FLINDERS UNIVERSITY

HONOURS THESIS

Security through Simplicity: Implementing Secure Compartments for the MEGA65

Author:
Jayden GRIGG

Supervisor:
Dr. Paul
GARDNER-STEPHEN

A thesis submitted in fulfilment of the requirements
for the degree of Bachelour of Engineering(Electronics)(Honours)

October 15, 2018

Declaration of Authorship

I, Jayden GRIGG, declare that this thesis titled, "Security through Simplicity: Implementing Secure Compartments for the MEGA65" and the work presented in it are my own. I confirm that:

- This work was done wholly while in candidature for a degree of Bachelour of Engineering(Electronics)(Honours).
- This document is in accordance with the plagiarism policy of Flinders 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:

_____

Date:

_____

"Strange how paranoia can link up with reality now and then."

Phillip K. Dick

FLINDERS UNIVERSITY

Abstract

College of Science and Engineering

Bachelour of Engineering(Electronics)(Honours)

Security through Simplicity: Implementing Secure Compartments for the MEGA65

by Jayden GRIGG

Modern computers have become too complex to fully understand, let alone to fully verify. Therefore it has ceased to be rational to trust their security and integrity. With billions of transistors and billions of lines of code, even the large corporations who design and manufacture computer processors and write operating systems and applications are unable to verify their security, as evidenced by hardware vulnerabilities in processors such as Spectre and MELTDOWN, and the endless stream of software security updates. In short, computers are vulnerable to attacks from the surrounding environment, and defence strategies are heuristic and reactionary, placing the integrity of communications and commerce at risk. However this has not always been the case: The computers of the 1980s and early 1990s were much simpler, with software measured in kilobytes, not giga-bytes, and with processors with thousands, not billions of transistors. This thesis asks the question as to whether it is possible to create modern usable computing devices based on architectures so simple that they can be adequately verified by a determined user, and in that context, what architectural extensions might be used to strengthen the security of simple architectures, without making the resulting systems unnecessarily complex. A proof-of-concept system is presented based on the MEGA65 retro-computer, to which a robust secure compartmentalisation facility is added, together with a use-controlled out-of-band memory inspector, that ensures the user has complete sovereignty over the system, so that even were it compromised, that the compromise can be discovered and dealt with.

Acknowledgements

I would like to than Paul Gardner-Stephen for providing me with this very unique opportunity to work on the MEGA65 project as well as for his support and advice on many of the tasks that I was required to perform. I would also like to thank Tim Kirby for his previous work on the MEGA65 project, without which, the secure compartment implementation would have been impossible. In addition, I would like the thank my fellow students Lachlan McDonald and Tanguy Nodet, who helped me and worked along side me on the MEGA65.

Contents

List of Figures

Chapter 1

Introduction

1.1 Paradise Lost

The earliest epochs of the digital era were based on a sense of control and security: A computational device could be created, and could be expected to behave as expected, and not to be subverted at a distance by invisible hands. This began with the first computers that were the size of a small building, and continued until the advent of personal computers that were capable of easy exchange of data between them, initially by floppy disk, by modem or other forms of networking [4]. Essentially, whenever a means of communications was established between computers, worms, virii and other pathological software soon followed. The pathological software was initially simple in nature, and could easily be detected and removed, and into the 1990s it was not uncommon to be able to obtain anti-virus software that could detect and remove all extant pathological software from a system. Figure 1.1 shows an example advertisement from 1991, which explains that

if users suspect the existence of a new virus, they can call a "24-hour virus newsline", betraying how relatively sedate the development and spread of computer virii was at that time.

The advent of ubiquitous internet access marked a dramatic shift: Where as previously viruses were mostly reliant on the movement of physical artefacts, such as floppy disks, the internet allowed for the infection of systems anywhere in the world, without coming into physical contact. That is, it introduced a new and highly effective transmission mode, that was all the more insidious in that in the floppy-era you could determine whether your computer might be at risk, based on whether it had been in contact through the insertion of a floppy disk. This allowed for individuals to effectively manage the sanitation of their computers in a rational and understandable manner. The Internet allowed for invisible infection at a distance, a problem which persists to the present day, and which has created the conditions for the wide range of malware, viruses, trojan software, ransomware, rootkits, backdoors and other pathological software to develop and spread.

This evolution of the pathogenic material to take advantage of the available modes of delivery is directly mirrored in the world of biology [3]. Indeed, what we see with computers now, is that the defences against these infectious threats have come to resemble the immune system of complex animals: A variety of intrinsic protections, heuristic measures and the development of immunity only after infection have become accepted norms for computer security. However, this is a dangerous situation, just as animals are made sick or die or are subverted by biological pathogens, e.g., Figure 1.2), the compromise of our digital systems has significant effects for individuals and society. And just as it is impossible to quickly appraise whether an animal is free of all disease, it has become impossible to determine authoritatively whether a computer is free of all digital pathogens.

Not only are there connections between pathogens and computer viruses in general, but eerily enough, there are some computer viruses that behave almost identically to a biological counterpart. The Intel management engine is a small CPU that has been in almost all Intel chips for the past 18 years. Late in 2017, a proof of concept subversion on the Intel management engine was done by Mark Ermolov and Maxim Goryachy [40]. In this subversion, the engine was proved to be directly controllable with out being visible to the user, meaning that a hacker could potentially cause your machine to attack itself. This is the equivalent of an auto-immune disease in the biological world, as the greater organism, whether that is man or machine, is being

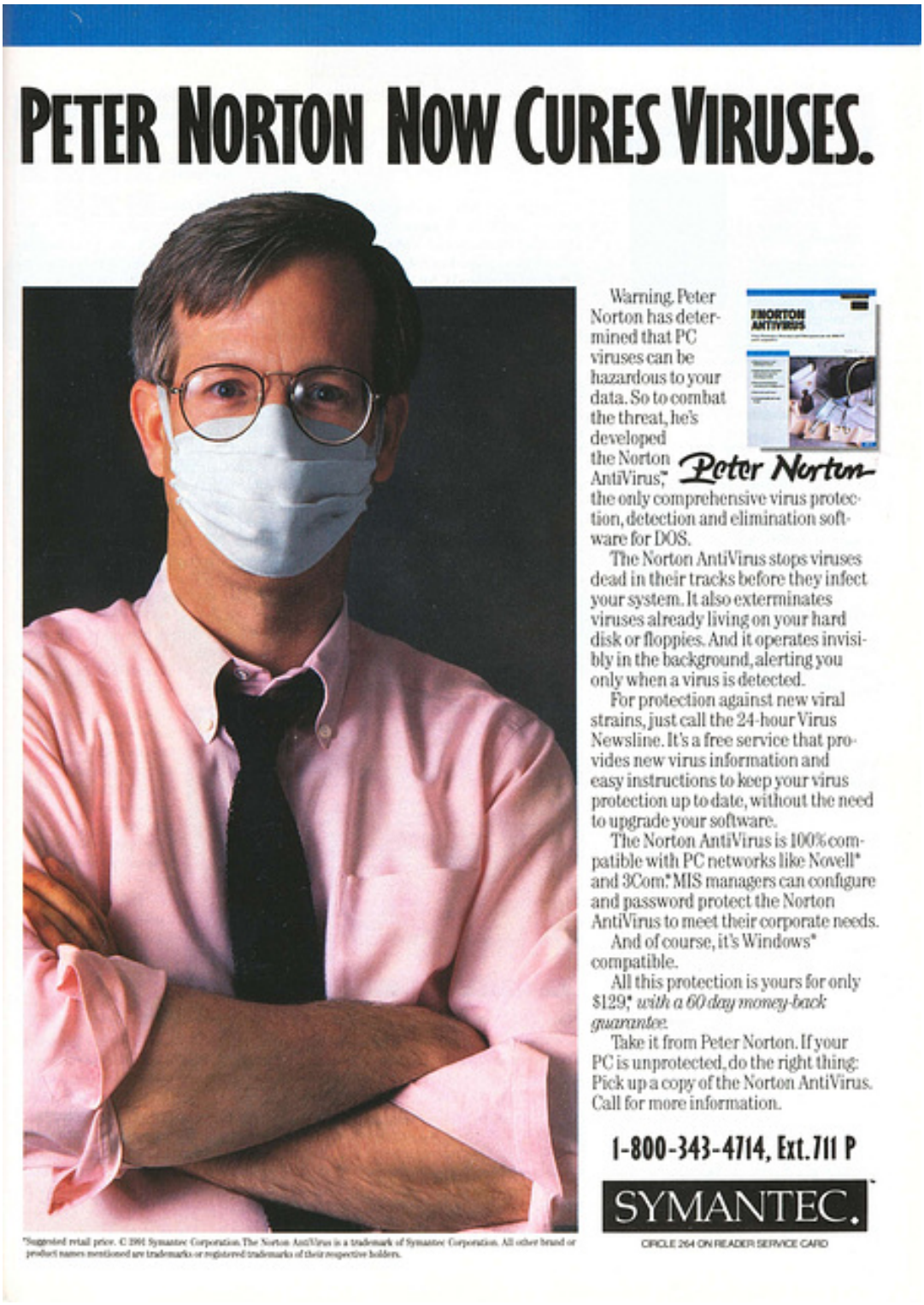


FIGURE 1.1: Early advertisement for Norton Anti-Virus for DOS. New viruses spread slowly at the time, as they could only be spread by floppy disk. As a result, users could be reasonably directed to call a telephone number if they suspected that they had a new virus on their computer. Source: http://www.ipernity.com/doc/78280/6670939.

FIGURE 1.2: *Ophiocordyceps unilateralis* is perhaps better known as the "zombie ant fungus". When it infects an ant, it modifies the ant's behaviour, so that it leaves its ground-based nest-mates, climbs a tree and finds a leaf, and bites into the underside of the leaf so that it hangs upside down. Several days later the fungus emerges from the head of the ant in a fruiting body that showers its nest-mates with spores, thus spreading quickly through entire colonies. Source: https://upload.wikimedia.org/wikipedia/commons/8/85/Ophiocordyceps_unilateralis.png

attacked by part of itself. The parallels do not stop here however, HIV/AIDS are immune system weakening diseases. These diseases are often not the cause of death themselves, but are often contributers [39]. Much like these diseases, the "BadBIOS" phenomenon does not directly cause the death of a computer. This virus instead secretly and with out user authorisation, opens the communication channels of any computer it infects [10]. The similarities between "BadBIOS" and HIV/AIDS does not stop there, both are extremely resistant to treatment to the point where they are almost permanent. In the case of "BadBIOS" this infects the main program that controls all the actions of the computer. Because all other operations occur on the computer through this master program, there is no way to remove all of the infection which could be lurking anywhere [10].

Were this not depressing enough, the problem is complicated by the fact that computers have, like biological organisms, become so complex, that it is impossible to construct them in such a way that they can effectively resist being subverted. The interactions in modern software are now so rich – like the interactions of proteins, genes and other elements that make up the control machinery of living cells – that it is impossible to fully understand them in all but the most trivial of circumstances.

This problem has been compounded by Moore's Law [26], which has made it cheaper to create ever more complex computers, and by human psychology which have created an emphasis on the number of features or functions of a given piece of software, rather than on its correctness and security. Thus software has exploded in size from kilo-bytes in the early 1980s to tens of giga-bytes in the current era. That is, software is now often millions of times larger and more complex than it once was, to the point where verification of correctness has become effectively impossible.

As bad as this situation is for software, it is even worse when hardware systems are considered, and the central processing units (CPUs) of computers in particular. CPUs have also grown in complexity from thousands of transistors to tens of billions of transistors over the same time period. However, unlike a defect in software, a defect in a CPU cannot necessarily be corrected without physical replacement.

The fact that the Spectre and MELTDOWN vulnerabilities existed in CPUs for almost 20 years before being discovered [25] is strong evidence that we have long since passed the point where even a well-resourced company like Intel or AMD can ensure the correct operation of a CPU. If these companies with many thousands of verification engineers cannot achieve it, what hope

do individuals and small organisations have for verifying the correct operation of their computing devices?

What is clear is that the problem is one of complexity: A single transistor can be easily verified. The simple early CPU designs like the 6502, that consisted of only a few thousand transistors can also be easily verified. But modern complex CPU designs cannot. There must exist somewhere a tipping point where verification ceases to be realistic for a determined user. Beyond that point security is merely a hopeful dream, because we can no longer prove correctness and immunity to the challenges that it might face, just as avoiding the common cold is also no more than a hopeful dream, and certainly nothing that can be guaranteed if we are to remain in communications with the outside world.

We are therefore forced to accept the unpalatable truth that we are no more sovereign over our computing devices than our bodies are over the biological world that surrounds us: We have been cast from the Eden of security for eating the forbidden fruit of ever increasing complexity and functionality.

The question is whether we can go back, to create computing devices that are sufficiently simple as to be securable, and yet remain sufficiently useful in practice. This is the motivating question for this thesis.

1.2 Research Questions

What are the missing or non-functional sub-systems that the MEGA65 requires to be implemented?
Which will be answered in the form of a survey of the current sub-systems of the MEGA65, and a survey of the sub-systems required to create a functional prototype.

How can these sub-systems be implemented?
Which will be answered by the creation of plans for the implementing of the missing or incomplete sub-systems.

How can the simplicity, understandability and hence, the security of these sub-systems be maximised?
Which will be answered by considering each sub-system, qualitatively appraising its simplicity, understandability and security, and where appropriate, making well researched recommendations for refining those components to improve one or more of these axes, and time permitting, acting on those recommendations.

How can the complete MEGA65 architecture be physically prototyped on the bench?
Which will be answered by examination of the current partial bench prototype and comparing it with the sub-systems identified through the other research questions, and designing and realising a complete bench prototype. This will occur through coordination with Mr. Lachlan McDonald, who is undertaking the designing of the PCB for the MEGA65 smart-phone device.

How can the secure compartmentalisation's architecture planned for the MEGA65 be realised?
Which will be answered by considering this architecture and the current state of the MEGA65 system to derive and execute a method for implementing a secure compartmentalisation architecture.

Overall the success of the project will be measured against the creation of a functioning bench prototype device that, through the architecture, implements the secure and understandable compartmentalisation of hardware to the point of demonstrability.

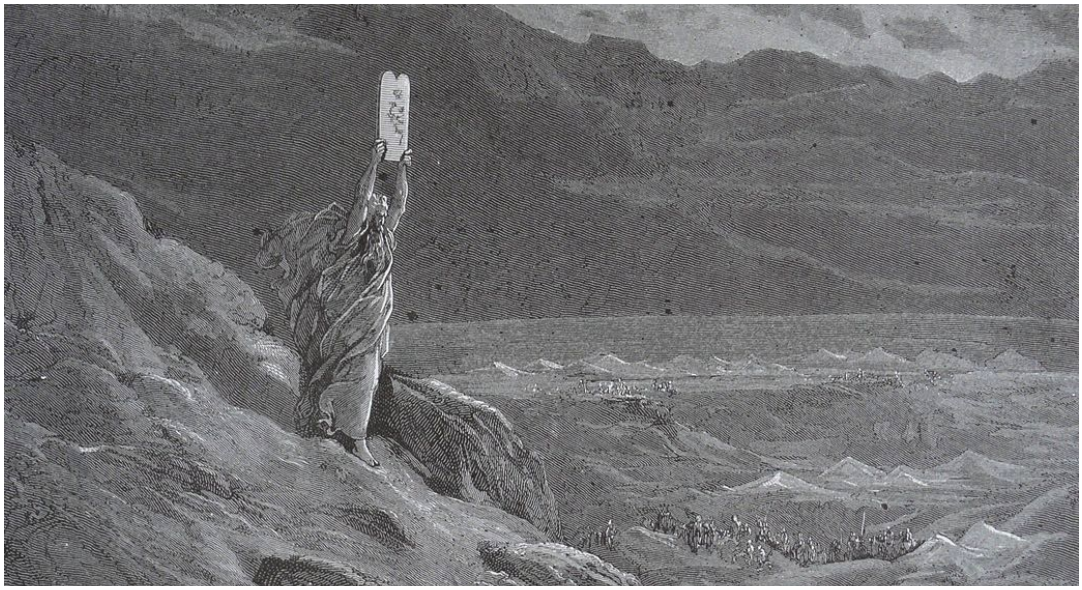
1.3 Thesis Layout

Following this chapter, "Chapter 1 : Introduction", is the chapter, "Chapter 2 : Literature Review". In which relevant background information regarding complexity and cyber security, mobile devices, isolative security and the MEGA65 project will be given. After this, "Chapter 3 : Methods and Materials" will go on to describe how this project was undertaken and what tools were used. This will be immediately followed by "Chapter 4 : Project Set-up". In this chapter, details about the issues faced immediately after joining the project will be made clear. Following this is "Chapter 5 : Matrix Mode Corrections", in this chapter the issues encountered with the matrix mode and their fixes will be made clear. "Chapter 6 : Secure Compartmentalisation" will follow, in which an overview of how the secure containers in the phone were implemented will be given. This will be followed by "Chapter 7 : Results and Discussion" where the challenges, what was achieved and research questions will be talked about. In addition to this, the future works for the secure compartment and matrix mode portions of the project will be outlined. Finally, "Chapter 8 : Conclusion" will discuss the various details about the findings of this

document, as well as answers to the research questions outlined in section 1.2.

Chapter 2

Literature Review



2.1 Overview

Since the introduction of general-purpose mobile operating systems, such as Symbian, Android, Windows 10 and iOS, and especially throughout the last decade, mobile phones have evolved dramatically [2]. The introduction of feature-rich and complex operating system has not only brought the benefits of a computer, but also the risks of one too [27]. Since phones have become computerised people have become more trusting of what data they can store on their phone, such as location, bank details, etc [7]. With private data increasingly being stored on phones, it becomes necessary for mobile security to receive a larger amount of attention. This literature review explores this issue, as follows:

- Section 2.2 explores the interrelationship between complexity and security.
- Section 2.3 documents a wide variety of the security threats facing modern computers, with a particular emphasis on mobile devices.

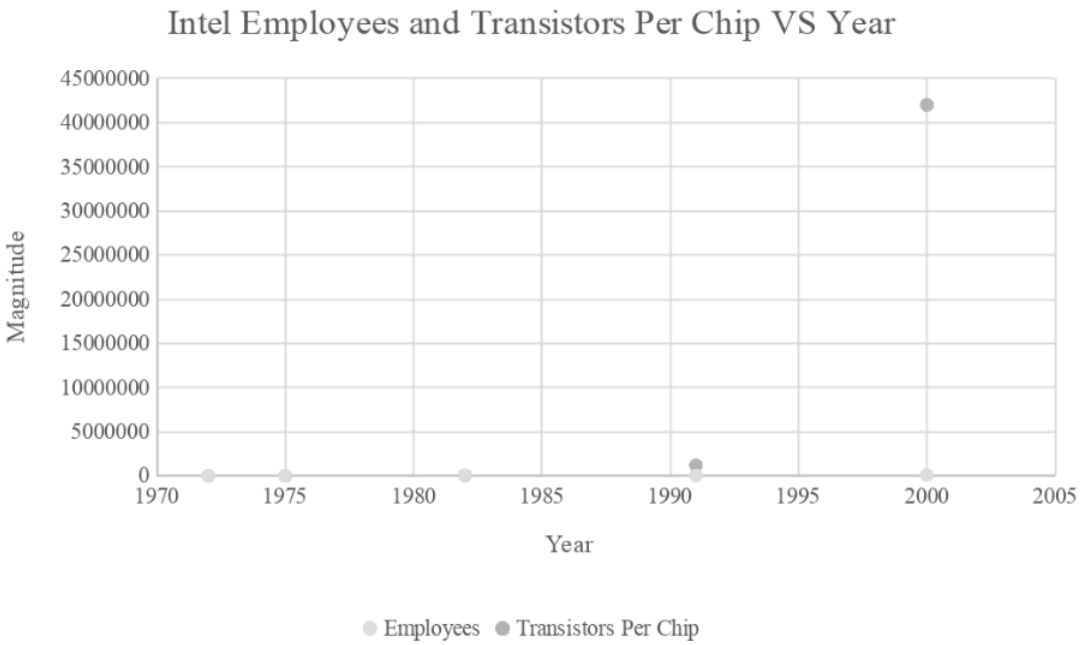


FIGURE 2.1: Graph of Intel's CPU transistor count vs the amount of Intel employees.
[15][16][17][14][13][28][1][30]

- Section 2.4 relates the application of isolation and compartmentalisation to improve security.
- Section 2.5 documents the history and status of the MEGA65 project, including the historical Commodore 64 platform on which it is based, as well as the recent work towards introducing compartmentalisation and related security features to the platform.

2.2 Complexity and Security

One of the major issues with the current state of security is that the complexity involved is far beyond what any one person can comprehend. Intel embodies the complexity issue perfectly when examining the number of employees vs the number of transistors on one of their CPUs.

From 1972 to 2000 Intel's workforce grew by 86 times, comparing this growth with the transistor growth of 1200 times over the same period, a huge disparity can be seen. If every employee at Intel was dedicated to testing transistors, in 1972 each employee needed to test 3.5 transistors, compare this to the 487 they need to test now. This clearly demonstrates the impossibility for, not only one person, but for all of Intel to successfully verify that their

CPU is working as intended. With modern computers unable to be understood by a single person, they are inherently not to be trusted [9]. Not only are they not to be trusted, but they can never be trusted, because while they are not understood by a single person there is always the possibility of interactions in the CPU that remain undiscovered [10].
As systems become more complex, they too see more issues arise within them. A key property of this is the amount of failure modes that are present in the system. Larger, more complex, systems naturally have more security vulnerabilities as they have more interactions, or opportunities for failure [33]. These larger systems also require more time and effort to test. More worryingly is that for the end user, this complexity aids malicious attempts [18]. For the same reasons that testing is difficult, so too is finding indicators of a security breach, or even malicious activity after a breach has been detected [35].
To combat this increase in security flaws there are a few methods that can be used. One such method is to patch the security flaws with work-arounds. As these work-arounds were not originally designed as part of the system, there is a possibility that they can make the entire system less secure overall.
This has been beautifully demonstrated by the recent exploits found in speculative execution processors. This exploit, named Meltdown, was a massive cause of panic as it allowed privileged information to be read at megabytes a second. The Windows 7 January 2018 patch for the exploit resulted in permissions for the user being set erroneously. This allowed the user to read memory that was normally only accessible by the processor [8].
The other method of reducing security flaws is to reduce the complexity of the system. Less interactions allows more time dedicated to ensuring each one works correctly. In addition to this, simpler systems allow for users to better understand them, and thus are more easily verified as secure. This was listed as one of the key aspects of a trusted computer system by the Department of Defence as it is only then that understandable and maintainable protection is achieved [6].

2.3 Mobile Security Threats

When analysing current mobile (smart) phones there are multiple avenues under which data can be maliciously obtained, or normal service can be disrupted. These avenues can be categorised under the following labels:

- Mobile Hardware

- Mobile Software
- Mobile Networks
- Physical Access
- Mobile Enterprise

2.3.1 Mobile Hardware

Mobile hardware refers to the various sensors and physical devices built into the phone. By accessing the hardware directly on the phone, most software security provided by the operating system is bypassed, allowing malicious users to access data provided by cameras, microphones, etc. This form of attack is particularly difficult to protect against in some cases due to the advent of users rooting or jail-breaking their phones [21]. Jail-breaking/rooting is where the user deliberately uses exploits vulnerabilities to gain more control over the phone's systems.
Currently, the best way to defend against mobile hardware exploitation is to ensure that the operating system is up to date and that patches for publicly known exploits are installed. This does not prevent mobile hardware exploits but rather makes them nontrivial to perform [12].
Even with all the updates and patches installed, there still exists improvements that could be made to further reduce hardware exploits. As most hardware on a phone does not have an indicator for when it is active, it is difficult to determine its status; implementing this in hardware would allow a user to tell decisively whether hardware was on or off [19].

2.3.2 Mobile Software

Mobile software refers mainly to the applications running on the phone, but it also refers to the background processes that run while the user isn't using their phone. While these programs give the user more control over their phone, they are also a potential source for coding errors.
Through these errors such as, files being stored in an unprotected location or files being stored in an insecure format, it is possible to leak sensitive data [5]. Not only is it possible to read stored data, but through an insecure network or a vulnerable third-party software library, it is possible to read and alter all data being sent to and from the phone.
To protect against vulnerable software there are limited options available.

Keeping the operating system up to date is the best solution as more security monitoring helps catch exploits [12]. The process of application analysis involves two different methods, static analysis and dynamic analysis. In static analysis the source code for an application is evaluated without running it to detect possible exploits. This method relies on the source code being available, which isn't always the case, and, if it is computer assisted, can provide false positives [36]. To a lesser extent, static analysis can be performed on compiled code, but it is limited to program interface, compiler optimisations and software library checking. Dynamic analysis is like source code static analysis with the exception that it is done with the code running. This allows behaviour that is not apparent from source code examination to be identified [19].

In addition to coding errors that allow third parties to exploit security vulnerabilities, some applications are developed intentionally for malicious activity. These less than reputable applications often exploit underlying vulnerabilities in the operating system or trick the user into giving the application privileges that exceed those needed for proper function [38].

Malicious applications can breach phone security in many ways to extort money from the user or gain access to protected data. Ransomware is the name given to an application that encrypts all the user's data and demands payment to decrypt it. This form of extortion poses extreme risk when dealing with cooperate data due to the loss in productivity. The unauthorised access of user data can be done through many methods such as jail-breaking the phone so that hardware can be exploited, subverting login details, sharing data with other applications (such as Dropbox) and logging private data [19].

Malicious applications are difficult to protect against. In almost all situations they require some form of active security that is user reliant to effectively protect against them. In addition to understanding the threats present and keeping all software up to date, users can:

- Use dynamic analysis in a safe environment
- Isolate the application from sensitive data
- Use authentication protocols

These methods would allow for adequate protection against malicious applications or applications used for a malicious purpose, but there is still more that can be done. Better application development practices and regulations would

prevent many of the issues with software exploitation. This is unrealistic though as most applications are so complex there will always be a missed bug. More transparent applications are the solution to this as it would allow users to not only verify the application but improve upon it [19].

2.3.3 Mobile Networks

A mobile network is the interconnection between all phones that allows for phone calls, messages and internet access to the mobile device. These connections are facilitated by the numerous radios and modem. The mobile network can be broken down into three major sections, the radio access network, the core network and the external services network.
The radio access network facilitates the connection between the mobile device and the service provider for the phone. This network is used to connect the mobile device to the telephone tower which then leads to the core network. The core network is responsible for tracking billing, signal routing and connecting the mobile device to the external services network, such as the internet or mobile devices from a different carrier [19].
Radio access networks are susceptible to three basic types of exploits, denial of service attacks, eavesdropping and device tracking. Denial of service attacks is used as a large umbrella term for all the attacks that can block or hinder normal mobile phone operation. These attacks can be conducted by filling up all available radio access slots on a telephone tower, impersonating a telephone tower to intercept emergency calls or flooding the area with radio waves to obscure legitimate calls [32]. Eavesdropping, as the name suggests, involves the gaining access to data being transmitted between the telephone tower and mobile device. This is possible because it is not required that data between the mobile device and the telephone tower be encrypted, in the case that end to end encryption is not used, all the user's data is free to access by those that can [19]. For superior data transmission a shorter radio access network is desired, this bring forth a need for mobile device locational services, so the closest radio tower can be used in data transfer. This can be exploited to track a mobile device and by extension, a user's location [32].
The core network, once subverted, offers attackers many avenues of attack. Most mobile networks use some form of management system in order to control the entire network. This provides the ability to intercept or block phone calls and texts, as well as denying service to users were it subverted [19].

External networks are how a phone connects to the internet among other networks. This brings all the benefits, but also all the dangers of an internet connection.
To defend against data interception and theft, end to end encryption must be used. This negates any chance of the radio access network and core network leaking data, as the leaked data would still be encrypted. Denial of service attacks can be mitigated by alternative methods of data transmission. Though it is highly unlikely that an entire carrier will have it's services denied, using an alternative method would be able to negate any and all effects of such an attack. Unfortunately, mobile device tracking cannot be defended against due to it being so intertwined with the mobile network. Even bouncing a connection across a privately-owned network would not result in much, as it can be traced by a skilled attacker [19].

2.3.4 Physical Access

While mobile phones are almost always on person all the time, there are certain situations in which contact must be broken. These instances occur during certain security checks, like police or airport searches, or during charging and communication sensitive operations. During these times it would be possible to access data from the phone or perform other malicious activity.
One avenue of attack is the combined charging/data port of the phone. By plugging the phone into a PC, it is possible to abuse vulnerabilities to gain access to data. This is not limited to PCs however, by modifying a charger with a small PC board it is possible to execute complex attacks on the device whenever the phone is charging. If done right, this attack could also work in the reverse, infecting a host computer that the phone is connected to [19].
Another avenue is the phone itself, NowSecure released a report in 2016 that stated 43% of mobile users do not use a pass-code, PIN, or pattern lock on their device [29]. This, coupled with cached memory and browser cookies could result in banking details, among other things being stolen.

The best defence against physical access attacks is to use screen locks and exclusively use personal charging equipment. These negate the threat of physical access attacks and data port attacks.

Despite the ease of ensuring physical security, more needs to be done to encourage users to use this security. Currently, most phones use authentication

methods that don't fully capitalise on all the mobile sensors available. While using sensors like fingerprint scanners and facial recognition is emerging in mobile phones, there exists sensors like the capacitive sensor, gyroscope and accelerometer that are still unused in lock screen technology [19].

2.3.5 Mobile Enterprise

When mobile devices are integrated into business there comes a need for servers, processes and systems that manage these devices. This enterprise, such as a company application, can serve as an infection source, should the enterprise itself become infected. To counteract these flaws mobile device managers for enterprise purposes have been developed. These managers enforce security policies, remote access, remote wiping, etc. for the device. Because these device managers have a higher level of privilege than the user, exploiting the device manager to gain access to the entire enterprise is a serious threat, as doing so would allow infection of all devices connected to the enterprise. Once compromised, it is possible for all data within the enterprise to be stolen, manipulated or for all services to be blocked completely. These attacks can be mitigated using comprehensive authentication protocols to prevent impersonation, protected execution environments and network monitoring. While authentication prevents most forms of attack, it is still subject to authentication information being discovered, which is why the execution environment is used to contain malicious activity and network monitoring is used to look for such activity [19].

2.4 Security Through Isolation

One approach to securing a computer has been to have sections of it "air gapped", such that infection in one area cannot affect other areas [23].

One example of an operating system built on this principle is the Qubes OS, which has gone about this through compartmentalisation. By isolating things, such as untrusted websites, in their own qube it is possible stop any attempt to compromise the whole system as they are contained in that qube. These qubes are not limited to software, they work with hardware too, meaning that USB controllers and network cards are secured in their own qubes. This reduces the effectiveness of using hardware as a security exploit

[31].

Isolative computing can be emulated by hosting multiple virtual machines on one host machine, thus recreating the Qubes OS concept. This comes one major disadvantage over using Qubes OS however, once the host OS is subverted all the virtual machines are subverted too [11]. As Qubes OS uses a hyper-visor to manage all the qubes it is inherently more secure than virtual machine recreation. This is due to the increased difficulty of subverting the hyper-visor as compared to subverting an operating system.

Another method for air gapping a computer is to have it physically isolated from external connections. By removing the computer from external networks almost all attack vectors are removed too, thus ensuring the computer is safe. This form of protection is mainly used in critical systems, as by isolating a computer a lot of the functionality of it, such as web browsing, is removed [41].

While air gapping computers appears to make data exfiltration impossible, there are a few ways in which this can happen. The simplest method is via physical access and a USB or USB device. The best know device is the USB Rubber Ducky, this device can be used to imitate a keyboard inputs via a script loaded onto the device [34]. This allows the rubber ducky full control over the host device and can even be used to install malicious software [34]. Another method of circumventing an air gap is to use high frequency sound from a speaker to transmit and infect air gapped computers [37] "AirHopper" is another method for data exfiltration, instead of using sound, it uses radio waves generated by cables from the computer [20]. Even when Faraday shielded to prevent these sound and electromagnetic signals from escaping, it is still possible to transmit data from the computer by using the CPU. The CPU can be used to generate low frequency magnetic waves by altering the load on specific cores. These magnetic waves more readily penetrate Faraday shielding, allowing the transmission of data from the air gapped computer [22].

2.5 The MEGA65 Project

The MEGA65 project was born from one Paul Gardner-Stephen in 2014. The scope of the project was to create an accelerator for the commodore 64. This

changed when the Museum of Electronic Games and Art sponsored the project and connected him with various other experts [24]. The project was expanded to recreating and archiving the commodore 65, a prototype computer that was never officially released. In addition to the restoration of this computer, updating the hardware was done wherever possible, so long as they didn't compromise the identity of the computer [24]. This was done so an external accelerator was not needed for the project.

The MEGA65 computer itself runs on a field programmable gate array (FPGA) dev board. This option was explored in depth due to a few factors such as, cost, onboard SD card slot, VGA output and the Artix-7 chip. The cost of boards was a large factor in the decision to use them as they would make the C65 affordable. The SD card slot on the board also helped this affordability as they are inexpensive and, conveniently, able to easily be interfaced with many computers. The VGA interface on the board was considered noteworthy as many monitors still have a port for them and there are readily available adaptors for VGA to various other ports [24].

In addition to the MEGA65 project, another project was run along side it, the MEGAphone project. This project aimed to create a commodore 65 based smart phone with a security focus. Instead of allowing the technology gap between the MEGAphone and modern phones to be detrimental, it would be one of the key features of the phone. The vastly simplified way in which the computer worked as well as the age would allow for many advantages over modern systems.

The simplicity would allow for the entire device to be verifiable, this is not normally explored by other security development teams. By being able to verify the phone is working as intended, it becomes almost impossible to compromise. In addition to this, as the phone is security focused, malicious activity is limited by the phone design itself. Not only this, but many of the security flaws being found in systems today are just not present in the commodore 64 because it doesn't have the supporting hardware.

Through the 36 years of activity that the commodore 64 has seen, many bugs have been found and documented in the system. As such, it is monumentally unlikely that there are new exploits in the system that are unfound, making it excellent to use for security purposes.

Chapter 3

Methods and Materials

3.1 Overview

This chapter describes how work on the project was conducted. It does this by first describing the methodology used, then by following with the hardware and software used. Finally, an overview of the thoughts about this chapter will be given.

3.2 Methodology

This project included several tasks, in which, debugging skills and a deep programming knowledge base was necessary. While debugging there were a few techniques that were used to enabled fast and more accurate pinpointing of the offending pieces of hardware. Chief among these debugging techniques was to visually expose relevant signals. By exposing a signal in real time it could be quickly and easily verified as either working or not working. Should

this fail, another useful debugging technique was to simulate the modules with report statements. While this was not always viable, it did allow for much more real time data to be exposed; it also allowed exact inputs to the hardware to be monitored. The final debugging technique used was to reduce the complexity of the hardware. By removing more complex elements of the hardware it is possible to isolate the issue and then fix it.

3.3 Materials

3.3.1 Hardware Used

During the development of this project multiple hardware devices were used, all of this hardware enabled either the creation, interaction or programming of the MEGA65. In addition to this some of the work on the project required specific hardware in order for full functionality of the MEGA65 to be present. This hardware is listed as follows:

- Digilent Nexys 4 DDR Artix-7 FPGA Trainer Board: This field programmable gate array (FPGA) development board, as seen in figure 3.1, was one of the vital components while conducting this project. It was used as the remappable hardware, on which, the MEGA65 prototypes could be developed and tested.
- Acer Aspire V Nitro: This laptop, as seen in figure 3.2, was the most important piece of hardware used in this project. This piece of hardware was used to interface with the FPGA, as well as make changes to the MEGA65 hardware.
- Dell UltraSharp 2408WFP 24-inch LCD Monitor: This monitor, while not critical, was a required piece of hardware. This was used to host the VGA output from the MEGA65.
- Dell KB1421 Keyboard: Much like the monitor, this piece of hardware was not critical but without it, or a similar product, progress on the project would be exceedingly difficult.
- SanDisk Ultra 16Gb MicroSDHC Micro SD card: This SD card, like the keyboard and monitor, was not critical, but it was necessary for full operation of the MEGA65 however.

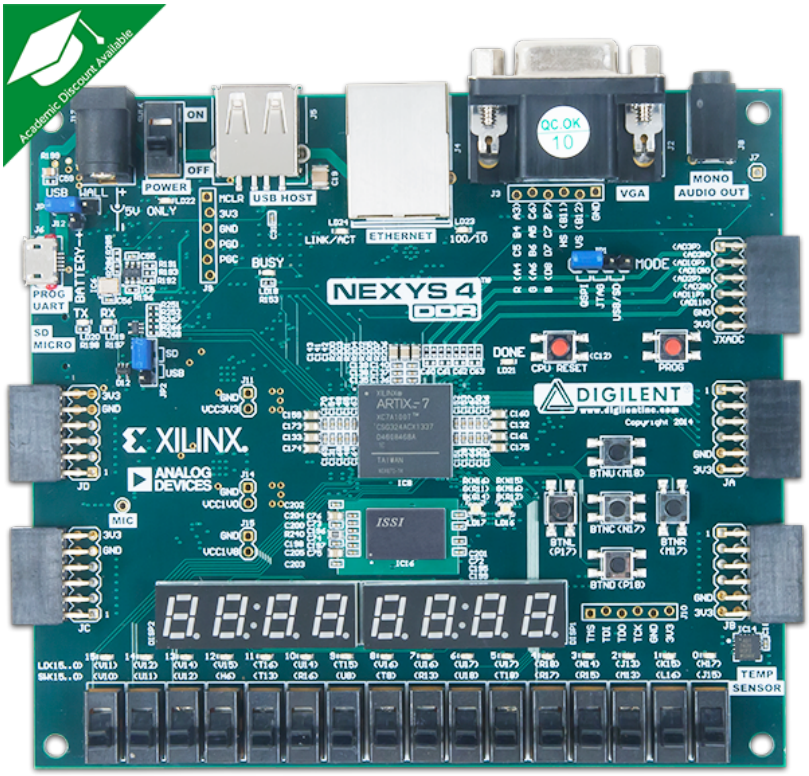


FIGURE 3.1: The Nexys 4 DDR Artix-7 FPGA Trainer Board

FIGURE 3.2: The Acer Aspire V Nitro Laptop

3.3.2 Software Used

During development of the MEGA65, several software packages were used in order to interact with the hardware listed above. These packages include:

- Ubuntu 18.04: This operating system was required for work on the project. It was through it that commands required to program and interact with the FPGA were able to function.
- Vivado HLx Webpack Edition: The development tools that this webpack provided were critical for the function of the MEGA65. These tools were used to program and make changes to the FPGA.
- Git: This version control tool, while not necessary, was extremely useful. This allowed changes to be tracked and shared between developers on the project. The hosted repositories are found on the GitHub website
- GHDL: This VHDL simulation software was very useful when debugging during the project. It allowed for more easily accessible signals during run time.

3.4 Final Thoughts

During this project several methods of development were used, most useful among these methods was to expose the signals physically on the FPGA. This allowed for clear confirmation of the states of the various finite state machines on the board. Secondary to this the GHDL simulation of the hardware also proved useful as it allowed greater exposure of the inner signals of the FPGA.
During this project several hardware and software packages were used. The most important hardware packages consisted of the FPGA, keyboard, monitor and computer. The key software packages included the Ubuntu Linux operating system, the hardware development software Vivado and the source control package Git.

Chapter 4

Project Set-up

4.1 Overview

This chapter describes the issues encountered prior to beginning development on answering the research questions posed in section 1.2. This section start by discussing why there were dependency issues experienced when starting the project. It then goes on to specifically identify the issues and the steps taken to correct these them. Then, in section 4.3, the issues that were experienced with fdisk are highlighted and the solution discussed. This chapter will end with the final thoughts about the issues encountered and the solutions given.

4.2 Dependency Issues

The MEGA65 project is an open source project, this works to the benefit, but also to the detriment of the project. Because there are many developers on the project, it is not well documented and the source code is prone to dependency issues that are only problems for newer developers. In addition to the the

```
$(CBMCONVERT):
        git submodule init
        git submodule update
        ( cd cbmconvert && make -f Makefile.unix )

$(CC65):
        git submodule init
        git submodule update
        ( cd cc65 && make -j 8 )

$(OPHIS):
        git submodule init
        git submodule update

$(GHDL):
        git submodule init
        git submodule update
        ( cd ghdl && sudo ./configure --prefix=$PWD/build && make && make install )
```

FIGURE 4.1: The makefile with sub-module initialisation and make additions.

large developer base, the iterative development cycle and trial and error bug fixing has also caused dependency issues to be present in the source code.

4.2.1 Missing Sub-Modules

During synthesis errors would often occur due to missing commands that were used in the makefile, but were not natively supported by Ubuntu Linux. The missing commands were the cc65 command, the ca65 command, the ld65 command, cbmconvert command and the ophis command. The cc65 command was used as the C compiler for the 6502 CPU. The ca65 was used in conjunction with the cc65 compiler as the assembler for the 6502 CPU. The ld65 was used as the linker to create an executable file from the object files. In addition to the missing compilation commands the cross-assembler, ophis, and the Commodore binary converter, cbmconvert, commands were not present in the project. These commands were developed separately from the MEGA65 and accessible through their own github repositories. This separation from the MEGA65 project caused each developer to reference the required commands differently, making the use of one universal makefile impossible. In order to correct this, as seen in figure 4.1, the source code for each required command was initialised as a sub-module of the project. This allowed the making of the project to automatically download, update and build the commands required later during synthesis.

Additionally, for these new sub-modules to be used, the pathing in the makefile was changed to correctly reference the required commands. Figure

```
COPT=    -Wall -g -std=gnu99
CC=      gcc
OPHIS=   Ophis/bin/ophis
OPHISOPT=        -4

VIVADO= ./vivado_wrapper

CC65=    cc65/bin/cc65
CA65=   cc65/bin/ca65 --cpu 4510
LD65=   cc65/bin/ld65 -t none
GHDL=   ghdl/build/bin/ghdl

CBMCONVERT=      cbmconvert/cbmconvert
```

FIGURE 4.2: The updated referencing of the sub-modules

```
349       - $(SDCARD_DIR)/MEGA65.D81:     $(UTILITIES)
      354 + $(SDCARD_DIR)/MEGA65.D81:     $(UTILITIES) $(CBMCONVERT)
350   355         $(warning ============================================================)
351   356         $(warning ~~~~~~~~~~~~~~~~> Making: $(SDCARD_DIR)/MEGA65.D81)
352   357         mkdir -p $(SDCARD_DIR)
353       -       cbmconvert -v2 -D8o $(SDCARD_DIR)/MEGA65.D81 $(UTILITIES)
      358 +       $(CBMCONVERT) -v2 -D8o $(SDCARD_DIR)/MEGA65.D81 $(UTILITIES)
```

FIGURE 4.3: The updated function calls that reference the code seen in 4.1.

4.2 shows that, thanks to sub-modules being located in the project directory, absolute pathing to the commands can now be done.

For the initialising and updating additions to the makefile to be useful, each instance where a sub-module is being used was updated to include a call to the sections seen in 4.1. One such example of these updates can be seen in figure 4.3.

4.3 Fdisk

In order to partition the MEGA65, the command line tool fdisk was created; this tool was already made a sub-module of the MEGA65 project. Similar to the MEGA65 source code, the fdisk source code had dependency issues

```
- CC65= cc65
- CL65= cl65
+ CC65= cc65/bin/cc65
+ CL65= cc65/bin/cl65
  COPTS=        -t c64 -O -Or -Oi -Os --cpu 65c02
  LOPTS=

@@ -28,12 +28,16 @@ HEADERS=   Makefile \

  DATAFILES=    ascii8x8.bin

-
- %.s:  %.c $(HEADERS) $(DATAFILES)
+ %.s:  %.c $(HEADERS) $(DATAFILES) $(CC65)
        $(CC65) $(COPTS) -o $@ $<

  all:  $(FILES)

+ $(CC65):
+       git submodule init
+       git submodule update
+       (cd cc65 && make -j 8)
+
  ascii8x8.bin: ascii00-7f.png pngprepare
        ./pngprepare charrom ascii00-7f.png ascii8x8.bin

@@ -45,7 +49,7 @@ ascii.h:     asciih
  pngprepare:   pngprepare.c
        $(CC) -I/usr/local/include -L/usr/local/lib -o pngprepare pngprepare.c -lpng

- m65fdisk.prg: $(ASSFILES) $(DATAFILES)
+ m65fdisk.prg: $(ASSFILES) $(DATAFILES) $(CL65)
        $(CL65) $(COPTS) $(LOPTS) -vm -m m65fdisk.map -o m65fdisk.prg $(ASSFILES)
```

FIGURE 4.4: fdisk sub-module dependency changes.

in it due to it's open source nature. Specifically, due to the cc65 and cl65 commands being required and the unusuallity of fdisk tool being compiled separately from the MEGA65 project. The pathing for the cc65 and cl65 commands in the mega65-fdisk makefile was incorrect. As seen in figure 4.4, previously the cc65 and cl65 commands was just copied into the repository, which was not ideal as the fdisk repository was supposed to be able to be built independently from the rest of the mega65 project. Similar to section 4.2, the cc65 repository was made a sub-module of the fdisk repository and the fdisk makerfile was changed to automatically initialise it and build it while making fdisk. These changes are shown in figure 4.4.

4.4 Final Thoughts

During the start up of the project there were many dependency issues present. This was due to the multiple active branches of the project working at once. Because of this there was no uniform way, in which, the project could be made. In order to correct this issue several third party github repositories were added as sub-modules to the MEGA65 project. In addition, the makefile was changed to update the pathing to the tools used and to automatically download and update these new sub-modules when they were used.

Chapter 5

Matrix Mode Corrections

5.1 Overview

In the following chapter the issues discovered in matrix mode are outlined, analysed and then the solutions to the problems are described. This begins with the serial interface locking issue and then moves onto the UART monitor, matrix mode compositor handshaking issue. The letterboxing implementation is then discussed and followed by the smaller bug fixes to matrix mode. Finally, the thoughts on the work done in this chapter are given.

5.2 Serial Locking

One of the existing issue with the phone was that when it entered matrix mode, the serial interface with the board would lock up. This would render all attempts at interfacing with the MEGA65 useless. Locking the serial input to the device was expected whilst in matrix mode, as this would prevent

```
@@ -122,6 +122,7 @@ entity gs4510 is
122 122         monitor_map_enables_low : out std_logic_vector(3 downto 0) := "1111";
123 123         monitor_map_enables_high : out std_logic_vector(3 downto 0) := "1111";
124 124         monitor_interrupt_inhibit : out std_logic := '0';
    125 +       monitor_memory_access_address : out unsigned(31 downto 0);
125 126
126 127         ----------------------------------------------------------------------
127 128         -- Memory access interface used by monitor
```

FIGURE 5.1: Addition of monitor memory access address port to the nocpu.vhdl file.

subversion of during sensitive program execution. The failure to return serial functionality to the device upon leaving matrix mode was showed that there was some handshaking protocols that were not behaving as expected.This phenomenon was especially strange as the code to disable the serial input while in matrix mode was removed for debug purposes. This proved that it was not entering or exiting matrix mode that caused the bug, but some protocol within the character processing or character displaying.

5.2.1 Creating the Test-bench

Due to the long synthesis times, in excess of an hour, a light weight version of the phone needed to be created. Thanks to debugging in the past, this problem had already been encountered and solved. In order to save time during synthesis previously, a version of the phone was created without the CPU. This version was outdated and required some additional ports to trick the components into working correctly. A CPU free version of the phone, some of which can be seen in figures 5.1, 5.2, 5.3, was possible thanks to matrix mode being composited over the video feed in hardware, rather than being done in software. Once the CPU free components were working, a Vivado technical command language (tcl) file was acquired from one of the other contributors to the project. This file was used to set up and execute the vivado tasks required to create and compile the CPU-less version of the phone. This resulted in a bit stream that was able to be run on the Nexys4ddr boards in the lab.

5.2.2 Timing Issues

One of the theorised causes for the serial locking issue was that the timing between the handshaking protocols was misaligned and was causing a state in which the serial output was not able to be read. In order to find these

```
143 144       -- Interface to ChipRAM in video controller (just 128KB for now)
144 145       ------------------------------------------------------------------------
145 146       chipram_we : OUT STD_LOGIC := '0';
146     -     chipram_address : OUT unsigned(16 DOWNTO 0) := "00000000000000000";
147     -     chipram_datain : OUT unsigned(7 DOWNTO 0) := (others => '0');
    147 +
    148 +     chipram_clk : IN std_logic;
    149 +     chipram_address : IN unsigned(19 DOWNTO 0) := to_unsigned(0,20);
    150 +     chipram_dataout : OUT unsigned(7 DOWNTO 0);
148 151
149 152       cpu_leds : out std_logic_vector(3 downto 0) := "1111";
150 153 +
```

FIGURE 5.2: Addition of the chip ram clock and modification to the chip ram address and data ports in nocpu.vhdl.

```
170 173       -- fast IO port (clocked at core clock). 1MB address space
171 174       ------------------------------------------------------------------------
172 175       fastio_addr : inout std_logic_vector(19 downto 0);
    176 +     fastio_addr_fast : inout std_logic_vector(19 downto 0);
173 177       fastio_read : out std_logic := '0';
174 178       fastio_write : out std_logic := '0';
175 179       fastio_wdata : out std_logic_vector(7 downto 0);
176 180       fastio_rdata : in std_logic_vector(7 downto 0);
177 181       kickstart_rdata : in std_logic_vector(7 downto 0);
    182 +     kickstart_address_out : out std_logic_vector(13 downto 0);
178 183       sector_buffer_mapped : in std_logic;
179 184       fastio_vic_rdata : in std_logic_vector(7 downto 0);
180 185       fastio_colour_ram_rdata : in std_logic_vector(7 downto 0);
```

FIGURE 5.3: Fast I/O address port and kickstart address port additions to the nocpu.vhdl file.

```
623  623    }
624  624    set obj [get_report_configs -of_objects [get_runs impl_1] impl_1_init_report_timing_summary_0]
625  625    if { $obj != "" } {
626       - set_property -name "is_enabled" -value "0" -objects $obj
     626  + set_property -name "is_enabled" -value "1" -objects $obj
627  627
628  628    }
629  629    # Create 'impl_1_opt_report_drc_0' report (if not found)
```

FIGURE 5.4: The tcl file change that allowed the generation of timing reports.

issues a timing report needed to be generated during synthesis. This was achieved by altering the tcl file and enabling the timing report generation, as seen in figure 5.4. This change was done to all of the tcl files as it would allow any type of synthesis, no CPU or otherwise, to generate timing reports. This would assist in identifying signals that extend synthesis due to timing issues in the future. In the timing reports generated, several video signals were found to be causing timing issues; to solve these an intermediary signal was created to host the video data. As seen in figure 5.5, this would cause the data to be delayed by one clock cycle, the loss in time was determined to be acceptable as the difference of a pixel appearing a clock cycle late would not be perceptible to the human eye. This fix was not just applied to the vga signal, it was also applied to some signals in the frame generator, pixel mapper, io mapper and top level assembly. Some of the signals that were causing timing issues were unable to be driven, this was because they were determined to be critical. These signals required if statement reduction to reduce the logic delay between the source and destination registers. This was as the delay between the source and destination caused values in the outer statements to become stale before the inner logic could be resolved. In order to perform these reductions, the arguments in the inner statements were merged into the outer statements. Unfortunately, the resolution of the timing issues had no impact on the serial locking, but it did result in faster synthesis and a clearer image on the screen.

5.2.3 Handshaking Issues

As the timing issues were cleared, it was determined that the locking issue must be an error in the handshaking rather than a signal error. Firstly the keycode data that was passed from the terminal emulator to the rain compositor, as well as the iomapper key data and terminal emulator state were examined. This was done via the seven segment display on the nexys4ddr development

```
-         vgared <= x"FF";
-         vgagreen <= x"00";
-         vgablue <= x"00";
+         vgared_driver <= x"FF";
+         vgagreen_driver <= x"00";
+         vgablue_driver <= x"00";
        else
          drive_led_out <= '0';
-         vgared <= vga_out_red(7 downto 0);
-         vgagreen <= vga_out_green(7 downto 0);
-         vgablue <= vga_out_blue(7 downto 0);
+         vgared_driver <= vga_out_red(7 downto 0);
+         vgagreen_driver <= vga_out_green(7 downto 0);
+         vgablue_driver <= vga_out_blue(7 downto 0);
        end if;
```

FIGURE 5.5: Addition of vga driving signals to the viciv.vhdl file.

board, as seen in figure 5.6. This revealed that the correct data was being passed prior to matrix mode and even after the lock had occurred. Further more, it showed that the terminal emulator was getting stuck in the print banner stage, in which, a predefined string was sent to the rain compositor. This suggested that the rest of the terminal emulator was working correctly and it was the output function that was causing the issue. Removing the printing section from the print banner state resulted in the serial output not locking up upon leaving matrix mode. Instead, the serial output would only lock after the first character was sent to the screen.

After further examining some of the flags associated with the output character function, it was noted that the terminal emulator would set two flags when sending characters, as seen in figure 5.7. The first is the character valid flag, it was observed to go high after a single character input and then never go low between subsequent inputs. This was also true for the second flag, the terminal emulator just sent flag. Looking for other instances of the both flags revealed that they were supposed to be reset while the terminal emulator was not ready, as seen in figure 5.8. This terminal emulator ready flag was followed into the matrix rain compositor and it was discovered that the terminal emulator was only not ready when scrolling or when processing a character in the specific case seen in figure 5.10. As these cases were correctly setting the terminal emulator to not ready when it was busy, the handshaking error was determined to be in the statement seen in figure 5.8. Examining figure 5.8, figure 5.10 and figure 5.7 together, it can be seen that there is a logic clash when sending a character. The output character function sets the just sent and character valid flags high, which is then used to set the ready flag low, which is used to set the just sent and character valid flags low. This error was corrected by flipping the reset logic, as seen in figure 5.9. This resulted in matrix mode no longer locking up the serial input after processing a single character.

5.3 Letterboxing

Another of the issues with matrix mode was that the overlay was exceeding the desired bounds. This resulted in junk data to be loaded into the areas of the screen that were supposed to be blank. It also caused the top of the matrix mode display to appear above the visible portion of the screen and

```
seg_led_data(31 downto 24) <= cpuspeed;
if cpuis6502 = '1' then
  seg_led_data(23 downto 16) <= x"65";
else
  seg_led_data(23 downto 16) <= x"45";
end if;
-- XXX temporary debug
seg_led_data(23 downto 16) <= uart_monitor_state;
seg_led_data(15 downto 8) <= uart_char;
seg_led_data(7 downto 0) <= monitor_char;
```

FIGURE 5.6: Temporary debug output added to enable keyboard and serial keycode and terminal emulator state visualisation.

```
procedure output_char(char : in character) is
begin
  -- XXX MATRIX - Disable UART output when in matrix mode in production
  -- UART output (even if we are in matrix mode)
  tx_data <= to_std_logic_vector(char);
  tx_trigger <= '1';

  if (protected_hardware_in(6)='1') then
    -- matrix mode terminal emulator output
    monitor_char_out <= unsigned(to_std_logic_vector(char));
    monitor_char_valid <= '1';

    -- Then mark terminal emulator busy long enough for it to
    -- accept the character, and clear the ready flag, if required.
    -- (this is required because the terminal emulator might be busy
    -- drawing pixels at the time).
    terminal_emulator_just_sent <= '1';
    terminal_emulator_ready_counter <= 511;
```

FIGURE 5.7: The output character function within the UART monitor subsystem.

```
if terminal_emulator_ready = '0' then
   terminal_emulator_just_sent <= '0';
   monitor_char_valid <= '0';
 end if;
```

FIGURE 5.8: The monitor character valid and terminal emulator just sent flag reset statement.

```
if terminal_emulator_ready = '1' then
   terminal_emulator_just_sent <= '0';
   monitor_char_valid <= '0';
 end if;
```

FIGURE 5.9: The corrected monitor character valid and terminal emulator just sent flag reset statement.

```
if terminal_emulator_fast = '1' then
  terminal_emulator_ready <= '1';
  terminal_emulator_fast <= '0';
  screenram_we <= '0';
elsif monitor_char_valid = '1' and screenram_busy = '0'
  and monitor_char_primed = '1' then
  terminal_emulator_ready <= '0';
  monitor_char_primed <= '0';
  report "Terminal emulator processing character $"
    & to_hstring(monitor_char_in);
```

FIGURE 5.10: The matrix rain compositor character sequencing code.

the end of matrix mode the continue off the screen, all of which can be seen in figure 5.11.

5.3.1 Makefile Correction

When looking at the error it was believed the the junk data was caused by rouge input from somewhere, as this input was always the same, it was determined to be some predefined string. Through an examination of matrix mode, it was discovered that the only source of input to the mode was the terminal emulator. Disconnecting the terminal emulator from the matrix rain compositor did not have any effect on the junk data being displayed however. This confirmed that the issue was not with the input to matrix mode, but in the compositor itself. Following the input characters into the matrix mode compositor lead to the case statement that handles the character processing and in particular, the case seen in figure 5.12. The monitor character can be seen being assigned to the screen ram write data, this write data signal is then used to write to the terminal memory, as seen in figure 5.13. This terminal memory, as seen by the use of screenram_rdata in figure 5.14, is then used to load character bits, which are then used to output characters to the screen. A joint examination of the terminal memory file with Dr. Gardner-Stephen, revealed that there were many preinitilized values in it via hardware. This was done in the makefile and the loaded values were the ASCII font and the matrix banner text. Removing both these from the terminal memory resolved the issue, but also resulted in no characters being able to be displayed. Further examination of the character set by Dr. Gardner-Stephen showed that two copies of it were being loaded into the terminal emulator and that one character set only required memory from location $000 to location $500. In order to remove this second character set from the terminal memory, after the character sets were loaded, zeros were loaded into the address locations from $800 to $4095. This change, as seen in figure 5.15, resulted in the matrix compositor only displaying unwanted characters in the lower portion of the screen. Figure 5.16 shows change in issue, the remaining junk characters are either from the character set itself or the banner, both of which are unable to be removed from the terminal memory.

5.3.2 Letterbox Implementation

As the matrix mode display was still exceeding the desired bounds, a limiting signal was used to keep the visible output within a letterbox. As revealed

FIGURE 5.11: Matrix mode prior to letterbox fixes.

by Dr. Gardner-Stephen, one such signal already existed in the VICIV and was being used for a similar purpose with the vga output. With the help of Dr. Gardner-Stephen, this signal was taken from the VICIV and sent to the matrix mode compositor, where it was used in place of vsync. As seen in figure 5.17, in addition to the replacement of vsync, the letterbox signal was used to blank the region outside of the letterbox, this can be seen in figure 5.18. While the letterboxing partially fixed the issues, it introduced some more minor issues into matrix mode. These are further discussed in the following sections.

5.4 Bug Fixing

In addition to the matrix mode breaking bugs solved in sections 5.2 and 5.3, there were also less urgent bugs in matrix mode that were solved. These bugs were graphical in nature and fixing them did not affect the overall functionality of matrix mode.

5.4.1 Revolving Line

While restoring functionality to matrix mode it was noted that the input was not behaving as expected. A seen in figure 5.19, whenever the cursor would go to a new line while scrolling it would move left one horizontal space. This resulted in the line revolving around the screen, this is believed to be due to

```
  when others =>
    -- Simply put character into place, and advance cursor
    -- as for cursor right
    report "te_cursor_address = "
      & integer'image(te_cursor_address)
      & ", te_cursor_x = " & integer'image(te_cursor_x)
      & ", te_cursor_y = " & integer'image(te_cursor_y)
      & ", te_screen_start = "
      & integer'image(te_screen_start)
      & ", te_header_start = "
      & integer'image(te_header_start);
    screenram_addr <= te_cursor_address;
    screenram_wdata <= monitor_char_in;
    -- Prevent overwriting font
    if te_cursor_address >= te_header_start
      and te_cursor_address < 4096 then
      screenram_we <= '1';
    else
      screenram_we <= '0';
    end if;
    screenram_busy := '1';
    if te_cursor_x < te_x_max then
      -- stay on same line
      te_cursor_x <= te_cursor_x + 1;
      report "increment te_cursor_address, because cursor_x < x_max";
      te_cursor_address <= te_cursor_address + 1;
      terminal_emulator_fast <= '1';
    else
      -- advance to next line (and possibly scroll)
      te_cursor_x <= 0;
      if te_cursor_y < te_y_max then
        te_cursor_y <= te_cursor_y + 1;
        report "increment te_cursor_address, because not yet at bottom of screen "
          & "(te_cursor_y=" & integer'image(te_cursor_y)
          & ", te_y_max=" & integer'image(te_y_max) & ")";
        te_cursor_address <= te_cursor_address + 1;
        terminal_emulator_fast <= '1';
      else
        terminal_emulator_ready <= '0';
        scroll_terminal_up <= '1';
        erase_address <= te_screen_start;
        terminal_emulator_fast <= '0';
        te_cursor_address <= te_cursor_address - te_line_length + 1;
      end if;
    end if;
end case;
```

FIGURE 5.12: The matrix rain compositor character processing code.

```
screenram0:   entity work.termmem port map (
  clk => pixelclock,
  we => screenram_we,
  data_i => screenram_wdata,
  address => screenram_addr,
  data_o => screenram_rdata
  );
```

FIGURE 5.13: The matrix mode screen ram entity.

```
if matrix_fetch_chardata = '1' then
  if pixel_x_800 >= debug_x and pixel_x_800 < (debug_x+10) then
    report
      "x=" & integer'image(pixel_x_800) & ": " &
      "Reading char data = $" & to_hstring(screenram_rdata);
  end if;
  next_char_bits <= std_logic_vector(screenram_rdata);
elsif matrix_fetch_glyphdata = '1' then
  next_glyph_bits <= std_logic_vector(matrix_rdata);
else
      report "memory read data = $" & to_hstring(matrix_rdata);
end if;
```

FIGURE 5.14: The assignment of the character bits in the terminal memory to the output variable.

```
$(TOOLDIR)/mempacker/mempacker -n termmem -s 4095 -f $(VHDLSRCDIR)/termmem.vhdl $(BINDIR)/asciifont.bin@000 $(BINDIR)/matrix_banner.txt@A24
$(TOOLDIR)/mempacker/mempacker -n termmem -s 4095 -f $(VHDLSRCDIR)/termmem.vhdl $(BINDIR)/asciifont.bin@000 /dev/zero@800 $(BINDIR)/matrix_
```

FIGURE 5.15: The makefile change to remove the excess character set from the terminal memory.

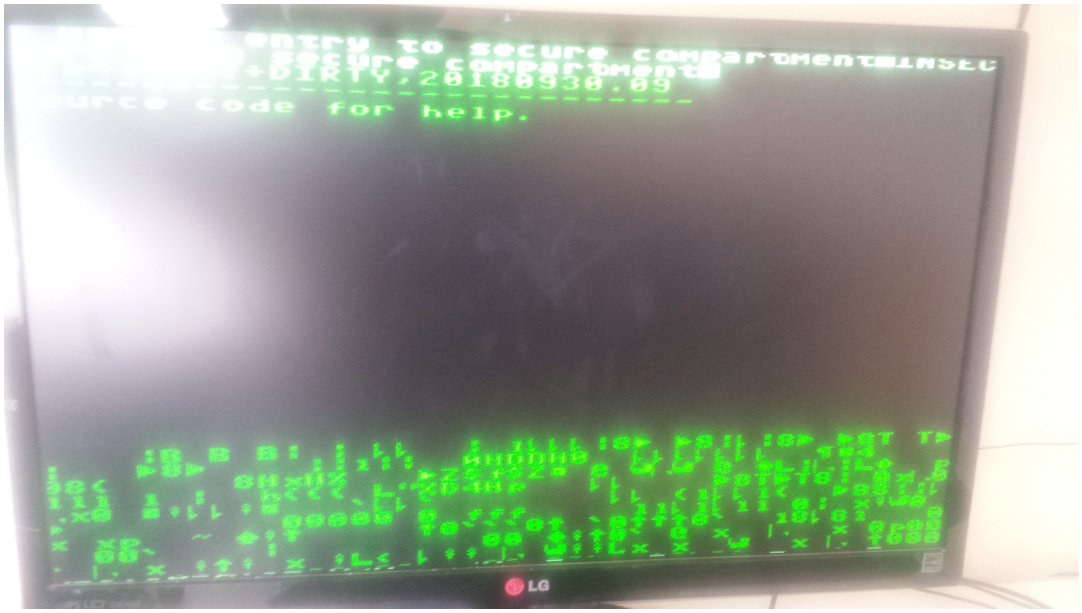


FIGURE 5.16: Matrix mode without the second character set loaded into memory during compilation.

```
246 246         hsync_out <= hsync_in;
247 247         vsync_out <= vsync_in;
248 248         last_hsync <= hsync_in;
249     -       last_vsync <= vsync_in;
    249 +       last_letterbox <= lcd_in_letterbox;
250 250
251 251         drop_row <= (to_integer(ycounter_in)+0)/16;
252 252
    @@ -749,6 +749,7 @@ begin  -- rtl
749 749             "x=" & integer'image(pixel_x_800) & ": " &
750 750             "source = " & feed_t'image(feed);
751 751         end if;
    752 +
752 753         case feed is
753 754           when Normal =>
754 755             -- Normal display, so show pixels from input video stream
    @@ -767,7 +768,13 @@ begin  -- rtl
767 768                 "  pixel_out = " & std_logic'image(char_bits(7))
768 769                 & ", char_bits=%" & to_string(char_bits);
769 770             end if;
770     -           if row_counter >= te_header_line_count then
    771 +           if last_letterbox='0' then
    772 +             -- outside of 800x480 LCD visible letterbox: Should be rain or black,
    773 +             -- not any of the underlying display
    774 +             vgared_out <= x"00";
    775 +             vgagreen_out <= x"00";
    776 +             vgablue_out <= x"00";
    777 +           elsif row_counter >= te_header_line_count then
771 778               -- In normal text area
772 779               if char_bits(0) = '1' then
773 780                 if is_cursor='1' and te_blink_state='1' then
    @@ -875,7 +882,7 @@ begin  -- rtl
875 882           lfsr_advance(1 downto 0) <= "11";
876 883           lfsr_advance(3 downto 0) <= "1111";
877 884         end if;
878     -       if last_vsync = '1' and vsync_in = '0' then
    885 +       if last_letterbox = '1' and lcd_in_letterbox = '0' then
```

FIGURE 5.17: The changes made to the matrix mode compositor to implement the letterboxing signal.

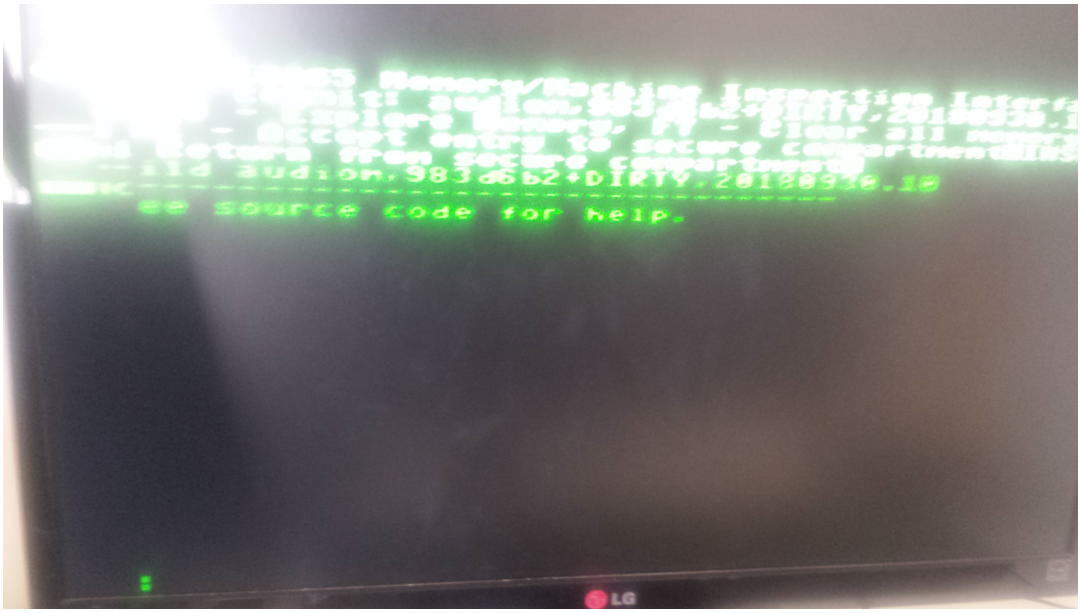


FIGURE 5.18: Matrix mode with the letterboxing signal.

under-flowing the horizontal position variable. Further examination of the variable, as seen in figure 5.20 and figure 5.21, show that, during the new line handling while scrolling, the cursor position is just rolled back to the start of the line. This does not take into account the full-stop used as an indication character at the start of each new line. The correction, as seen in figure 5.22, addresses this issue by starting the scrolled cursor position one increment to the right of the start of the line.

5.4.2 Bell Issue

During experimentation with the input to matrix mode it was noted that a strange character was being printed whenever the buffer overflowed. As identified by Dr. Gardner-Stephen, this character was intended to be the non printable bell character that would alert the user to an illegal action. This character can be seen as the "plus" symbols in figure 5.23. Examining the character output to the matrix mode compositor during when these bell characters were being output revealed that the key code being output was x"07". At the suggestion of Dr. Gardner-Stephen, an examination of the available keycodes showed that there was no case for this particular keycode. In order to stop this character from being printed, a case was added to handle it, as seen in figure 5.24. This case would do nothing when it detected the bell character, this allowed for the character to be used later, while also preventing it from being printed. This introduced another error however, whenever a

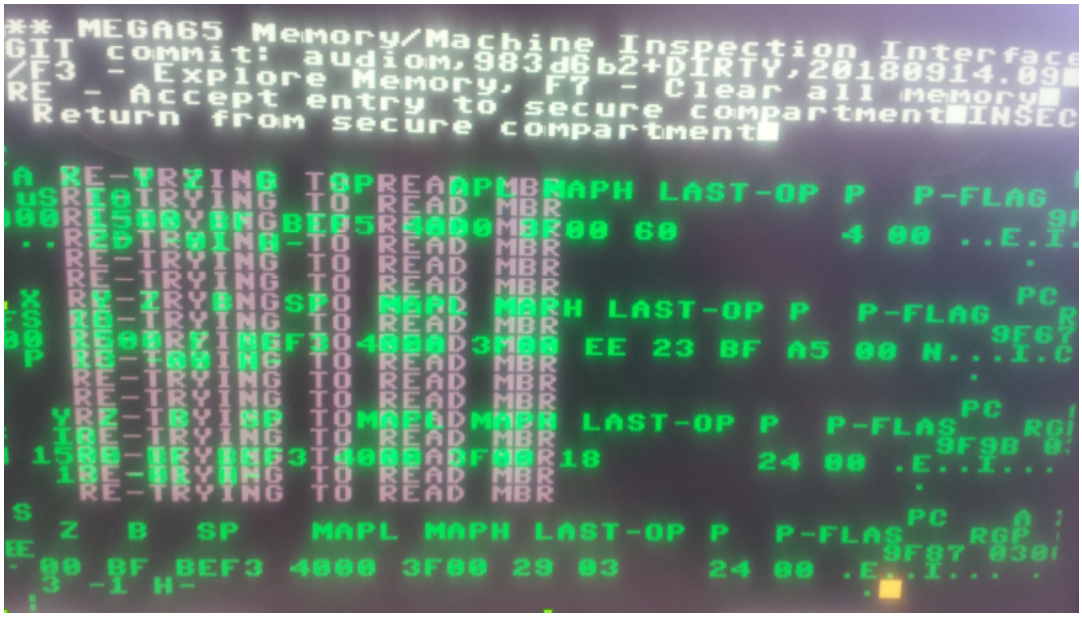


FIGURE 5.19: The matrix mode line revolution error.

```
if te_cursor_y < te_y_max then
  -- No need to scroll yet
  te_cursor_y <= te_cursor_y + 1;
  te_cursor_address <= te_cursor_address + 1;
  terminal_emulator_fast <= '1';
else
  -- We need to scroll
  terminal_emulator_ready <= '0';
  scroll_terminal_up <= '1';
  erase_address
    <= 4096 - (te_y_max+1) * te_line_length;
  terminal_emulator_fast <= '0';
  te_cursor_address <= te_cursor_address - te_line_length;
end if;
```

FIGURE 5.20: The code used to handle the cursor address when scrolling in matrix mode.

```
if te_cursor_y < te_y_max then
  te_cursor_y <= te_cursor_y + 1;
  report "increment te_cursor_address, because not yet at bottom of screen "
    & "(te_cursor_y=" & integer'image(te_cursor_y)
    & ", te_y_max=" & integer'image(te_y_max) & ")";
  te_cursor_address <= te_cursor_address + 1;
  terminal_emulator_fast <= '1';
else
  terminal_emulator_ready <= '0';
  scroll_terminal_up <= '1';
  erase_address <= te_screen_start;
  terminal_emulator_fast <= '0';
  te_cursor_address <= te_cursor_address - te_line_length;
```

FIGURE 5.21: The code used when characters overflow onto the next line and scrolling is required.

```
                erase_address
                  <= 4096 - (te_y_max+1) * te_line_length;
                terminal_emulator_fast <= '0';
-               te_cursor_address <= te_cursor_address - te_line_length;
+               te_cursor_address <= te_cursor_address - te_line_length + 1;
              end if;
            end if;
          when x"08" =>
@@ -505,7 +505,7 @@ begin  -- rtl
                scroll_terminal_up <= '1';
                erase_address <= te_screen_start;
                terminal_emulator_fast <= '0';
-               te_cursor_address <= te_cursor_address - te_line_length;
+               te_cursor_address <= te_cursor_address - te_line_length + 1;
              end if;
            end if;
        end case;
```

FIGURE 5.22: The matrix rain compositor changes to eliminate the revolving line.

bell character was recognised it would lock up the input to matrix mode. This was due to the terminal emulator never being ready. Examination of the other cases showed that they all set terminal_emulator_fast to high after they had performed their case specific action. Implementing this, as seen in figure 5.25, in the new case statement resulted in the issue being resolved.

5.4.3 Pixel Shift

When examining the characters displayed to the screen, as noted in figure 5.26, some of them had their right most pixels wrapped around to the very left of the of the character tile. This output, as seen in figure 5.27, is due to the char_bits variable. Tracing this variable back to its source revealed, as seen in figure 5.28, that the bits for each character were being rotated around and used to decide whether to display a pixel or not. Dr. Garner-Stephen suggested, as the issue was a horizontal rotation, a rotation of the character bits should be performed, as displayed in figure 5.29. This fixed the pixel shift issue.

5.4.4 Letterbox Fixing

In collaboration with Dr. Gardner-Stephen, the junk characters seen at the bottom section of matrix mode, as seen if figure 5.18, were able to be restricted from the display. To perform this restriction of the output to the visible

FIGURE 5.23: The matrix mode with printed bell characters.

```
+            when x"07" =>
+              -- Ignore bell character, instead of printing plus symbol
+              null;
```

FIGURE 5.24: The case statement to ignore bell characters.

```
              when x"07" =>
                -- Ignore bell character, instead of printing plus symbol
-               null;
+               terminal_emulator_fast <= '1';
```

FIGURE 5.25: The change to the case statement to prevent serial locking.

FIGURE 5.26: The matrix mode character shift.

```
if char_bits(0) = '1' then
  if is_cursor='1' and te_blink_state='1' then
    vgared_out(7 downto 6) <= "00";
    vgagreen_out(7 downto 6) <= "00";
    vgablue_out(7 downto 6) <= "00";
    vgared_out(5 downto 0) <= vgared_in(7 downto 2);
    vgagreen_out(5 downto 0) <= vgagreen_in(7 downto 2);
    vgablue_out(5 downto 0) <= vgablue_in(7 downto 2);
  else
    vgared_out(7 downto 6) <= "00";
    vgagreen_out(7 downto 6) <= "11";
    vgablue_out(7 downto 6) <= "00";
    vgared_out(5 downto 0) <= vgared_in(7 downto 2);
    vgagreen_out(5 downto 0) <= vgagreen_in(7 downto 2);
    vgablue_out(5 downto 0) <= vgablue_in(7 downto 2);
  end if;
else
  if is_cursor='1' and te_blink_state='1' then
    vgared_out <= "11111111";
    vgagreen_out <= "11111111";
    vgablue_out <= "00000000";
  else
    vgared_out(7) <= '0';
    vgagreen_out(7) <= '0';
    vgablue_out(7) <= '0';
    vgared_out(6 downto 0) <= vgared_in(7 downto 1);
    vgagreen_out(6 downto 0) <= vgagreen_in(7 downto 1);
    vgablue_out(6 downto 0) <= vgablue_in(7 downto 1);
  end if;
end if;
```

FIGURE 5.27: The character bits to vga output code.

```
elsif char_bit_count = 0 then
  -- Request next character
  if pixel_x_800 >=debug_x and pixel_x_800 < (debug_x+10) then
    report
      "x=" & integer'image(pixel_x_800) & ": " &
      "char_bits becomes $" & to_hstring(next_char_bits);
  end if;
  char_bits <= std_logic_vector(next_char_bits);
  is_cursor <= next_is_cursor;
  char_screen_address <= char_screen_address + 1;
  fetch_next_char <= '1';
  char_bit_count <= 16;
else
  -- rotate bits for terminal chargen every 2 640H pixels
  if (pixel_x_800 mod 2) = 0 and char_bit_count /= 1
    and pixel_x_800 /= last_pixel_x_800 then
    char_bits(7 downto 1) <= char_bits(6 downto 0);
    char_bits(0) <= char_bits(7);
  end if;
  if pixel_x_800 /= last_pixel_x_800
    and pixel_x_800 /= last_pixel_x_800 then
    char_bit_count <= char_bit_count - 1;
  end if;
end if;
```

FIGURE 5.28: The character bit rotation code.

```
777  777          vgablue_out <= x"00";
778  778        elsif row_counter >= te_header_line_count then
779  779          -- In normal text area
780       -        if char_bits(0) = '1' then
     780  +        if char_bits(7) = '1' then
781  781            if is_cursor='1' and te_blink_state='1' then
782  782              vgared_out(7 downto 6) <= "00";
783  783              vgagreen_out(7 downto 6) <= "00";
     @@ -810,7 +810,7 @@ begin  -- rtl
810  810        else
811  811          -- In header of matrix mode
812  812          -- Note that cursor is not visible in header area
813       -        if char_bits(0) = '0' then
     813  +        if char_bits(7) = '0' then
814  814            vgared_out <= (others => '0');
815  815            vgagreen_out <= (others => '0');
816  816            vgablue_out <= (others => '0');
```

FIGURE 5.29: The manual bit shift to counter the shifting error.

```
+           elsif row_counter >= (te_header_line_count + te_screen_height) then
+             -- Beyond the end of display, so don't display text of overlay.
+             vgared_out(7 downto 6) <= "00";
+             vgagreen_out(7 downto 6) <= "00";
+             vgablue_out(7 downto 6) <= "00";
+             vgared_out(5 downto 0) <= vgared_in(7 downto 2);
+             vgagreen_out(5 downto 0) <= vgagreen_in(7 downto 2);
+             vgablue_out(5 downto 0) <= vgablue_in(7 downto 2);
```

FIGURE 5.30: Correction to the rain compositor to stop the junk characters appearing in the lower section of the screen.

section of the screen, the row counter variable was used to determine if the current row was beyond the end of the matrix mode display. As seen in figure 5.30, if the current row was beyond the end of the writable matrix mode display, the vga input was darkened and passed out. This would hard code out any possibility of the matrix mode display being corrupted by junk characters being present in the terminal memory due to other faults.

5.4.5 Character Duplication

After the letterbox implementation as described in section 5.3, it could be seen that a strange phenomenon occurred on the left side of the screen when in matrix mode. It appears that there is a region, about four character wide, that contains duplicates of characters from the next line. In addition to this duplicate, there appears to be smearing of the first two characters in this duplicate region. These observations can be seen in figure 5.31. In addition to the issues with the left region, it is believed that these issues are causing the screen to become shifted to the right. As seen in figure 5.31, the last four characters of the line are continuing past the right most visible point of the screen. In order to correct this Dr. Gardner-Stephen introduced the column variable to monitor the matrix mode cursor position, as seen in figure 5.32. As seen in figure 5.33, this signal was used to determine what portion of the screen was visible. The column visible signal was then used as an additional condition for displaying the output to the screen. This fixed the matrix mode display issues and resulted in the clean display seen in figure 5.35.

5.5 Final Thoughts

During the beginning of this project matrix mode was unusable. The changes made to it have now restored it to a working order as the serial interface is no

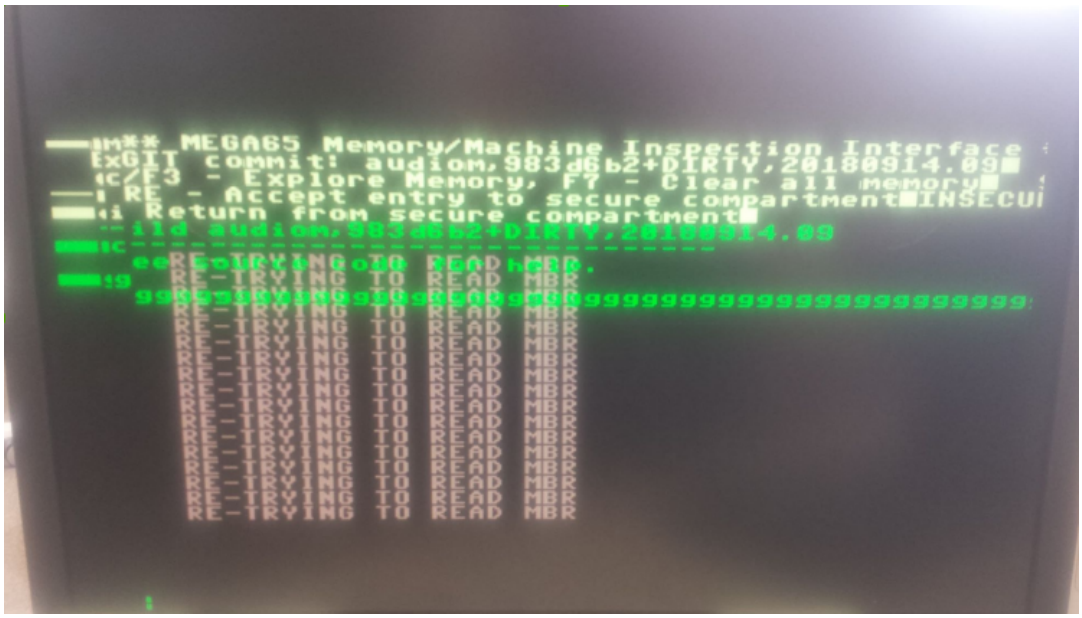


FIGURE 5.31: The matrix rain compositor smearing / duplication.

```
186  +   signal column_counter : integer := 0;
186  187     signal next_is_cursor : std_logic := '0';
187  188     signal is_cursor : std_logic := '0';
188  189
       @@ -587,6 +588,7 @@ begin  -- rtl
587  588           if external_frame_x_zero='1' and last_external_frame_x_zero = '0' then
588  589             char_bit_count <= 0;
589  590             fetch_next_char <= '1';
     591  +          column_counter <= 0;
590  592             -- reset fetch address to start of line, unless
591  593             -- we are advancing to next line
592  594             -- XXX doesn't yet support double-high chars
       @@ -611,6 +613,7 @@ begin  -- rtl
611  613             char_screen_address <= char_screen_address + 1;
612  614             fetch_next_char <= '1';
613  615             char_bit_count <= 16;
     616  +          column_counter <= column_counter + 1;
614  617           else
615  618             -- rotate bits for terminal chargen every 2 640H pixels
616  619             if (pixel_x_800 mod 2) = 0 and char_bit_count /= 1
```

FIGURE 5.32: The matrix rain compositor cursor column tracking variable.

```
616 +           -- The offset of 3 is to position the matrix mode overlay more
617 +           -- centrally on the screen.
618 +           if column_counter > 3 then
615 619             char_screen_address <= char_screen_address + 1;
616 620           end if;
621 +           if column_counter=3 then
622 +             column_visible <= '1';
623 +           elsif column_counter = (3 + te_line_length) then
624 +             column_visible <= '0';
625 +           end if;
617 626           fetch_next_char <= '1';
618 627           char_bit_count <= 16;
619 628           column_counter <= column_counter + 1;
```

FIGURE 5.33: The matrix rain compositor column visibility cases.

```
-- In normal text area
if (char_bits(0) = '1') and column_visible='1' then
  if is_cursor='1' and te_blink_state='1' then
    -- Display visual beep by making background of monitor
    -- terminal area flash red.
    vgared_out(7) <= invert_frame;
    vgared_out(6) <= invert_frame;
    vgagreen_out(7 downto 6) <= "00";
    vgablue_out(7 downto 6) <= "00";
    vgared_out(5 downto 0) <= vgared_in(7 downto 2);
    vgagreen_out(5 downto 0) <= vgagreen_in(7 downto 2);
    vgablue_out(5 downto 0) <= vgablue_in(7 downto 2);
  else
    if alternate_colour='1' then
      vgared_out(7 downto 6) <= alternate_row & alternate_row;
      vgagreen_out(7 downto 6) <= "11";
      vgablue_out(7 downto 6) <= "11";
    else
      vgared_out(7 downto 6) <= "00";
      vgagreen_out(7 downto 6) <= "11";
      vgablue_out(7 downto 6) <= "00";
    end if;
    vgared_out(5 downto 0) <= vgared_in(7 downto 2);
    vgagreen_out(5 downto 0) <= vgagreen_in(7 downto 2);
    vgablue_out(5 downto 0) <= vgablue_in(7 downto 2);
  end if;
```

FIGURE 5.34: The matrix mode column dependant output restrictions.

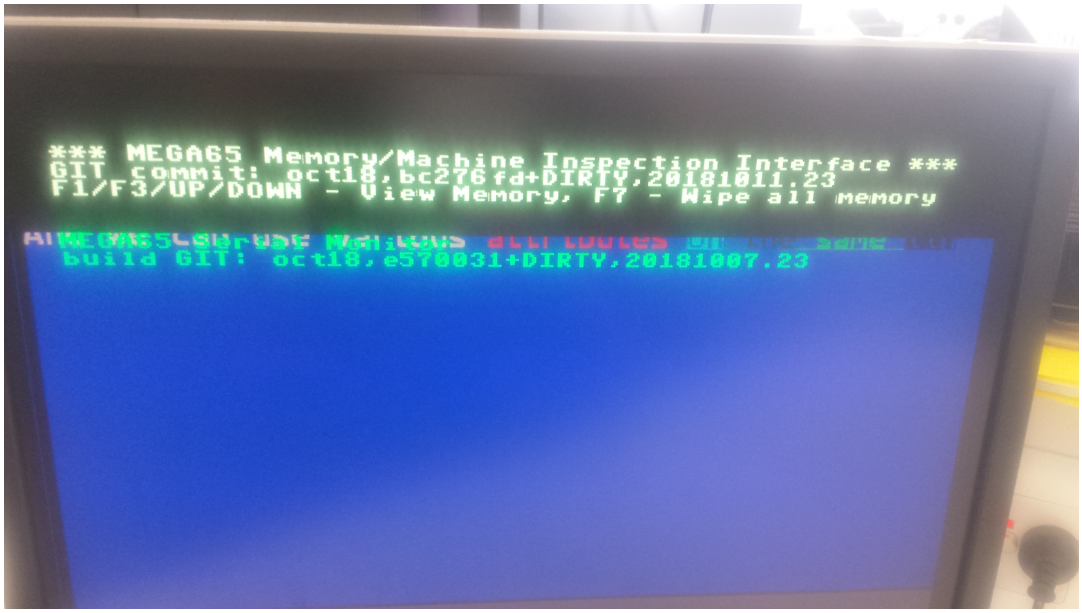


FIGURE 5.35: The clean matrix mode display.

longer disabled after leaving matrix mode. In addition to this, the changes made have resolved a lot of the graphical issues that were present. These changes include the restriction of the output to the writable section of the screen, the removal of the error character and character duplication. This resulted in making matrix mode more aesthetically pleasing to use.

Chapter 6

Secure Compartmentalisation

6.1 Overview

This chapter goes over the frame work for how secure compartments will be implemented in the MEGA65. It will then discuss the problems encountered during the implementation of the secure compartments. This chapter will then talk about the possible issues with the secure compartments. The final thoughts on the issues encountered and the solutions given then ends this chapter.

6.2 Theorised Operation

Thanks to the research done in a prior project by Tim Kirby, a frame work was provided for the operation of the secure compartments of the phone, as seen in figure 6.1. From this diagram, the finite state machine seen in figure 6.2 was created and the state transition actions were noted. As seen in both

figures 6.1 and 6.2, initially the user begins in the insecure user mode. This user mode is then halted via a hypervisor trap that causes the io registers and the RAM and ROM of the phone to be saved to the SD card. The non-transfer section of RAM and all of ROM are then erased. From one of the save-state slots on the SD card, the desired secure service is then loaded into ROM. As soon as this is finished secure mode and matrix mode are triggered, the hypervisor is left and the CPU is halted. Matrix mode causes all external input into the phone, apart from the keyboard, to be cut off. At this point the user is able to inspect ROM and the transfer area of RAM. If the user is satisfied, by typing "ACCEPT" Matrix mode is left and the secure service is then allowed to execute with the data provided in the transfer area of RAM. If the user is dissatisfied, by typing "REJECT" the secure service is erased from ROM, matrix mode is then left, the hypervisor is retriggered and the CPU is resumed. The hypervisor then uses the save-state created immediately prior to entering matrix mode to restore the io registers and load all the data that was saved back into RAM and ROM. If the secure container was accepted, any additional trap to the hypervisor will be seen as an exit request from the secure container and will once again trigger Matrix mode. During an exit request, the user will again be prompted to inspect the transfer area of RAM and then accept or reject the escaping of the of that data from the secure container. If satisfied, by typing "ACCEPT" the non-transfer section if RAM and all of ROM are once again erased. Then, identically to the user rejecting entry into the secure container, matrix mode is left, the hypervisor is retriggered and the CPU is resumed. The hypervisor then loads the save-state created immediately prior to entering matrix mode from the SD card, which is used to restore the io registers and reload data back into non-transfer RAM and ROM. If the user is dissatisfied with data escaping the secure container, by typing "REJECT" all of RAM and all of ROM are erased before the matrix mode exit. Then the save-state is loaded identically to the accepted exit case by the hypervisor and the resumed CPU. Finally, an exit status flag is raised depending on how the secure service transaction went. A basic diagram of the secure mode functionality can be seen in figure 6.3

6.3 Initial Issues

As the secure compartments were entirely separate from the matrix mode corrections and during the time a major update for another section of the phone was finished, a new branch of the project was created. Switching to

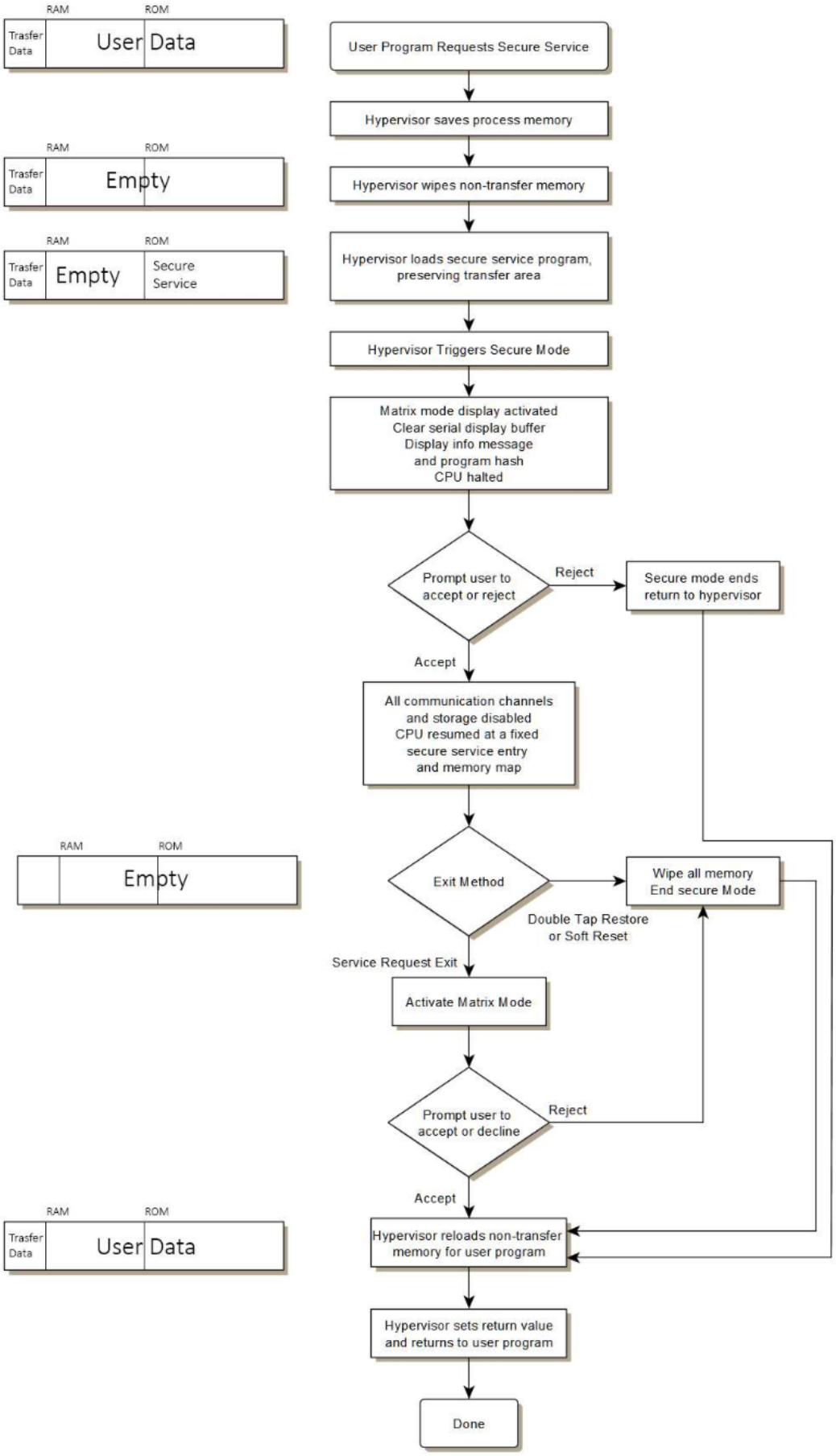


FIGURE 6.1: The secure compartment frame work provided by Tim Kirby's thesis "Design and Development of a Secure Compartmentalised 8-bit Architecture".

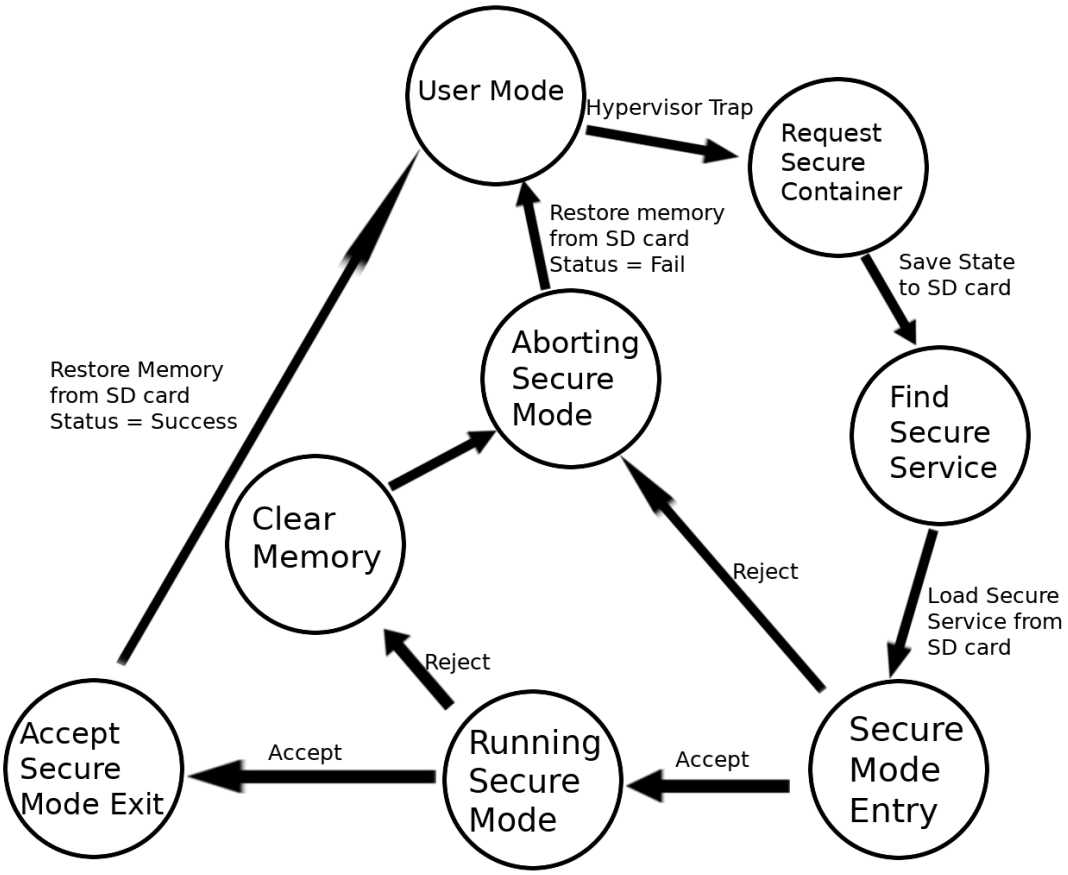


FIGURE 6.2: The secure mode finite state machine.

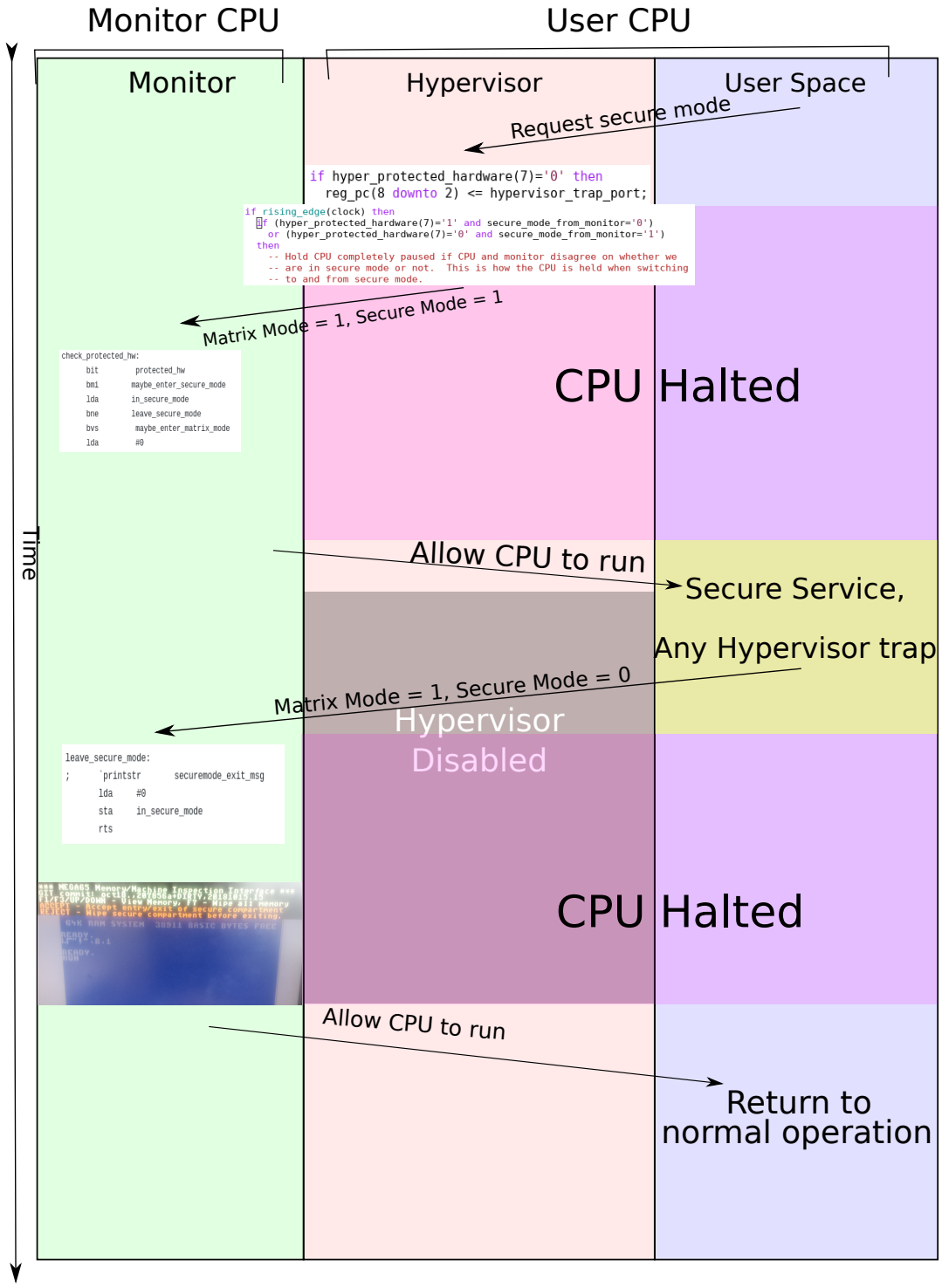


FIGURE 6.3: The secure mode signal diagram.

this new branch resulted in overhead issues that needed to be solved in order for the project to proceed further.

6.3.1 Repairing the Branch

As the project was open source and had many developers on it, there were multiple active branches all pursuing various goals. This wide array of active branches meant that the other developers worked in isolation from the changes described in chapter 5, many of these changes had to be reimplemented in the merged branch.
The timing fixes done to possibly correct the matrix mode issue were all removed, including the generation of the timing reports, these were reimplemented. The implementation was identical to that described in section 5.2.2 of chapter 5.
All of the letterboxing changes were lost, so, as seen in section 5.3 of chapter 5, the erasing of the second character set was done one again and the letterbox signal was one again used to limit the output of matrix mode.
The revolving line issue due to discrepancies in line length returned and was fixed as described in section 5.4.1 of chapter 5.

In addition to the minor issues, matrix mode was not behaving as intended again. When attempting to enter or exit matrix mode with the tab + C= key combination, the key combination would be read and re-read rapidly and continuously. This made entering and exiting matrix mode unreliable as it was up to luck whether the key combination was scanned an odd or even amount of times. Odd and the state would be toggled, even and the state would not be toggled. Tracing the matrix mode hypervisor trap from the CPU to the io mapper shows, as seen in figure 6.4, there are latching and unlatching conditions for the matrix mode trigger. When monitoring the ascii_key_valid and ef_latch signals via oscilloscope, the cause of the issue was clear. The ascii_key_valid signal was not a latching signal, and thus, it would only be high for a single clock cycle. As seen in the logic in figure 6.4, this would cause the latch signal to pulse similarly, allowing the matrix mode hyper trap to be triggered every second clock cycle. A quick solution to this issue, as seen in figure 6.5 was written by Dr. Gardner-Stephen in the form of a timeout clock that blocks repeated instances of tab + C= from triggering the matrix mode hypervisor trap.

```
matrix_mode_trap <= '0';
if ascii_key_valid='1' and ascii_key = x"EF" and  ef_latch='0' then
  -- C= + TAB
  -- This replaces the old ALT+TAB task switch combination
  matrix_mode_trap <= '1';
  ef_latch <= '1';
end if;
if ascii_key_valid='0' or ascii_key /= x"ef" then
  ef_latch <= '0';
end if;
```

FIGURE 6.4: The matrix mode hypervisor trap latch.

```
-- Generate 50Hz signal for TOD clock
-- (Note that we are a bit conflicted here, as our video mode is PALx4,
--  but at 60Hz.  We will make our CIAs take 50Hz like in most PAL countries
-- so that we don't confuse things too much.  We will probably add a 50Hz
-- raster interrupt filter to help music and games play at the right rate.)
if counter50hz<divisor50hz then
  counter50hz <= counter50hz + 1;
else
  clock50hz <= not clock50hz;
  counter50hz <= 0;
end if;

seg_led(12) <= eth_keycode_toggle;
seg_led(11) <= last_scan_code(12);
seg_led(10 downto 0) <= unsigned(last_scan_code(10 downto 0));

-- Buffer ASCII keyboard input: Writing to the register causes
-- the next key in the queue to be displayed.
matrix_mode_trap <= '0';
if ascii_key_valid='1' and ascii_key = x"EF" and  ef_latch='0' then
  -- C= + TAB
  -- This replaces the old ALT+TAB task switch combination
  matrix_mode_trap <= '1';
  ef_latch <= '1';
  -- But don't allow multiple triggerings of this too quickly
  -- (This is to work around a very weird bug where C= + TAB on PS/2
  -- keyboards at least, triggers repeatedly.  But C= + ~ (like <- on a
  -- C64 keyboard) doesn't have that problem. Maybe the PS/2 keyboard triggers
  -- the TAB key more frequently?  Anyway, we will just block repeats
  -- from occurring too quickly
  ef_timeout <= 20; -- x 1/100th of a second
end if;
if (ascii_key_valid='0' or ascii_key /= x"ef") and ef_latch='1' then
  if counter50hz = 0 then
    if ef_timeout /= 0 then
      ef_timeout <= ef_timeout - 1;
    else
      ef_latch <= '0';
    end if;
  end if;
end if;
```

FIGURE 6.5: The matrix mode hypervisor trap latch with the new timeout period to prevent constant triggering.

6.3.2 SD Card Restoration

One of the existing functions on the phone was the freeze function, this allowed the user to save state to a "Slot" located on the SD card. This freeze function would be the basis for the saving to and loading from the SD card when entering and exiting secure mode. The functionality for the standard capacity SD cards (SDSC) was entirely as expected, it was only the high capacity SD cards (SDHC) that were causing issues. When attempting to write to an SDHC card with the existing code, no visible errors occurred. It was only when attempting to read from this card that an issue occurred. When attempting to read after the first time the SD card would read no data, each read required a reset of the SD card between them in order for correct functionality. The extreme capacity SD card (SDXC) was not supported by the current hardware, and indeed, was not recognised as an SD card at all when reading and writing was attempted. At the suggestion of Dr. Garner-Stephen, the issue with the SDHC cards was examined from the finite state machine located in the sdcard.vhdl file. As seen in figure 6.6 when reading or writing, the SD card is selected by making the cs_bo bit low, and when it is not doing either of them, it is deselected by making cs_bo high. Comparing this to the reset process seen in figures 6.7, 6.8 and 6.9, it can be seen that there are several signals that are not being propagated when deselecting the SD card. At the suggestion of Dr. Gardner-Stephen, the simple solution of not deselecting the SD card was used. This solution, as seen in figure 6.10, does not provide hot swapping of the SD card, but has the functionality required for the secure mode implementation.

While implementing the secure compartmentalisation, it was noted that there was an anomaly when attempting to write to the SD card. When attempting to write to the SD card, the write actions would perform as expected, but no data would be written; instead the sector that was to be written to was erased. As no dedicated erase function had be developed, this was expected to be the FPGA attempting to write to the SD card, not meeting the power requirements to successfully write. This would cause the data seen from the SD card to be all logic low and the write process to complete successfully, thus throwing no errors. Changing the FPGA power supply from USB to the mains outlet resolved these issues.

```
when WAIT_FOR_HOST_RW =>  -- Wait for the host to read or write a block of data from the SD card.
  clkDivider_v     := SCLK_PHASE_PERIOD_C - 1;  -- Set SPI clock frequency for normal operation.
  getCmdResponse_v := true;  -- Get R1 response to any commands issued to the SD card.
  if rd_i = '1' then  -- send READ command and address to the SD card.
    cs_bo <= '0';              -- Enable the SD card.
    if continue_i = '1' then  -- Multi-block read. Use stored address.
      if CARD_TYPE_G = SD_CARD_E then  -- SD cards use byte-addressing,
        addr_v := addr_v + BLOCK_SIZE_G;  -- so add block-size to get next block address.
      else                     -- SDHC cards use block-addressing,
        addr_v := addr_v + 1;  -- so just increment current block address.
      end if;
      txCmd_v := READ_BLK_CMD_C & std_logic_vector(addr_v) & FAKE_CRC_C;
    else                       -- Single-block read.
      txCmd_v := READ_BLK_CMD_C & addr_i & FAKE_CRC_C;  -- Use address supplied by host.
      addr_v  := unsigned(addr_i);  -- Store address for multi-block operations.
    end if;
    bitCnt_v   := txCmd_v'length;  -- Set bit counter to the size of the command.
    byteCnt_v  := RD_BLK_SZ_C;
    state_v    := START_TX;  -- Go to FSM subroutine to send the command.
    rtnState_v := RD_BLK;  -- Then go to this state to read the data block.
  elsif wr_i = '1' then  -- send WRITE command and address to the SD card.
    cs_bo <= '0';              -- Enable the SD card.
    if continue_i = '1' then  -- Multi-block write. Use stored address.
      if CARD_TYPE_G = SD_CARD_E then  -- SD cards use byte-addressing,
        addr_v := addr_v + BLOCK_SIZE_G;  -- so add block-size to get next block address.
      else                     -- SDHC cards use block-addressing,
        addr_v := addr_v + 1;  -- so just increment current block address.
      end if;
      txCmd_v := WRITE_BLK_CMD_C & std_logic_vector(addr_v) & FAKE_CRC_C;
    else                       -- Single-block write.
      txCmd_v := WRITE_BLK_CMD_C & addr_i & FAKE_CRC_C;  -- Use address supplied by host.
      addr_v  := unsigned(addr_i);  -- Store address for multi-block operations.
    end if;
    bitCnt_v   := txCmd_v'length;  -- Set bit counter to the size of the command.
    byteCnt_v  := WR_BLK_SZ_C;    -- Set number of bytes to write.
    state_v    := START_TX;  -- Go to this FSM subroutine to send the command ...
    rtnState_v := WR_BLK;  -- then go to this state to write the data block.
  else              -- Do nothing and wait for command from host.
    cs_bo   <= '1';            -- Deselect the SD card.
    busy_o  <= '0';  -- SD card interface is waiting for R/W from host, so it's not busy.
    state_v := WAIT_FOR_HOST_RW;  -- Keep waiting for command from host.
  end if;
```

FIGURE 6.6: The SD card idle state code.

```
if reset_i = '1' then             -- Perform a reset.
  state_v          := START_INIT;  -- Send the FSM to the initialization entry-point.
  sclkPhaseTimer_v := 0;  -- Don't delay the initialization right after reset.
  busy_o           <= '1';  -- Busy while the SD card interface is being initialized.
```

FIGURE 6.7: The reset SD card code.

```
when START_INIT =>  -- Deselect the SD card and send it a bunch of clock pulses with MOSI high.
  error_o          <= (others => '0');  -- Clear error flags.
  clkDivider_v     := INIT_SCLK_PHASE_PERIOD_C - 1;  -- Use slow SPI clock freq during init.
  sclkPhaseTimer_v := INIT_SCLK_PHASE_PERIOD_C - 1;  -- and set the duration of the next clock phase.
  sclk_r           <= '0';     -- Start with low clock to the SD card.
  hndShk_r         <= '0';     -- Initialize handshake signal.
  addr_v           := (others => '0');  -- Initialize address.
  rtnData_v        := false;  -- No data is returned to host during initialization.
  bitCnt_v         := NUM_INIT_CLKS_C;  -- Generate this many clock pulses.
  state_v          := DESELECT;  -- De-select the SD card and pulse SCLK.
  rtnState_v       := SEND_CMD0;  -- Then go to this state after the clock pulses are done.
```

FIGURE 6.8: The SD card initial start up code.

```
when DESELECT =>  -- De-select the SD card and send some clock pulses (Must enter with sclk at zero.)
  doDeselect_v     := false;  -- Once the de-select is done, clear the flag that caused it.
  cs_bo            <= '1';     -- De-select the SD card.
  mosi_o           <= '1';  -- Keep the data input of the SD card pulled high.
  state_v          := PULSE_SCLK;  -- Pulse the clock so the SD card will see the de-select.
  sclk_r           <= '0';  -- Clock is set low so the next rising edge will see the new CS and MOSI
  sclkPhaseTimer_v := clkDivider_v;  -- Set the duration of the next clock phase.
```

FIGURE 6.9: The deselect SD card code.

```
458 458                 state_v    := START_TX;  -- Go to this FSM subroutine to send the command ...
459 459                 rtnState_v := WR_BLK;  -- then go to this state to write the data block.
460 460               else              -- Do nothing and wait for command from host.
461     -               cs_bo   <= '1';            -- Deselect the SD card.
    461 + --              cs_bo   <= '1';            -- Deselect the SD card.
462 462                 busy_o  <= '0';  -- SD card interface is waiting for R/W from host, so it's not busy.
463 463                 state_v := WAIT_FOR_HOST_RW;  -- Keep waiting for command from host.
464 464               end if;
    ❄ @@ -573,7 +573,13 @@ begin
573 573               end if;
574 574               sclk_r           <= not sclk_r;    -- Toggle the SPI clock...
575 575               sclkPhaseTimer_v := clkDivider_v;  -- and set the duration of the next clock phase.
576     -
    576 +           if clear_error = '1' then
    577 +             sclk_r     <= '0';
    578 +             bitCnt_v   := 2;
    579 +             state_v    := DESELECT;
    580 +             rtnState_v := WAIT_FOR_HOST_RW;
    581 +           end if;
    582 +
577 583             when RX_BITS =>              -- Receive bits from the SD card.
578 584               if sclk_r = '1' then    -- Bits enter after the rising edge of SCLK.
579 585                 rx_v := rx_v(rx_v'high-1 downto 0) & miso_i;
    ❄ @@ -598,7 +604,10 @@ begin
598 604
599 605             when DESELECT =>  -- De-select the SD card and send some clock pulses (Must enter with sclk at zero.)
600 606               doDeselect_v     := false;  -- Once the de-select is done, clear the flag that caused it.
601     -           cs_bo            <= '1';     -- De-select the SD card.
    607 + --            cs_bo            <= '1';     -- De-select the SD card.
    608 +           -- PGS XXX Make it controllable whether we deselect card when done
    609 +           -- or not.
    610 +           cs_bo <= clear_error;
602 611               mosi_o           <= '1';  -- Keep the data input of the SD card pulled high.
603 612               state_v          := PULSE_SCLK;  -- Pulse the clock so the SD card will see the de-select.
604 613               sclk_r           <= '0';  -- Clock is set low so the next rising edge will see the new CS and MOSI
```

FIGURE 6.10: The fixed read, write and idle code for the SD card controller.

```
hyper_trap_combined <= hyper_trap and monitor_hyper_trap;
```

FIGURE 6.11: The combining of the keyboard hypervisor trap and the monitor hypervisor trap.

6.4 Secure Compartment Implementation

6.4.1 Secure Mode Trap

As discussed in section 6.2, in order for the user to enter secure mode a hypervisor trap needs to be sent to the CPU. All hypervisor traps work in a single standard way, a set of conditions, usually a key combination, pulls the hypervisor trap signal low. This signal is then fed out of the triggering hardware where it is combined with the the other hypervisor traps to form a singular, active low, hypervisor trap. This hypervisor trap signal, seen in figure 6.11, is then sent to the CPU, where it is used to determine the type of hypervisor trap occurring. As seen in figure 6.12, this trap input is then used to generate an edge signal which is further used to latch the hypervisor trap pending signal. This prevents the hypervisor trap from being lost if it is not active for long enough. This is then used in figure 6.13 to cause the next instruction of the CPU to enter the hypervisor mode. When entering the hypervisor the secure mode bit is checked to see if secure mode is already active, this is used for exiting secure mode and further discussed in section 6.4.4. As seen in figure 6.14, when not in secure mode, the hypervisor_trap_port signal is used as a pointer to set the program counter to the desired hypervisor trap location in memory. These memory locations start at $8000 and the secure mode trap is trap 11, which starts at memory location $8044 and can be seen in figure 6.15. This location in memory is where the next stage of the secure mode process, as described in section 6.4.2, will take place.

6.4.2 Save and Load State

Once trapped to the hypervisor, as described in section 6.2 and seen in figure 6.1, the io registers, RAM and ROM all need to be saved to a save state slot. Upon entering the hypervisor and attempting to create a save state, as seen in figure 6.16, nothing occurs at all. All that happens is that access to the cartridge is disabled, the CPU is forced into 50MHz mode and the CPU mode is forced into C65 mode. Then the secure mode and matrix mode flags are set high. This is not how the saving should occur in the final design. In

```
                              --Check for system-generated traps (matrix mode, and double tap restore)
 if hyper_trap = '0' and hyper_trap_last = '1' then
   hyper_trap_edge <= '1';
 else
   hyper_trap_edge <= '0';
 end if;
 hyper_trap_last <= hyper_trap;
 if (hyper_trap_edge = '1' or matrix_trap_in ='1' or hyper_trap_f011_read = '1' or hyper_trap_f011_write = '1')
   and hyper_trap_state = '1' then
   hyper_trap_state <= '0';
   hyper_trap_pending <= '1';
   if matrix_trap_in='1' then
     matrix_trap_pending <='1';
   elsif hyper_trap_f011_read='1' then
     f011_read_trap_pending <='1';
   elsif hyper_trap_f011_write='1' then
     f011_write_trap_pending <='1';
   end if;
 else
   hyper_trap_state <= '1';
 end if;
```

FIGURE 6.12: The hypervisor trap edge generation, pending trap generation and special trap input generation.

```
when InstructionFetch =>
  if (hypervisor_mode='0')
    and ((irq_pending='1' and flag_i='0') or nmi_pending='1')
    and (monitor_irq_inhibit='0') then
                              -- An interrupt has occurred
    pc_inc := '0';
    state <= Interrupt;
                              -- Make sure reg_instruction /= I_BRK, so that B flag is not
                              -- erroneously set.
    reg_instruction <= I_SEI;
  elsif (hyper_trap_pending = '1' and hypervisor_mode='0') then
                              -- Trap to hypervisor
    hyper_trap_pending <= '0';
    state <= TrapToHypervisor;
```

FIGURE 6.13: The use of the hyper_trap_pending flag to enter the hypervisor mode.

```
if hyper_protected_hardware(7)='0' then
  reg_pc(8 downto 2) <= hypervisor_trap_port;
else
  -- Any hypervisor trap from in secure mode instead calls one
  -- specific trap, and automatically generates a signal to the
  -- monitor interface to say that the user wishes to exit the
  -- secure compartment.
  -- Trap $11 is enter secure mode, and trap $12 is exit.
  -- $12 x 4 = $48
  reg_pc(15 downto 0) <= x"8048";
  -- But we also clear the secure mode flag on the CPU, and
  -- reactivate matrix mode, so that the monitor can triage the exit
  -- before the hypervisor gets activated again.
  -- (this will implicitly cause the processor to stop, because the
  -- monitor will be indicating secure mode to us, until such time
  -- as it receives and ACCEPT or REJECT command.
  hyper_protected_hardware(7 downto 6) <= "01";
end if;
```

FIGURE 6.14: The statement that determines the type of hypervisor trap while entering hypervisor mode.

```
; Traps $10-$17 (user callable)
;
jmp nosuchtrap
nop
jmp securemode_trap
nop
jmp leave_securemode_trap
nop
jmp nosuchtrap
nop
jmp nosuchtrap
nop
jmp nosuchtrap
nop
jmp nosuchtrap
nop
jmp nosuchtrap
nop
```

FIGURE 6.15: The secure mode hypervisor trap service call.

```
securemode_trap:

; XXX - The following is what we SHOULD do for the complete system to work:

        ; XXX Freeze current process to slot

        ; XXX Find the requested service

        ; XXX Load the requested service

        ; Set secure mode flag, and set PC and memory map in the secure service

; XXX - What we WILL do for now, is just enable secure mode, and set the PC to
; $8000.

        ; First, disable access to cartridge, force 50MHz mode and 4502 CPU personality
        LDA     #$32
        STA     hypervisor_feature_enables
        ; Second, disable all protecteed IO access, and mark matrix mode and secure mode.
        ; This also freezes the CPU until the monitor acknowledges that the CPU is in
        ; secure mode.  Only after that will the remainder of this routine proceed,
        ; and thus allow the secure program to run.
        ; XXX - This means that a little piece of the hypervisor is still running when we
        ; go into the secure compartment.  For this reason, the CPU needs to be blocked
        ; from writing to hypervisor_secure_mode_flags when in that state.
        LDA     #$c0
        STA     hypervisor_secure_mode_flags
```

FIGURE 6.16

the final design saving should occur similar to the saving done in the freeze routine shown in figures 6.17, 6.18 and 6.19. In this process the a freeze slot is specified, this would be how the user selects the secure service they wish to load. From this point the slot is checked to see if it is valid. Then the SD card is prepared for writing and finally the slot in the SD card is written to.

When loading a state, as seen in figure 6.20, nothing happens. This is due to the unfreezing not yet being implemented in the MEGA65. This is not what should occur in the final design. In the final design loading a save state from the SD card should be implemented as the similar to saving. First the CPU should be forced into the 50MHz and C65 modes with the cartridge disabled. Then the most recent save slot should be selected. This ensures that the program the user was executing prior to secure mode is loaded after secure mode. This slot should then be validated and the SD card should be prepared for reading. Finally the SD card should be read from.

```
freeze_to_slot:
       ; Freeze current running process to the specified slot

       ; Slot in XXYY

       jsr syspart_locate_freezeslot
       bcc ffail
       jmp freeze_save_to_sdcard_immediate
ffail: rts
```

FIGURE 6.17

```
 syspart_locate_freezeslot:
        ; Get the first sector of a given freeze slot
        ; X = low byte of slot #
        ; Y = high byte of slot #

        phx
        phy

        ; Check that we have a system partition
        lda syspart_present
        bne splf1
        lda #syspart_error_nosyspart
        sta syspart_error_code
        rts
```

FIGURE 6.18

```
freeze_save_to_sdcard_immediate:

       ; Save the current process to the SD card. $D681-4 are expecxted to
       ; already be pointing to the first sector of the freeze slot

       ; Stash SD card registers to scratch area
       jsr do_freeze_prep_sdcard_regs_to_scratch

       ; Save current SD card sector buffer contents
       jsr freeze_write_sector_and_wait

       ; Save each region in the list
       ldx #$00
freeze_next_region:
       jsr freeze_save_region
       txa
       clc
       adc #$08
       tax
       lda freeze_mem_list+7,x
       cmp #$ff
       bne freeze_next_region

       rts
```

FIGURE 6.19

```
1   ;      Un-freeze routines.
2   ;      These routines reverse the process of freezing a task.
3   ;      Un-freezing has some special challenges, because we have to
4   ;      restore the state exactly, including that of the SD card
5   ;      interface.  In practice this means that we need to do a little
6   ;      bit of post-loading tweaking, in addition to simply loading the
7   ;      previously frozen regions.
8   ;      Obviously for compatibility with the freeze process, we need to
9   ;      match the order of actions with that.
10
11  ; XXX - Copy sector buffer and SD and DMAgic regs to spare space in 4KB SD sector buffer BRAM first,
12  ; then just need that stash and unstash routine, and the rest can remain completely orthoganal.
```

FIGURE 6.20

6.4.3 Secure Mode Entry

With the io registers, RAM and ROM all saved and the secure service loaded into RAM and ROM while preserving the transfer area, entry into secure mode can begin. This starts with the hypervisor toggling the secure mode and matrix mode bits of the protected hardware flags. As soon as there is a mismatch between the monitor secure mode flag and the CPU secure mode flag, as seen in figure 6.23 the CPU is halted. This allows the secure mode signal to propagate out of the CPU to the monitor.

The monitor is like the Intel Management Engine (IME), but opposed to the IME being a service CPU, the monitor is a servant CPU. The key difference between the two being that where the IME is able to autonomously perform commands, the monitor CPU requires the user to issue commands. The monitor CPU was introduced due to hardware limitations of the FPGA. Previously the monitor was entirely hardware with no CPU at all, this took up lots of space on the FPGA and would prevent the rest of the phone from being able to fit. By converting the monitor into a servant CPU the large amount of specialised hardware was able to be condensed into a multipurpose unit.

As seen in figure 6.21, when the secure mode flag reaches the monitor CPU, it is eventually scanned and used to branch and enter the secure mode routine. In this routine, as seen in figure 6.22, currently only the local secure mode variable is set high and the routine returns to figure 6.21. In the future iterations, an indicator as to how much memory the transfer area occupies will need to be implemented.

In parallel to the monitor functions, the protected hardware signal is fed into the matrix mode compositor. As seen in figure 6.27, this displays the secure mode screen entirely independently from the secure mode verification; it also locks down the inputs and outputs from the MEGA65. Once the monitor CPU verification has completed, as seen in figure 6.24 the user is required to enter "ACCEPT" or "REJECT" to approve or disapprove entry into the secure container. In the "ACCEPT" case, this will parse the local secure mode variable out and to the CPU. In the "REJECT" case, the io registers, RAM and ROM are be restored from the save state created in section 6.4.3 and then the local variable for secure mode will be parsed to the CPU. The "ACCEPT" case can be seen in figure 6.25 and the "REJECT" case can be seen

in figure 6.26.

While the "ACCEPT" case has been fully implemented, the "REJECT" case has not been implemented at all. Figure 6.26 shows that in the "REJECT" case, all that is occurring is the memory is being erased. Once the memory has been erased, the "REJECT" case continues on into the "ACCEPT" case where the local secure mode value is sent to the CPU, entering secure mode anyway. This is not how the check should function. In the future this reject command should be dependant on the state of the local secure mode bit. If this bit is high, hence we are attempting to enter secure mode, then memory should not be erased. The "REJECT" case should instead retrigger the hypervisor and bypass the exit secure mode confirmation, causing the hypervisor to reload the save state created in section 6.4.2 once the CPU is resumed.

Once the input has been read and either the "ACCEPT" case or the "REJECT" case has been recognised and acted upon, the monitor will then leave the matrix mode screen, as seen in figure 6.25. In addition to leaving the matrix mode screen, the CPU is unhalted, this allows the secure service or loaded save state to function.

6.4.4 Secure Mode Exit

While in secure mode, any hypervisor trap will be seen as a request for exit from secure mode. This, as seen in figure 6.28, overwrites the hypervisor trap determination seen in figure 6.13 with the exact location of the exit secure mode trap and sets the matrix mode flag high and secure mode flag low. Once again the CPU is halted due to the mismatch seen in figure 6.23 between the monitor secure mode state and the CPU secure mode state.

The code seen in figure 6.21 is once again run, but this time branches to figure 6.29. As in section 6.4.3, while the monitor CPU is performing the verification check, the matrix mode compositor is displaying the secure mode exit screen and prompting the user to enter "ACCEPT" or "REJECT". As seen in figures 6.24 and 6.25, in the "ACCEPT" case the CPU is allowed to run by setting the monitor secure mode state to be identical to the CPU. In the "REJECT" case, demonstrated by figure 6.26, all the top megabyte of address space is erased and io address registers are also erased.

```
check_protected_hw:
      bit         protected_hw
      bmi        maybe_enter_secure_mode
      lda        in_secure_mode
      bne        leave_secure_mode
      bvs         maybe_enter_matrix_mode
      lda         #0
update_protected_hw:
      sta         in_matrix_mode
check_protected_hw_exit:
      rts
leave_secure_mode:
;      `printstr       securemode_exit_msg
       lda     #0
       sta     in_secure_mode
       rts
```

FIGURE 6.21: The routine that checks the protected hardware to determine if the phone should be in secure mode

```
maybe_enter_secure_mode:
      lda        in_secure_mode
      bne        +
      ; XXX - We should indicate how much space is transfer
;      `printstr         securemode_entry_msg
*     lda        #$80
      sta        in_secure_mode
      rts
maybe_enter_matrix_mode:
      lda         in_matrix_mode
      bne         check_protected_hw_exit
      `printstr   banner_msg
      lda         #1
      bra         update_protected_hw
```

FIGURE 6.22: The routine that sets the secure mode flag to non-zero value.

```
if rising_edge(clock) then
  -- We this awkward comparison because GHDL seems to think secure_mode_from_monitor='U'
  -- initially, even though it gets initialised to '0' explicitly
  if (hyper_protected_hardware(7)='1' and secure_mode_from_monitor='0')
    or (hyper_protected_hardware(7)='0' and secure_mode_from_monitor='1')
  then
    -- Hold CPU completely paused if CPU and monitor disagree on whether we
    -- are in secure mode or not.  This is how the CPU is held when switching
    -- to and from secure mode.
    report "SECUREMODE: Holding CPU paused because cpusecure=" & std_logic'image(hyper_protected_hardware(7))
      & ", but monitorsecure=" & std_logic'image(secure_mode_from_monitor);
    io_settle_delay <= '1';
```

FIGURE 6.23: The CPU halting code that halts when the CPU and monitor disagree on if the phone is in secure mode.

```
; Check if "ACCEPT" or "REJECT" were typed.
accept_or_reject_command:

        ldx     #0
*       lda     cmdbuf,x
        cmp     accept_string,x
        bne     +
        inx
        cpx     #6
        bne     -
        jmp     accept_command
*       ldx     #0
*       lda     cmdbuf,x
        cmp     reject_string,x
        bne     +
        inx
        cpx     #6
        bne     -
        jmp     reject_command
*       bra     not_accept_or_reject

accept_string: .byte "ACCEPT"
reject_string: .byte "REJECT"
```

FIGURE 6.24: The monitor CPU code that compares the input string against a string containing "ACCEPT" and one containing "REJECT".

```
accept_command:

        ; Accept is like reject, except that we don't erase memory first.
        ; It is up to the hypervisor to spot the difference.

        ; Resume CPU by confirming mode change to/from secure mode
        lda   in_secure_mode
        sta   $901c
        ; And then cancel matrix mode
        sta   $900a

        jmp     next_command
```

FIGURE 6.25: The monitor CPU "ACCEPT" case.

```
reject_command:
        ; Erase all memory
        jsr     erase_memory
        ; Return empty secure container back to hypervisor
        ; (The hypervisor should already have been triggered, and the CPU stopped,
        ;  so all we should need to do is to resume the CPU)
```

FIGURE 6.26: The monitor CPU "REJECT" case.

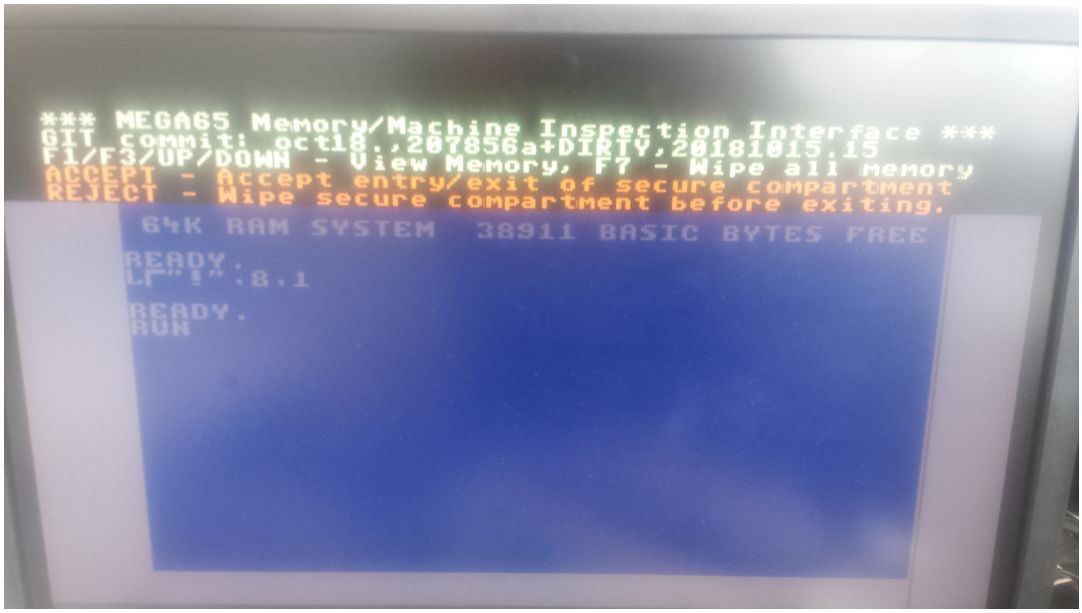


FIGURE 6.27: The secure mode interface.

```
if hyper_protected_hardware(7)='0' then
  reg_pc(8 downto 2) <= hypervisor_trap_port;
else
  -- Any hypervisor trap from in secure mode instead calls one
  -- specific trap, and automatically generates a signal to the
  -- monitor interface to say that the user wishes to exit the
  -- secure compartment.
  -- Trap $11 is enter secure mode, and trap $12 is exit.
  -- $12 x 4 = $48
  reg_pc(15 downto 0) <= x"8048";
  -- But we also clear the secure mode flag on the CPU, and
  -- reactivate matrix mode, so that the monitor can triage the exit
  -- before the hypervisor gets activated again.
  -- (this will implicitly cause the processor to stop, because the
  -- monitor will be indicating secure mode to us, until such time
  -- as it receives and ACCEPT or REJECT command.
  hyper_protected_hardware(7 downto 6) <= "01";
end if;
```

FIGURE 6.28: The CPU hypervisor trap determination code.

The "REJECT" case, as seen in figure 6.30, has not entirely been implemented. Currently, the "REJECT" case is not dependant on the secure mode bit, causing it to call the erase memory function on entry or exit. In the future it needs to overwrite the transfer area RAM only when the local secure mode bit is low, hence it is leaving secure mode. Once all the erasing has been done, the "REJECT" case can continue, as implemented, into the "ACCEPT" case.

Like the "REJECT" case, the "ACCEPT" case needs to be fully implemented. Currently, accepting a leave request from secure mode allows all memory to escape the secure container. In future development, the io registers, ROM and non-transfer area RAM need to be erased. This, when compounded with the "REJECT" case would erase all memory when rejecting exit from a secure container, but would only erase non-transfer area memory when accepting exit from a secure container. This "ACCEPT" case would also require some modification in it's current state to function differently when entering and exiting secure mode. As when entering secure mode, memory does not need to be erased.

6.4.5 Restore User Mode

Once either the "ACCEPT" or the "REJECT" actions have occurred and the CPU is running again, the hypervisor then is able to restore to the user program. As seen in figure 6.31, restoring to the user session has not yet been implemented. In the future, what should occur in this trap is the io registers, non-transfer

```
leave_secure_mode:
;       `printstr       securemode_exit_msg
        lda     #0
        sta     in_secure_mode
        rts
```

FIGURE 6.29: The monitor CPU leave secure mode routine.

```
erase_memory:
       ; XXX - Implement me.
       ; Erase all RAM.
       ; Erase all IO registers (including all banks of palette, and
       ; all the other funny bits, like ethernet frame buffer, SD card buffers etc.
       ; basically we have to prevent ANY state being leaked out.  This is actually
       ; the hardest part of the whole system to get right.

       ; 1. Fill the first 1MB of address space
       lda #$00
       ldx #3
*      sta mem_addr,x
       sta top_addr,x
       dex
       bpl -
       ; Set top to 1MB mark
       ldy #$10
       sty top_addr+2
       jsr fill_loop

       ; 2. Fill the IO address space $FF00000-$FFEFFFF
       ; ($FFFxxxx has the hypervisor memory, which we need to leave alone)
       ;
       lda #$0F
       sta mem_addr+3
       sta top_addr+3
       lda #$f0
       sta mem_addr+2
       ldy #$fe
       sty top_addr+2
       jsr fill_loop

       ; 3. XXX - Fill other colour palettes and other funny memories around
       ; the place.
       ; XXX - We could lock colour palette bank in secure mode to make life
       ; simpler for us here.

       rts
```

FIGURE 6.30: The monitor CPU erase routine.

```
leave_securemode_trap:

        ; If we get here, we have left a secure compartment, with either memory erased
        ; or intact.  Either way, we should hand control back to the user, and disable
        ; matrix mode display.

        ; XXX - Debug
        inc $d021

        lda     #$00
        sta     hypervisor_secure_mode_flags

        jmp     nosuchtrap
```

FIGURE 6.31: The hypervisor leave secure mode trap.

area RAM and all of ROM should be loaded from the save state created in section 6.4.2.

6.5 Secure Compartment Considerations

6.5.1 Hypervisor

During the development of the secure compartments, a possible issue with the design was made apparent by Dr. Gardner-Stephen. Originally, when entering or exiting secure mode, the hypervisor was in complete control of the phone's operation. This posed a problem thanks to the inability to transition between secure mode and the hypervisor instantaneously. Between the loading of the secure service and the entering of secure mode, there was a small window where the user was able to perform any actions they desired. To correct this issue, control over the halting of the CPU was given to the monitor and was performed while in the hypervisor trap. This allowed the halt to occur before leaving the hypervisor, locking the user out of running code in the small window. At the same time to the monitor halting the CPU, the hypervisor is forced to return to user mode, as seen in figure 6.32. This system takes control away from the user and hypervisor, making the action of entering secure mode atomic.

6.5.2 Monitor CPU

When using the monitor CPU it is especially important that the capabilities of this CPU be considered in terms of security. As erroneously allowing

```
if last_write_address = x"FFD3672" and hypervisor_mode='1' then
  hyper_protected_hardware <= last_value;
  if last_value(7)='1' then
    -- If we attempt to enter secure mode, then we are forced out of
    -- the hypervisor, to make sure that the hypervisor cannot do
    -- naughty things to the secure container, like re-enable IO
    -- devices.
    state <= ReturnFromHypervisor;
  end if;
  if last_value(6)='1' then
    matrix_rain_seed <= cycle_counter(15 downto 0);
  end if;
end if;
```

FIGURE 6.32: If the CPU is in secure mode and hypervisor mode, the CPU is forced from hypervisor mode.

this CPU free access could undermine the security centred design of the MEGA65. With this in mind, the monitor CPU was handled as a servant CPU to the greater MEGA65 system. They key features of this CPU that makes it a servant rather than a service, is the manner in which it operates. The monitor CPU is entirely controlled by the user and only the user. As seen in figure 6.33, the monitor CPU has been implemented in a simple single loop. This loop only checks the protected hardware bits and scans pressed keys which are then acted upon. This results in the monitor CPU being entirely user controlled, as the only ways to interact with it are through the keyboard. In addition to being user controlled, this CPU has it's own memory entirely separate from the main CPU. This memory is also entirely write protected. The benefit of having the monitor CPU in its own ROM is that it is now more simple and therefore easily verified. Additionally, ROM can not be overwritten and thus the monitor CPU is immune to a wide range of malicious behaviour.

6.6 Final Thoughts

The secure compartmentalisation of the MEGA65 has almost entirely been implemented in the hardware aspects. The hypervisor trap, secure mode entry/exit and monitor CPU have all been implemented successfully. The software side still requires some work. While the saving only requires the relevant software to be referenced, the loading has not even been started and the accepting/rejecting of secure compartments still needs their secondary cases implemented. Overall work is still required in these aspects for the

```
accepting_input:
      jsr         check_protected_hw

; XXX - When we are in secure mode, we should ignore the UART, only
; read from the keyboard
      lda        in_matrix_mode
      bne        +
      bit         uart_status
      bmi         get_uart_char
; Check for keyboard input
*     lda        uart_status
      and        #$20
      bne        get_keyboard_char

      bit         monitor_char_status ; Anything from the monitor?
      bmi         get_monitor_char

*     bit         mon_trace         ; See if we got a watch or break match
      bvc         +
      jmp         watch_match
*     bpl         +
      jmp         break_match

      ; Nothing else to do, see if we're doing continuous tracing
*     lda         trace_continuous
      beq         +
      jmp         trace_step

*     bra         accepting_input
```

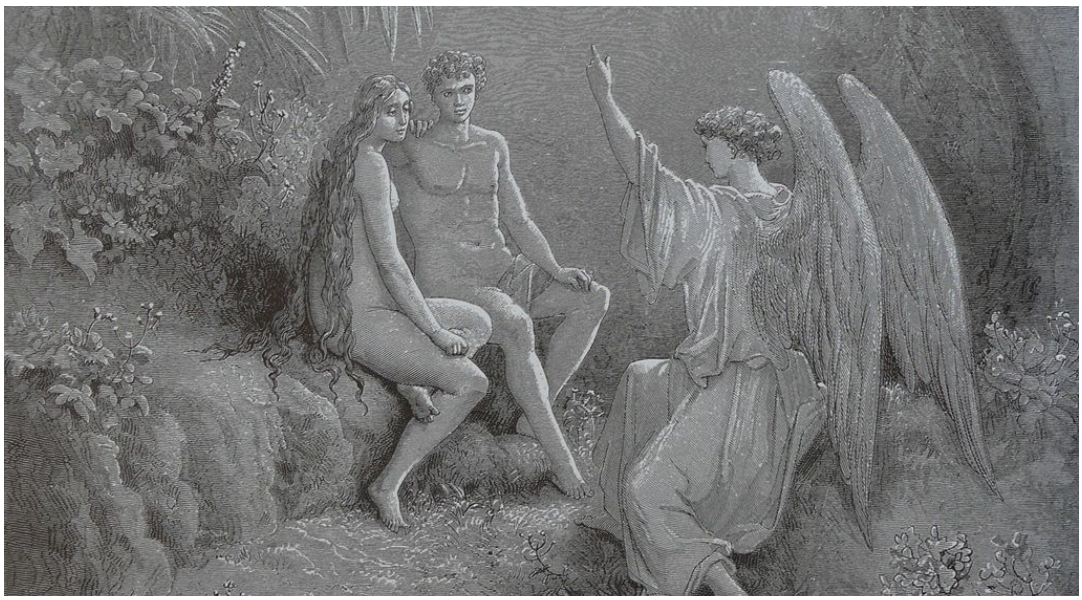
FIGURE 6.33: The main monitor CPU execution loop.

secure compartmentalisation to be complete.

In addition to this, any work done on the secure compartmentalisation should be done with a security focus. The hypervisor exiting is a great example of this security focus; instead of allowing it to exit naturally the CPU is stopped and it is forced out. This prevents anything from executing between the hypervisor and the entry to secure mode. The monitor CPU is another good example. It is as simple as it can be while performing the required tasks and only accepts input from the user.

Chapter 7

Results and Discussion



7.1 Overview

In the following chapter what was achieved over the project will be discussed. This will begin with the challenges encountered during and final state of the matrix mode and secure compartment sections. This will then continue on to discussing the research questions and will end with a discussion about the future works of the project regarding matrix mode and the secure compartments.

7.2 Matrix Mode Operation

When initially starting the project, matrix mode was in a unusable state. The serial interface with the device was locked after input, only allowing a single character to be read. In addition to the locking, there were several visual issues that inhibited the function of matrix mode. While these issues

were corrected over the course of the project, they did come with their own challenges along the way.

7.2.1 New Developer Issue

The biggest issue encountered when attempting to correct any of the parts of matrix mode was the lack of comments. This project has multiple developers in it and while some sections are well documented, there are some sections that are not obvious to new developers.
In addition to the lack of foundation available, the range of development tools in use in this project requires an advanced knowledge of programming and hardware design. Throughout this project the design language VHDL was the main language used, but despite this a detailed knowledge was required of Verilog, GS4510 assembly, Vivado tool command language (tcl) and Make. As I had no knowledge of the latter languages this made some sections of the project difficult to design.
Despite these challenges changes to the tcl, Make and Assembly files were able to be done and an understanding of the Verilog files was developed.

7.2.2 Serial Locking Issues

During the correction of the serial locking issue, correcting the offending code proved to be very challenging. As the MEGA65 did not have all the excessive resources of modern devices, after identifying the issue, it was difficult to implement a solution to the problem that was efficient. This lead to brainstorming of possible corrections and their security implications. In the end, the solution that was favoured was the edit code that was already present, adding the minimal amount of my own code. This issue was entirely overcome after implementing the first change to the MEGA65. Upon suggesting my change to Dr. Gardner-Stephen I was informed that any change made should be added to the project. This was as the optimisation of the device had not yet occurred, so any changes I make will inevitably be improved.

7.2.3 Letterboxing

During the correction of the visual issues in matrix mode through a letterbox signal, a lack of project knowledge proved to be very tough to overcome. This lack of knowledge resulted in several attempts to correct the issue through offsets and other, ultimately useless, tricks to move the terminal memory

around. If a greater knowledge of the other sub-systems in the MEGA65 was possessed then the corrections to the device would have proceeded more rapidly.

7.2.4 Visual Bug Chasing

During corrections to the visual bugs, the biggest challenges faced were the exposing of signals and understanding of what was happening. While running the hardware, exposure of the signals was difficult due to the limited amount of LEDs on the board. This made identification of some of the bugs slow. To combat this GHDL, a VHDL simulator, was used. This exposed the signals more readily as there were report statement littered through the project, this would make signal tracking trivial. GHDL, unfortunately, did not provide a graphical display of what was occurring on the hardware, making discrimination of expected behaviour from unexpected behaviour difficult.

7.3 Secure Compartment Implementation

7.3.1 SD card Fixing

Fixing the SD card functions, as described in section 6.3.2, was where the first major issue in the project was encountered. This was due to the multiple branches of knowledge required to even verify id the SD card was working or not. Firstly, a knowledge of the VHDL files was required, in addition to this, knowledge of where the SD card was saving and reading from was also required. This allowed me to understand how reading and writing was occurring. But to check that they were occurring correctly, I needed to understand and use assembly to query the CPU through the matrix mode commands. This allowed me to manually write, read and reset the SD card.

7.3.2 Hypervisor Trap

During the creation of the hypervisor trap for secure mode a communication error occurred between Dr. Gardner-Stephen and I. I believed that the hypervisor traps were all done in hardware, as there were hardware implementations for some hypervisor traps in the CPU. This was not the case, the majority of hypervisor traps were to be executed in software. This miscommunication resulted in a time loss on the project.

7.3.3 Secure mode Entry/Exit

During implementation of the secure mode entry and exit the biggest challenge of the project was encountered. The sequencing for events and interactions between the various sub-systems required very careful consideration. As discussed in section 6.5.1, in order to eliminate the possibilities of exploitation, control needed to be taken away from the user to make certain processes atomic. This posed such a problem as my inexperience in security caused me to miss possible exploitation avenues. It was actually Dr. Gardner-Stephen that brought the issue of the user window between hypervisor and secure mode to my attention.

7.4 Research Question Satisfaction

What are the missing or non-functional sub-systems that the MEGA65 requires to be implemented?
Over the course of the project, several missing and/or non-functional sub-systems were found within the MEGA65 smart phone. Of those sub-systems, the ones that were fixed are detailed in chapters 4, 5 and 6 of this document. Despite the fixes applied, there are still many sub-systems of the phone that require additional attention in order for them to function as intended.

One of the functions of the phone is being able to create and load save states that are images of the phone at a particular point in time. These save states are mostly functional, but there are still some strange cases, in which, their functionality is not entirely as expected or missing. It was noted, but not explored, that creating a save state in the most recent version of the phone causes the partition table to be invalidated. This is suspected to be the result of some change to the save state code causing the slot pointer to point to the partition table instead of the desired slot.

The save state error also has a cascading effect into the implementation of secure mode, as discussed in chapter 6. As described in section 6.4.2, the secure mode implementation plans to use this save state process as a part of it, this error will therefore cause the secure state to corrupt the partition table.

In addition to the partition corrupting, the secure mode is also non-functional for a number of other reasons. The string comparison seen in figure 6.24,

only compares the input to the upper case version of accept or reject. While this could be intentional to ensure that the user truly desires the result, by entering reject in lower case an instruction is invoked. This is due to the lower case 'r' character being an instructional prefix. Another issue with secure mode was when successfully rejecting secure compartments. When the SD card has been removed from the device, there is currently no protection to stop the phone from loading despite this. This causes the secure mode to load from nothing and produces unpredictable results.

Although functionality to matrix mode has mostly been restored, it is not working entirely as intended. From within matrix mode it is possible to halt and single step the CPU, if the CPU is halted, then a request to leave matrix mode is sent and stepped through, the user is left in user mode with a frozen CPU and no way to return to matrix mode. When exiting from matrix mode with the secondary exit keycode, superkey + , the key is printed in user mode. This is believed to be due to double scanning of the non-special input keys.

When experimenting with the SD card compatibility, it was noted that the SDXC card was not recognised as an SD card at all. Investigation into the SD card hardware showed that there was no case for this type of card. In addition to this, in the latest version of the phone, the SD card detection no longer rescans for the SD card if one is not initially detected upon start-up.

How can these sub-systems be implemented?

- Save Sate
  The save state partition table corruption issue should be solved by reviewing the location at which the save state is attempting to be saved to. This location should be confirmed as save state slot. Additionally, the size of the data being saved should be compared to the size of the save state slot to ensure that other data is not being overwritten.
- ACCEPT/REJECT Strings
  To ensure the secure compartment sub-system is functional and in accordance with the user's expectations, the lower case versions of accept and reject should also have the same functionality as the upper case versions. Conversely to this, if the accept and reject cases are restricted to upper

case only, inputting a lower case version should send a warning to the user and, in the reject case, not attempt to execute a command.

- CPU Locking
  To prevent CPU locking and restore some functionality to matrix mode, exiting matrix mode should be tied into unfreezing the CPU.

- SD card Protection
  To protect against loading junk data from the SD card slot when there is no SD card present, a verification should occur prior to reading. If a fail state occurs, the MEGA65 should prompt the user to insert an SD card and specify the slot they wish to load from.

- SDXC Support
  The SDXC SD cards should have their data sheet evaluated and the functionality required to used them added to the SD card handling hardware.

How can the simplicity, understandability and hence, the security of these sub-systems be maximised?
When considering how to ensure the simplicity, understandability and security of each sub-system, it was decided that the UNIX philosophy should be employed. "Do one thing and do it well." This was employed in all aspects mentioned in this document. The matrix rain compositor only places the matrix video over the screen. When performing the secure mode process, only the secure mode trap is being acted upon, only save state function is occurring, only the user's input is being processed, etc. This philosophy removes the obscuration from performing tasks that may otherwise be complex. In addition to this, by using smaller and more simple sub-systems they are inherently more easily understood.
How can the complete MEGA65 architecture be physically prototyped on the bench?
When implementing the MEGA65 architecture there are only a few ways that it can be done. Insanely, the entire architecture could be prototyped on breadboards or prototyping specific circuit boards. This would require a monumental amount of work to verify that the devices are working correctly, would also have to be done manually and changes to the design would not be easily implemented. Another way would be to design a motherboard and have all the sub-modules as separate boards that all connect to it. This solution is not ideal for prototyping as the design of the boards is very difficult to modify, this also causes the time required to make changes excessively

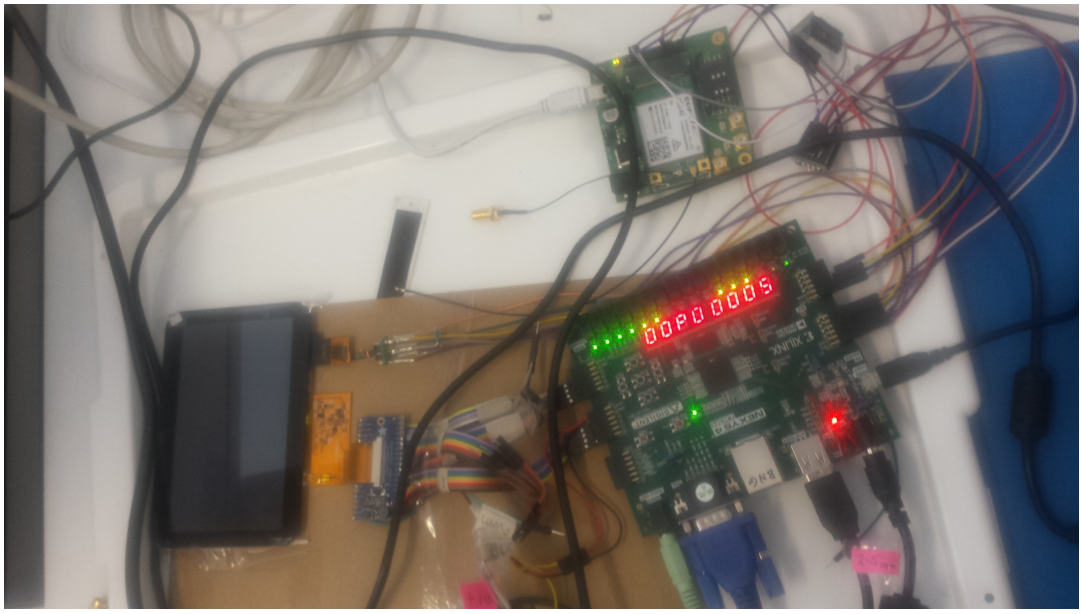


FIGURE 7.1: The MEGA65 bench prototype.

long. Conversely, this method would produce the highest quality product when compared with the other methods. The chosen method was to use an FPGA to map the MEGA65 hardware, which is then supplemented by specific hardware modules. This method makes changes to the design almost trivial to do and would even reduce the time taken to make them. The MEGA65's architecture can be seen prototyped in figure 7.1.

How can the secure compartmentalisation's architecture planned for the MEGA65 be realised?

The planned secure architecture for the MEGA65 can be realised through the combination of hardware and software security protocols. The hardware sections of the compartment were used to perform atomic actions and restriction of access to the MEGA65.

The atomic actions of the phone include:

- Secure mode hypervisor trap triggering.
- Hypervisor switching
- CPU halting
- CPU resuming

While in secure mode all of the external access ports, with the exception of the keyboard, and all of the internal wireless communication devices were restricted from communication. This prevents the escaping of data from the

container.

The software section of the secure compartments were used to perform the actions that required user input, or were not atomic. The user input actions include the accepting/rejecting of entering/exiting the secure compartment. In addition to this, the specification of the secure service that is to be run must also be done in software.
The non-atomic actions of the secure mode process include the saving and loading of the save states as well as the erasing of memory.

7.5 Future Work

Upon completion of the project tenure it is clear that there are still aspect of the project that are unfinished. In it's current state, the secure compartments and matrix mode of the MEGA65 are not ready for the final product of the smart phone.

7.5.1 Matrix mode

In addition to the changes, as described in chapter 5, matrix mode requires some more attention to its functionality. In particular, the exiting of matrix mode with the super and key should be further evaluated as to remove the bug where a is printed upon exit. In addition to this the super and tab key latching should be looked at again. In the current implementation seen in section 6.4.1 a wait state is used between rescanning the keys. This should be replaced with a latching statement that will only scan the combination once and not scan when they are held down.

7.5.2 Secure mode

Secure mode requires lots of additional work in order to be ready for final version of the MEGA65. As described in section 6.4.2, both of the save state functions, such as all of loading and some of saving, need to be implemented into the secure mode process. In addition to this the loading from the freeze commands also needs to be implemented so it can be used in the secure mode process. The "ACCEPT" and "REJECT" commands also require some more work in order for them to be fully functional. While both commands require

modification to them based on the current secure mode state, "ACCEPT" only requires implementation of the secure mode exit case. "REJECT" requires a method for implementing the denial of entry into a secure container when entering secure mode. It also requires memory clearing capabilities when exiting a secure container.

Chapter 8

Conclusion

8.1 Overview

This final chapter will begin by going over the project as a whole and what was achieved during it. Then it will provide an evaluation against the motivation of the project to determine if what was achieved was in line with the aims. Finally, the research questions will be answered.

8.2 Paradise Restored?

Using the techniques described in chapter 4, chapter 5 and chapter 6, the MEGA65 project was made more developer friendly, matrix mode was corrected and secure compartmentalisation was almost entirely implemented. New developers for the project are now able to more easily build and develop the MEGA65 through the use of a single makefile. The MEGA65 is now able to successfully enter and exit matrix mode without disabling any functionality

of the rest of the phone. In addition, all of the visual issues in matrix mode have been eliminated. The secure mode hypervisor trap has been implemented as well as some of the secure mode saving. The accepting and rejecting of a secure container has also been partially implemented and the CPU halting a resuming has been fully implemented.

To truly restore paradise, the following need to be implemented:

- Matrix mode entering and exiting needs to be latched
- Matrix mode exiting needs to stop printing to the screen
- Secure mode needs to save and load save states
- Secure mode entry needs to erase the RAM, ROM and io registers
- Secure mode entry needs to support rejecting entry to a secure container
- Secure mode exit needs to erase RAM, ROM and io registers
- SD card read protection
- SDXC Compatibility

Overall, it can be seen that while significant progress has been made on the project, it is not entirely complete. The additions required and explained in this document will result in a production functional version of matrix mode and a functional version of secure mode. Additionally, it will extend functionality of the MEGA65 to SDXC cards and prevent reading from empty SD card slots.

8.3 Research Question Answers

What are the missing or non-functional sub-systems that the MEGA65 requires to be implemented?
The missing and/or non-functional sub-systems of the MEGA65 are:

- Save state Saving and Loading
- Secure mode Saving and Loading
- Secure mode ACCEPT and REJECT Implementation
- SD card Read Protection

- CPU Lock Protection
- Matrix mode Key Scanning
- SDXC Compatibility

How can these sub-systems be implemented?
These sub-systems can be implemented as follows:

- Save state Saving and Loading
  The save and load states can be implemented in the place holder files. Since the supporting hardware has already been developed, all that is required is the assembly implementations.

- Secure mode Saving and Loading
  The secure mode saving and loading require the implementation of the save state functions. Additionally, they require some of their own assembly code. This code is required in the hypervisor traps and, like the save states, already has the supporting hardware around it.

- Secure mode "ACCEPT" and "REJECT" Implementation
  Much like the previous two points, the accepting and rejecting functionality only requires the software implementation.

- SD card Read Protection
  This can be simply implemented through manipulation of the SD card software. Enabling a software SD card check can be done by reading the partition table and if it is corrupted, disallowing all further writes and reads. In hardware, a similar check could possibly be done, this depends on the functionality of the SD card controller.

- CPU Lock Protection
  In order to prevent accidental locking of the CPU, matrix mode could be tied to the halting and resuming of the CPU.

- Matrix mode Key Scanning
  The matrix mode key scanning can be implemented by implementing a working latch for the enter and exit key combinations. This latch

should cause the iomapper to ignore the held version of these key combinations.

- SDXC Compatibility
  Adding SDXC support, such as reset, read and write protocols, to the existing SD card hardware would enable this functionality.

How can the simplicity, understandability and hence, the security of these sub-systems be maximised?
By implementing the sub-systems in accordance with the UNIX philosophy of "Do one thing and do it well.", it was possible to create highly understandable and secure units that are able to make up the whole that is the MEGA65
How can the complete MEGA65 architecture be physically prototyped on the bench?
The complete architecture of the MEGA65 was prototyped via an FPGA. This FPGA contained the all of the MEGA65 architecture and was supplemented by 3rd party hardware, such as the modem.
How can the secure compartmentalisation's architecture planned for the MEGA65 be realised?
Through a combination of hardware and software the MEGA65's plan for secure compartments can be realised. The hardware can handle the actions that are desired to be atomic, as well as the disabling of data escaping mediums. The software can handle the actions that require user input and that are not atomic.

Bibliography

[1] H. El-Aawar. *Increasing the transistor count by constructing a two later crystal square on a single chip*. Tech. rep. The Lebanese International University, June 2009.

[2] D. Adams. *The History of Mobile Operating Systems [Infographic]*. July 2011. URL: http://www.bitrebels.com/technology/the-history-of-mobile-operating-systems-infographic/. (accessed: 31.05.2018).

[3] Janis Antonovics et al. "The evolution of transmission mode". In: *Phil. Trans. R. Soc. B* 372.1719 (2017), p. 20160083.

[4] Thomas M Chen and Jean-Marc Robert. "The evolution of viruses and worms". In: *Statistical methods in computer security* 1 (2004).

[5] L. Constnatin. *Attackers exploited ColdFusion vulnerability to install Microsoft IIS malware*. Dec. 2013. URL: https://www.pcworld.com/article/2080721/attackers-exploited-coldfusion-vulnerability-to-install-microsoft-iis-malware.html. (accessed: 31.05.2018).

[6] The Department of Defense. *Trusted Computer System Evaluation Criteria*. Tech. rep. The Department of Defense, 1985.

[7] B. Ellis. *Your most dangerous possession? Your smartphone*. Jan. 2011. URL: https://money.cnn.com/2011/01/12/pf/saving/smartphone_dangers/index.htm. (accessed: 31.05.2018).

[8] U. Frisk. *Total Meltdown?* Mar. 2018. URL: http://blog.frizk.net/2018/03/total-meltdown.html?m=1. (accessed: 26.05.2018).

[9] B. Gerblich. "Computers Must be Understandable to be Secure". MA thesis. Adelaide: Flinders University, 2017.

[10] D. Goodin. *Meet "badBIOS," the mysterious Mac and PC malware that jumps airgaps*. Jan. 2013. URL: https://arstechnica.com/information-technology/2013/10/meet-badbios-the-mysterious-mac-and-pc-malware-that-jumps-airgaps/. (accessed: 05.06.2018).

[11] D. Goodin. *Virtual machine escape fetches $105000 at Pwn2Own hacking contest*. Mar. 2017. URL: https://arstechnica.com/information-technology/2017/03/hack-that-escapes-vm-by-exploiting-edge-browser-fetches-105000-at-pwn2own/. (accessed: 5.06.2018).

[12] D. Graham-Smith. *12 ways to hack-proof your smartphone*. Mar. 2017. URL: https://www.theguardian.com/technology/2017/mar/26/12-ways-to-hack-proof-your-smartphone-privacy-data-thieves. (accessed: 31.05.2018).

[13] Intel. *2000 Annual report*. Tech. rep. Intel, 2000.

[14] Intel. *Annual report 1991*. Tech. rep. Intel, 1991.

[15] Intel. *Intel annual report 1972*. Tech. rep. Intel, 1972.

[16] Intel. *Intel corperation annual report 1975*. Tech. rep. Intel, 1975.

[17] Intel. *Intel corperation annual report 1982*. Tech. rep. Intel, 1982.

[18] P. Kervalishvili J. Ramsden. "Security and Complexity". In: IOS Press, 2008, pp. 249–347.

[19] Dr. R. Griffin Jr. *Study on Mobile Device Security*. Tech. rep. Washington: The Department of Homeland Security, Apr. 2017.

[20] E. Kovacs. *"AirHopper" Malware Uses Radio Signals to Steal Data from Isolated Computers*. Oct. 2014. URL: https://www.securityweek.com/airhopper-malware-uses-radio-signals-steal-data-isolated-computers. (accessed: 3.06.2018).

[21] S. Lemon. *Jailbroken IPhones Leave Users More Vulnerable*. July 2009. URL: https://www.pcworld.com/article/167760/article.html. (accessed: 31.05.2018).

[22] et al. M. Guri. *ODINI Escaping Sensitive Data from Faraday-Caged, Air-Gapped Computers via Magnetic Fields*. Tech. rep. The University of Negev, Feb. 2018.

[23] S. Mansfield-Devine. "Security Through Isolation". In: *Computer Fraud and Security* (May 2010), pp. 8–12.

[24] *MEGA65 Design Choices*. Mar. 2018. P. Gardner-Stephen, [Interviewee].

[25] *Meltdown and Spectre*. [Online; accessed 12. Oct. 2018]. Jan. 2018. URL: https://meltdownattack.com.

[26] *Moore's Law*. [Online; accessed 12. Oct. 2018]. URL: http://www.mooreslaw.org.

[27] J. Murphy. *Security Minister Warns of Mobile Phone Hacker Risk*. June 2009. URL: http://go.galegroup.com/ps/i.do?id=GALE%5C%7CA203154593&v=2.1&u=flinders&it=r&p=ITOF&sw=w. (accessed: 31.05.2018).

[28] The CPUSHAKE Museum. *IntelC8008-1*. 2009. URL: http://www.cpushack.com/chippics/Intel/8008/IntelC8008-1.html. (accessed: 31.05.2018).

[29] NowSecure. *2016 NowSecure Mobile Security Report*. Tech. rep. NowSecure, 2016.

[30] K. Okuly. *Intel Processor Transistor Count (By Year)*. Sept. 2014. URL: https://www.coursehero.com/file/11042546/Intel-Processor-Transistor-Count-by-Year/. (accessed: 01.06.2018).

[31] Qubes OS. *An Introduction to Qubes OS*. Aug. 2017. URL: https://www.qubes-os.org/intro/. (accessed: 23.03.2018).

[32] et al. S. Kami Makki. *Mobile and Wireless Network Security and Privacy*. Springer, 2007.

[33] *Security and Complexity*. IT NOW, Dec. 2015. R. Anderson, [Interviewee].

[34] B. Slash. *Use the USB Rubber Ducky to Disable Antivirus Software and Install Ransomeware*. Nov. 2017. URL: https://null-byte.wonderhowto.com/how-to/use-usb-rubber-ducky-disable-antivirus-software-install-ransomware-0180418/. (accessed: 3.06.2018).

[35] McCabe Software. *More Complex Equals Less Secure*. Tech. rep. 41 Sharpe Drive, Cranston, RL 02920: McCabe Software, 2015.

[36] McCabe Software. *Software Security Analysis: Control Flow Security Analysis with McCabe IQ*. Tech. rep. 41 Sharpe Drive, Cranston, RL 02920: McCabe Software, 2015.

[37] K. Than. *Computers can be hacked using high-frequency sound*. Dec. 2013. URL: https://www.insidescience.org/news/computers-can-be-hacked-using-high-frequency-sound. (accessed: 3.06.2018).

[38] K. Thompson. "Reflections on Trusting Trust". In: *Communications of the ACM* 27.8 (Aug. 1984), pp. 761–763.

[39] *What Are HIV and AIDS?* [Online; accessed 12. Oct. 2018]. May 2017. URL: https://www.hiv.gov/hiv-basics/overview/about-hiv-and-aids/what-are-hiv-and-aids.

[40] *What You Need To Know About The Intel Management Engine*. [Online; accessed 12. Oct. 2018]. Dec. 2017. URL: https://hackaday.com/2017/12/11/what-you-need-to-know-about-the-intel-management-engine.

[41] K. Zetter. *Hacker Lexicon: What is an air gap?* Dec. 2014. URL: https://www.wired.com/2014/12/hacker-lexicon-air-gap/. (accessed: 2.06.2018).