bits long float samples.
- **Sample Rate** The default value is 48KHz which is greater than CD's.
- **Channels** Both channels (Right,Left) are recorded [Stereo].
- User is given the option to turn on/off the sound stream.
- User is given the option to adjust the recording volume [0-1 range in float format]. This option should not be confused with the speaker volume as it only affect the volume of the sound stream and not the PC's native sound.

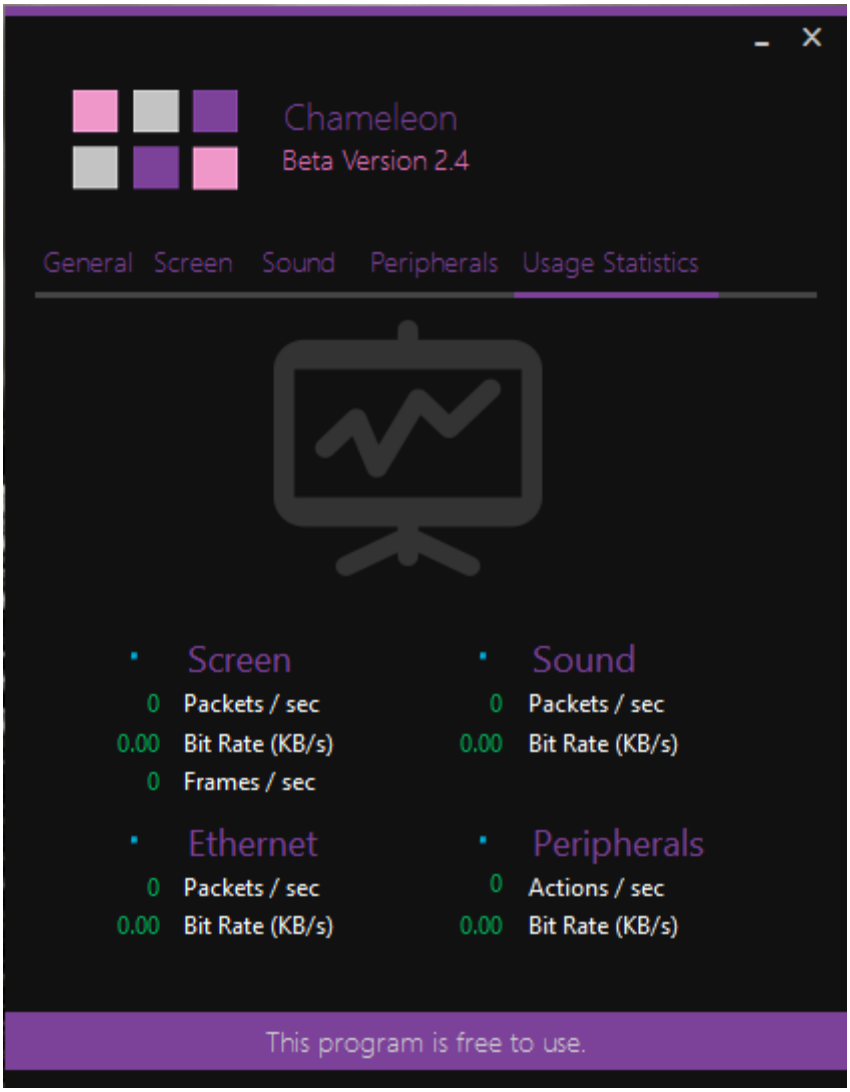
8.2.4 PERIPHERALS



- User is given the option to turn on/off the peripheral (mouse, keyboard) data reception.

Warning: Devices with USB Hub are not supported. Although, a USB to PS/2 switch can be used for USB devices without Hub.

8.2.5 STATISTICS



All streams are cut into smaller chunks in order to be sent from the sockets. These chunks are called packets. Knowing the packet size allows us to measure the bit rate of every stream.

Packets / sec	How many packets of data are sent through the sockets.
Actions / sec	How many peripheral actions are received.
Frames / sec	How many frames are examined.

Bit Rate (KB/s) Calculated using this formula:

(# packets * packet size) / 1024. Packet size is given in Bytes.

Video packet size	100
Audio packet size	960
Peripheral event packet size	24

8.3 FPGA-PC CONNECTION

For the FPGA programming check Chapter 8.4

Our devices are connected through the Internet, so theoretically there is no need for physical connection between them. Both of them are connected with a router to access the network. Warning: The router must support 1 Gigabit Ethernet connections.

If the Cloud Server is accessing the Internet with a Static IP address provided by the ISP, then the FPGA can identify its target and exchange packets with it. Nowadays most ISP's provide Dynamic IP addresses to their subscriptions so it is impossible for the devices to identify each other and remotely connect.

Another more viable option to test this design is to connect both the devices with an Ethernet cat6 cable. Both of them are assigned with Local IP address and is easier to operate. Although, the FPGA needs to use the ARP Protocol to establish itself on the network and the routing tables.

8.4 PREPARING FPGA

First of all, physical connections need to be established.

- Connect the DVI cable to the DVI port of the FPGA.
- Connect the jack speaker cable to the Line Out port. Headphones port can be used too.
- Connect the PS/2 peripherals, mouse and keyboard.
- Connect the JTAG to USB peripheral which is used to program the FPGA.
- Connect the power plug.

The target's (Cloud Server) IP address and port number have to be known beforehand. Locate these parameters in the TX Data and RX Data Verilog files and change them accordingly. Then regenerate the design. Next step is to turn on the device and download the .bit to the FPGA. Finally, iMPACT is used to program the FPGA with our design.

8.5 FPGA UI

After completing the steps described in Chapter 8.4 the FPGA is ready for the remote connection. The User Interface is pretty straightforward:

- Top push-button sends the reset signal to device.
- Middle push-button sends the authorization packet in order to remotely access the Server.

Here's a table with LED debugging information:

	On	Off
LED 0	Keyboard Working	Reset the device
LED 1	Mouse Working	Reset the device
LED 2	Ethernet Working	Check cable connection
LED 3	AC 97 Working	Reset the device
LED 4	RAM Working	Check RAM placement
LED 5	Screen Working	Reset the device
LED 6	Connection Successful	Check LED 2 & Chapter 8.3
LED 7	Streaming Online	Check LED 2

Ethernet Chip (Marvell 88E1111) has its own LED interface on the FPGA.

BIBLIOGRAPHY

[1] An Analysis of Power Consumption in a Smartphone, Aaron Carroll - Gernot Heiser.

[2] Implementation Scenario for Teaching Partial Reconfiguration of FPGA. Pierre Leray, Amor Nafkha,Christophe Moy

[3] Implementing a Real Time Music Pedal in FPGA, Thanos Sariggelos, Efstathios Alexandros Karatzaferis

[4] Streaming Media – Wikipedia

[5] Remote Desktop Software – Wikipedia

[6] Ethernet Protocol – Wikipedia

[7] UDP Protocol – Wikipedia

[8] ARP Protocol – Wikipedia

[9] Digital Video Interface - Wikipedia

[10] XPS Thin Film Transistor (TFT) Controller, (v2.01a), December 02, 2009 - Xilinx

[11] Xilinx High-Performance DDR2 SDRAM Interface in Virtex-5 Devices - XAPP858 (v2.2), September 14, 2010

[12] ML505/ML506/ML507 Evaluation Platform User Guide Xilinx [UG347] (v3.1.2) May 16, 2011

[13] Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC User Guide UG194 (v1.10) February 14, 2011 Xilinx

[14] Memory Interface Solutions User Guide UG086 (v3.6) September 21, 2010 Xilinx

[15] CH7301C DVI Transmitter Device Datasheet - 2010

[16] LM4550 AC '97 Rev 2.1 Datasheet May 2004

[17] PS/2 Core in Verilog, Piotr Foltyn, OpenCores.org

[18] AC 97 Core + Interface, Javier Valcarce.

[19] FPGA NES System, by Brian Bennett 2012

[20] TX Data Verilog File + IP Header Checksum, Joel Williams.

[21] NAudio Library, Mark Heath

[22] MetroFramework Library, Sven Walter, Dennis Magno

[23] Slim-DX Library, 2009-2011, Slim-DX Group.