ARIZONA STATE UNIVERSITY

HONORS THESIS

Building a Mobile Device that Leverages the Power of a Desktop Computer

---

Author:
Dylan LATHRUM

Director:
Dr. Robert HEINRICHS

Second Reader:
Ruben Acuña

Third Reader:
Dr. Shawn Jordan

A thesis submitted in fulfillment of the requirements for the degree of Software Engineering

for

Barrett, The Honors College

April 24, 2022

Declaration of Authorship

I, Dylan LATHRUM, declare that this thesis titled, "Building a Mobile Device that Leverages the Power of a Desktop Computer" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Dylan Lathrum

Date: 4/15/2022

"Premature optimization is the root of all evil."

Sir Tony Hoare

ARIZONA STATE UNIVERSITY

Abstract

Barrett, The Honors College

Software Engineering

Building a Mobile Device that Leverages the Power of a Desktop Computer

by Dylan LATHRUM

With the recent focus of attention towards remote work and mobile computing, the possibility of taking a powerful workstation wherever needed is enticing. However, even emerging laptops today struggle to compete with desktops in terms of cost, maintenance, and future upgrades. The price point of a powerful laptop is considerably higher compared to an equally powerful desktop computer, and most laptops are manufactured in a way that makes upgrading parts of the machine difficult or impossible, forcing a complete purchase in the event of failure or a component needing an upgrade.

In the case where someone already owns a desktop computer and must be mobile, instead of needing to purchase a second device at full price, it may be possible to develop a low-cost computer that has just enough power to connect to the existing desktop and run all processing there, using the mobile device only as a user interface.

This thesis will explore the development of a custom PCB that utilizes a Raspberry Pi Computer Module 4, as well as the development of a fork of the Open Source project Moonlight to stream a host machine's screen to a remote client. This implementation will be compared against other existing remote desktop solutions to analyze it's performance and quality.

Acknowledgements

I would like to express my graditude to my thesis director, Dr. Robert Heinrichs, and to each of my committee members, Ruben Acuña and Dr. Shawn Jordan, for their help in the development of this thesis. Without their guidance, I would not have been able to step so far outside of my comfort zone and learn about so many new topics to make this thesis happen. I would also like to thank my friends and family who were always there to bounce ideas off of and inevitably tell me when I was trying to fit too many words into a single section.

Contents

Declaration of Authorship iii

Abstract vii

Acknowledgements ix

List of Abbreviations xvii

1 Introduction 1
- 1.1 Power and Portability . . . 1
- 1.2 Purpose of this Thesis . . . 1
- 1.3 Thesis Overview . . . 1

2 Background 3
- 2.1 Specialization of Computers . . . 3
  - 2.1.1 Power of the Desktop . . . 4
  - 2.1.2 Convenience of the Laptop . . . 4
  - 2.1.3 Rise of the Gaming Laptop . . . 5
- 2.2 Thin Clients . . . 5
- 2.3 Application to Modern Day . . . 5

3 State of the Art 7
- 3.1 Software Solutions . . . 7
  - 3.1.1 Remote Desktop Protocol . . . 7
  - 3.1.2 Virtual Network Computing . . . 8
  - 3.1.3 Chrome Remote Desktop . . . 8
  - 3.1.4 Secure Shell Protocol . . . 8
- 3.2 Hardware Solutions . . . 9
  - 3.2.1 Thin Clients . . . 9
  - 3.2.2 Nvidia Shield and GameStream . . . 9
- 3.3 Research Questions . . . 10

4 Developing the Hardware 11
- 4.1 Requirements . . . 11
  - 4.1.1 Performance . . . 11
  - 4.1.2 Cost . . . 12
- 4.2 Choosing Parts . . . 12
- 4.3 Prototyping . . . 16
- 4.4 Designing the Printed Circuit Board . . . 16
  - 4.4.1 Power . . . 17
  - 4.4.2 USB . . . 18
  - 4.4.3 HDMI . . . 19
  - 4.4.4 CM4 and Miscellaneous . . . 20
- 4.5 Manufacturing the Circuit Board . . . 21

4.5.1 First Attempt . . . 21
4.5.2 Second Attempt . . . 22
4.5.3 Third Attempt . . . 23

5 Developing the Software **25**
5.1 Requirements . . . 25
5.2 Utilizing NVIDIA GameStream . . . 25
5.3 Developing for ARM . . . 27
5.4 Developing Testing and Measurement Tools . . . 29

6 Evaluation **31**
6.1 Testing Methodology . . . 31
6.2 Responsivity and Latency . . . 33
6.3 Quality . . . 36
6.3.1 Moonlight . . . 36
6.3.2 Chrome Remote Desktop . . . 37
6.3.3 Remote Desktop Protocol . . . 37
6.3.4 Virtual Network Computing . . . 37
6.4 Real World Testing . . . 38
6.5 Summary . . . 42

7 Conclusion **45**
7.1 Summary . . . 45
7.2 Limitations . . . 46
7.3 Future Research . . . 46

A PCB Specifications **47**
A.1 PCB Schematic . . . 47
A.2 PCB Layout . . . 51

B Bill of Materials **53**

C Open Source Repository **55**

Bibliography **57**

List of Figures

2.1 Thin Client . . . 5

3.1 Nvidia Shield . . . 9

4.1 Raspberry Pi Compute Module 4 . . . 13
4.2 Raspberry Pi Compute Module 4 Pin out . . . 13
4.3 USB Controller . . . 16
4.4 SMD Breakout Board . . . 16
4.5 PCB Power Circuit . . . 17
4.6 Power Schematic . . . 17
4.7 PCB USB Circuit . . . 18
4.8 USB Schematic . . . 18
4.9 PCB HDMI Circuit . . . 19
4.10 PCB CM4 Circuit . . . 19
4.11 CM4 Schematic . . . 20
4.12 PCB Stencil . . . 21
4.13 Hand-soldered mezzanine connector . . . 22
4.14 Two versions of the QFN-20-1EP_4x4mm package . . . 23
4.15 Assembled PCB . . . 24

6.1 Input Delay Data . . . 33
6.2 Color Delay Data . . . 34
6.3 Color Delay Loss Data . . . 35
6.4 CRD Data . . . 37
6.5 Streaming Electronics Design software . . . 38
6.6 Streaming the training of an AI . . . 40
6.7 Streaming a 3D modeling, animation, and rendering program . . . 41
6.8 Streaming the development of a video game level . . . 42

A.1 Top Level Schematic . . . 47
A.2 Power Schematic . . . 48
A.3 CM4 Schematic . . . 49
A.4 USB Schematic . . . 50
A.5 PCB Front Layout . . . 51
A.6 PCB Back Layout . . . 52

List of Tables

4.1 Selected PCB Components 15

B.1 Bill of Materials 54

List of Abbreviations

BOM	**B**ill **O**f **M**aterials
CM4	Raspberry Pi **C**ompute **M**odule **4**
CPU	**C**entral **P**rocessing **U**nit
CRD	**C**hrome **R**emote **D**esktop
CSV	**C**omma **S**eparated **V**alues
FFC	**F**lat **F**lexible **C**able
FPS	**F**rames **P**er **S**econd
GPU	**G**raphics **P**rocessing **U**nit
IDE	**I**ntegrated **D**evelopment **E**nvironment
PC	**P**ersonal **C**omputer
PCB	**P**rinted **C**ircuit **B**oard
RDP	**R**emote **D**esktop **P**rotocol
RTC	**R**eal **T**ime **C**lock
SBC	**S**ingle **B**oard **C**omputer
SMD	**S**urface **M**ounted **D**evice
SSH	**S**ecure **Sh**ell (Protocol)
VNC	**V**irtual **N**etwork **C**omputing

For my parents, and their endless support, encouragement, and guidance.

Chapter 1

Introduction

1.1 Power and Portability

As computers continue to grow to become more and more integrated into our daily lives, purchasing a computer has always been a balance. Whether its deciding between something cheap or something long-lasting, a machine that is pre-built or a machine that is custom tailored and custom built, or, a tradeoff that has become much more common in recent years, something powerful or something portable. As with anything, compromises can always be made and a middle ground may be found, but the only way to get the best of both worlds currently is to pay a price premium. If someone were to look for a computer that is powerful, long-lasting, and upgradable, they would be directed towards a desktop computer. However, if they needed something with the same power but portable, their only option would be to purchase a laptop costing much more than its desktop counterpart and often with limitations in upgradeability down the line. Without purchasing both an expensive desktop and an expensive laptop, there currently is no way to get the benefits of both power and portability.

1.2 Purpose of this Thesis

The primary goal of this thesis is to create a portable hardware and software solution that can be used to control a desktop computer remotely while leveraging all of it's power. There are existing software solutions that give a user remote control of their desktop computer from a different location, which will be introduced in Chapter 3, but such solutions may not provide a performant and high-quality experience for the user. These solutions will be tested while running highly demanding applications such as rendering animation, training Artificial Intelligence models, or developing video games to see if they are capable of working in such situations.

In other words, this thesis seeks to provide a user who already owns a powerful desktop computer with a mobile solution that gives them the ability to leverage it's power remotely.

1.3 Thesis Overview

This thesis is split into seven chapters. Chapter 2 introduces background knowledge surrounding existing computers, the shift to a more mobile computing world, and the modern day applications for this technology. Chapter 3 discusses the current the state of the art in terms of existing solutions for remote computer access, and presents a list of questions that this thesis seeks to answer. Chapters 4 and 5 detail the requirements, design, and production of the hardware and software portions of the

project respectively. This design is tested against the requirements and the existing solutions in Chapter 6. Finally, Chapter 7 concludes the thesis with a summary of the project's innovations, limitations, and future research.

Chapter 2

Background

Computers have a long history of being built to serve a particular purpose, often compromising some aspects of the system to bolster others. Chapter 2.1 discusses the history of how computers were built for a particular function, and how the advancement of technology has brought the "Personal Computer" seen know today. Chapter 2.1.1 discusses the benefits of a desktop computer, Chapter 2.1.2 discusses the benefits and drawbacks of a laptop computer, and Chapter 2.1.3 discusses the rise of the gaming laptop and the closing gap between power and portability at the cost of increased price. Chapter 2.2 briefly discusses thin clients, a corporate solution for remote access to a computer. Chapter 2.3 introduces current world wide context that has an impact on the computing world and the need for power and portability.

2.1 Specialization of Computers

In the age of punch cards and monolithic computers the size of entire rooms, it was expected of a computer to only be capable of serving a single function. Before the technology could be miniaturized, every single computer had to be specialized for the job at hand. As time passed and technology progressed, a single computer could be produced that served multiple functions as well as be adaptable programmable to handle new tasks that were not considered when the machine was originally built. This gave rise to what was known as "Autonomic Computing", a system of characteristics that are built into a computer to help it self-manage it's resources and adapt to it's administrator's requirements without the need to be redesigned for it's new purpose [22]. Autonomic Computing was designed to combat the exponential complexity crisis that came from the widespread availability of computers in different disciplines, since now anyone who could afford a computer could program it for any purpose. As new use cases were found for computers in day-to-day life and new technologies were being developed to interface with the world around us, the need for componentization became ever more apparent. Researchers at IBM knew that the best way to address the looming problem of runaway complexity in computers was to develop a way for the computers to automatically interface with any new components that are installed, and to configure itself to only use what is necessary for the job at hand [22]. This allowed hardware and software developers to focus on building their products to work with a common standard rather than having to manually integrate their products with every computer.

Once the idea of modularity began to take hold in the computer industry, attention was turned once again towards specializing individual computers. Now that a computer's physical footprint can be minified, and peripheral components can be added and removed without a complete reengineering of the device, a single computer can be optimized to handle a single task without the overhead cost of

building a monolithic computer [6]. Now a computer can be specialized to serve a particular purpose or set of purposes while keeping development time comparatively short. The world has seen this realized through numerous applications such as cell phones, designed to be user-friendly portable devices, computing clusters, built for high-performance computing, or even internet routers, which are built to be a plug-and-play solution for a problem posed to users of all levels of familiarity with computers. Without this idea of Autonomic Computing, each of these devices would have to be reengineered from the ground up every time a new use case was developed.

This leads into the specialization of home computers in from the perspective of a consumer. It used to be that a "home computer" was a device that was bought off the shelf as-is and served it's purpose. Now a home computer can be a desktop PC that can be upgraded with time and seldom moves from it's place, or it could be a laptop that is portable and easy to cary around. These specializations bring more choices for the consumer to pick a computer that best suits their needs, but they often come with compromises.

2.1.1 Power of the Desktop

The classical manifestation of a personal computer is the desktop computer. Known for it's componentization, user repairability, and direct connections to power and the network, desktop computers are the best option for a user looking for a workhorse system. Pieces of the computer could be bought separately, and single parts could be upgraded or replaced by anyone will a little hardware knowledge. Even though they aren't very portable, desktop computers allow consumers to access greater processor power for a lower cost compared to portable devices [26]. Due to this, a desktop PC is often seen at one's home or place of work hardwired to the wall where it remains unmoving unless it needs upgrading, cleaning, or repairs. Adding on the fact the desktop is modular, it can be easily customized to fit any user's needs, as well as enable particular parts to be replaced or upgraded without needing to rebuild the entire system.

While desktops were the staple of personal computers for decades, advancing technology and the rising need of portability and flexibility has led to the growth of laptops.

2.1.2 Convenience of the Laptop

As the world moved towards a more mobile lifestyle, the laptop shifted from being a luxury to being a necessity. People began to need computing power on the go, whether it be for working remotely, attending classes and following along, or simply needing more power than their phone can offer. Companies began to take note too and started building applications to work fully within a web browser, such as Google's implementation of Google Docs, allowing users to edit documents on the go without having to install software like Microsoft Word directly to their computers. This shift towards mobility provided the general public the ability to be more flexible with how they used computers, no longer requiring them to dedicating a spot in their house to be taken up by a desktop computer.

Reducing the barrier to entry for computers, such as the knowledge to buy the computer the best fits their needs and the restrictions that come with setting up and keeping a desktop, led to the widespread adoption of laptops. In fact, by 2021, laptops accounted for 80% of total computer sales [33]. For most consumers, the

laptop is all they need; it may not have the raw power of a desktop, but it handles everything they need it to do. The biggest limiting factor then becomes technology, with manufacturers constantly trying to minimize existing desktop technology into the footprint of a laptop while keeping energy efficiency high enough to be feasible in a mobile device.

2.1.3 Rise of the Gaming Laptop

All computer products are a balance, it simply isn't possible to have the perfect combination of portable, powerful, serviceable, and cheap. While desktops focus on power and serviceability, laptops often focus portability at the expense of other attributes. As mobility becomes ever more important, gaming laptops have become more popular to bring power to the portable form factor. This understandably comes with a cost, with the literal cost of a gaming laptop being the most obvious compromise. The componentization of such a laptop also often takes a hit with parts being directly soldered to the motherboard, reducing a user's options for repairs or upgrades. It is more expensive, and often less user-serviceable, to buy a gaming laptop that has the same performance of a desktop computer – but given that it is an all-in-one device, it's portability gives it a niche that a desktop cannot hope to rival. The compromise presented by a gaming laptop helps close the gap between power and portability, but it still falls short of being an all around good solution.

2.2 Thin Clients

On the corporate side of computing, the solution for portability has been the thin client. Rather than integrating the power of a desktop to a portable machine, a thin client will connect to a powerful server and provide a thin interface to the user. This allows the user to utilize the power of the host machine from a cheap computer that usually doesn't have enough power to be used as a standalone computer. Because of this, the thin client cannot be used as a standalone computer, but it's low cost and portability make it a great solution for a corporate user. While these clients work well for simple interfacing tasks such as data entry or system management, they are not designed to be used for complex tasks that require quick response times or graphical fidelity.

FIGURE 2.1: A thin client next to a traditional desktop computer [13].

2.3 Application to Modern Day

In 2022, the world is currently going through some chaotic times that demand flexibility and adaptability like it hasn't faced before. From the COVID-19 pandemic to massive chip shortages and supply line disruptions, people are now more than ever looking for a way to be flexible in their work and personal lives now and in the future. The COVID-19 pandemic has forced people to be able to operate remotely, whether that be for work, school, or communicating with friends and family [24]. Many office workers used to working on a desktop computer onsite are now needed

to work from home, or classes usually taught in person are now held remotely [9]. This shift to remote operation has fueled the need for mobile computers, many of which need to be powerful enough to handle the workload of a desktop computer from anywhere in the world. The global chip shortage is also causing issues for the purchase of new devices [20]. Parts which are usually readily available for purchase are out of stock with months of delay on back order. Resellers are taking advantage of the situation by hiking prices to two or three times the original listing price, and it's become harder than ever to build a new PC [34]. All of these factors combined lead to a very difficult situation for anyone looking to purchase a new computer or parts for their existing computer.

It doesn't make sense to build or purchase a powerful PC for use at home and another power laptop to handle working on the go, especially with rising prices, so many people are moving to exclusively powerful laptops or looking for other solutions.

Chapter 3

State of the Art

This chapter seeks to provide a summary of existing solutions for remote computer access. It will not cover specific applications, but rather the protocols that are used to communicate with a remote computer. Generally speaking, there are two types of solutions that have been developed, software solutions, which are covered in Chapter 3.1 and hardware solutions, which are covered in Chapter 3.2. Various protocols and implementations are covered in the subsections of those two categories.

3.1 Software Solutions

While there are many applications in existence that allow a user to remotely control a computer from another location, few of them offer the capabilities required to stream high-performing applications to a remote device. The following subsections will introduce existing solutions to this problem as well as drawbacks that come with each implementation.

3.1.1 Remote Desktop Protocol

Microsoft's Remote Desktop Protocol (RDP), is a protocol that defines communication between a terminal server and terminal server client for multimedia purposes [37]. Coming pre-installed on every Windows machine since Windows XP and with clients available for Windows, Mac, and Linux, RDP is an easy to use solution for remotely accessing Windows machines graphically.

Although it is the built-in solution for Windows machines, it isn't without it's drawbacks. Firstly, only one graphical session is active at one time while using windows. This means if a user is logged into the host computer and someone initiates a connection with RDP, the user using the host computer will be logged out. While this isn't always an issue, sensitive programs that don't take well to being logged out may run into issues. Secondly, RDP does not support relative mouse movement [7]. This means every mouse input is sent to the host computer as an absolute position on the screen, rather than as a relative distance from it's previous location. This means any program that moves the user's mouse for them, such as 3D applications with a virtual camera, will be sent the absolute position of the client's mouse. This will cause the application to detect a large movement of the mouse instantly, which can result in erratic camera movement as the program interprets the mouse moving a large distance every time the cursor is reset to the center of the screen. Again, while this isn't always an issue, tasks such as 3D modeling and animation or game development cannot be controlled properly.

3.1.2 Virtual Network Computing

Virtual Network Computing (VNC), is a platform-independent system originally developed by Olivetti & Oracle Research Labs and later bought and shelved by AT&T [35]. While the original implementation is no longer used in a wide capacity, the protocol it was developed on has been expanded and improved to become one of the most flexible yet simple ways to control a computer remotely. VNC can run on any modern operating system, and even a web browser can serve as a VNC client. It was originally built as the simplest form of remote computer control over LAN, allowing system administrators to control almost any kind of computer with a simple, robust, and extremely compatible system. However, since its use has gradually outgrown it's original scope, VNC isn't usually the right tool for any job greater than remote system management. Because of it's simplicity and focus on being as straightforward as possible, VNC is often found to be lacking in terms of security and speed [35]. It is not recommended to use VNC over the internet without some secondary form of security, such as SSL or a secure VPN.

3.1.3 Chrome Remote Desktop

Google Chromoting, implemented publicly as Chrome Remote Desktop, is an open source protocol that enables remote desktop control through any web browser running on the Chromium engine [8]. It prioritizes low bandwidth usage and ease of access, allowing easy server install on every major desktop operating system, and enables any web browser or mobile device to act as a client. By employing the VP8 compression format (Most commonly seen in internet video streaming), Chromoting is able to keep network traffic low by only sending a few full images of the server's screen as keyframes and transmitting motion with compressed partial frames [16]. These partial frames are calculated by the server using the difference between the previous and current frame, allowing the server to only send the difference between the two frames.

This save in bandwidth and processing done by the client makes it the perfect solution for Google's ChromeOS laptops coming out around the same time. These low powered machines were built to be laptops running on the power of a web browser with a link back to more powerful computers when the tasks were too great [38]. While the protocol works well for remote access, it is limited in certain applications by it's conservative approach to bandwidth usage and reliance on the web browser.

3.1.4 Secure Shell Protocol

The Secure Shell Protocol (SSH), is a protocol for securely transmitting data over an insecure network [39]. Usually used for remote server management, SSH has become the de facto standard for connecting to a remote computer when nothing more than terminal access is needed. Paired with the associated SSH file transfer (SFTP) or secure copy (SCP) protocols, SSH provides an incredible amount of flexibility for working with remote computers. It's largest and most obvious drawback is limited graphical support. While not a problem for it's traditional use-case, it does limit what high-intensity applications can be used with it.

One recent innovation in remote computing has been the introduction of Remote Development using SSH through the Visual Studio Code IDE. By opening multiple SSH connections to the host machine at once, Visual Studio Code is able to run the IDE's User interface on the client's machine, and simply send all the written code and commands through to the host machine, utilizing it's power to compile

code, run applications, and debug software [11]. This enables a development experience comparable to working on a local machine by only transmitting the data that is needed over the internet and constructing the visual user interface on the client. Because of this, high-intensity computational programs such as Artificial Intelligence training and simulation can be run on a powerful host machine and only need to transmit the output to the lower-powered client, but graphical applications are still limited.

3.2 Hardware Solutions

While Hardware solutions are not as popular or widespread as software solutions, there has been some innovation in the field that is worth considering. Hardware solutions usually focus on building a device that has just enough power to connect and stream data to a remote computer in order to leverage its power for the user to use.

3.2.1 Thin Clients

As discussed in Section 2.2, Thin Clients are a family of devices commonly seen in the corporate world to allow employees to access data centers and computing clusters from their desks. Due to their low cost and ease of deployment, they are often the device of choice to enable employees to access the full power of the business' computing infrastructure without needed to purchase a powerful device for each employee. However, since a typical thin client is a headless device, meaning it doesn't come with a monitor, keyboard, mouse, or other peripherals, they are often less portable as one might think. A thin client is usually set up once, and then left in that place for as long as a desktop would stay. Especially with laptop sales on the rise, thin clients do not have the benefit of being more portable than the alternatives.

3.2.2 Nvidia Shield and GameStream

The Nvidia Shield game console was Nvidia's first foray into the realm of remote streaming. Though it was focused on gaming and media streaming, it was one of the first attempts at using a portable device to stream intensive applications such as games from a host computer [3]. At the steep price point of $349, it definitely was enthusiast hardware for a device that focused on mobile and desktop gaming over running console games or demanding titles locally. But while the the physical device was praised for performance, battery life, and the experience of streaming games to the console, it didn't perform too well in terms of sales. It was still widely regarded as a technological breakthrough though, and Nvidia continued to develop the technology into a new device called the Nvidia Shield TV [10]. Focused on bringing the power of a gaming desktop to a TV, the Nvidia Shield TV dropped the idea of portability in favor of filling a different market focused on comfort. While this is no longer a valid device for the purposes of

FIGURE 3.1: An open Nvidia Shield displaying the home page [4].

this paper, the developments of the proprietary GameStream technology that powered the Shield and the Shield TV proved to be even more exciting than the physical devices themselves.

Though the GameStream technology that powers the Nvidia Shield devices is closed source and only officially compatible with the products Nvidia releases, its enticing power and potential drew a community to reverse engineer the protocol. In 2013, a group of students at Case Western Reserve University developed Moonlight, an open source implementation of the GameStream protocol [27]. The project is structured as a core library written in *C* with a number of community built clients built on top of it for various platforms. In essence, it turns the closed-source black-box protocol developed by Nvidia into a usable programming interface. By reverse engineering the protocol originally implemented by the Nvidia Shield, Moonlight enables the development of a GameStream client for almost any operating system. The Moonlight community has already developed fantastic clients for many platforms such as Windows, Mac, Linux, Android, iOS, Web, and even other gaming consoles. This stands as a good starting place to develop the solution described in this paper.

3.3 Research Questions

While there are many existing solutions for accessing a computer remotely, many struggle to be performant enough or efficient enough to stream demanding applications to the client without compromising on the user's experience. This thesis will seek to build a solution that enables the use the power of a desktop computer from a mobile device in a way that is performant and responsive in response to the following questions:

1. **Hardware Feasibility:** *What hardware is needed in order to power a mobile device capable of acting as a client?*
2. **Hardware Cost:** *Is it feasible to construct such as device at a lower or comparable cost to existing solutions?*
3. **Communication Protocol:** *Does a protocol exist that is efficient enough to stream demanding applications to a client?*
4. **Software Application:** *Can a software solution be built to utilize this protocol in a manner that is performant enough?*

Chapter 4

Developing the Hardware

There are a lot of factors that go into designing and building an electrical system that fits the needs for this project. This process begins by outlining the requirements in Section 4.1, followed by describing the process and methodology of choosing parts in Section 4.2. Then, the feasibility of prototyping such as system is discussed in Section 4.3, and finally the system is designed and manufactured in Sections 4.4 and 4.5 respectively.

The entirety of the hardware schematics and designs developed for this project are available on GitHub at: `https://github.com/Dylancyclone/thesis/kicad`.

4.1 Requirements

This section will explain the different constraints taken into account while developing the hardware portion of this thesis. First, the hardware must be performant enough to provide a positive user experience (4.1.1), and secondly the product must be cheap enough to justify it over similar alternatives (4.1.2).

Since no specific studies regarding possible users of the system were conducted, the requirements laid out in this section are based on the author's own requirements and preferences. The requirements are constructed with the target user being someone who already has a working desktop computer and wishes to leverage it's power remotely without sacrificing the user experience.

4.1.1 Performance

When considering how much power needs to be incorporated in to the miniature computer, it is important to consider a number of factors, such as: having enough power to stream audio and video over the internet, not consuming too much power so as to require an expensive battery, have a fast enough connection to the internet to deliver data with the least possible latency, and having a reasonable footprint to be used as a mobile device.

Immediately, the device must be capable of decoding and rendering video that is streamed over the internet. Such video processing is a relatively expensive process that cannot easily be done with something such as a micro controller or even a single core CPU seen on Single Board Computers (SBC) such as the Raspberry Pi Zero due to the lack of working memory and processing power [15]. Though at the same time, it is important to not simply build a powerful system akin to a laptop that will consume more power. It makes no sense to built a powerful mini-computer when all it will ever be used to is to connect to another computer, and the added power draw would require a larger battery in order to stay operational for decent periods of time. Instead it should be cut down and distilled into something that can do it's job as efficiently as possible.

The second major performance requirement of the hardware is the ability to stream this data over the internet as quickly as possible. Because all of the data being sent to the device is live information from the host machine, the video and audio streams cannot simply be buffered in advance and played when it's ready like a recorded video. Instead, the data must be displayed as it is received so that the user can have a smooth experience without having to wait for their keystrokes or mouse movements to be registered. While Wi-Fi is growing ever quicker and more reliable, and should absolutely be included as a way to connected to the internet, the best way to ensure a stable and speedy connection is to use an Ethernet Cable. This will help the device work on a consistent connection when used at a desk, while also providing a way to connect while on-the-go.

This leads to the desire for a portable design that can be used both in a conducive environnement such as a desk or while mobile. The hardware should have the ability to be powered by a chord plugging into the device, or by an internal battery. This alone presents a number of difficult concerns which will be addressed in Section 4.3. Such a requirement leads to a clear benefit of integrating crucial computers such as a battery and a screen into the hardware design, while leaving as many ports open such as ethernet and USB for the user to take advantage of.

4.1.2 Cost

Because this project consists of both a Hardware and Software component to make up a complete product, there must be reason for both to be developed. While the software has the benefit of being the foundation for the connection between host and client machines, there are already hardware solutions that would theoretically be able to run such software. Since the goal of this project is to save money by not needing to purchase an expensive mobile device to connect to a powerful desktop computer, there must be benefit in the hardware to warrant it's existence over repurposing something like an old laptop. The most obvious metric of this is the actual cost of the hardware. If the hardware can be produced at a high enough quality at a low enough cost to bring value over just a software solution, then it should be considered moving forward. Otherwise it may prove that the development of the software should focus on compatibility so the hardware can be provided by the end user.

Generally speaking, the goal for the hardware of this project is to cost approximately $100-$200 to produce a single unit. This comes with the understanding that hardware production in bulk often costs significantly less than assembling a single product. The reasoning for this cost comes down to feasibility: the price of the final product should be less than what it would cost to purchase or refurbish an existing computer to do the job of this project. Otherwise, as stated before, the software of this project may prove more valuable than the hardware as long as compatibility is kept in the forefront of development.

4.2 Choosing Parts

When starting to choose parts for the hardware, there are many difficult questions to face. First and foremost, the hardware had to be built with the time and resources available for this project. Specifically, building a Single Board Computer from scratch would be completely infeasible, but the goal was also not to simply

purchase an off the shelf SBC that was not tailored to the task at hand such as a traditional Raspberry Pi. Instead, a better suited solution would be to utilize a *Raspberry Pi Compute Module 4 (CM4)* which has "The power of Raspberry Pi 4 in a compact form factor for deeply embedded applications" [30]. As this perfectly lines up with the requirements for the project, this thesis will be built using the CM4. However, due to it's small size and deeply embedded nature, the only way to communicate with the Compute Module is through two tiny mezzanine connectors as shown in Figure 4.2. With no other pin outs for anything such as power, HDMI, or USB, the rest of the hardware must be built on a custom Printed Circuit Board (PCB) that tailors the CM4 for this thesis.

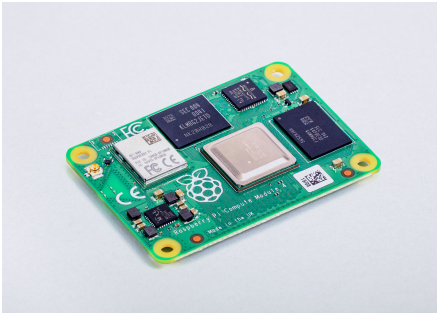


FIGURE 4.1: Raspberry Pi Compute Module 4, sizing 55 x 40mm

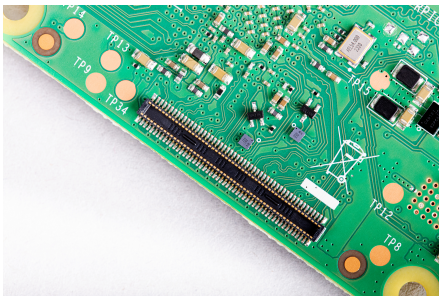


FIGURE 4.2: Close-up of the pin out, with a 0.04mm pitch between pins

This PCB would have to be designed so that the CM4 could be plugged into the board using the mezzanine connectors and rely on it for all inputs and outputs. This, however, is only the beginning of what is required to develop the product. Since the project would not use an off the shelf SBC like a regular Raspberry PI, the project would require only designing the PCB but also picking out all the components that would go onto the PCB, such as the USB ports themselves, the USB driver chip, and related components needed for operation. Because the CM4 is designed to be as low-powered and compact as possible, some features that would usually be expected of such a device, like the USB ports, are disabled by default and must instead be built by the board designer. While this helps greatly by removing extraneous components and features that aren't needed for this project, it also provides a great opportunity to learn exactly what parts are being used in the hardware.

Thought must also be given to the peripherals that will be included in the design. Most importantly, if a screen is going to be included in the design, what kind of screen will be used? For projects of this scale, the usual candidates for a relatively performant screen are HDMI screens or DSI screens. While HDMI screens are much more common and recognizable, DSI screens are a new development that promises 60Hz performance in low cost and low power applications such as embedded systems. However, when looking into existing DSI screens, there are a couple of issues that prevent it from being the best choice for this project. Even though DSI screens are more power efficient and have built in touch support, it is a closed format which greatly restricts manufactures from producing their own screens [18]. Due to this, there is only one screen available for purchase outside of the official implementation, and both have a lower resolution that what is possible with HDMI screens of the same size. So even though HDMI screens are more expensive at this size, they are a more reliable choice compared to the still-developing DSI standard.

Another potential feature to consider is whether to integrate a Real Time Clock (RTC) into the design. An RTC is a component commonly found within laptop and desktop computers that is used to keep track of time even though the computer is powered off. This is done with a small button battery powering an RTC chip integrated into the motherboard to keep the system's clock in sync with the real world. While this is almost never necessary in embedded systems, it is worth considering for this project because of it's use as a mobile computer. However, when considering the primary use case of connecting to another computer over the internet, and the potential of needing a linux kernel patch in order to add support for the RTC [21], it would be much simpler to tell the operating system to simply resync it's clock with an internet clock every time it connects to a network.

An invaluable resource while designing the PCB was the Official I/O board for the CM4 [31]. Though the PCB board would still have to be custom built and components would need to be replaced due to the current ongoing chip shortage, it provides a great foundation instead of starting from scratch. Another massive benefit from working within the Raspberry Pi Ecosystem is their incredible documentation and community surrounding the boards [32]. Around the time this project was being started, the CM4 was just beginning to get into the hands of makers and developers, and only a couple custom boards had actually been made by anyone other than Raspberry Pi themselves. Even though this leaves the community with less of a bulk of knowledge to work with, it has provided a sort of comradery with everyone helping each other learn the new Compute Module together. Members of the CM4 engineering team would even assist those on the forums to help them pick parts and design their own boards [1, 21].

Following the example put forward by the existing boards, PCB parts were chosen to be integrated into the circuit as safely as possible. Thankfully due to most parts following international standards, many parts were chosen simply because they followed the same standard as the CM4 (such as USB 2.0, or *1PORT 1000 BASE-T* Ethernet) and were in stock. What follows is a table of some components that were used in the custom PCB that may differ from those used by the CM4's official board, and the reason for their selection. A full Bill of Materials (BOM) can be found in Appendix B.

Purpose	**Manufacturer's Number**	**Reason**
USB-C Power in	*USB4125-GF-A* by "GCT"	Most projects of this scale are powered by micro usb, but with USB-C gaining popularity, this component was picked for its low price and exclusive ability to transfer power without transfering data.
Ethernet Port	*ARJM11D7-502-AB-EW2* by "Abracon LLC"	One of the few in-stock ethernet ports that supports *1000 BASE-T* usage. The CM4 recommends the *MagJack-A70-112-331N126* produced by "LINK-PP", but that part is out of stock for at least another year as of writing.

USB Port	*SS-52100-001* by "Stewart Connector"	Single flat port that supports USB 2.0.
HDMI Port	*SS-53000-001* by "Stewart Connector"	Single flat port that supports HDMI 1.4.
Flat Flexible Cable Connector	*FFC3B07-20-T* by "GCT"	A generic Flat Flexible Cable port that can be connected to anything. In this case, one is connected to the built-in screen's HDMI port, and another is connected to the screen's touch/power USB port.
2-Pin Through Hole Header	*PREC002SAAN-RC* by "Sullins Connector Solutions"	A generic 2-pin header than can be used to bypass the power circuit as will be discussed in Section 4.3.
HDMI Power Regulator	*RT9742ENGJ5* by "Richtek USA Inc."	The CM4 recommends the *RT9742SNGV* from the same manufacturer, but it is out of stock. This part has slight differences in functionality that require adjustments from the official I/O board design.
SD-Card Power Regulator	*RT9742VGJ5* by "Richtek USA Inc."	The CM4 recommends the *RT9742GGJ5F* from the same manufacturer, but it is out of stock. This part has slight differences in functionality that require adjustments from the official I/O board design.
Battery Voltage Regulator and Boost Converter	*TPS61090RSAR* by "Texas Instruments"	Used to balance power usage from the USB-C port or the attached battery, favoring the external power when available.
Li-Po Battery Charge Management Controller	*MCP73871-2CCI_ML* by "Microchip Technology"	Used to charge the attached battery when attached to external power through the USB-C port.
4 Port USB Hub Controller	*USB2504A-JT* by "Microchip Technology"	Used to control the four USB 2.0 ports and send the data to the CM4. The CM4 recommends the *USB2514B-I/M2* by the same manufacturer, however it and all the parts in it's family are out of stock with at least a 52-week back order. Out of all the components, this has the highest potential to be problematic.

TABLE 4.1: A non-exhaustive list of components that were chosen for this project.

4.3 Prototyping

Whenever working with electrical projects such as this, it is always a good idea to build up prototypes before committing to a final design that will be produced at the highest quality. Usually breadboards and some breakout boards for Surface Mounted Devices (SMD) are a great way to develop a circuit without creating a full PCB; but due to the nature of the complex microchips required for this project as well as the dependence on the CM4's tiny mezzanine connectors, building a prototype using a breadboard is impractical given the time and resources available. For example, the USB controller chip only has 0.5mm of space between each of it's 64 pins (Figure 4.3), which would require a breakout board at least 7 times larger than the chip so that each each pin can be easily accessed by hand. Not to mention the difficulty in wiring up that many pins along with the other related components which would need their own breakout boards either purchased or custom-designed.

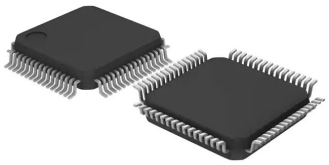


FIGURE 4.3: The USB controller, with 0.5mm between pins

Instead, the first board was built out in a piecewise fashion, allowing each section of the board a way to fail without interrupting the rest of the board. This allows each function of the board to be tested individually and in isolation so that changes could be made incrementally without needing to worry about their integration with the rest of the board. For example, in order to build a system that would allow the board to be powered via USB or an internal battery, as well as charge the battery simultaneously, a complex circuit would have to be built that managed Lithium Polymer (LiPo) batteries which could cause fires if mishandled. Since power is obviously a critical component of the board, A secondary way of feeding power directly the board was built to bypass all that circuitry in case it fails. In doing so, the board can still be powered and it's other features can be tested while a fix for that portion of the circuitry is developed for the next iteration.

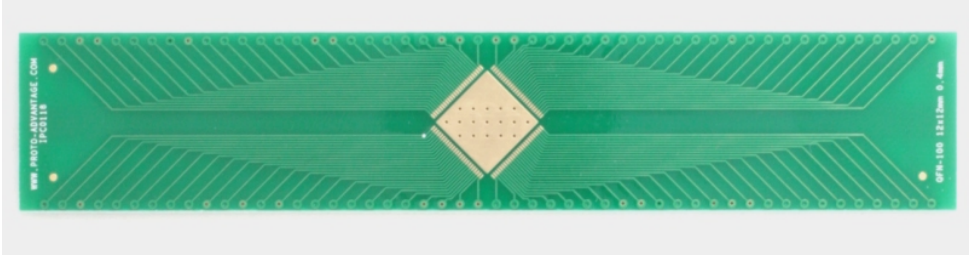


FIGURE 4.4: A breakout board similar to what would be needer for the USB controller used on the custom PCB

4.4 Designing the Printed Circuit Board

Because of the lack of prototyping options discussed in Section 4.3, and ultimate decision to build the board in a way that could safely fail in any way, it became clear that multiple versions of the project were going to need to be designed and produced. Since the time and budget for this project was limited, the amount of

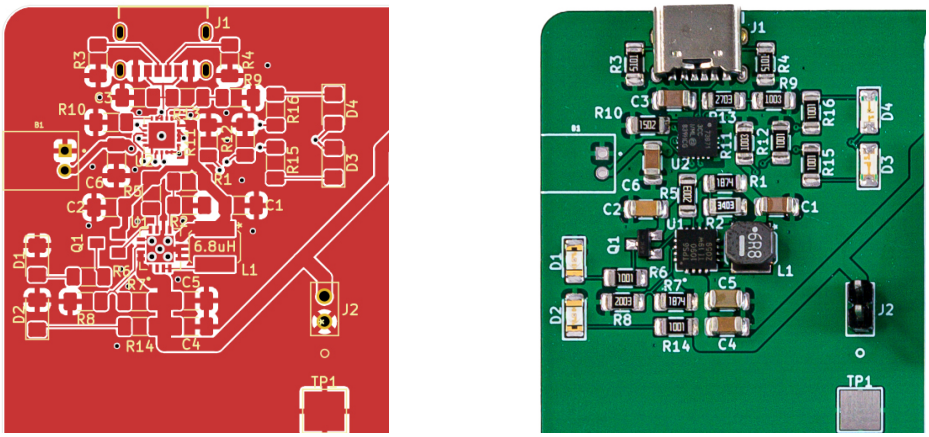


FIGURE 4.5: The front side of the PCB's power circuit

revisioning was minimized by running the circuit by electrical engineering students and professors to catch any potential flaws before the board was manufactured.

4.4.1 Power

Development started with arguably the most important part of the board: the power supply (schematic in Figure 4.6). If the CM4 could not receive power, none of the project would work. A feature of modern devices that has become ubiquitous as much as it has been taken for granted is the ability to charge a battery at the same time it is in use by the device. This enables the device to be used portably until the battery runs out and then charge the device without needing to stop using it before moving again. Though it proves to be slightly more complicated than simply connecting the two power supplies. As seen in Figure 4.5, the source of external power, the USB-C port labeled *J1*, has to travel through two integrated circuits before reaching the CM4 through the trace on the right side. The first chip, *U2*, detailed in Table 4.1 as *MCP73871-2CCI_ML*, is used to charge the internal battery attached to

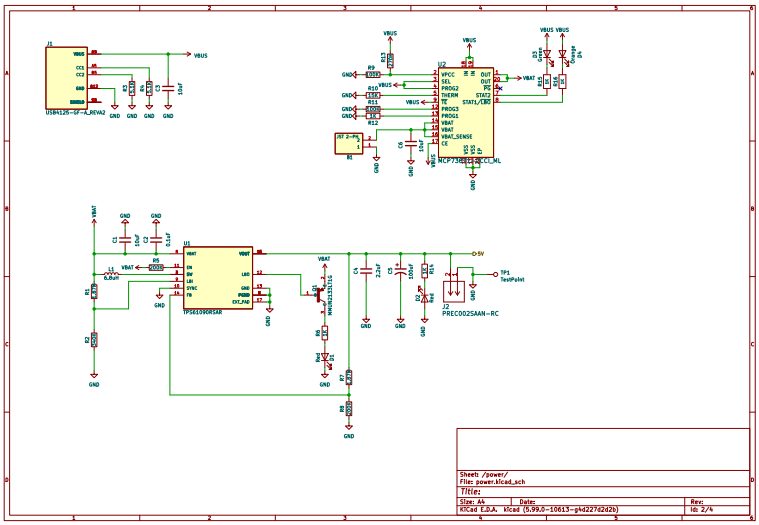


FIGURE 4.6: Schematic layout for the power circuit, including battery management. Expanded in Appendix A.2

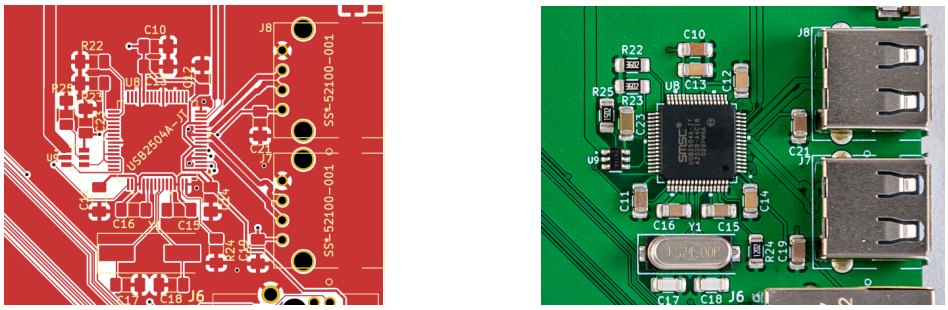


FIGURE 4.7: The front side of the PCB's USB circuit

the left side of the board through the connector labeled *B1*. The second chip, *U1*, also detailed in Table 4.1 as *TPS61090RSAR*, is used to balance the power draw from the USB-C port or the attached battery, favoring the external power when the device is plugged in. Thankfully, due to the sensitivity of the two chips, their provided datasheets offer specific configurations that they should be designed in to ensure proper functionality. Although very dense, these blueprints promised individual functionality for each chip, all that was left was connecting them together. The two of these chips must work in tandem for power to be successfully delivered to the CM4, if either failed, none of the other parts of the board could be tested. In the event this happened, a secondary method of powering the board directly was added in the form of the generic two-pin header *J2*. These two pins are simply 5v input and ground through-hole pins that bypass all the power circuitry just described. Once the CM4 could be powered, the rest of the I/O could be designed.

4.4.2 USB

The next prominent circuit on the board is the section that controls the USB ports. As seen in Figure 4.7 (schematic in Figure 4.8), the dense circuit is mostly comprised of the USB controller chip, *U8*, it's associated components, and the four USB ports, two of which are pictured as *J7* and *J8*. Similarly to the power circuit, the USB controller

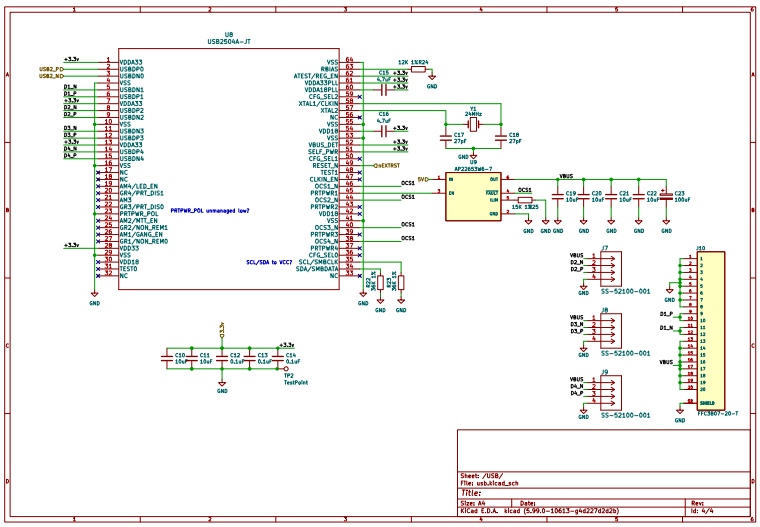
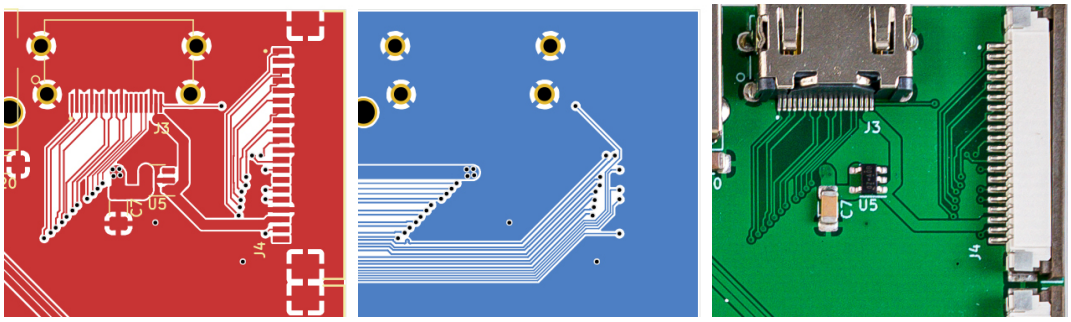


FIGURE 4.8: Schematic layout for the USB hub and ports. Expanded in Appendix A.4



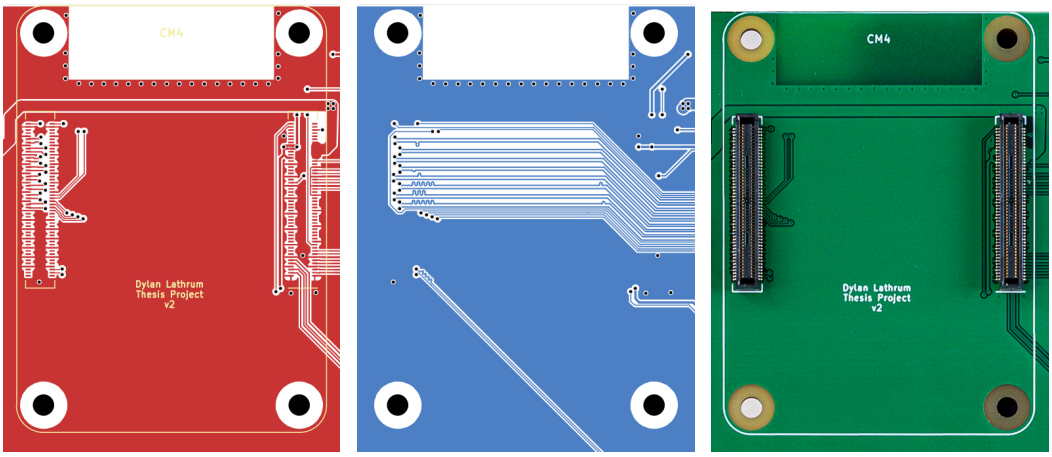
(A) The front side of the PCB (B) The back side of the PCB (C) The physical PCB

FIGURE 4.9: The PCB's HDMI circuit

chip is a very sensitive component that must be arranged in relation to other components in a certain way in order to ensure it will function properly. This includes surrounding the chip with numerous capacitors to normalize any small power fluctuations, using a specific load balancing power distribution switch (*U9*), and an oscillating crystal to provide a stable clock for the chip. The chip connects to each of the four downstream USB ports and handles all of the digital signal processing before sending the data to the upstream CM4.

4.4.3 HDMI

Another vital part of the board is the two HDMI ports provided by the device. Thankfully, unlike the USB ports, HDMI ports send information in only one direction, so an extra chip is not necessary to handle the data as an intermediary. All that is necessary is to connect the two HDMI ports to the CM4 in the right order and the device will be able to output the rendered video to a display. Because this device is intended to be a standalone machine, it makes sense to have essential components like a screen integrated as part of the design. The only issue with simply attaching an HDMI cable inside of the device is that the ports are relatively bulky and inflexible, meaning that another solution must be created to fit in the tight space. This is



(A) The front side of the PCB (B) The back side of the PCB (C) The physical PCB

FIGURE 4.10: The circuitry surrounding the CM4's two mezzanine connectors

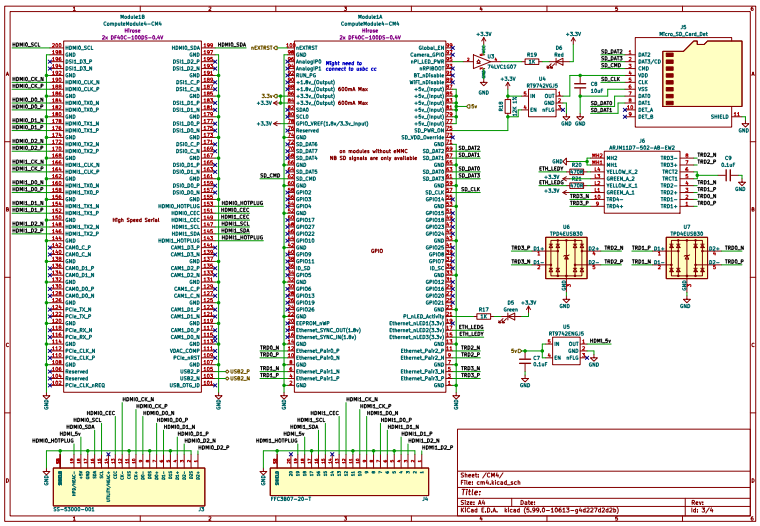


FIGURE 4.11: Schematic layout for interfacing with the Compute Module 4. Expanded in Appendix A.3

accomplished with the Flat Flexible Cable Connector, seen on the right side of Figure 4.9a as *J4*. Flat Flexible Cables (FFC) are generic bendable cables that can be used in tight spaces. Even though their connectors are completely generic, meaning that they have no set pinout, it is possible to use them given that the receiving screen will take in the FFC in the same order as a HDMI cable. Using this knowledge, an internal FFC can be wired for the device's primary screen, while a second traditional HDMI port can be exposed to the user in case they want to use an external monitor. This, along with the FFC used for one of the four USB ports, allows a screen to be hooked up to the CM4 internally without needing to expose anything to the user.

4.4.4 CM4 and Miscellaneous

Finally, everything must connect to the CM4. Figure 4.10 shows the the two mezzanine connectors the CM4 uses to communicate with the rest of the PCB (schematic in Figure 4.11). Remember that there is only 0.04mm of space in between each of the pins, leaving very little space to route the traces where they need to go. The first thing to notice is the large stretch of traces on the back side of the board coming from the HDMI ports. Because most of the traces come in positive and negative pairs, called differential pairs, the two tracks must be roughly the same length as each other. This is because as data is transmitted across the pair of tracks, they need to arrive at their destination at the same time based on the physical distance the signals need to cover. Since the speed of electricity is constant throughout the PCB board, the traces may need to wind back and forth so that the lengths of the positive and negative tracks are equal. Also note the blank area surrounded by vias at the top of Figure 4.10. This is where the WiFi and Bluetooth antennas are located on the CM4 board. By removing the copper pour from this area on the PCB board and stitching the perimeter with vias, it prevents the board from acting like another antenna and interfering with the signals being sent to and from the CM4. Finally, each of the other components are connected to the CM4 board in the same manner as described throughout this chapter, such as the Ethernet adapter, the SD card reader, and remaining ports. These ports are connected by simply connecting the

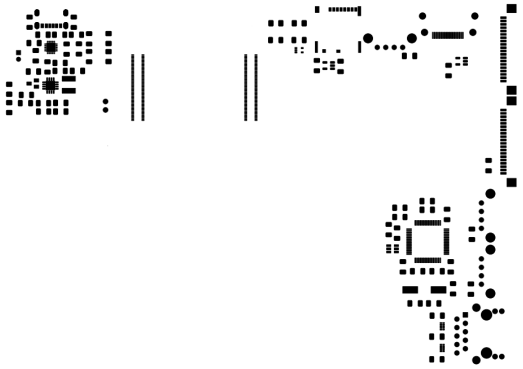


FIGURE 4.12: The PCB's stencil. Areas colored black are where solder paste is applied to solder SMD parts to the board

appropriate pin on the CM4's mezzanine connector to the corresponding pin on the component. Apart from the concerns brought up by keeping the differential pairs in line and the power carefully managed as with the previous components described in detail, there are no special considerations that need to be made while designing the layout for these components. See Appendices A and B for further information.

4.5 Manufacturing the Circuit Board

Once the board's design has been completed and checked against all electrical, functional, and manufacturing constraints, the Printed Circuit Board can be produced. The first step in fabrication is to print the actual PCB. While ASU offers a basic printing service for use in classes and academic projects, the requirements of this board, specifically the tiny traces and pads required for the CM4's mezzanine connectors (pictured in Figure 4.2), are outside of the printer's capabilities. Instead, by recommendation of a professor, the printing was done through an external company that provides quick turnaround and low costs for printing PCBs [29]. A set of five boards was ordered, allowing for multiple boards to be assembled in case of error, as well as a stencil. A PCB stencil is a thin sheet of metal with holes cut in it that line up with the pads on the PCB board. This enables the easy application of solder paste onto all of the pads on the board as once, including the difficulty small pads of the CM4's mezzanine connectors. While the PCB was being printed, all of the parts found in the Bill of Materials (found in Appendix B) were ordered from sites such as Digi-Key and Mouser [12, 28]. In order to account for the production of multiple prototype boards and the possibility of broken parts, enough parts were ordered to produce five complete boards.

4.5.1 First Attempt

Once the PCB and it's component parts arrived, the board could be assembled. The first board was assembled by hand using the PCB stencil and a handheld heat gun. The stencil, shown as an image in Figure 4.12, is used by securing it the PCB board so that the holes line up with the pads, and spreading solder paste across it allowing the paste to cover the exposed pads through the holes. The stencil was then removed, and each of the surface mounted components were then placed onto the board using a set of tweezers. For tricky parts such as the CM4's mezzanine connectors and other small parts with the pads below the component itself, a microscope was used

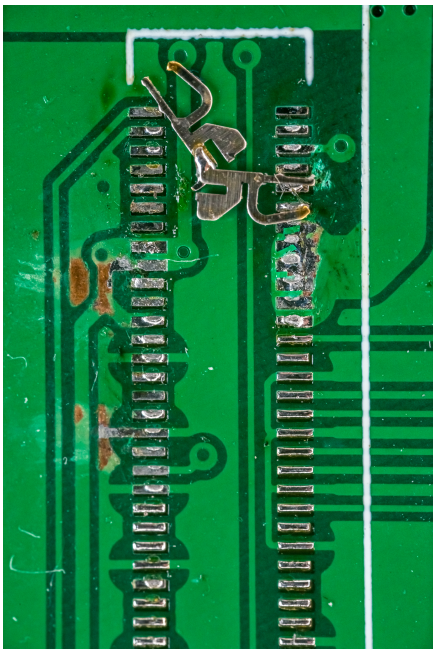


FIGURE 4.13: The leftovers of the broken-off mezzanine connector from the hand-soldered first board. Notice the exposed copper and warped pads.

to assist in lining up the pads with the components. Once the parts were placed, a heat gun was used to melt the solder paste and fuse the components to the board. Following the SMDs, the through-hole components, such as the ports lining the side of the board, were then placed and soldered using a hand-held soldering iron.

The first board ran into two issues that prevented it from working. Firstly, due to improper use of the heat gun, most of the LEDs had popped, resulting in all hardware-level diagnostic indicators being inoperable. Secondly, the CM4 connectors were not properly connected to the board, and in an attempt to repair it, the connector components were irreparably damaged and the board itself was damaged slightly, seen in Figure 4.13. Instead of attempting to assemble another board by hand, a local third-party company was contacted to assemble the board professionally. After delivering the board, stencil, and parts, the second board was assembled.

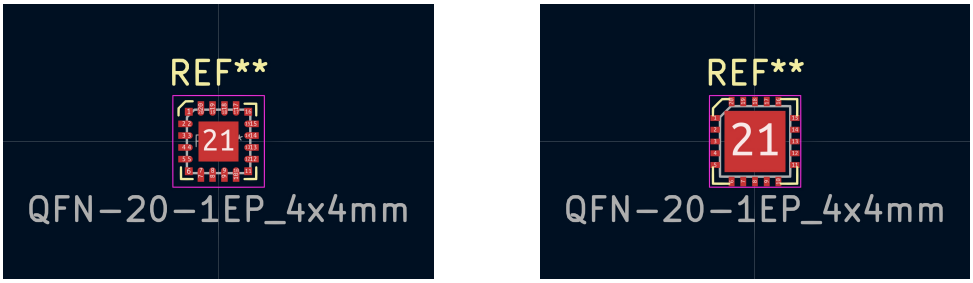
4.5.2 Second Attempt

The second board was assembled mostly successfully, and allowed the first stage of testing to begin. The only issue apart from some ports being too close together with real-world cables was one component having the wrong footprint on the board. Unfortunately, the component *U2 (MCP73871-2CCI_ML)*, is described as using the *QFN-20-1EP_4x4mm* footprint in its datasheet [14], but actually uses a revised version of the footprint, shown in Figure 4.14. This discrepancy meant that that component could not be soldered, and it's associated circuit would not function properly. In this case, the component was a part of the battery charging circuit, causing the entire primary power supply to be unusable. Thankfully, as described in Section 4.3, the secondary method of powering the board could be used to test the remaining features of the board. After making the necessary fixes, a second round of five PCB boards and a stencil was ordered. Because there was still enough components to assemble three more boards, and no components needed to be changed, no more

components were ordered at this stage. Once the boards were delivered, the revised board was again sent to be professionally assembled.

4.5.3 Third Attempt

Once the third board was assembled, the board could be tested in full. While each part of the board was at least partially functional, there were still issues preventing it from working completely. In particular, USB devices plugged into any of the ports on the board were able to receive power from the board but were unable to transfer data in either direction. Thankfully, this did not prevent testing of the board from being completed, but it did stop the board from being considered a final complete product. This board could still be used for testing by connecting peripherals such as the keyboard and mouse through bluetooth and powering the screen through USB. Figure 4.15 shows the final assembled board, with the final PCB layouts in Figures A.5 and A.6.



(A) *QFN-20-1EP_4x4mm* (B) *QFN-20-1EP_4x4mm_RevA*

FIGURE 4.14: The two footprints for components with the *QFN-20-1EP_4x4mm* package. The original footprint is shown in Figure 4.14a, while the revised footprint is shown in Figure 4.14b

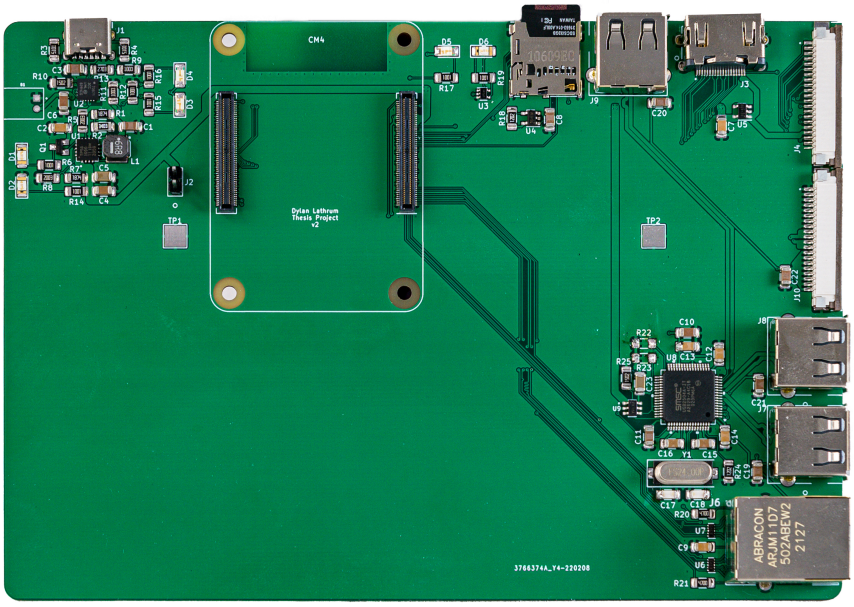


FIGURE 4.15: The assembled PCB

Chapter 5

Developing the Software

The second half of the project is composed of software that is used to connect the client machine to a host computer. Designing the software begins with identifying it's requirements in Section 5.1. Then, the Nvidia GameStream Protocol is analyzed in depth in Section 5.2 so that the software does not need to be built from scratch. Following that, special considerations must be made while developing the software for the CM4 running on an *ARM* processor in Section 5.3, and the development of the suite of tools used to record data is detailed in Section 5.4.

The entirety of the code developed for this project is available on GitHub at: `https://github.com/Dylancyclone/thesis/code`.

5.1 Requirements

Much like the hardware requirements in section 4.1, the primary requirement of the software is to be performant enough to provide the user with a positive experience. This is broken down to include being able to stream and decode video over the internet with as little latency as possible, stream inputs from the user to the remote computer, and retain visual fidelity while the host machine is under heavy load. While much of the latency between the host and client is due to the network both devices are connected to, there is still some optimizations that can be made to reduce the time between data being send and images being displayed on the screen. Similarly, while the rate at which data is transferred between the host and client cannot exceed the speed of the slowest internet connection of the two devices, there can still be a balance between the bandwidth-saving but lower quality streaming seen in Chrome Remote Desktop (3.1.3), and the local network but higher quality Nvidia Shield (3.2.2). To accomplish this, a compression method similar to CRD will be used, but it will not be bound by the traditional web browser which limit CRD due to the software solution being developed as a native application.

As a secondary objective, it would also be beneficial for the software to be able to run on a variety of different hardware platforms in the event the hardware is determined to be infeasible to develop, as discussed in Section 4.1.2.

5.2 Utilizing NVIDIA GameStream

As with any software project, there are many potential ways to solve the problem. As evident by the number of existing solutions that attempt to solve this problem (Chapter 3), taking into special account Nvidia's recent foray into remote gaming with the Shield (3.2.2), it is not reasonable to attempt to create a brand new software solution from scratch with the time and resources available for this thesis. It has taken Nvidia, a company worth hundreds of billions of dollars, years to get to the

point they are at now [3]. Developing a rival protocol or new software platform is simply not possible for this project. Instead, given that Nvidia's GameStream technology is the current leader in the realm of high-performance remote computing, the best way to build a software product that meets the requirements laid out in Section 3.3 is to build upon what already exists.

Unfortunately, the Nvidia GameStream protocol is both proprietary and a closed source project. Very little is officially known about the protocol, other than it uses specific video encoders built into Nvidia Graphics cards from the 600 series from 2012 and newer [19]. Apart from that, everything has been researched and reverse engineered by the community. A GameStream session is started by first connecting to a host computer which has enabled the GameStream service through Nvidia's GeForce Experience application. Once the service is enabled, the computer will listen on port *47989* for the following web requests:

- /serverinfo
- /pair
- /unpair
- /applist
- /launch
- /resume
- /cancel

Once a session has been started with a */launch* request, raw data covering video, audio, and input is passed back and forth over a number of ports until the session is terminated. An end-to-end example of the protocol's usage is as follows:

1. A user knows that a capable host computer is ready to stream at the address *192.168.1.123*.
   (a) This can be confirmed by sending a HTTP GET request */serverinfo* to the host computer on port *47989*. For example:
   http://192.168.1.123:47989/serverinfo?uniqueid=1234
2. The user can make a request to */pair* with a number of parameters to authorize itself with the host.
   (a) Pairing is a two step-process. First, a request to pair is made, at which point the host machine will display a four digit code on it's screen. Then the client must send that code back to the host as a second form of authentication.
   (b) The authentication created by this pairing process lasts until either the client makes an */unpair* request with it's unique ID or a user on the host machine unpairs the client.
3. The user can make a request to */applist* to get a list of applications that the host is able to stream.
4. The user makes a request to */launch* with a number of parameters to instruct the host to create a session with the given settings and launch the specified application.

   (a) The configuration options given include the application to launch, the video resolution to stream, audio setup, input mapping, and multiple forms of authentication.

5. If the user unexpectedly or accidentally leaves a session without closing it, they can make a request to */resume* to instruct the host to reopen and resume the session the previous session.

6. Once the user is finished, they make a request to */cancel* to instruct the host to close the session.

7. If the user is completely finished with the host, they can return it to the state they found it in by making a request to */unpair* with their unique ID, unauthorizing themselves with the host.

The most difficult part of this process is handling the transfer of data after a session has begun. Once a session has launched, data is sent directly between the host and client over a number of ports without going through an easy to understand web protocol. This is where the community project Moonlight steps in.

By taking advantage of the Moonlight library briefly described in Section 3.2.2, the previously described process is translated into a usable programming interface. To accomplish this, a fork of the Moonlight project is created so the GameStream protocol can be leveraged in this thesis. With the fork, much of the code required to create the software solution is already provided, the next step is to build on top of the existing code to ease development for the *ARM* architecture and to write a suite of testing tools that hook into Moonlight to extract statistics.

5.3 Developing for ARM

The CM4 runs on a 64-bit ARM Cortex-A72 processor [30], meaning that applications must be compiled for the *ARM* architecture in order for the CPU to be able to run it. If a developer were to write a program on their *x86* architecture Windows desktop computer, they could compile their program for any other computer running on the same *x86* architecture, but such a program would be incompatible with an *ARM* processor. Similarly, if the program were compiled for *ARM*, that program could be run on an *ARM* computer but not one running *x86*.

In order to run the code for the project, given that the CM4 is the only *ARM* computer available for this project, the code must either be compiled on the CM4 or built in a cross-compilation environment on an *x86* machine. Using a cross-compilation environment would be overall quicker than that developing on the CM4 itself since it can use the full power of a desktop PC. Thankfully, setting up a cross-compilation environment for projects written in *C* is straightforward. *CMake*, a common compilation tool for *C*, has built-in support for compiling for different target architectures with easy to use libraries. What isn't as straightforward however, is building a toolchain to compile the rest of the code for *ARM*.

In order to create an out-of-the-box experience for someone using the hardware for this project, a custom operating system image is created that has all the dependencies, libraries, and applications pre-installed and pre-configured. Manjaro provides good tools for building and managing custom images of the Manjaro operating system for *ARM* architectures [25], with the only drawback being that it must be run on an *ARM* processor, and is much more difficult to run on another architecture. To make development easier, a toolchain was built that could be run on the CM4 to

build the code that is written on other computers. All the developer needs to do is to pull the git repository of this project onto a machine running the *ARM* architecture [23], run a set up script to install dependencies, and then run the build script to compile everything into an immediately deployable OS image. This build script will automatically compile all the necessary code using *cmake*, package all the necessary files using *makepkg*, and build a fresh OS image that is ready to flash. The following excerpt details the three functions used to make this work.

Excerpt from */scripts/build.sh*

```
50 runMake()
51 {
52   mkdir ./moonlight/build # Create build directory
53   cd ./moonlight/build # Go to build directory
54   cmake ../ # Run cmake on the project directory
55   make # Build the project
56   sudo make install # Install the project
57   sudo ldconfig /usr/local/lib # Link libraries
58   cd "$parent_path" # Go back to original execution location
59 }
60
61 runPackage()
62 {
63   cd ./pkgbuild # Go to pkgbuild directory
64   tar -cf moonlight.tar ../moonlight/ # Create tar archive of the
       project
65   makepkg -sLfc # Build the package
66   rm moonlight.tar # Remove the tar archive
67   cd "$parent_path" # Go back to original execution location
68 }
69
70 runBuild()
71 {
72   pkgname=$(sed -n 's/^pkgname=//p' ./pkgbuild/PKGBUILD)
73   pkgver=$(sed -n 's/^pkgver=//p' ./pkgbuild/PKGBUILD)
74   pkgrel=$(sed -n 's/^pkgrel=//p' ./pkgbuild/PKGBUILD)
75   # Copy local package to cache to be installed into new image
76   sudo cp ./pkgbuild/$pkgname-$pkgver-$pkgrel-aarch64.pkg.tar.zst /var/
       cache/pacman/pkg/
77   # Build image
78   sudo buildarmimg -d rpi4 -e thesis -i $pkgname-$pkgver-$pkgrel-
       aarch64.pkg.tar.zst
79   read -s -n 1 -p "Image built, press a key to continue (for sudo)"
80   # Remove local package from cache
81   sudo rm /var/cache/pacman/pkg/$pkgname-$pkgver-$pkgrel-aarch64.pkg.
       tar.zst -f
82   # Move build image to build folder
83   sudo mv /var/cache/manjaro-arm-tools/img/* ./build/
84   cd "$parent_path" # Go back to original execution location
85 }
```

This script starts by building the forked version of moonlight using *cmake* and *make* and links it's libraries using *ldconfig*. Once the application is built, it then builds it into an installable package using *makepkg*, before installing it into the newly created image. Finally, the image is moved to the project's build folder where it is ready to be flashed and installed like any other operating system.

5.4 Developing Testing and Measurement Tools

In order to evaluate the performance of the project, a suite of tools must be developed to test both the newly developed application and the existing solutions. To do this, two sets of data will be collected: First, data is recorded directly from the forked version of moonlight that logs statistics such as the number of frames received, dropped, or rendered, and data that is recorded from a separate application that logs the amount of time it takes for the client machine to receive updates after it sends data to the host machine. The first set of data is important to track the raw performance of the application, with statistics like Frames Per Second (FPS) and decoding time which are difficult to record otherwise. These statistics provide insight into how performant the application is from a technical perspective. The following is a structure that was developed for this thesis to hold the statistics recorded over a session of connecting to a host computer. Corresponding code was written throughout the fork of the Moonlight application to populate this structure while a streaming session is running. The data recorded is evaluated in Chapter 6.

Excerpt from *moonlight/src/video/stats.h*

```
23 typedef struct _VIDEO_STATS {
24   uint32_t receivedFrames;
25   uint32_t decodedFrames;
26   uint32_t renderedFrames;
27   uint32_t totalFrames;
28   uint32_t networkDroppedFrames;
29   uint32_t totalDecodeTime;
30   uint32_t totalRenderTime;
31   uint32_t lastRtt; // Rtt = Round Trip Time
32   uint32_t lastRttVariance;
33   float receivedFps;
34   float decodedFps;
35   float renderedFps;
36   uint32_t measurementStartTimestamp;
37 } VIDEO_STATS;
```

The second set of data is recorded by measuring the total amount of time it takes for the host to receive data from the client, and for the client to receive updates back from the host. This is done using the following steps once the client is connected to the host:

1. When a user hits a key on the client, record which key was pressed and at what timestamp.
2. When the host receives the instruction to press the key from the client, record which key and the timestamp.
3. Change the color of the host's screen to a different color.
4. When the client detects that the color of the screen has changed, record the current timestamp.

This records two points of data, the time it takes for the host to implement the actions done by the user, and the time that the user sees the result of their actions. Because this method of data collection runs as a separate application on both the client and the server, it can be used to measure the performance of any of the remote desktop applications for comparison. An important obstacle to using this data is

the difficulty in synchronizing the host and client machines' clocks. Even if each machine accurately recorded the amount of time between each event and the order in which they occur, the data is useless if the two machines do not agree at which time the test began. This is again discussed in Chapter 6, with the issue of clock synchronization being discussed in Section 6.1.

Both the application running on the client and the server do the same thing, every time a key is pressed, record which key what pressed and at what time, and every time the pixel in the center of the screen changes color, record the new color and at what time it changed. This data is written to a Comma Separated Values (CSV) file for later processing. To change the color of the screen, a simple web page is used that changes color any time a key is pressed. It's source in it's entirety follows:

/datalogger/web/index.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <script>
5        let nextColor = "green";
6        document.addEventListener("keydown", () => {
7          document.body.setAttribute("style", "background-color:" +
               nextColor);
8          nextColor = nextColor === "green" ? "red" : "green";
9        });
10     </script>
11   </head>
12   <body></body>
13 </html>
```

This is sufficient to change the color of the screen back and forth between easily distinguishable colors for the data loggers of both the client and the host to record. With both of these datasets recorded, the next step is to analyze the data and evaluate the performance of the application.

Summary

Chapter 6

Evaluation

In order to evaluate the performance of the forked version of Moonlight developed for this paper, it must be tested against the other existing solutions. The testing methodology employed to ensure the accuracy of the testing is described in Section 6.1. Then, the responsivity and latency of each solution is evaluated in Section **??**, and the quality of each solution's stream is evaluated in Section 6.3. Then, the forked version of Moonlight is evaluated in real world scenarios in Section 6.4. Finally, the results are summarized and verdicts on each of the solutions' applicability to the research questions will be presented in Section 6.5.

6.1 Testing Methodology

In order to ensure that the data collected for analysis is accurate and directly comparable, the testing environment must be kept exactly the same between trials and between different applications. This gives each application the exact same starting conditions, and the best chance to demonstrate it's unabated performance. For these tests, the host machine is a desktop computer running Windows 10 with a AMD Ryzen 9 5900X CPU and a Nvidia RTX 3070 GPU running the latest GeForce Experience version *3.25.0.84*. This configuration does not change throughout testing. The client machine is the CM4 board attached to the PCB developed in Chapter 4, running the custom Manjaro linux distribution and forked version of Moonlight developed in Chapter 5. This configuration also does not change throughout testing. Each test is performed with both the host computer and the client machine running on the same network, with both devices connected to the same router using an ethernet cable. This ensures that the speed of the network is not a factor in the tests. The tests are performed multiple times under the following conditions:

- Ideal Conditions: No other applications are running during testing.
- CPU Stressed: The CPU is constantly at 100% load during the entire test.
- GPU Stressed: The GPU is constantly at 100% load during the entire test.
- Both Stressed: Both the CPU and the GPU are constantly at 100% load during the entire test.

This is to examine whether each application can perform under intensive conditions. For each test, the following steps are performed:

1. Both the host and client computers are fully rebooted to ensure no other application is running.

2. The host computer starts up a single browser window with the web page described in Section 5.4.

3. If the test calls for the CPU or the GPU to be stressed, the application "Blender" is started on the host machine, and a benchmark is performed for the CPU or the GPU. This benchmark will run for longer than the duration of the test, and will keep the CPU and/or GPU at 100% load for the entire duration of the benchmark.

4. Both devices start up their respective data collection tool that will record keystrokes and changes in the screen's color.

5. The client machine connects to the host computer using the application being tested.

6. The tester presses the "=" key on the client machine to begin the automated testing process. Due to the wide variance in what the user can do using the software, it is nearly impossible to test every possible real world scenario. Instead, the automated test simulates an active working environment by sending a repeatable series of keystrokes that cause the entire screen to change colors to mimic high-intensity applications. For consistent testing, 400 keystrokes are automatically sent with a 500ms delay between inputs.

7. Once the trial is complete, both the client and the host computers close their data recording tools and the client ends the streaming session.

This process is repeated for each of the following applications:

- The forked version of Moonlight developed for this project.
- Chrome Remote Desktop (CRD, Section 3.1.3).
- Microsoft's Remote Desktop Connection (RDP, Section 3.1.1).
- Virtual Network Computing (VNC, Section 3.1.2).

Once all the data is collected, the data from the host and client machines for a single trial are organized into a single CSV file for processing. In order to ensure that the data is comparable, all extraneous factors must be removed from the data. Namely, the network latency and difference in computer clocks must be removed. By keeping the host and client computers attached to the same network through ethernet, the network latency is minimized. But the task of keeping two computer's internal clocks synchronized is not trivial. Even after instructing the operating system to synchronize its clock with an internet time server, the two computers may still report different times. In some cases, the difference between the host and client computer's system time was noticed to be upwards of a full second after resynchronizing both clocks [36]. This is partially accounted for in step *6* of the above process, and the remaining inaccuracy can now be removed by subtracting the time difference between the when the client presses the button and the time the host executes that command. It is important to not subtract the entire round trip time of the input, as that is what is being testing in the following section, but rather just the time between when the client sends the input and when the host receives it. This ensures that any inaccuracies in each computer's internal clock will not impact the data.

After the data is cleaned up, the data can be processed by matching each keystroke and change in the screen's color detected by the client and host machines and calculating how much time elapsed between the events. This reveals the amount of delay the user experiences between when they press a key and when the host executes the command, and between when they press a key and can visually see the response.

6.2 Responsivity and Latency

The first factor in determining the performance of the application is to take a look at how responsive it is to use. Responsivity is defined as how quick the application responds to inputs and how quickly the user can see the results of their inputs. This is important for general user experience and applications where continuous input is required. This is also what is generally referred to when determining whether an application has a positive user experience, since a program that can take seconds to respond to the user begets lower productivity and a more frustrating experience. To analyze this, two points of data are recorded: First, the amount of time between when the client records a keystroke and when it is actually inputted on the host machine, and secondly, the amount of time between the host machine updating their screen and when the client actually displays that update. The first datum is generally referred to as the latency of the application, since even though each trial is done using the same network setup different solutions may have different optimizations built in that reduce the time needed for the host to implement keystrokes made by the client. This means that as the network delay is removed from the data, all that is left is the time that it takes the software on the host's machine to actually execute the input. Each of the following figures shows data over one trial for each category. Each trial consists of the client machine programmatically sending 400 keystrokes with 500ms between each input. Limitations in this data is discussed in Section 7.2. Figure 6.1 shows the amount of time between the client and the host registering the

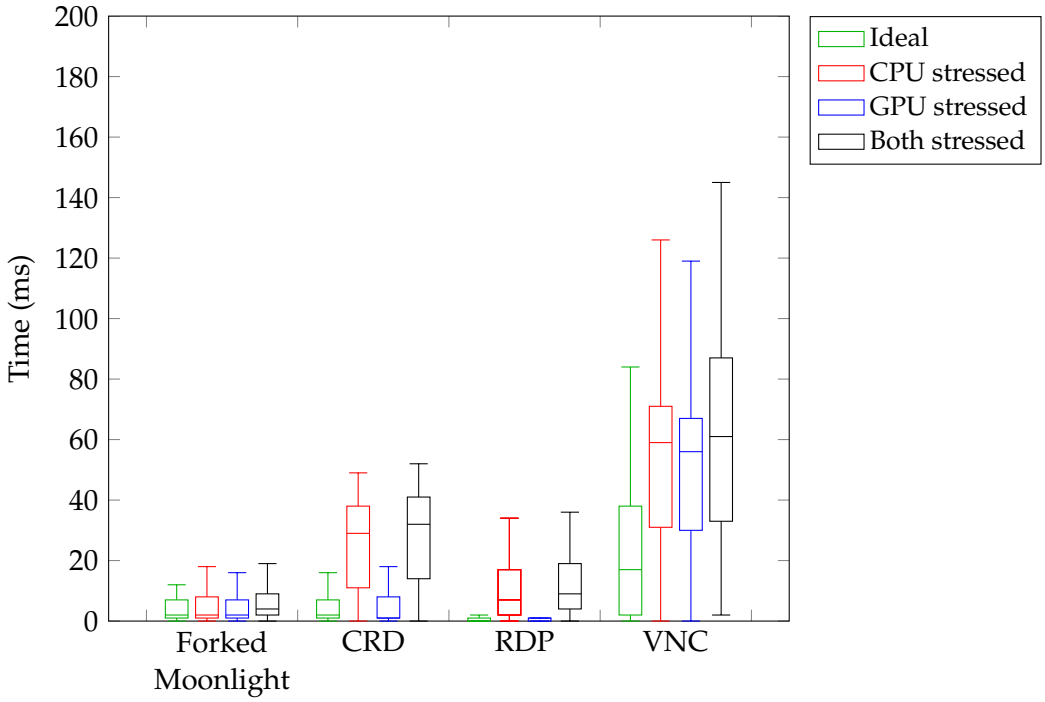


FIGURE 6.1: Time for host to press key from client. n=400.

keystroke under the different test conditions.

Each application does a good job sending data from the client to the host in a timely manner, with the majority of keystrokes being realized within a tenth of a second of being pressed on the client machine. The forked Moonlight implementation keeps a latency under 20ms regardless of the load of the host machine, with more than 75% of all keystrokes being realized within 10ms. CRD proves to be a CPU limited application, with latency while idle or under GPU load also staying below 20ms, but climbing up above 50ms when under CPU load. This is important to keep in mind when thinking about what sort of situations the host machine will be in while being accessed. If the user is going to be rendering video remotely, a GPU intensive process, the latency of the stream wont be impacted, but if the user is going to be running a CPU intensive process, the latency can be expected to be negatively impacted. RDP has a similar results, with extremely low latency while idle or under GPU load, and upwards of 40ms while under CPU load. VNC has the highest variance, with no discernable affinity to any kind of load. This lines up with it's intended use described in Section 3.1.2, being the simplest and most straightforward method of remote desktop control without much focus on performance or security. That's not so say VNC is unusable or even difficult to use; latency of 140ms is still usable in all but the most precise situations.

Though this is only one metric to consider, the other is the amount of time it take for the client to display the results of their input. This is recorded as the amount of time it takes between the host machine changing the color of it's screen and the client rendering the change. As noted before, the color of the host's screen changes with each input sent by the client, which occurs every 500ms for 400 keystrokes. Figure 6.2 shows the amount of time for each application under each condition.

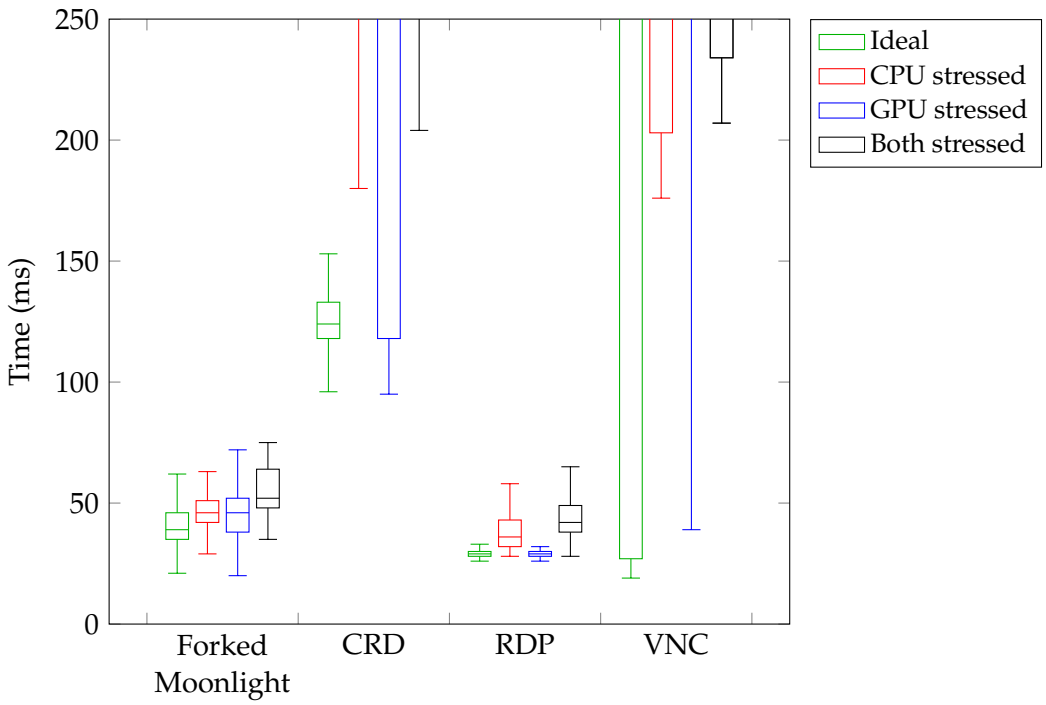


FIGURE 6.2: Time for client to render update to the screen. Values over 250ms push the usability of the system. n=400.

This data demonstrates a similar but more extreme pattern to the input delay figure, with most application performing well under ideal conditions but struggling

a little under high stress. The forked Moonlight implementation is able to render updates on the client device in under 70ms in nearly all cases, with an even spread regardless of the load of the host machine. CRD is able to render all updates in under 160ms while the host is idle, but struggles to keep up while under load. Again, CRD struggles the most while under CPU load, with the quickest update happening in just under 180ms, and the median update taking over a full second. CRD performs a little better under GPU load, but still experiences much high delays with the median update taking over half of a second. Once both the CPU and the GPU are under load, the worst of both scenarios is realized as the median update taking nearly three quarters of a second. All three of these values push the usability of the system as any kind of intensive program will bring the responsivity of the application to a crawl. RDP again performs extremely well, with all render updates taking less than 50ms while under ideal conditions or GPU load, and all updates taking less that 70ms while under full CPU load. VNC is again the slowest and most inconsistent application, with all conditions resulting in a frequent render delays of more than a quarter of a second. This makes VNC considerably less suitable for the task at hand.

The delay, however, is only half the story in this case. Another factor to consider is the percentage of updates that are not rendered by the client. This, often referred to as dropped frames, records how much data is lost when the host sends it's screen to the client device. A higher percentage results in a more choppy and difficult to use experience, with a 50% loss meaning that 50% of the updates sent by the host are not rendered by the client. Figure 6.3 shows the percentage of color changes not rendered by the client for each of the load conditions.

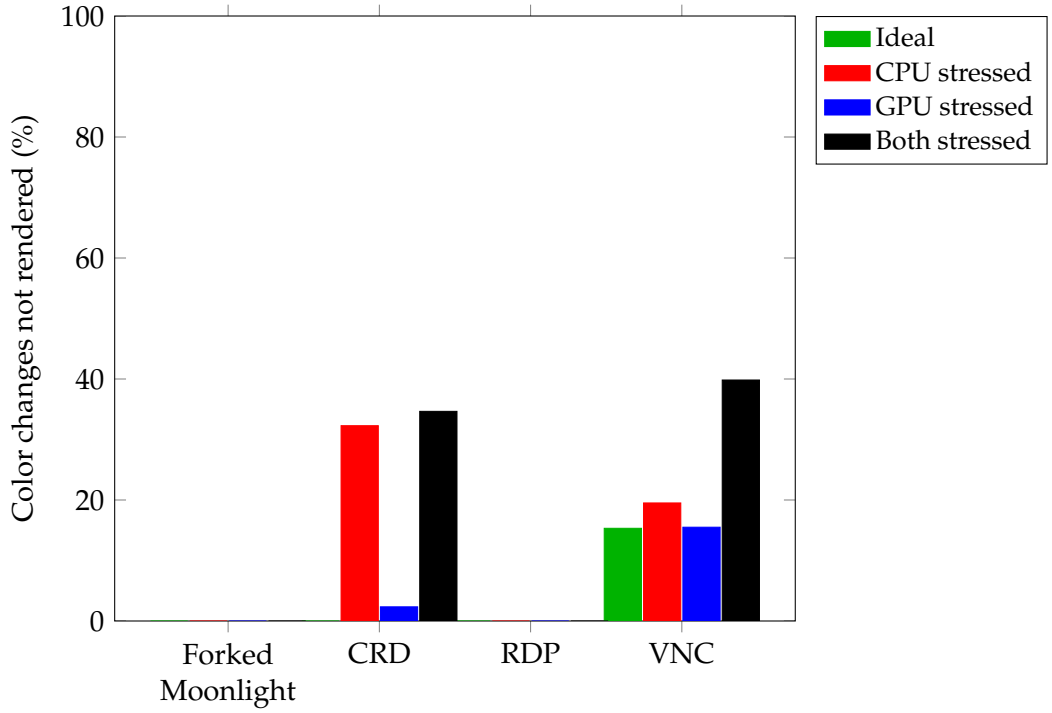


FIGURE 6.3: Percentage of color changes not rendered. n=400.

These data points adds context to the previous figures, showing where the applications struggle to keep up with sending data back to the client device. The forked Moonlight implementation does not experience any frame drops, as does CRD under ideal conditions. But once the host machine is put under heavy load, CRD begins to drop frames, with a few being dropped under GPU stress, and 32% of frames

being dropped while under CPU stress. This can make CRD unpleasant and frustrating to use while the host is running a CPU intensive program as not only does the client take much more time to update the screen to reflect that status of the host computer, but many of the updates are lost completely and not shown to the user. RDP is also able to display all the data sent by the host without dropping any frames under any condition. VNC is again proved to be the least performant, with frames being dropped regardless of the load being put on the host computer. This is still in line with it's intended use of basic remote control, though it is disappointing to see about 15% loss even under ideal conditions.

6.3 Quality

Apart from the performance of each application, the quality of the user experience must also be taken into account when considering each application. Each application will be assessed individually, and then compared in Section 6.5. Due to the closed nature of some of the protocols, only limited amounts of data could be extracted and compared. This prevents direct comparisons, but can still contribute to the understanding of the overall performance of the solution.

6.3.1 Moonlight

When analyzing the forked version of Moonlight, specific quantitative data described in Section 5.4 can be collected directly from the application while it is running. Once the streaming session concludes, the statistics are collated into human-readable numbers and presented in the console. What follows is an example output from one of the test streaming sessions:

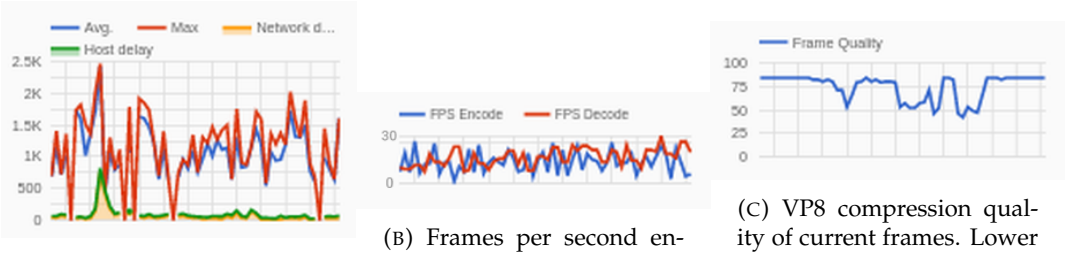
Statistics generated after a streaming session

```
Global Video Stats
------------------------------------------------------------
Incoming frame rate from network: 62.18 FPS
Decoding frame rate: 62.18 FPS
Rendering frame rate: 62.18 FPS
Frames dropped by your network connection: 0.00%
Average network latency: 1 ms (variance: 0 ms)
Average decoding time: 4.27 ms
Average rendering time (including monitor V-sync latency): 0.06 ms
Total dropped frames: 0
```

This data shows that Moonlight is able to render all frames sent by the host without dropping any frames due to the network or decoding delays. It is also able to keep the stream running at over 60 FPS throughout the entire session, which far exceeds the requirements of a pleasing to use system. Taking into account the average frame decoding time of $4.27ms$ and the average rendering time of $0.06ms$, the program still has plenty of time in-between frames in case the streaming conditions are less favorable: $(4.27ms + 0.06ms) = 4.33ms < 16.\overline{66}ms = (\frac{1000ms}{60fps})$. It is also worth noting that while testing the streaming quality, there were very few visual artifacts while monitoring the screen. There were very few times where the screen appeared to be run through a compression filter, and it for the most part kept the visual fidelity of the host machine.

6.3.2 Chrome Remote Desktop

Chrome Remote Desktop also provides some quantitative data during streaming, which gives an insight into it's performance usage and speed. Figure 6.4 shows screenshots of the stream during the most demanding test session.



(A) delay of image rendering in milliseconds

(B) Frames per second encoded by host and decoded by client

(C) VP8 compression quality of current frames. Lower means CRD is struggling to keep up

FIGURE 6.4: Statistics from CRD during the most demanding test session. Data recorded over 60 seconds.

Unfortunately, as there is no official documentation for this data, analysis on these figures is limited but insights on the stream quality can still be found. As seen in Figure 6.4a, the delay of image rendering climbs as the CPU is put under stress, with frames sometimes taking entire seconds to be rendered on the client machine. It is unclear on how much of the delay is due to the host machine versus the client machine, as the delay obviously rises as the host's CPU is put under stress, but it is clear that as the host machine is under heavy load the frames can take entire seconds to be rendered on the client machine. Figure 6.4c shows the quality setting of the VP8 compression algorithm being adjusted over time to keep up with demand. This is a decent indication of the quality of the video stream over time, as the quality drops in an attempt to keep the stream updating the client with as much frequency as possible instead of sending full frames. With the knowledge that CRD leverages the VP8 compression algorithm [16], it is possible to infer that as the quality goes down, the frames being sent contain less information about the current frame and instead more temporal data to construct the new frame from the previous frame. This is also seen with visual smudging on the client's screen as the stream slows down.

6.3.3 Remote Desktop Protocol

Microsoft's Remote Desktop Protocol does not provide quantitative data to analyze it's quality, but speaking from a user experience perspective, and considering the data provided in the previous section, RDP performed very well under each of the testing conditions. By dropping zero frames and visually keeping up with the host machine, it is clear that RDP works extremely well over a local network. Similar to Moonlight, there was very little loss of visual fidelity in the client's screen compared to the host's screen. However, noticing that the bandwidth usage can climb upwards of 10 Mb/s, such streaming may cause issues while attempting to access the host machine while outside of the local network [17]. This will be further discussed in Section 7.2.

6.3.4 Virtual Network Computing

VNC provides little in the form of quality statistics, but given it's performance and responsiveness discussed in the previous section, it is clear that the quality of the

stream suffers heavily whenever the host is under load. As discussed in Section 3.1.2, VNC was never built to be a highly performant protocol. The testing done for this paper really stretches the limits of the protocol and pushes the use case of VNC to the extreme, but it is still worth looking into a common and widely used tool.

6.4 Real World Testing

While the testing conducted in the previous two sections provide an overview of how each application performs under controlled conditions, testing in the real world provides better insight into how a user may experience the final product. Unfortunately, there are many limitations in how testing can be conducted in the real world. First and foremost, due to the closed nature of of the existing solutions, collecting data while using them in real-world scenarios is impossible while leveraging the tools from the previous sections. Because of the reliance on using exact and repeatable testing conditions, a dynamic test of each application is much more difficult to conduct without direct access to the performance of the program, all data that could be collected otherwise would amount to quantitative data subject to interpretation. However, because of the logging tools that were developed in Section 5.4, it is possible to collect data from the forked version of moonlight that shows the performance of the application over an entire streaming session. While it is impossible to predict and exhaustively test what the user may try to do with the final product, it is possible to test common use cases and evaluate whether the streaming experience is satisfactory.

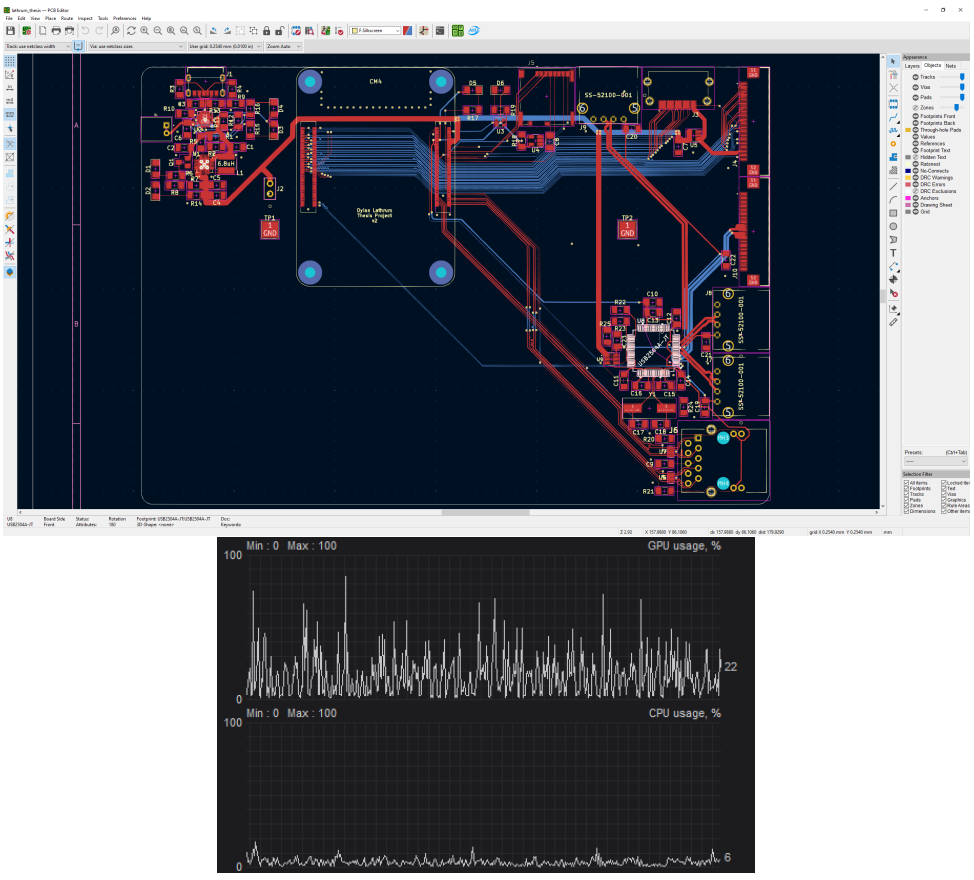


FIGURE 6.5: Working on the PCB for this thesis using KiCad.

The first use case that was tested was streaming KiCad to the client machine to inspect the PCB layout of the custom board (Figure 6.5). This was to test the ability to work with highly accurate CAD programs to ensure that the small client device could be used for technical work. The first thing that was noticed was that since the chosen screen for the client device was small, everything had to be zoomed in much further than normal to be comfortable. This is where the secondary HDMI output could be used to plug in a larger, more conventional monitor, but in the event the user did not have access to such a hardware, the small screen needed to be sufficient. Once zoomed in, KiCad was readable and usable to without any further compromise. The client's inputs were responsive and accurate enough for precise movements to be made without error and without the worry of needing to wait for the screen to catch up to render the changes. Once the five minute testing session was complete, the following statistics were recorded:

Statistics recorded while streaming KiCad (Figure 6.5)

```
Global Video Stats
------------------------------------------------------------
Incoming frame rate from network: 61.23 FPS
Decoding frame rate: 61.23 FPS
Rendering frame rate: 61.23 FPS
Frames dropped by your network connection: 0.00%
Average network latency: 1 ms (variance: 0 ms)
Average decoding time: 4.24 ms
Average rendering time (including monitor V-sync latency): 0.06 ms
Total dropped frames: 0
```

These results are very similar to the results recorded in the previous sections, where the frame rate is able to be kept above 60 FPS throughout the test, and zero frames were dropped.

The next use case that was tested was programing while training an AI model (Figure 6.6). While programming on it's own is not a very intensive process, and processes such as compiling can be done while the user isn't actively interacting with the computer, training an AI model can be a very time consuming process that may need to continue in the background while the user continues to do other work. As this scenario is extremely similar to the CPU stress test from Section 6.2, the same results are expected.

Statistics recorded while training an AI (Figure 6.6)

```
Global Video Stats
------------------------------------------------------------
Incoming frame rate from network: 62.05 FPS
Decoding frame rate: 62.05 FPS
Rendering frame rate: 62.05 FPS
Frames dropped by your network connection: 0.00%
Average network latency: 1 ms (variance: 0 ms)
Average decoding time: 4.26 ms
Average rendering time (including monitor V-sync latency): 0.05 ms
Total dropped frames: 0
```

As with before, the stream remains performant and usable throughout the test. There was no point during the test that the stream started to struggle as the AI training process kept the CPU at 100% usage.

The next use case that was tested was streaming the 3D modeling, animation, and rendering program Blender (Figure 6.7). To test this, a demo scene provided for

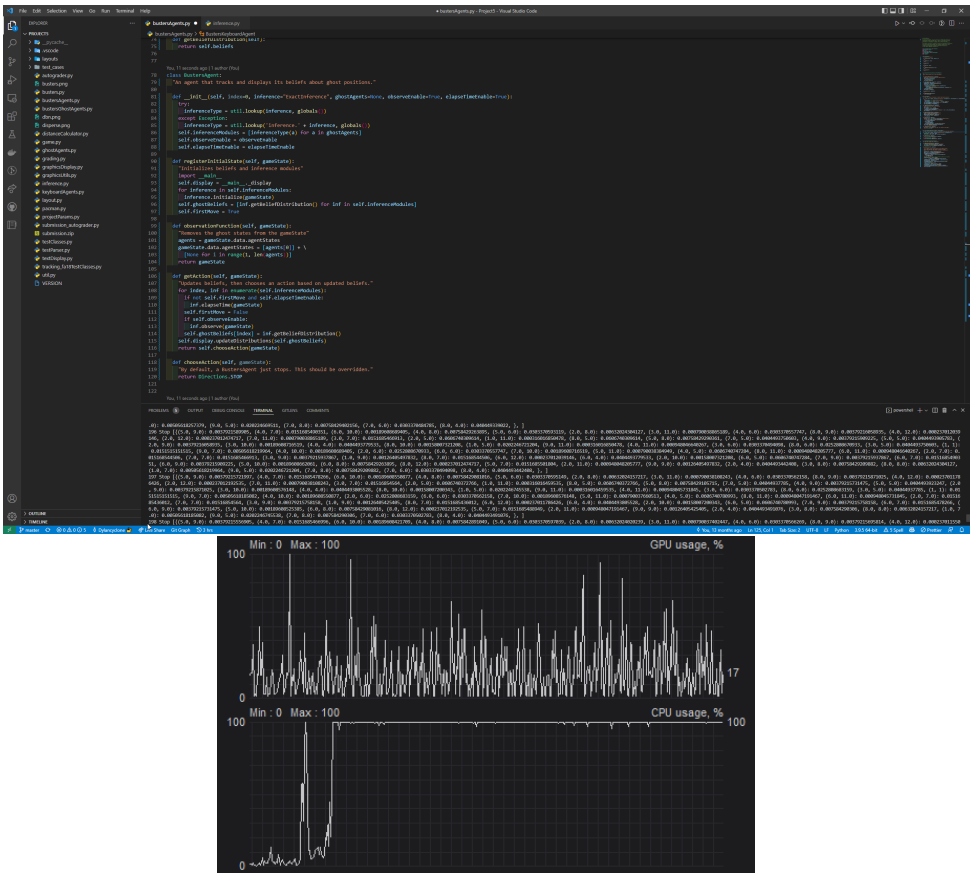


FIGURE 6.6: Programing while training an AI model.

free by the Blender Foundation was loaded and edited in one window while being rendered from the camera's point of view in real time in another window [2]. This put inconsistent strain on the computer as it constantly had to rerender the scene as changes were made. Whenever the scene was rendered to a reasonable resolution, Blender would automatically stop rendering to keep resource usage low, resulting in the CPU and GPU usage to bounce back and forth between high and low usage. This proved to be the most strenuous test of the application:

Statistics recorded while streaming Blender (Figure 6.7)

```
  Global Video Stats
------------------------------------------------------------
Incoming frame rate from network: 57.30 FPS
Decoding frame rate: 57.30 FPS
Rendering frame rate: 57.30 FPS
Frames dropped by your network connection: 0.06%
Average network latency: 1 ms (variance: 0 ms)
Average decoding time: 4.70 ms
Average rendering time (including monitor V-sync latency): 0.05 ms
Total dropped frames: 16
```

This was the only test in which not all frames rendered successfully, though the majority 99.4% of frames rendered as expected. In fact, due to the stream running at an average 57 FPS, the 0.06% loss of frames was imperceivable during the test and only became known once the statistics were produced. During the test, the stream remained performant and stable. Though applications such as this are more difficult

FIGURE 6.7: Working on a blender animation while rendering the scene.

to use on a smaller screen, it was by no means unusable; and because of the minimal delay from the stream, there was no point during the test where it felt like the output of the tester's inputs were not visible as quick as they normally would.

The last use case that was tested was streaming the development of a video game level while inspecting in the game (Figure 6.8). This was intended to be an general all-round stress test that would put the host under multiple kinds of stress at the same time. Since the host computer had to run not only the video game's development tools but also the game itself, the stream had to keep up with the precise nature of geometric level design as well as interacting with the game. During the test, the client machine always felt as though it was not compromising the experience of working on the host machine save for the small display; inputs in the development tools stayed precise and accurate, and interacting with the game was as smooth as playing on the host machine. Once the test concluded, the following statistics were recorded:

Statistics recorded while streaming the development of a video game level (Figure 6.8)

```
Global Video Stats
------------------------------------------------------------
Incoming frame rate from network: 61.93 FPS
Decoding frame rate: 61.93 FPS
Rendering frame rate: 61.93 FPS
Frames dropped by your network connection: 0.00%
Average network latency: 1 ms (variance: 0 ms)
```

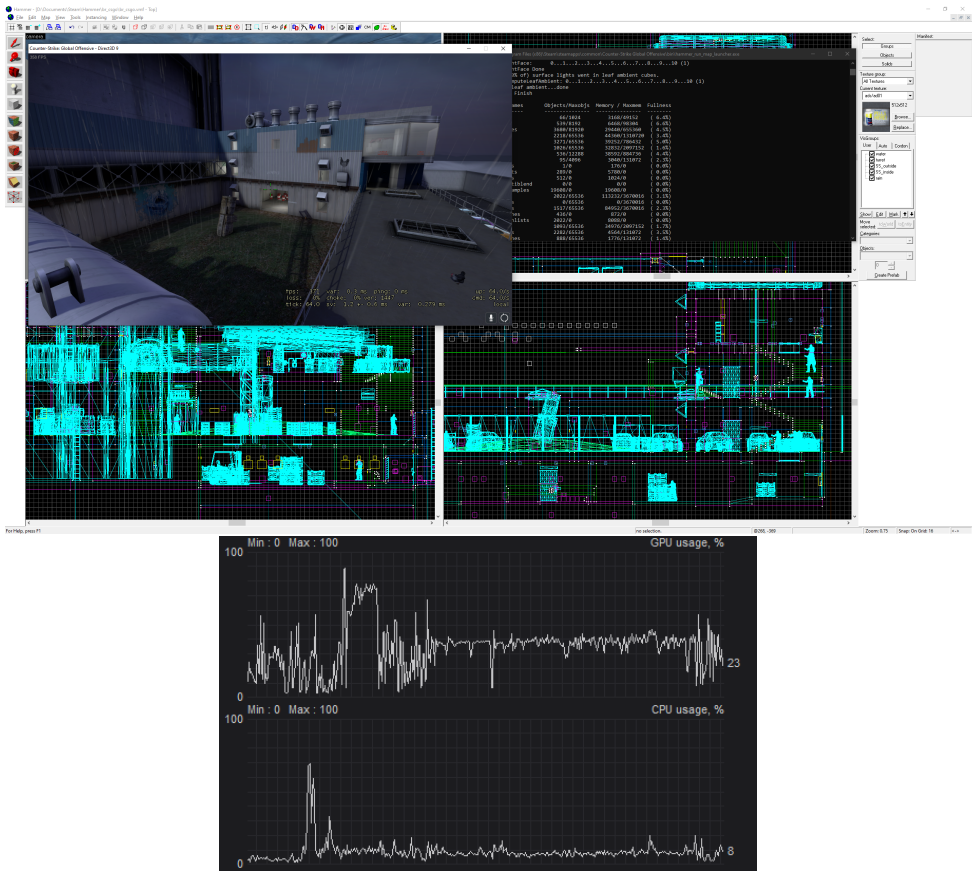


FIGURE 6.8: Developing a video game level while inspecting it in game.

```
Average decoding time: 4.27 ms
Average rendering time (including monitor V-sync latency): 0.06 ms
Total dropped frames: 0
```

As with the most of the previous tests, the stream kept a constant frame rate above 60 FPS and never dropped any frames.

6.5 Summary

By combining the information from the previous three sections, a wholistic picture of the performance of each application can be drawn. Firstly, Virtual Network Computing works as a bare-bones method of connecting to a host machine from a client device in the simplest way possible, but struggles at keeping up with even mildly demanding tasks. As seen in Figure 6.3, VNC failed to provide a smooth experience for the user even in ideal conditions by simply not rendering updates to the screen in time for the next frame to be shown. This, combined with a slow response time and delayed inputs, causes a poor experience for the user when attempting to use demanding applications or anything that requires speed, making it unsuitable for the purposes of this paper.

Secondly, Chrome Remote Desktop performs well under ideal conditions and even succeeds at at maintaining steady performance under some GPU load, but it struggles to keep up whenever the host's CPU is under stress. Due to CRD's reliance on web technologies, it too suffers from the same limitations as seen in other

web streaming applications such as visual smearing and dependence on the CPU. In fact, these issues are something technology companies like Google have been trying so solve, with the release of their new VP9 codec used for YouTube starting in 2014, but such technology has not yet been introduced to CRD [5]. Unfortunately, due to the dependence on the CPU, and difficulty in handling conditions where the host computer is running strenuous tasks, Chrome Remote Desktop is not a suitable software solution for the purposes of this paper.

Thirdly, Microsoft's Remote Desktop Protocol proves to be a very performant protocol that can handle nearly every situation without a problem. It is slightly CPU limited, as seen in Figure 6.1, but even then it is able to keep input latency under a very reasonable 40ms, and it still manages to render the host's screen without dropping any frames. It even proves to be the most stable and consistent solution in Figure 6.2, producing the smallest variance in the render delay of the client's screen. If it weren't for the limitations brought up in Section 3.1.1, RDP would be a great solution for streaming demanding applications to a remote client. However, because of these drawbacks, RDP is not suitable for every application.

Finally, Moonlight proves to be a high-performant solution that isn't limited by the host's CPU or GPU being under stress. Each test resulted in highly similar results, and while it does on average have slightly higher response times compared to RDP (Figures 6.1 and 6.2), it only falls behind by 5-10ms – quicker than the blink of an eye. By not dropping any frames, and numerically proving it's ability to keep the stream above 60 frames per second, Moonlight proves to be a great solution for streaming demanding applications to a remote client without compromising its usability.

Chapter 7

Conclusion

7.1 Summary

This thesis has proved that it is possible to produce a hardware and software solution that is capable of leveraging the power of a desktop computer remotely without sacrificing the user's experience. It improves upon existing solutions by continuing to offer a performant and high-quality stream of the host's screen to the user even as the host is running highly demanding applications. Each of the questions posed in Section 3.3 can be answered individually.

1. **Hardware Feasibility:** *What hardware is needed in order to power a mobile device capable of acting as a client?*

This thesis project is able to run on embedded Single Board Computers and hardware available for purchase today. While this project built a custom PCB for the Raspberry Pi Compute Module 4 (Discussed in section 4.2), the software could also run on consumer-grade main-line Raspberry Pi models as well.

2. **Hardware Cost:** *Is it feasible to construct such as device at a lower or comparable cost to existing solutions?*

Yes, the total cost for a single hardware unit, including the board and all its components, totalled in under $150, and could even be produced for less (Discussed in Section 4.1.2). This cost is considerably cheaper than buying a new laptop device, and often cheaper than buying a used laptop with comparable power.

3. **Communication Protocol:** *Does a protocol exist that is efficient enough to stream demanding applications to a client?*

Yes, Nvidia's GameStream protocol proved itself to be capable of streaming highly demanding applications to a client device without any loss of quality or performance (Discussed in Section 6.5). Even though it's original purpose and namesake is to stream video games, the high demands of such a use case have made it applicable to much more than just that. While Microsoft's Remote Desktop Protocol proved to be powerful enough to stream demanding applications to a client device, it's limitations and closed-source nature prevent it from being the viable solution for certain use cases (Discussed in 3.1.1).

4. **Software Application:** *Can a software solution be built to utilize this protocol in a manner that performant enough?*

Yes, the solution produced in this thesis is built upon the open source project Moonlight, which utilizes the GameStream protocol, to provide a software application that is capable of streaming demanding applications to a client device while remaining performant enough to provide a positive user experience (Discussed in Section 6.5).

7.2 Limitations

While this thesis was constructed with the utmost attentiveness with regards to the time and resources available, there are still limitations in what could be accomplished. The largest limitation is the lack of testing in alternative environments. All of the testing performed in Chapter 6 was performed in an ideal network environment, where there was minimal network latency between the host and client machines due to them both being connected to the same network by ethernet. While this is a good indicator of the best possible performance, it is not very indicative of the performance that could be expected in a real-world environment. More testing should be done where the host and client machines are located on different networks akin to the use case described in Section 2.3.

Secondly, due to the ongoing chip shortage and the lack of experience in developing PCB boards, it is very likely that there are better ways to develop the PCB board for this project. While the board was able to meet the requirements for this project, there we're many instances (some described in table 4.1) where alternative parts had to be chosen and designed around because the first-choice was out of stock. It is very possible that the board could have been constructed quicker and at a cheaper price if more parts were available for purchase.

7.3 Future Research

Should this project be continued in the future, there are multiple ways that additional time and resources could further improve this project. The first improvement to make is to fix the issue with USB data described in section 4.5.3. This is the only flaw in the hardware preventing it from being considered a completed product. Similar to the previous section, if more time were available, more testing could be done in various environments to identify any weaknesses in the software that may have gone unnoticed.

Another aspect of the project worth testing in more detail is the performance of the hardware produced in Chapter 4 compared to the hardware performance of the development board that is sold with the CM4 (Section 4.2). The ideal outcome is that the two boards provide the same performance, meaning that there is no unneeded overhead introduced by the board produced in this thesis that should be fixed. In the event that there is slowdown somewhere within the custom board, the specific problem should be identified and redesigned to provide optimal performance.

After experimenting with streaming over different networks and testing the project under different conditions, it would be beneficial to look at polishing the hardware of the project into a self-contained product. As it stands, it is currently a PCB with peripherals attached that make it cumbersome to carry around. The groundwork for this has already been laid out in Section 4.4.3, with FFC connectors established for one HDMI and one USB port for connection to an embedded screen, and with the footprint of the final PCB and battery fitting within the footprint of the chosen screen. All that is left is to build an enclosure to house the final product.

Appendix A

PCB Specifications

A.1 PCB Schematic

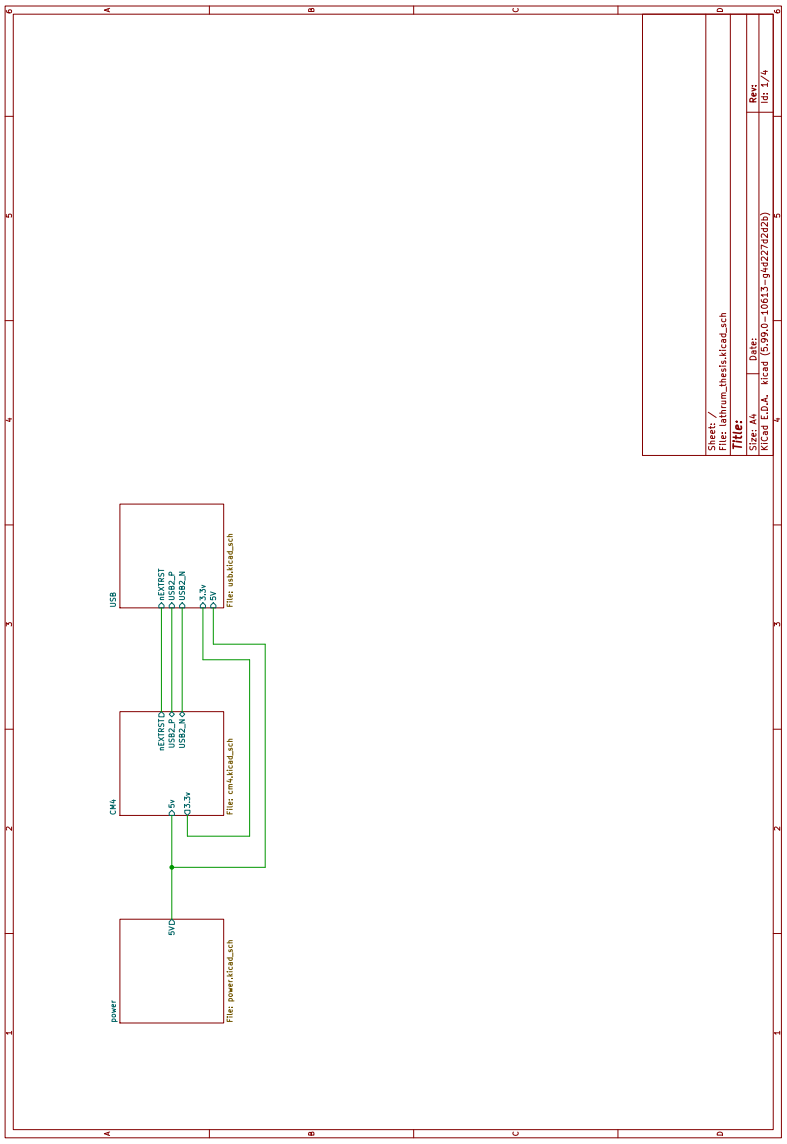


FIGURE A.1: The top level of the PCB board schematic

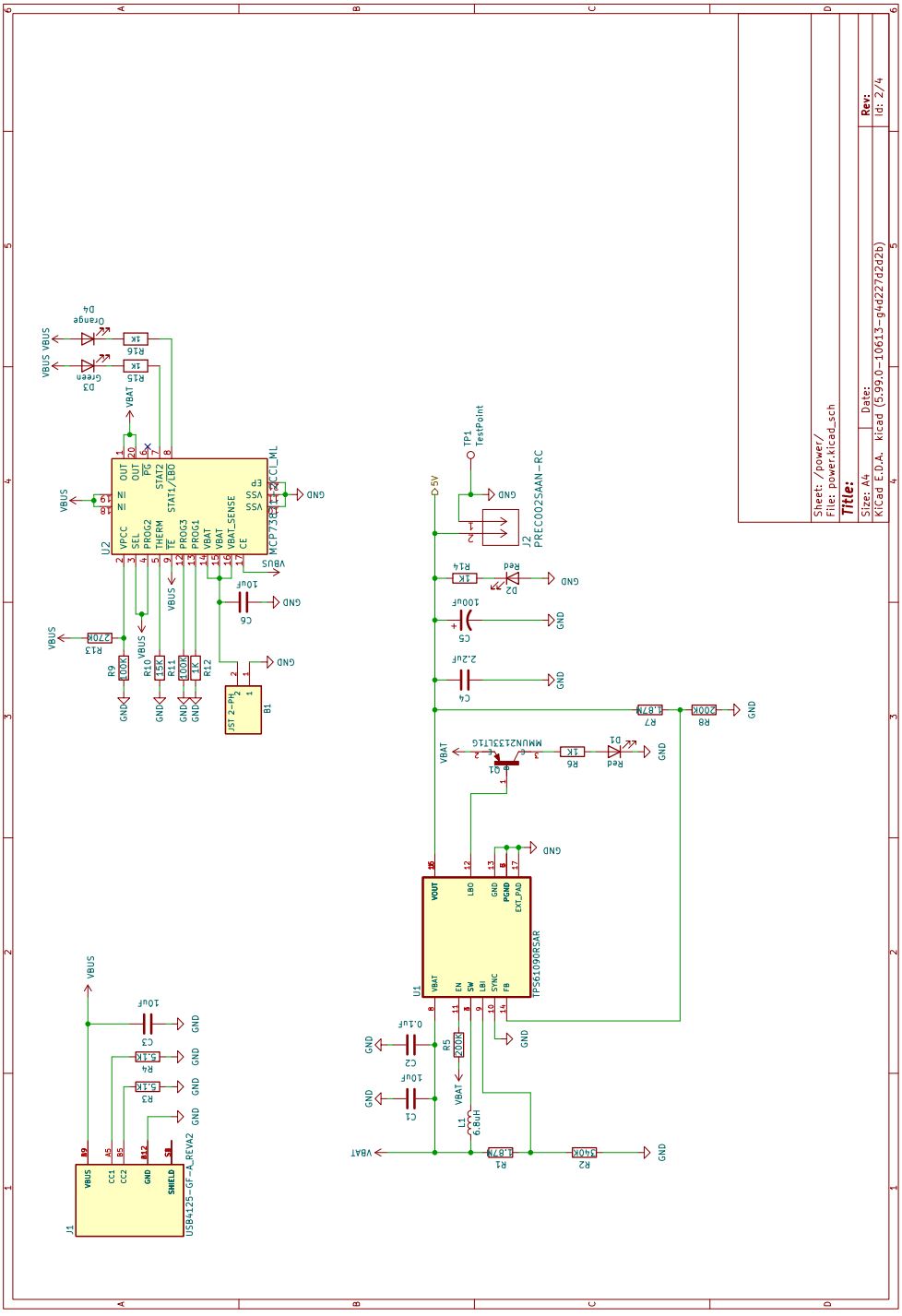


FIGURE A.2: Schematic layout for the power circuit, including battery management

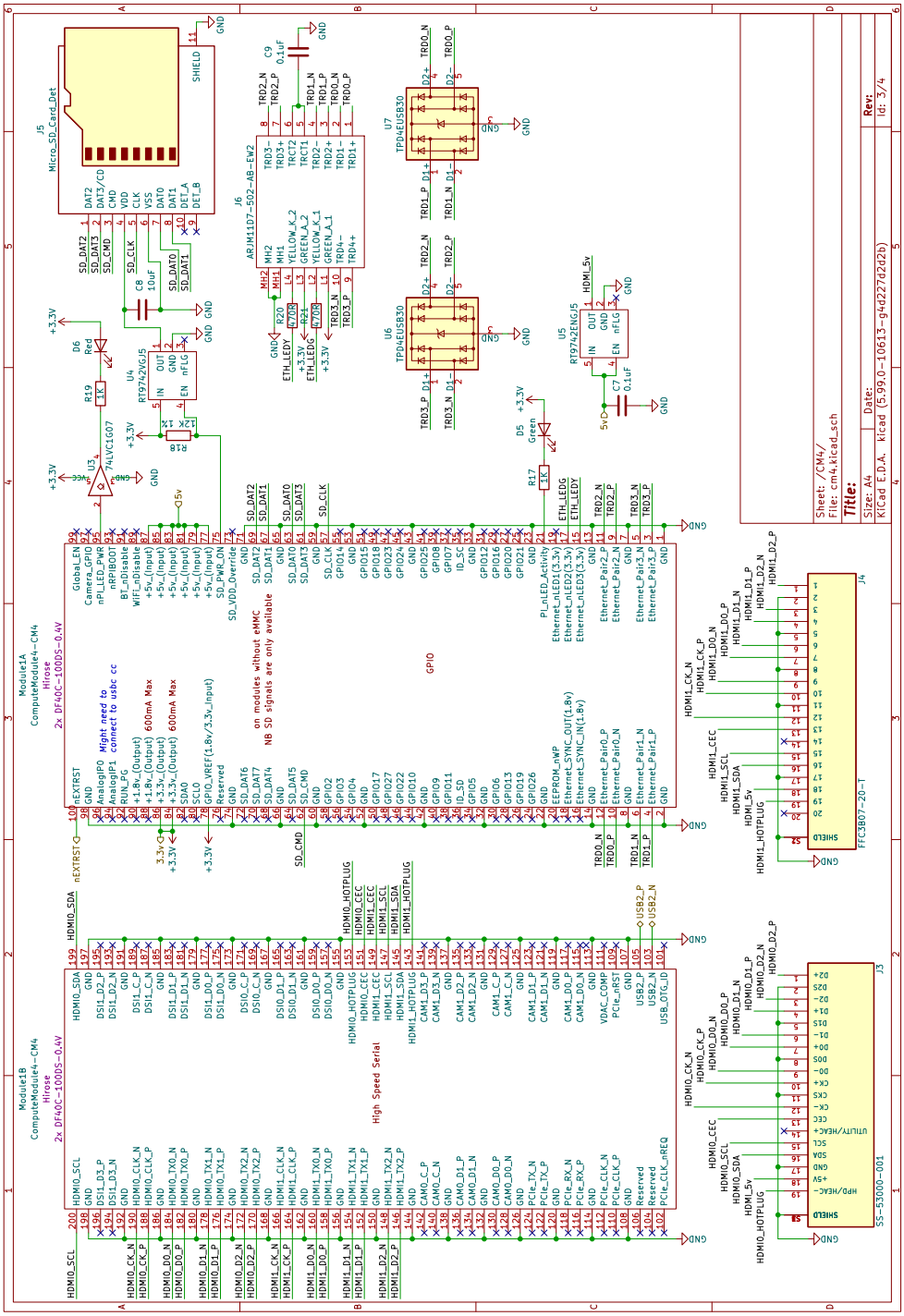


FIGURE A.3: Schematic layout for interfacing with the Compute Module 4

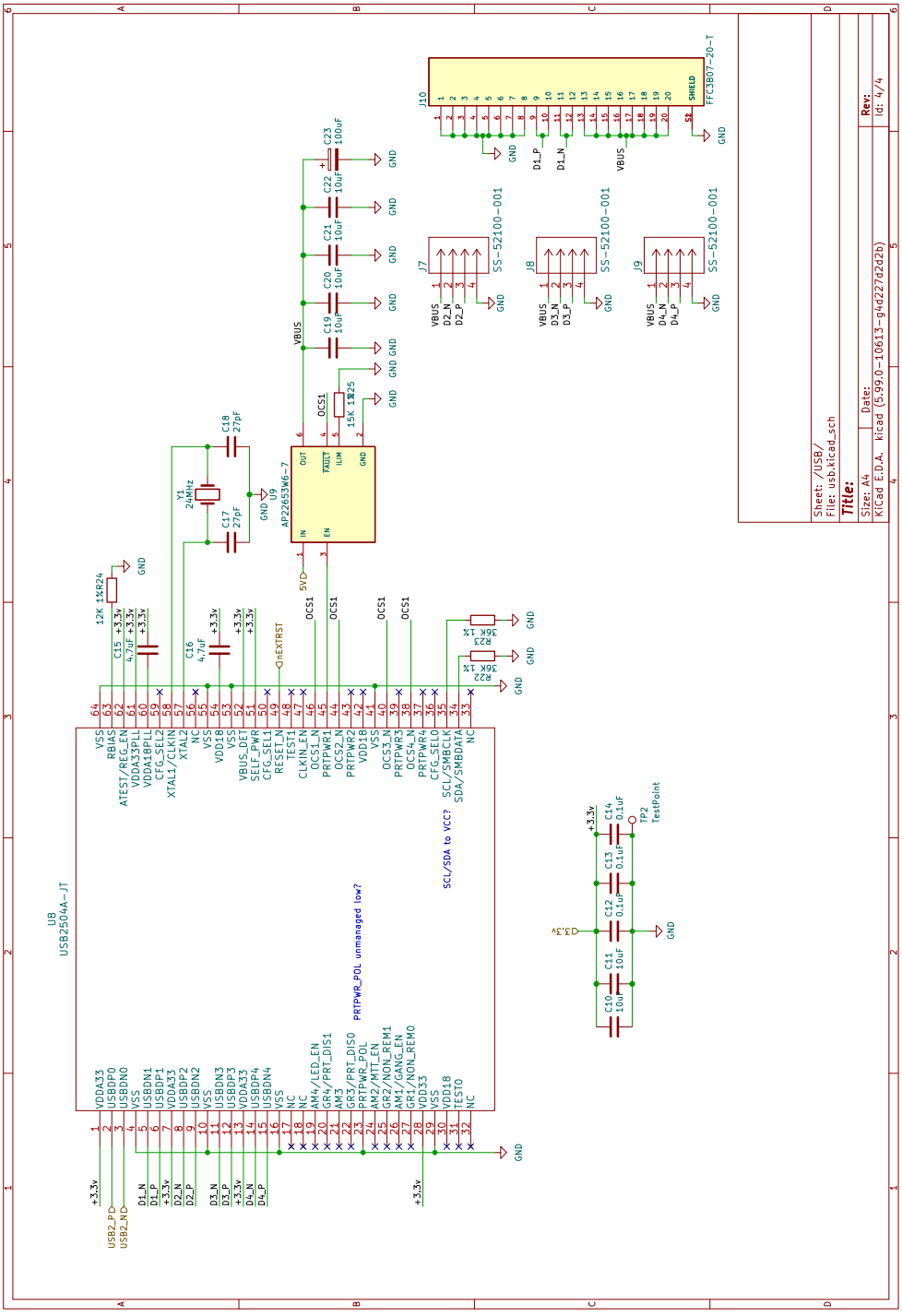


FIGURE A.4: Schematic layout for the USB hub and ports

A.2 PCB Layout

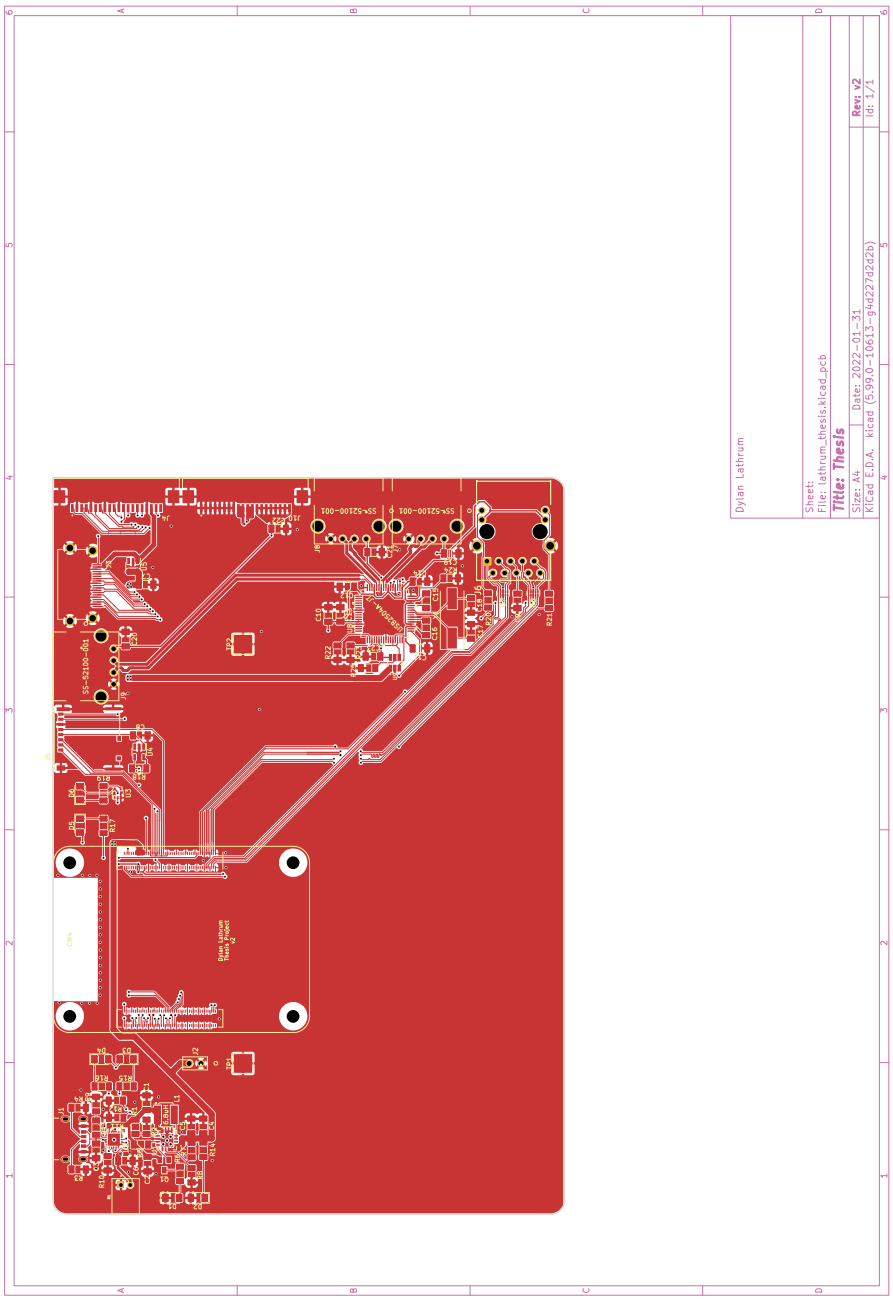


FIGURE A.5: Physical board layout for the front side of the PCB

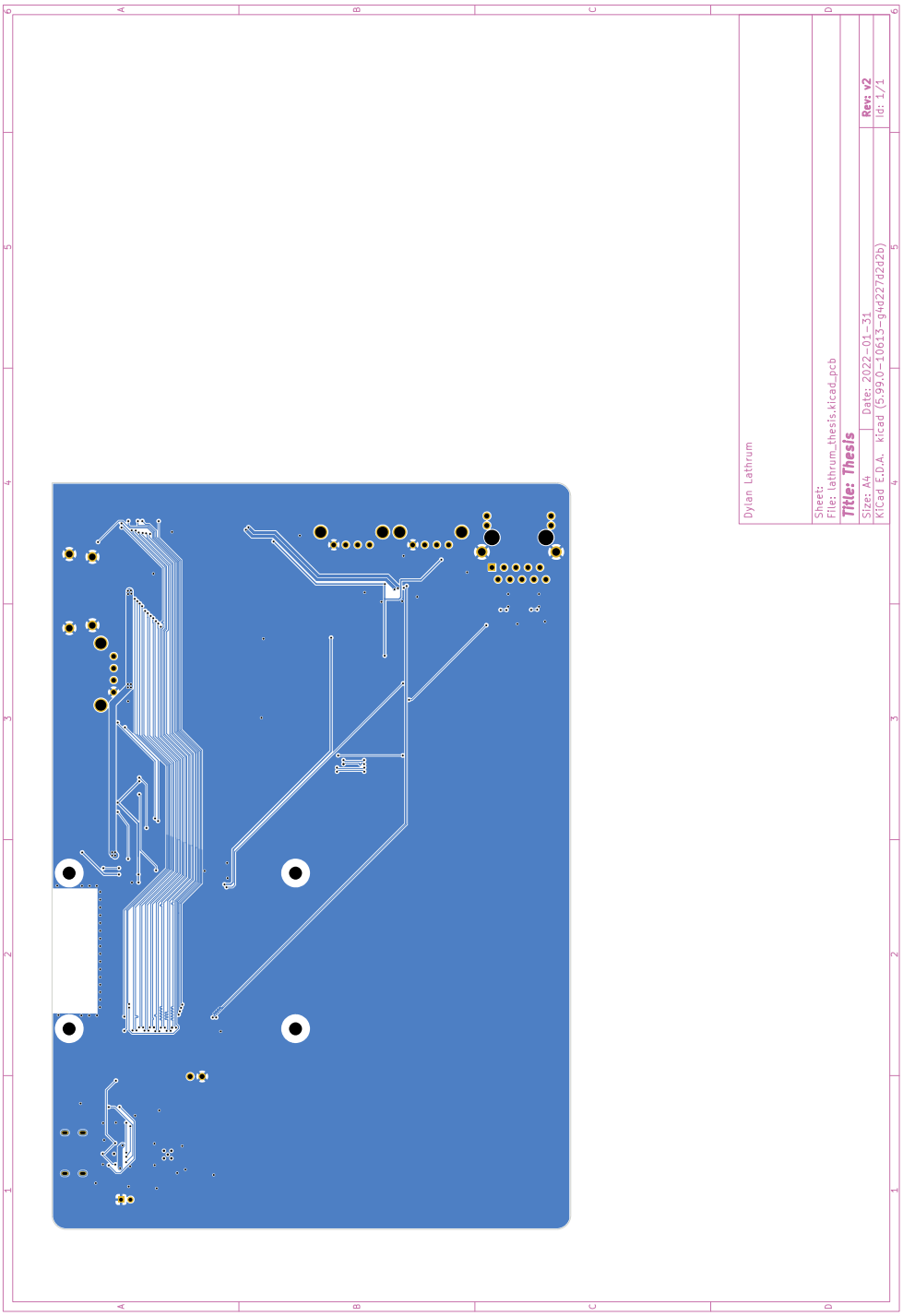


FIGURE A.6: Physical board layout for the back side of the PCB

Appendix B

Bill of Materials

Reference Designator	Count	Manufacturer's Number	Value
B1	1	B2B-PH-K-S(LF)(SN)	JST 2-PH
C1 C3 C6 C8 C10 C11 C19 C20 C21 C22	10	CL31B106MOHNNNE	10uF
C15 C16	2	CL31B475KAHNFNE	4.7uF
C17 C18	2	C1206C270K5GAC7800	27pF
C2 C7 C9 C12 C13 C14	6	CL31B104KBCNFNC	0.1uF
C4	1	CL31B225KOHNNNE	2.2uF
C5 C23	2	GRM31CR60G107ME39L	100uF
D1 D2 D6	3	156120RS75300	Red
D3 D5	2	150120GS75000	Green
D4	1	150120AS75000	Orange
J1	1	USB4125-GF-A	USB4125-GF-A_REVA2
J2	1	PREC002SAAN-RC	PREC002SAAN-RC
J3	1	SS-53000-001	SS-53000-001
J4 J10	2	FFC3B07-20-T	FFC3B07-20-T
J5	1	5033981892	Micro_SD_Card_Det
J6	1	ARJM11D7-502-AB-EW2	ARJM11D7-502-AB-EW2
J7 J8 J9	3	SS-52100-001	SS-52100-001
L1	1	NRS5030T6R8MMGJ	6.8uH
Module1	1	CM4104000	ComputeModule4-CM4
Q1	1	MMUN2133LT1G	MMUN2133LT1G
R1 R7	2	RC1206FR-071M87L	1.87M
R10	1	RC1206FR-0715KL	15K

R13	1	RC1206FR-07270KL	270K
R18 R24	2	RC1206FR-0712KL	12K 1%
R2	1	ERJ-8ENF3403V	340K
R20 R21	2	RC1206FR-07470RL	470R
R22 R23	2	RC1206FR-0736KL	36K 1%
R25	1	RC1206FR-0715KL	15K 1%
R3 R4	2	RMCF1206FT5K10	5.1K
R5 R8	2	RC1206FR-07200KL	200K
R6 R12 R14 R15 R16 R17 R19	7	RC1206FR-071KL	1K
R9 R11	2	RMCF1206FT100K	100K
U1	1	TPS61090RSAR	TPS61090RSAR
U2	1	MCP73871-2CCI/ML	MCP73871-2CCI_ML
U3	1	74LVC1G07SE-7	74LVC1G07
U4	1	RT9742VGJ5	RT9742VGJ5
U5	1	RT9742ENGJ5	RT9742ENGJ5
U6 U7	2	TPD4EUSB30DQAR	TPD4EUSB30
U8	1	USB2504A-JT	USB2504A-JT
U9	1	AP22653W6-7	AP22653W6-7
Y1	1	FC4SDCBMF24.0-T1	24MHz

TABLE B.1: Bill of Materials for a single PCB Board.

Appendix C

Open Source Repository

The entirety of this project, including hardware schematics and software code, is available online on GitHub at: https://github.com/Dylancyclone/thesis.

Bibliography

[1] *AnalogIP0 AnalogIP1 pins on CM4*. URL: https://forums.raspberrypi.com/viewtopic.php?t=289344#p1748915.

[2] *Blender 3.0*. 2021. URL: https://cloud.blender.org/p/gallery/617933e9b7b35ce1e1c01066.

[3] Eric Brown. *Nvidia Shield game console runs Android on tegra 4*. 2013. URL: https://linuxgizmos.com/nvidia-shield-android-game-console-shipping-soon/.

[4] Eric Brown. *Nvidia Shield: Shipped, praised, critiqued, dissected*. 2013. URL: https://linuxgizmos.com/nvidia-shield-reviews-and-teardown/.

[5] Ronald Bultje. "WebM and the New VP9 Open Video Codec". In: *Google I/O 2013*. 2013. URL: https://www.youtube.com/watch?v=K6JshvblIcM.

[6] Steve Burbeck. "Complexity and the Evolution of Computing : Biological Principles for Managing Evolving Systems". In: *Complexity and the Evolution of Computing : Biological Principles for Managing Evolving Systems*. 2007. URL: https://evolutionofcomputing.org/Complexity%20and%20Evolution%20of%20Computing%20v2.pdf.

[7] Bryan S Burgin. *RelativeMouseMode*. 2013. URL: https://social.msdn.microsoft.com/Forums/en-US/ae516842-1e3d-4943-8144-1b225e74684e/relativemousemode.

[8] *Chromoting Build Instructions*. 2020. URL: https://chromium.googlesource.com/chromium/src/+/refs/tags/82.0.4058.2/docs/old_chromoting_build_instructions.md.

[9] *Covid-19 announcements archive*. URL: https://eoss.asu.edu/health/announcements/coronavirus/announcements-archive.

[10] Chris Daniel. *Shield TV brings the smart home to a TV near you: Nvidia blog*. 2017. URL: https://blogs.nvidia.com/blog/2017/01/04/shield-ai-home/.

[11] *Developing on remote machines using SSH and Visual Studio Code*. 2021. URL: https://code.visualstudio.com/docs/remote/ssh.

[12] *DigiKey Electronics Home*. URL: https://www.digikey.com/.

[13] Djjajah. *Clientron U700*. 2008. URL: https://commons.wikimedia.org/wiki/File:ClientronU700.jpg.

[14] *Dual Low Dropout Voltage Regulator*. MCP73871-2CCI/ML-ND. Rev. E. Microchip Technology. Apr. 2019. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/MCP73871-Data-Sheet-20002090E.pdf.

[15] *Everything about raspberry pi zero 2 W*. 2021. URL: https://picockpit.com/raspberry-pi/everything-about-raspberry-pi-zero-2-w/.

[16] *Google Chromoting - remote desktop management*. URL: https://www.miniorange.com/google-chromoting-remote-desktop-management.

[17] Gundarev. *Remote Desktop Protocol Bandwidth Requirements*. 2021. URL: https://docs.microsoft.com/en-us/azure/virtual-desktop/rdp-bandwidth.

[18] *HDMI vs DSI, Pros and Cons?* 2018. URL: https://raspberrypi.stackexchange.com/questions/88075/hdmi-vs-dsi-pros-and-cons.

[19] *How GameStream Works*. URL: https://www.nvidia.com/en-us/support/gamestream/how-gamestream-works/.

[20] *How long will the chip shortage last?: J.P. Morgan Research*. 2021. URL: https://www.jpmorgan.com/insights/research/supply-chain-chip-shortage.

[21] *How to make the PCF85063AT work with CM4?* 2020. URL: https://forums.raspberrypi.com/viewtopic.php?t=293632.

[22] J.O. Kephart and D.M. Chess. "The vision of autonomic computing". In: *Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1160055.

[23] Dylan Lathrum. *ASU Barrett Undergraduate Thesis*. https://github.com/Dylancyclone/thesis. 2022.

[24] Gad Levanon. "Remote work: The biggest legacy of covid-19". In: *Forbes* (2020). URL: https://www.forbes.com/sites/gadlevanon/2020/11/23/remote-work-the-biggest-legacy-of-covid-19/?sh=699b0767f590.

[25] *manjaro-arm-tools*. URL: https://gitlab.manjaro.org/manjaro-arm/applications/manjaro-arm-tools/-/tree/master/.

[26] L. Meyer. "5 Reasons Schools Still Need Desktop Computers: Despite the Growth of Mobile Learning, Desktops Still Play Important Roles in the 21st Century Classroom". In: *T.H.E. Journal Technological Horizons in Education* 41 (2014), p. 20.

[27] *Moonlight*. URL: https://moonlight-stream.org/.

[28] *Mouser Electronics - electronic components distributor*. URL: https://www.mouser.com/.

[29] *PCB prototype & PCB fabrication manufacturer*. URL: https://jlcpcb.com/.

[30] Raspberry Pi. *Compute Module 4*. URL: https://www.raspberrypi.com/products/compute-module-4/.

[31] Raspberry Pi. *Compute Module 4 IO Board*. URL: https://www.raspberrypi.com/products/compute-module-4-io-board/.

[32] *Raspberry pi documentation*. URL: https://www.raspberrypi.com/documentation/computers/compute-module.html.

[33] *Solid notebook demand in consumer and education marks 2021 Q1 as the fourth consecutive quarter of double-digit growth in EMEA PC market, says IDC*. 2021. URL: https://www.idc.com/getdoc.jsp?containerId=prEUR147632021.

[34] Maxim Tamarov. *Chip shortage driving up PC prices, Wait Times*. 2021. URL: https://www.techtarget.com/searchmobilecomputing/news/252499402/Chip-shortage-driving-up-PC-prices-wait-times.

[35] *The VNC family of Remote Control Application*. URL: http://ipinfo.info/html/vnc_remote_control.php.

[36] *Time.is*. URL: https://time.is/.

[37] *Understanding remote desktop protocol (RDP) - windows server*. URL: https://docs.microsoft.com/en-us/troubleshoot/windows-server/remote/understanding-remote-desktop-protocol.

[38] Linus Upson and Caesar Sengupta. *Next step in the chrome OS journey*. 2012. URL: https://blog.google/products/chromebooks/next-step-in-chrome-os-journey/.

[39] Tatu Ylonen. *RFC 4251 - the secure shell (SSH) protocol architecture*. Ed. by ChrisEditor Lonvick. 2006. URL: https://datatracker.ietf.org/doc/html/rfc4251.