Faculté des Sciences et Ingénierie
Master Informatique
Systèmes Électroniques, Systèmes Informatiques

Laboratoire d'Informatique Paris 6 - CIAN

Hardware initialization of modern computers

A review on the importance of firmware in modern computing and a documentation on the Asus KGPE-D16 RAM initialization

August, 2024

Adrien 'neox' Bourmault (neox@gnu.org)

Under the supervision of Franck WAJSBÜRT (franck.wajsburt@lip6.fr)

This is Edition 0.0.

Copyright (C) 2024 Adrien 'neox' Bourmault <neox@gnu.org>

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Contents

Abstract **5**

1 Introduction to firmware and BIOS evolution **6**

1.1 Historical context of BIOS 6
1.1.1 Definition and origin 6
1.1.2 Functionalities and limitations 7
1.2 Modern BIOS and UEFI 8
1.2.1 Transition from traditional BIOS to UEFI (Unified Extensible Firmware Interface) 8
1.2.2 An other way with *coreboot* 8
1.3 Shift in firmware responsibilities 10

2 Characteristics of ASUS KGPE-D16 mainboard **11**

2.1 Overview of ASUS KGPE-D16 hardware 12
2.2 Chipset 13
2.3 Processors 15
2.4 Baseboard Management Controller 16

3 Key components in modern firmware **18**

3.1 General structure of coreboot 18
3.1.1 Bootblock stage 19
3.1.2 Romstage 21
3.1.3 Ramstage 22
3.1.3.1 Advanced Configuration and Power Interface 22
3.1.3.2 System Management Mode 23
3.1.4 Payload 23
3.2 AMD Platform Security Processor and Intel Management Engine 24

4 Memory initialization and training algorithms [WIP] **25**

4.1 Importance of memory initialization 25
4.2 Memory training algorithms 25
4.3 Practical examples 26
4.3.1 RAM Initialization Preparation 26
4.3.2 RAM Initialization 26
4.3.2.1 Memory Controller Initialization 26
4.3.2.2 Memory Module Training 27

5 Virtualization of the operating system through firmware abstraction **28**

5.1 ACPI and abstraction of hardware control 28
5.2 SMM as a hidden execution layer 28
5.3 UEFI and persistence 28
5.4 Intel and AMD: control beyond the OS 29
5.5 The OS as a virtualized environment 29

Conclusion 30

Bibliography 31

List of Figures 37

List of Listings 38

GNU Free Documentation License 39

Abstract

The global trend is towards the scarcity of free software-compatible hardware, and soon there will be no computer that will work without software domination by big companies, especially involving firmware like BIOSes.

A Basic Input Output System (BIOS) was originally a set of low-level functions contained in the read-only memory of a computer's mainboard, enabling it to perform basic operations when powered up. However, the definition of a BIOS has evolved to include what used to be known as Power On Self Test (POST) for the presence of peripherals, allocating resources for them to avoid conflicts, and then handing over to an operating system boot loader. Nowadays, the bulk of the BIOS work is the initialization and training of RAM. This means, for example, initializing the memory controller and optimizing timing and read/write voltage for optimal performance, making the code complex, as its role is to optimize several parallel buses operating at high speeds and shared by many CPU cores, and make them act as a homogeneous whole.

This document is the product of a project hosted by the *LIP6 laboratory* and supported by the *GNU Boot Project* and the *Free Software Foundation*. It delves into the importance of firmware in the hardware initialization of modern computers and explores various aspects of firmware, such as Intel Management Engine (ME), AMD Platform Security Processor (PSP), Advanced Configuration and Power Interface (ACPI), and System Management Mode (SMM). Additionally, it provides an in-depth look at memory initialization and training algorithms, highlighting their critical role in system stability and performance. Examples of the implementation in the ASUS KGPE-D16 mainboard are presented, describing its hardware characteristics, topology, and the crucial role of firmware in its operation after the mainboard architecture is examined. Practical examples illustrate the impact of firmware on hardware initialization, memory optimization, resource allocation, power management, and security. Specific algorithms used for memory training and their outcomes are analyzed to demonstrate the complexity and importance of firmware in achieving optimal system performance. Furthermore, this document explores the relationship between firmware and hardware virtualization. Security considerations and future trends in firmware development are also addressed, emphasizing the need for continued research and advocacy for free software-compatible hardware.

Chapter 1

Introduction to firmware and BIOS evolution

1.1 Historical context of BIOS

1.1.1 Definition and origin

The BIOS (Basic Input/Output System) is firmware, which is a type of software that is embedded into hardware devices to control their basic functions, acting as a bridge between hardware and other software, ensuring that the hardware operates correctly. Unlike regular software, firmware is usually stored in a non-volatile memory like ROM or flash memory. The term "firmware" comes from its role: it is "firm" because it's more permanent than regular software (which can be easily changed) but not as rigid as hardware.

The BIOS is used to perform initialization during the booting process and to provide runtime services for operating systems and programs. Being a critical component for the startup of personal computers, acting as an intermediary between the computer's hardware and its operating system, the BIOS is embedded on a chip on the motherboard and is the first code that runs when a PC is powered on. The concept of BIOS has its roots in the early days of personal computing. It was first developed by IBM for their IBM PC, which was introduced in 1981 [40]. The term BIOS itself was coined by Gary Kildall, who developed the CP/M (Control Program for Microcomputers) operating system [90]. In CP/M, BIOS was used to describe a component that interfaced directly with the hardware, allowing the operating system to be somewhat hardware-independent.

Figure 1.1: The eight-striped wordmark of IBM (1967, public domain, trademarked)

IBM's implementation of BIOS became a de facto standard in the industry, as it was part of the IBM PC's open architecture [42][14], which refers to the design philosophy adopted by IBM when developing the IBM Personal Computer (PC), introduced in 1981. This architecture is characterized by the use of off-the-shelf components and publicly available specifications, which allowed other manufacturers to create compatible hardware and software. It was in fact a departure from the proprietary systems prevalent at the time, where companies closely guarded their designs to maintain control over the hardware and software ecosystem. For example, IBM used the Intel 8088 CPU, a well-documented and widely available processor, and also the Industry Standard Architecture (ISA) bus, which defined how various components like memory, storage, and peripherals communicated with the CPU. This open architecture allowed other manufacturers to create IBM-compatible computers, also known as "clones", which further popularized the BIOS concept. As a result, the IBM PC BIOS set the stage for a standardized method of interacting with computer hardware, which has evolved over the years but remains fundamentally the same in principle. IBM also published detailed technical documentation at that time, including circuit diagrams,

BIOS listings, and interface specifications. This transparency allowed other companies to understand and replicate the IBM PC's functionality [40].

1.1.2 Functionalities and limitations

When a computer is powered on, the BIOS executes a Power-On Self-Test (POST), a diagnostic sequence that verifies the integrity and functionality of critical hardware components such as the CPU, RAM, disk drives, keyboard, and other peripherals [108]. This process ensures that all essential hardware components are operational before the system attempts to load the operating system. If any issues are detected, the BIOS generates error messages or beep codes to alert the user. Following the successful completion of POST, the BIOS runs the bootstrap loader, a small program that identifies the operating system's bootloader on a storage device, such as a hard drive, floppy disk, or optical drive. The bootstrap loader then transfers control to the OS bootloader, initiating the process of loading the operating system into the computer's memory and starting it. This step effectively bridges the gap between hardware initialization and operating system execution. The BIOS also provides a set of low-level software routines known as interrupts. These routines enable software to perform basic input/output operations, such as reading from the keyboard, writing to the display, and accessing disk drives, without needing to manage the hardware directly. By providing standardized interfaces for hardware components, the BIOS simplifies software development and improves compatibility across different hardware configurations [14].

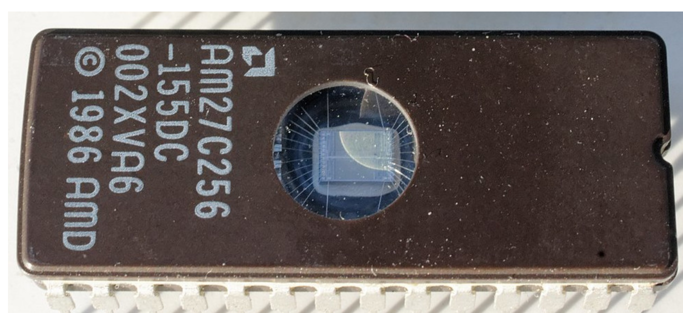

Figure 1.2: An AMI BIOS chip from a Dell 310, by Jud McCranie (CC BY-SA 4.0, 2018)

Despite its essential role, the early BIOS had several limitations. One significant limitation was its limited storage capacity. Early BIOS firmware was stored in Read-Only Memory (ROM) chips with very limited storage, often just a few kilobytes. This constrained the complexity and functionality of the BIOS, limiting it to only the most essential tasks needed to start the system and provide basic hardware control. The original BIOS was also non-extensible. ROM chips were typically soldered onto the motherboard, making updates difficult and costly. Bug fixes, updates for new hardware support, or enhancements required replacing the ROM chip, leading to challenges in maintaining and upgrading systems. Furthermore, the early BIOS was tailored for the specific hardware configurations of the initial IBM PC models, which included a limited set of peripherals and expansion options. As new hardware components and peripherals were developed, the BIOS often needed to be updated to support them, which was not always feasible or timely. Performance bottlenecks were another limitation. The BIOS provided basic input/output operations that were often slower than direct hardware access methods. For example, disk I/O operations through BIOS interrupts were slower compared to later direct access methods provided by operating systems, resulting in performance bottlenecks, especially for disk-intensive operations. This inflexibility restricts the ability to support new hardware and technologies efficiently[13]. Early BIOS implementations also had minimal security features. There were no mechanisms to verify the integrity of the BIOS code or to protect against unauthorized modifications, leaving systems vulnerable to attacks that could alter the BIOS and potentially compromise the entire system, such as rootkits and firmware viruses. Added to that, the traditional BIOS operates in 16-bit real mode, a constraint that limits the amount of code and memory it can address. This limitation hinders the performance and complexity of firmware, making it less suitable for modern computing needs [28]. Additionally, BIOS relies on the Master Boot Record (MBR) partitioning scheme, which supports a maximum disk size of 2 terabytes and allows only four primary partitions [35][87]. This constraint has become a significant drawback as storage capacities have increased. Furthermore, the traditional BIOS has limited flexibility and is challenging to update or extend. This inflexibility restricts the ability to support new hardware and technologies efficiently [13][1].

1.2 Modern BIOS and UEFI

1.2.1 Transition from traditional BIOS to UEFI (Unified Extensible Firmware Interface)

All the limitations listed earlier caused a transition to a more modern firmware interface, designed to address the shortcomings of the traditional BIOS. This section delves into the historical context of this shift, the driving factors behind it, and the advantages UEFI offers over the traditional BIOS.

The development of UEFI began in the mid-1990s as part of the Intel Boot Initiative, which aimed to modernize the boot process and overcome the limitations of the traditional BIOS. By 2005, the Unified EFI Forum, a consortium of technology companies including Intel, AMD, and Microsoft, had formalized the UEFI specification [35]. UEFI was designed to address the shortcomings of the traditional BIOS, providing several key improvements.

Figure 1.3: The UEFI logo (public domain, 2010)

One of the most significant advancements of UEFI is its support for 32-bit and 64-bit modes, allowing it to address more memory and run more complex firmware programs. This capability enables UEFI to handle the increased demands of modern hardware and software [28][89]. Additionally, UEFI uses the GUID Partition Table (GPT) instead of the MBR, supporting disks larger than 2 terabytes and allowing for a nearly unlimited number of partitions [29][87].
Improved boot performance is another driving factor. UEFI provides faster boot times compared to the traditional BIOS, thanks to its efficient hardware and software initialization processes. This improvement is particularly beneficial for systems with complex hardware configurations, where quick boot times are essential [28]. UEFI's modular architecture makes it more extensible and easier to update compared to the traditional BIOS. This design allows for the addition of drivers, applications, and other components without requiring a complete firmware overhaul, providing greater flexibility and adaptability to new technologies [1]. UEFI also includes enhanced security features such as *Secure Boot*, which ensures that only trusted software can be executed during the boot process, thereby protecting the system from unauthorized modifications and malware [13][21].

The industry-wide support and standardization of UEFI have accelerated its adoption across various platforms and devices. Major industry players, including Intel, AMD, and Microsoft, have adopted UEFI as the new standard for firmware interfaces, ensuring broad compatibility and interoperability [35].

1.2.2 An other way with *coreboot*

While UEFI has become the dominant firmware interface for modern computing systems, it is not without its critics. Some of the primary concerns about UEFI include its complexity, potential security vulnerabilities, and the degree of control it provides to hardware manufacturers over the boot process. Originally known as LinuxBIOS, *coreboot*, is a free firmware project initiated in 1999 by Ron Minnich and his team at the Los Alamos National Laboratory. The project's primary goal was to create a fast, lightweight, and flexible firmware solution that could initialize hardware and boot operating systems quickly, while remaining transparent and auditable[81]. As an alternative to UEFI, *coreboot* offers a different approach to firmware that aims to address some of these concerns and continue the evolution of BIOS.

One of the main advantages of *coreboot* over UEFI is its simplicity, as it is designed to perform only the minimal tasks required to initialize hardware and pass control to a payload, such as a bootloader or operating system kernel. This minimalist approach reduces the attack surface and potential for security vulnerabilities, as there is less code that could be exploited by malicious actors [86]. Another significant benefit of *coreboot* is its libre nature. Unlike UEFI, which is controlled by a consortium of hardware and software vendors, *coreboot*'s source code is freely available and can be audited, modified, and improved by anyone. This transparency ensures that security researchers and developers can review the code for potential vulnerabilities and contribute to its improvement, fostering a community-driven approach to firmware development[81]. This project also supports a wide range of bootloaders, called payloads, allowing users to customize their boot process to suit their specific needs. Popular payloads include SeaBIOS, which provides legacy BIOS compatibility, and Tianocore, which offers UEFI functionality within the *coreboot* framework. This flexibility allows *coreboot* to be used in a variety of environments, from embedded systems to high-performance servers [80].

Figure 1.4: The *coreboot* logo, by Konsult Stuge & coresystems (coreboot logo license, 2008)

Despite its advantages, *coreboot* is not without its challenges. The project relies heavily on community contributions, and support for new hardware often lags behind that of UEFI. Additionally, the minimalist design of *coreboot* means that some advanced features provided by UEFI are not available by default. However, the *coreboot* community continues to work on adding new features and improving compatibility with modern hardware or security issues [68]. For example, it provides a *verified boot* function, allowing to prevent rootkits and other attacks based on firmware modifications [79]. However, it's important to note that *coreboot* is not entirely free in all aspects. Many modern processors and chipsets require *proprietary blobs*, short for *Binary Large Object*, which is a collection of binary data stored as a single entity. These blobs are necessary for *coreboot* to function correctly on a wide range of hardware, but they compromise the goal of having a fully free firmware one day [62], since these blobs are used for certain functionalities such as memory initialization and hardware management.

Figure 1.5: The *GNU Boot* logo, by Jason Self (CC0, 2020)

To address these concerns, the GNU Project has developed GNU Boot, a fully free distribution of firmware, including *coreboot*, that aims to be entirely free by avoiding the use of proprietary binary blobs. GNU Boot is committed to using only free software for all aspects of firmware, making it a preferred choice for users and organizations that prioritize software freedom and transparency [63].

1.3 Shift in firmware responsibilities

Initially, the BIOS's primary function was to perform the POST, a basic diagnostic testing process to check the system's hardware components and ensure they were functioning correctly. This included verifying the CPU, memory, and essential peripherals before passing control to the operating system's bootloader. This process was relatively simple, given the limited capabilities and straightforward architecture of early computer systems [13].
As computer systems advanced, particularly with the advent of more sophisticated memory technologies, the role of firmware expanded significantly. Modern memory modules operate at much higher speeds and capacities than their predecessors, requiring precise configuration to ensure stability and optimal performance. Firmware now plays a critical role in managing the memory controller, which is responsible for regulating data flow between the processor and memory modules. This includes configuring memory frequencies, voltage levels, and timing parameters to match the specifications of the installed memory [35][9]. Beyond memory management, firmware responsibilities have broadened to encompass a wide range of system-critical tasks. One key area is power management, where firmware is responsible for optimizing energy consumption across various components of the system. Efficient power management is essential not only for extending battery life in portable devices but also for reducing thermal output and ensuring system longevity in desktop and server environments. Moreover, modern firmware takes on significant roles in hardware initialization and configuration, which were traditionally handled by the operating system. For example, the initialization of USB controllers, network interfaces, and storage devices is now often managed by the firmware during the early stages of the boot process. This shift ensures that the operating system can seamlessly interact with hardware from the moment it takes control, reducing boot times and improving overall system reliability [35]. Security has also become a paramount concern for modern firmware. UEFI (Unified Extensible Firmware Interface), which has largely replaced traditional BIOS in modern systems, includes features which prevents unauthorized or malicious software from loading during the boot process. This helps protect the system from rootkits and other low-level malware that could compromise the integrity of the operating system before it even starts [35]. In the context of performance tuning, firmware sometimes also plays a key role in enabling and managing overclocking, particularly for the memory subsystem. By allowing adjustments to memory frequencies, voltages, and timings, firmware provides tools for enthusiasts to push their systems beyond default limits. At the same time, it includes safeguards to manage the risks of instability and hardware damage, balancing performance gains with system reliability [13].

In summary, the evolution of firmware from simple hardware initialization routines to complex management systems reflects the increasing sophistication of modern computer architectures. Firmware is now a critical layer that not only ensures the correct functioning of hardware components but also optimizes performance, manages power consumption, and enhances system security, making it an indispensable part of contemporary computing.

This document will focus on *coreboot* during the next parts to study how modern firmware interact with hardware and also as a basis for improvements.

Chapter 2

Characteristics of ASUS KGPE-D16 mainboard

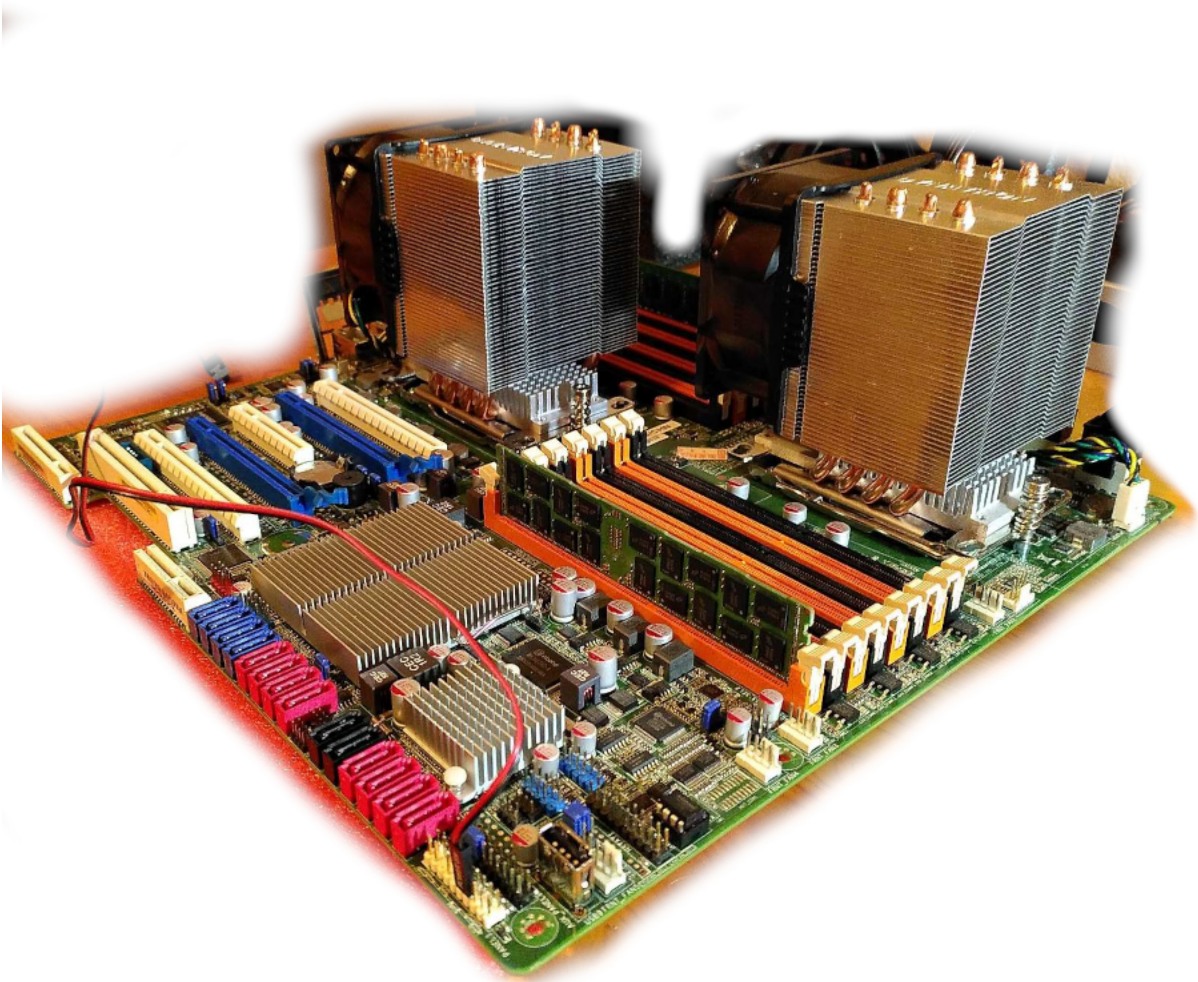

Figure 2.1: The KGPE-D16 (CC BY-SA 4.0, 2021)

2.1 Overview of ASUS KGPE-D16 hardware

The ASUS KGPE-D16 server mainboard is a dual-socket motherboard designed to support AMD Family 10h/15h series processors. Released in 2009, this mainboard was later awarded the *Respects Your Freedom* (RYF) certification in March 2017, underscoring its commitment to fully free software compatibility [38]. Indeed, this mainboard can be operated with a fully free firmware such as GNU Boot [64].

This mainboard is equipped with robust hardware components designed to meet the demands of high-performance computing. It features 16 DDR3 DIMM slots, capable of supporting up to 256GB of memory, although certain configurations may be limited to 192GB, with some reports suggesting the potential to support 256GB under specific conditions. In terms of expandability, the KGPE-D16 includes multiple PCIe slots, with five physical slots available, although only four can be used simultaneously due to slot sharing. For storage, the mainboard provides several SATA ports. Networking capabilities are enhanced by integrated dual gigabit Ethernet ports, which provide high-speed connectivity essential for data-intensive tasks and network communication [15]. Additionally, the board is equipped with various peripheral interfaces, including USB ports, audio outputs, and other I/O ports, ensuring compatibility with a wide range of external devices.

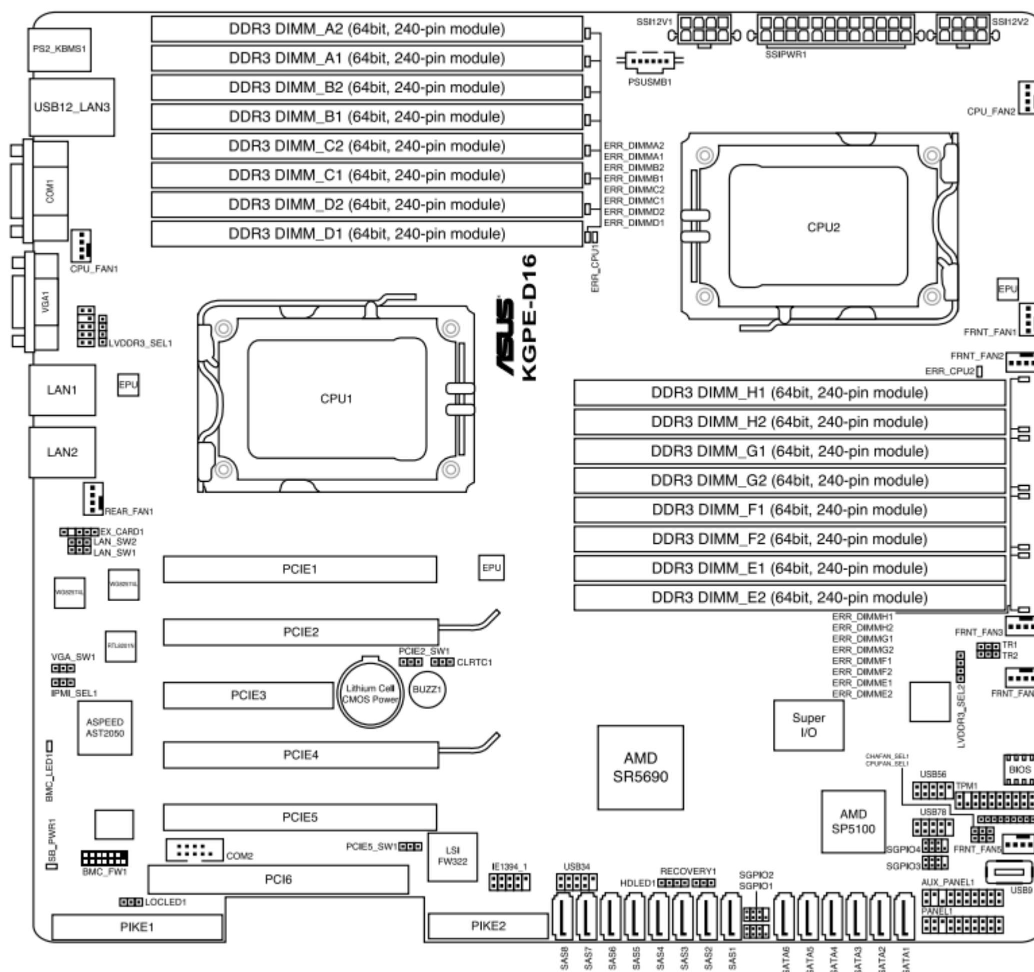

Figure 2.2: Basic schematics of the ASUS KGPE-D16 Mainboard, ASUS (2011)

The physical layout of the ASUS KGPE-D16 is meticulously designed to optimize airflow, cooling, and power distribution. All of this is critical for maintaining system stability, particularly under heavy computational loads, as this board was designed for server operations. In particular, key components such as the CPU sockets, memory slots, and PCIe slots are strategically positioned.

Figure 2.3: The KGPE-D16, viewed from the top (CC BY-SA 4.0, 2024)

2.2 Chipset

Before diving into the specific components, it is essential to understand the roles of the northbridge and southbridge in traditional motherboard architecture. These chipsets historically managed communication between the CPU and other critical components of the system [3].

The northbridge is a chipset on the motherboard that traditionally manages high-speed communication between the CPU, memory (RAM), and graphics card (if applicable). It serves as a hub for data that needs to move quickly between these components. On the ASUS KGPE-D16, the functions typically associated with the northbridge are divided between the CPUs internal northbridge and an external SR5690 northbridge chip. The SR5690 specifically acts as a translator and switch, handling the HyperTransport interface, a high-speed communication protocol used by AMD processors, and converting it to ALink and PCIe interfaces, which are crucial for connecting peripherals like graphics cards [11]. Additionally, the northbridge on the KGPE-D16 incorporates the IOMMU (Input-Output Memory Management Unit), which is crucial for ensuring secure and efficient memory access by I/O devices. The IOMMU allows for the virtualization of memory addresses, providing device isolation and preventing unauthorized memory access, which is particularly important in environments that run multiple virtual machines [3][120].

The southbridge, on the other hand, is responsible for handling lower-speed, peripheral interfaces such as the PCI, USB, and IDE/SATA connections, as well as managing onboard audio and network controllers. On the KGPE-D16, these functions are managed by the SP5100 southbridge chip, which integrates several critical functions including the LPC bridge, SATA controllers, and other essential I/O operations [3][123]. It is essentially an ALink bus controller and includes the hardware interrupt controller, the IOAPIC. Interrupts from peripheral always pass through the northbridge (fig. 2.4), since it translates ALink to HyperTransport for the CPUs and contains the IOMMU [11].

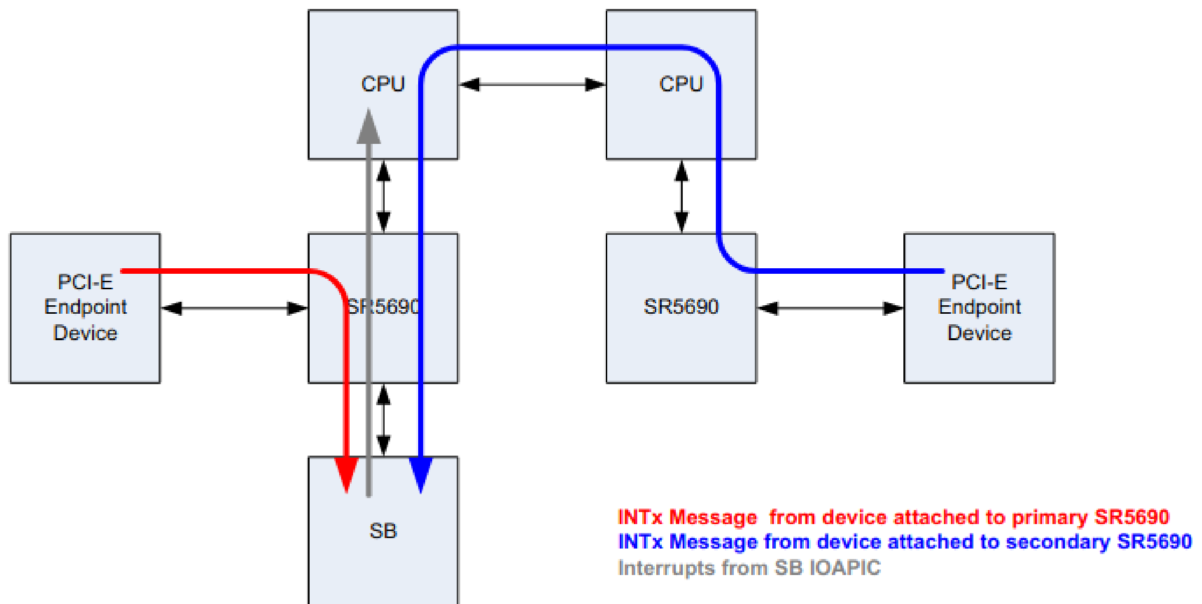

Figure 2.4: Functional diagram presenting the IOAPIC function of the SP5100, ASUS (2011)

In addition to the northbridge and southbridge, the KGPE-D16 also contains specialized chips for managing input/output operations and system health monitoring. The WINBOND W83667HG-A Super I/O chip handles traditional I/O functions such as legacy serial and parallel ports, keyboard, and mouse interfaces, but also the SPI chip that contains the firmware [127]. Meanwhile, the Nuvoton W83795G/ADG Hardware Monitor oversees the systems health by monitoring temperatures, voltages, and fan speeds, ensuring that the system operates within safe parameters [72]. On the KGPE-D16, access to the Super I/O from a CPU core is done through the SR5690, then the SP5100, as that can be observed on the functional diagram of the chipset (fig. 2.5) [11].

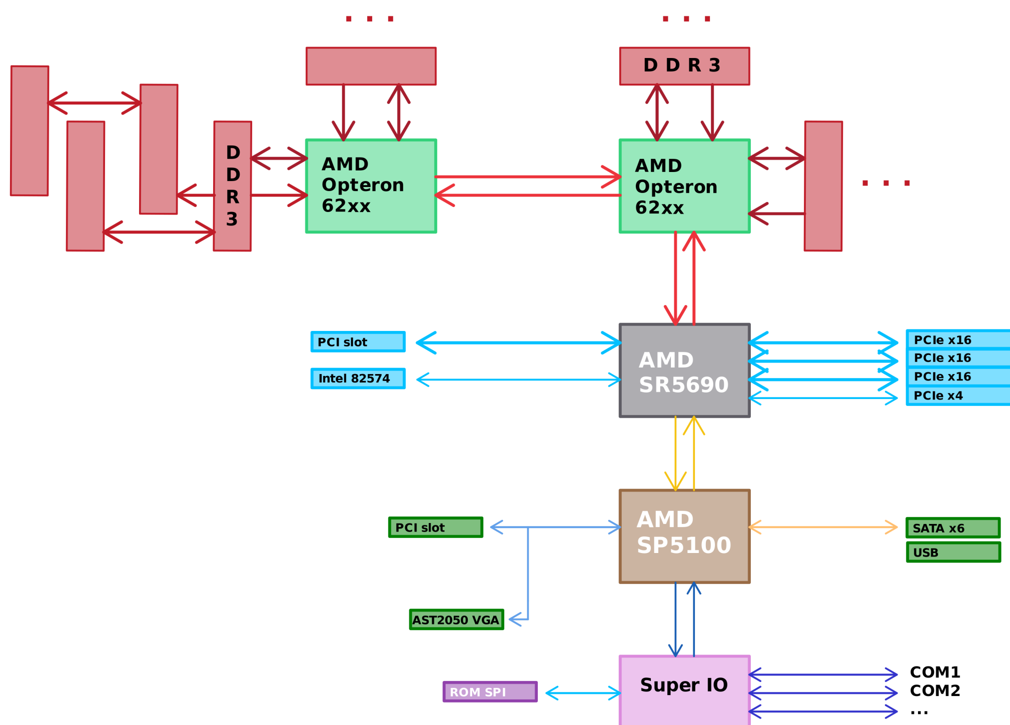

Figure 2.5: Functional diagram of the KGPE-D16 chipset (CC BY-SA 4.0, 2024)

2.3 Processors

The ASUS KGPE-D16 supports AMD Family 10h processors, but it is important to note that Vikings, a known vendor for libre-software-compatible hardware, does not recommend using the Opteron 6100 series due to the lack of IOMMU support, which is critical for security. Fortunately, AMD Family 15h processors are also supported. However, the Opteron 6300 series, while supported, requires proprietary microcode updates for stability, IOMMU functionality, and fixes for specific vulnerabilities, including a gain-root- via-NMI exploit. The Opteron 6200 series does not suffer from these problems and works properly without any proprietary microcode update needed [103].

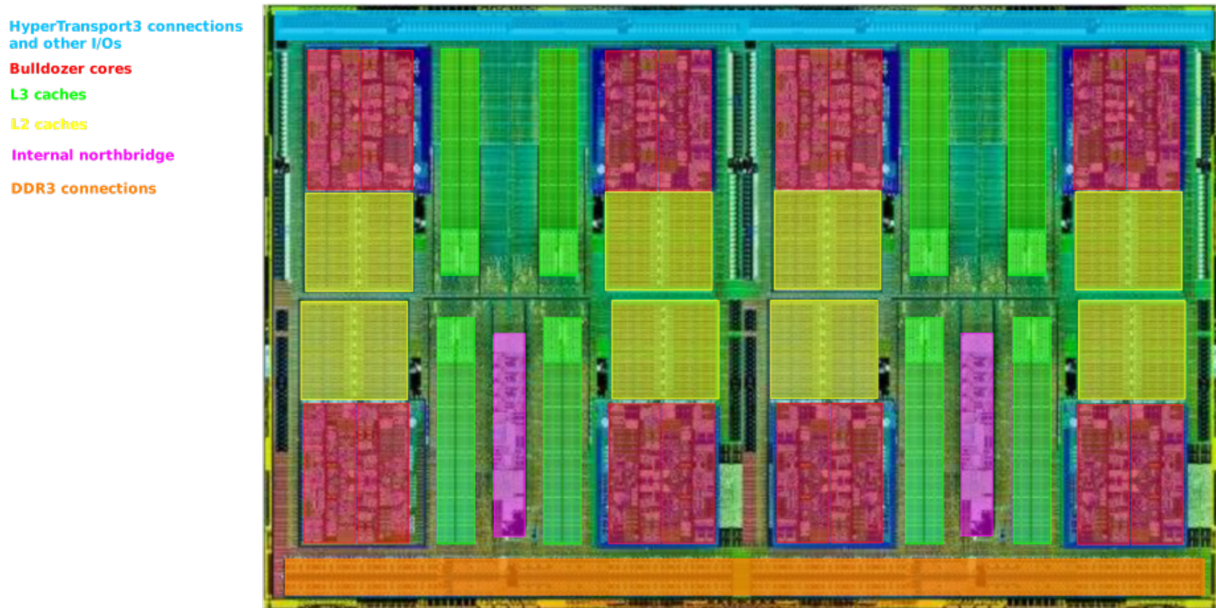

Figure 2.6: Annotated photography of an Opteron 6200 series CPU (2024), from a photography by AMD Inc. (2008)

The Opteron 6200 series, part of the Bulldozer microarchitecture, was designed to target high-performance server applications. These processors feature 16 cores, organized into 8 Bulldozer modules, with each module containing two integer cores that shared resources like the floating-point unit (FPU) and L2 cache (fig. 2.6, 2.7) [7][88]. The architecture of the Opteron 6200 series is built around AMD's Bulldozer core design, which uses Clustered Multithreading (CMT) to maximize resource utilization. This is a technique where each processor module contains two integer cores that share certain resources like the floating-point unit (FPU), L2 cache, and instruction fetch/decode stages. Unlike traditional multithreading, where each core handles multiple threads, CMT allows two cores to share resources to improve parallel processing efficiency. This approach aims to balance performance and resource usage, particularly in multi- threaded workloads, though it can lead to some performance trade-offs in single-threaded tasks. In the Opteron 6272, the processor consists of eight modules, effectively creating 16 integer cores. Due to the CMT architecture, each Opteron 6272 chip functions as two CPUs within a single processor, each with its own set of cores, L2 caches, and shared L3 cache. Here, one CPU is made by four modules, each module in it sharing certain components, such as the FPU and L2 cache, between two integer cores. The L3 cache is shared across these modules. HyperTransport links provide high-speed communication between the two sockets of the KGPE-D16. Shared L3 cache and direct memory access are provided by each socket [7][46].

This architecture also integrates a quad-channel DDR3 memory controller directly into the processor die, which facilitates high bandwidth and low latency access to memory. This memory controller supports DDR3 memory speeds up to 1600 MHz and connects directly to the memory modules via the memory bus. By integrating the memory controller into the processor, the Opteron 6200 series reduces memory access latency, enhancing overall performance [7][6]. It is interesting to note that Opterons incorporate the internal northbridge that we cited previously. The traditional northbridge functions, such as memory controller and PCIe interface management, are partially integrated into the processor. This integration reduces the distance data must travel between the CPU and memory, decreasing latency and improving performance, particularly in memory-intensive applications [7].

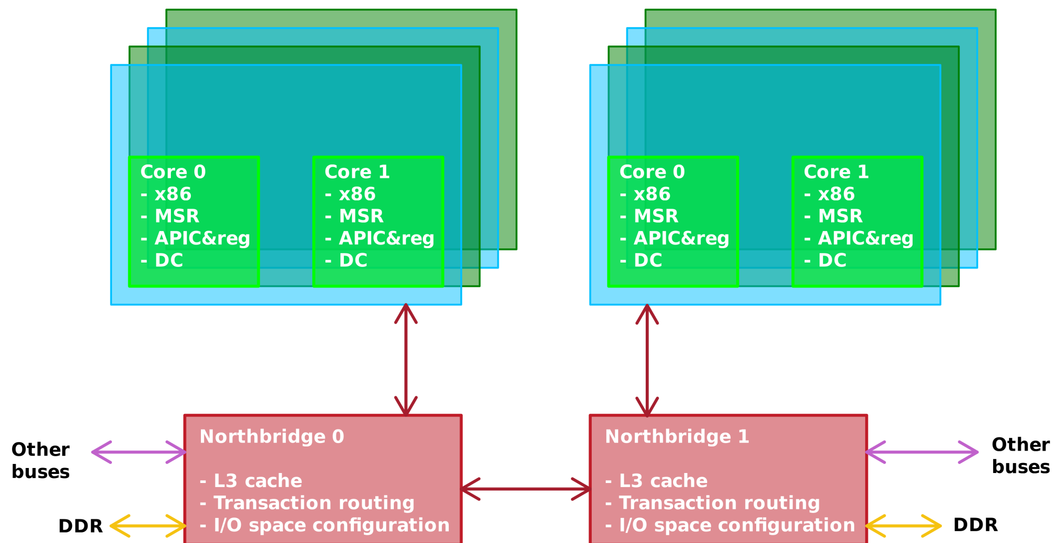

Figure 2.7: Functional diagram of an Opteron 6200 package (CC BY-SA 4.0, 2024)

Power efficiency was a key focus in the design of the Opteron 6200 series. Despite the high core count, the processor includes several power management features, such as Dynamic Power Management (DPM) and Turbo Core technology. These features allow the processor to adjust power usage based on workload demands, balancing performance with energy consumption. However, the Bulldozer architecture's focus on high clock speeds and multi-threaded performance resulted in higher power consumption compared to competing architectures [88]. A special model of the series, called *high efficiency* models, solve a bit this problem by proposing a bit less performant processor but with a power consumption divided by a factor from 1.5 to 2.0 in some cases.

The processor connected to the I/O hub is known as the Bootstrap Processor (BSP). The BSP is responsible for starting up the system by executing the initial firmware code from the reset vector, a specific memory address where the CPU begins execution after a reset [4]. Core 0 of the BSP, called the Bootstrap Core (BSC), initiates this process. During early initialization, the BSP performs several critical tasks, such as memory initialization, and bringing other CPU cores online. One of its duties is storing Built-In Self-Test (BIST) information, which involves checking the integrity of the processor's internal components to ensure they are functioning correctly. The BSP also determines the type of reset that has occurredwhether it's a cold reset, which happens when the system is powered on from an off state, or a warm reset, which is a restart without turning off the power. Identifying the reset type is crucial for deciding which initialization procedures need to be executed [4][9].

2.4 Baseboard Management Controller

The Baseboard Management Controller (BMC) on the KGPE-D16 motherboard, specifically the ASpeed AST2050, plays a role in the server's architecture by managing out-of-band communication and control of the hardware. The AST2050 is based on an ARM926EJ-S processor, a low-power 32-bit ARM architecture designed for embedded systems [94]. This architecture is well-suited for BMCs due to its efficiency and capability to handle multiple management tasks concurrently without significant resource demands from the main system.

The AST2050 features several key components that contribute to its functionality. It includes an integrated VGA controller, which enables remote graphical management through KVM-over-IP (Keyboard, Video, Mouse), a critical feature for administrators who need to interact with the system remotely, including BIOS updates and troubleshooting [92]. Additionally, the AST2050 integrates a dedicated memory controller, which supports up to 256MB of DDR2 RAM. This allows it to handle complex tasks and maintain responsiveness during management

operations [31]. The BMC also features a network interface controller (NIC) dedicated to management traffic, ensuring that remote management does not interfere with the primary network traffic of the server. This separation is vital for maintaining secure and uninterrupted system management, especially in environments where uptime is critical [95]. Another important architectural aspect of the AST2050 is its support for multiple I/O interfaces, including I2C, GPIO, UART, and USB, which allow it to interface with various sensors and peripherals on the motherboard [96]. This versatility enables comprehensive monitoring of hardware health, such as temperature sensors, fan speeds, and power supplies, all of which can be managed and configured through the BMC.

When combined with OpenBMC [121], a libre firmware that can be run on the AST2050 thanks to Raptor Engineering [82], the architecture of the BMC becomes even more powerful. OpenBMC takes advantage of the AST2050's architecture, providing a flexible and customizable environment that can be tailored to specific use cases. This includes adding or modifying features related to security, logging, and network management, all within the BMC's ARM architecture framework [51].

Chapter 3

Key components in modern firmware

3.1 General structure of coreboot

The firmware of the ASUS KGPE-D16 is crucial in ensuring the proper functioning and optimization of the mainboard's hardware components. For this to be done efficiently, *coreboot* is organized in different stages (fig. 3.1) [79].

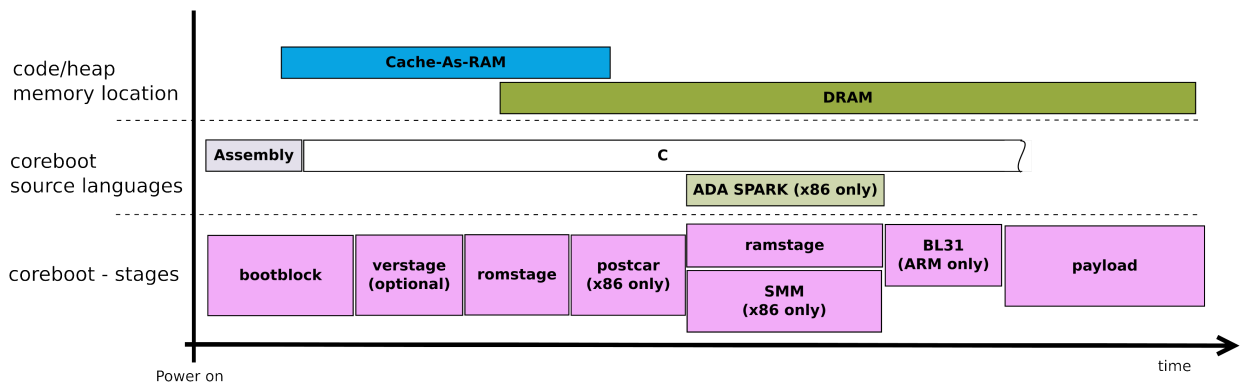

Figure 3.1: *coreboot*'s stages timeline, by *coreboot* project (CC BY-SA 4.0, 2009)

Being a complex project with ambitious goals, *coreboot* decided early on to establish an file-system-based architecture for its images (also called ROMs). This special file-system is CBFS (which stands for coreboot file system). The CBFS architecture consists of a binary image that can be interpreted as a physical disk, referred to here as ROM. A number of independent components, each with a header added to the data, are located within the ROM. The components are nominally arranged sequentially, although they are aligned along a predefined boundary (fig. 3.2).

Each stage is compiled as a separate binary and inserted into the CBFS with custom compression. The bootblock stage is usually not compressed, while the ramstage and the payload are compressed with LZMA. Each stage loads the next stage at a given address (possibly decompressing it in the process).

Some stages are relocatable and can be placed anywhere in the RAM. These stages are typically cached in the CBMEM for faster loading times during wake-up. The CBMEM is a specific memory area used by the *coreboot* firmware to store important data structures and logs during the boot process. This area is typically allocated in the system's RAM and is used to store various types of runtime information that it might need to reference after the initial boot stages.

In general, *coreboot* manages main memory through a structured memory map (fig. 3.1), allocating specific address ranges for various hardware functions and system operations. The first 640KB of memory space is typically unused by coreboot due to historical reasons. Graphics-related operations use the VGA address range and the text mode address ranges. It also reserves the higher for operating system use, ensuring that critical system components like the IOAPIC and TPM registers have dedicated address spaces. This structured approach helps maintain system stability and compatibility across different platforms and allows for a reset vector fixed at an address (*0xFFFFFFF0*), regardless of the ROM size. Payloads are typically loaded into high memory, above the

reserved areas for hardware components and system resources. The exact memory location can vary depending on the system's configuration, but generally, payloads are placed in a region of memory that does not conflict with the firmware code or the reserved memory map areas, such as the ROM mapping ranges. This placement ensures that payloads have sufficient space to execute without interfering with other critical memory regions allocated [24].

0x00000 - 0x9FFFF	Low memory (first 640KB). Never used.
0xA0000 - 0xAFFFF	VGA graphics address range.
0xB0000 - 0xB7FFF	Monochrome text mode address range. Few motherboards use it, but the KGPE-D16 does.
0xB8000 - 0xBFFFF	Text mode address range.
0xFEC00000	IOAPIC address.
0xFED44000 - 0xFED4FFFF	Address range for TPM registers.
0xFF000000 - 0xFFFFFFFF	16 MB ROM mapping address range.
0xFF800000 - 0xFFFFFFFF	8 MB ROM mapping address range.
0xFFC00000 - 0xFFFFFFFF	4 MB ROM mapping address range.
0xFEC00000 - DEVICE MEM HIGH	Reserved area for OS use.

Table 3.1: *coreboot* memory map

3.1.1 Bootblock stage

The bootblock is the first stage executed after the CPU reset. The beginning of this stage is written in assembly language, and its main task is to set everything up for a C environment. The rest, of course, is written in C. This stage occupies the last 20k (fig. 3.2) of the image and within it is a main header containing information about the ROM, including the size, component alignment, and the offset of the start of the first CBFS component. This block is a mandatory component as it also contains the entry point of the firmware.

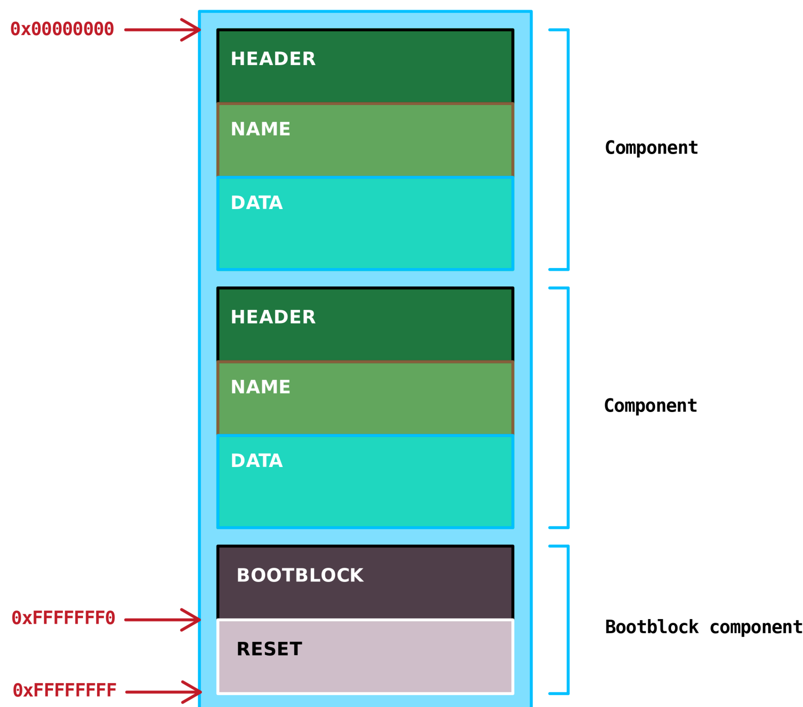

Figure 3.2: *coreboot* ROM architecture (CC BY-SA 4.0, 2024)

Upon startup, the first responsibility of the bootblock is to execute the code from the reset vector located at the conventional reset vector in 16-bit real mode. This code is specific to the processor architecture and, for our board, is stored in the architecture-specific sources for x86 within *coreboot*. The entry point into *coreboot* code

is defined in two files in the `src/cpu/x86/16bit/` directory: `reset16.inc` and `entry16.inc`. The first file serves as a jump to the `_start16bit` procedure defined in the second. Due to space constraints this function must remain below the 1MB address space because the IOMMU has not yet been configured to allow anything else.

During this early initialization, the Bootstrap Core (BSC) performs several critical tasks while the other cores remain dormant. These tasks include saving the results (and displaying them if necessary) of the Built-in Self-Test (BIST), formerly known as POST; invalidating the TLB to prevent any address translation errors; determining the type of reset (e.g., cold start or warm start); creating and loading an empty Interrupt Descriptor Table (IDT) to prevent the use of "legacy" interrupts from real mode until protected mode is reached. In practice, this means that at the slightest exception, the BSC will halt. The code then switches to 32-bit protected mode by mapping the first 4 GB of address space for code and data, and finally jumps to the 32-bit reset code labeled `_protected_start`.

Once in protected mode, which constitutes the "normal" operating mode for the processor, the next step is to set up the execution environment. To achieve this, the code contained in `src/cpu/x86/32bit/entry32.inc`, followed by `src/cpu/x86/64bit/entry64.inc`, and finally `src/arch/x86/bootblock_crt0.S`, establishes a temporary stack, transitions to long mode (64-bit addressing) with paging enabled, and sets up a proper exception vector table. The execution then jumps to chipset-specific code via the `bootblock_pre_c_entry` procedure. Once these steps are completed, the bootblock has a minimal C environment. The procedure now involves allocating memory for the BSS, and decompressing and loading the next stage.

The jump to `_bootblock_pre_entry` leads to the code files `src/soc/amd/common/block/cpu/car/cache_as_ram.S` and `src/vendorcode/amd/agesa/f15tn/gcccar.inc`, which are specific to AMD chipsets. It's worth noting that these files were developed by AMD's engineers as part of the *AGESA* project. The operations performed at this stage are related to pre-RAM memory initialization. All cores of all processors (up to a limit of 64 cores) are started. The *Cache-As-Ram* is configured using the Memory-type range registers. These registers allow the specification of a specific configuration for a given memory area [9]. In this case, the area that should correspond to physical memory is mapped to the cache, while other areas, such as PCI or other bus zones, are configured accordingly. A specific stack is set up for each core of each processor (within the arbitrary limit of 64 cores and 7 nodes, meaning 7 Core 0s). Core 0s receive 16KB, while the Bootstrap Core (BSC) gets 64KB. The other cores receive 4KB each. All cores except the BSC are halted and will restart during the romstage. Finally, the execution jumps to the entry point of the *bootblock* written in C, labeled `bootblock_c_entry`. This entry point is located in `src/soc/amd/stoneyridge/bootblock/bootblock.c` and is specific to AMD processors. It is the first C routine executed, and its role is to verify that the current processor is indeed the BSC, allowing the function `bootblock_main_with_basetime` to be called exclusively by the BSC.
We are now in the file `src/lib/bootblock.c`, written by Google's team, and entering the `bootblock_main_with_basetime` function, which immediately calls `bootblock_main_with_timestamp`. At this stage, the goal is to start the romstage, but a few more tasks need to be completed.
The `bootblock_soc_early_init` function is called to initialize the I2C bus of the southbridge. The `bootblock_fch_early_init` function is invoked to initialize the SPI buses (including the one for the ROM) and the serial and "legacy" buses of the southbridge. The CMOS clock is then initialized, followed by the pre-initialization of the serial console. The code then calls the `bootblock_mainboard_init` function, which enters, for the first time, the files specific to the ASUS KGPE-D16 motherboard: `src/mainboard/ASUS/kgpe-d16/bootblock.c`. This code performs the northbridge initialization via the `bootblock_northbridge_init` function found in `src/northbridge/amd/amdfam10/bootblock.c`. This involves locating the HyperTransport bus and enabling the discovery of devices connected to it (e.g., processors). The southbridge is initialized using the `bootblock_southbridge_init` function from `src/southbridge/amd/sb700/bootblock.c`. This function, largely programmed by Timothy Pearson from Raptor Engineering, who performed the first coreboot port for the ASUS KGPE-D16, finalizes the activation of the SPI bus and the connection to the ROM memory via SuperIO. The state of a recovery jumper is then checked (this jumper is intended to reset the CMOS content, although it is not fully functional at the moment, as indicated by the `FIXME` comment in the code). Control then returns to `bootblock_main` in `src/lib/bootblock.c`.
At this point, everything is ready to enter the romstage. *coreboot* has successfully started and can now continue its execution by calling the `run_romstage` function from `src/lib/prog_loaders.c`. This function begins by locating the corresponding segment in the ROM via the southbridge and SPI bus using `prog_locate`, which utilizes the SPI driver in `src/drivers/cbfs_spi.c`. The contents of the romstage are then copied into the

cache-as-ram by `cbfs_prog_stage_load`. Finally, the `prog_run` function transitions to the romstage after switching back to 32-bit mode.

3.1.2 Romstage

The *romstage* in *coreboot* serves the critical function of early initialization of peripherals, particularly system memory. This stage is crucial for setting up the necessary components for the platform's operation, ensuring that everything is in place for subsequent stages of the boot process. During this phase, *coreboot* configures the Advanced Programmable Interrupt Controller (APIC), which is responsible for correctly handling interrupts across multiple CPUs, especially in systems using Symmetric Multiprocessing (SMP). This includes setting up the Local APIC on each processor and the IOAPIC, part of the southbridge, to ensure that interrupts from peripherals are routed to the appropriate CPUs. Additionally, the firmware configures the HyperTransport (HT) technology, a high-speed communication protocol that facilitates data exchange between the processor and the northbridge, ensuring smooth data flow between these components.

The *romstage* begins with a call to the `_start` function, defined in `src/cpu/x86/32bit/entry32.inc` via `src/arch/x86/assembly_entry.S`. We then enter the `cache_as_ram_setup` procedure, written in assembly language, located in `src/cpu/amd/car/cache_as_ram.inc`. This procedure configures the cache to load the future *ramstage* and initialize memory based on the number of processors and cores present. Once this is completed, the code calls `cache_as_ram_main` in `src/mainboard/asus/kgpe-d16/romstage.c`, which serves as the main function of the *romstage*. In the `cache_as_ram_main` function, after reducing the speed of the HyperTransport bus, only the Bootstrap Core (BSC) initializes the spinlocks for the serial console, the CMOS storage memory (used for saving parameters), and the ROM. At this point, the HyperTransport bus is enumerated, and the PCI bridges are temporarily disabled. The port 0x80 of the southbridge, used for motherboard debugging with *Post Codes*, is also initialized. These codes indicate the status of the boot process and can be displayed using special PCI cards connected to the system. The SuperIO is then initialized to activate the serial port, allowing the serial console to follow *coreboot*s progress in real-time. If everything proceeds as expected, the code 0x30 is sent, and the boot process continues. If the result of the Built-in Self-Test (BIST), saved during the *bootblock*, shows no anomalies, all cores of all nodes are configured, and they are placed back into sleep mode (except for the Core 0s). If everything goes well, the code 0x32 is sent, and the process continues. Using the `enable_sr5650_dev8` function, the southbridges P2P bridge is activated. Additionally, a check is performed to ensure that the number of physical processors detected does not exceed the number of sockets available on the board. If any issues were detected during the BIST, the machine will halt, and the error will be displayed on the console. Otherwise, the process continues, and the default hardware information table is constructed, and the microcode of the physical processors is updated if necessary. If everything proceeds correctly, the code 0x33 and then 0x34 is sent, and the process continues. The information about the physical processors is retrieved using `amd_ht_init`, and communication between the two sockets is configured via `amd_ht_fixup`. This process includes disabling any defective HT links (one per socket in this AMD Family 15h chipset). If everything is working as expected, the code 0x35 is sent, and the boot process continues. With the `finalize_node_setup` function, the PCI bus is initialized, and a mapping is created (`setup_mb_resource_map`). If all goes well, the code 0x36 is sent. This is done in parallel across all Core 0s, so the system waits for all cores to finish using the `wait_all_core0_started` function. The communication between the northbridge and southbridge is prepared using `sr5650_early_setup` and `sb7xx_51xx_early_setup`, followed by the activation of all cores on all nodes, with the system waiting for all cores to be fully initialized. If everything is successful, the code 0x38 is sent.

At this point, the timer is activated, and a warm reset is performed via the `soft_reset` function to validate all configuration changes to the HT, PCI buses, and voltage/power settings of the processors and buses. This results in a system reboot, passing again through the *bootblock*, but much faster this time since the system recognizes the warm reset condition. Once this reboot is complete, the HyperTransport bus is reconfigured into isochronous mode (switching from asynchronous mode), finalizing the configuration process.

Memory training and optimization are also key functions of the firmware during the *romstage*. This process involves adjusting memory settings, such as timings, frequencies, and voltages, to ensure that the installed memory modules operate efficiently and stably. This step is crucial for achieving optimal performance, especially when dealing with large amounts of RAM and many CPU cores, as supported by the KGPE-D16. We'll see that in detail during the next chapter.

After memory initialization, the process returns to the cache_as_ram_main function, where a memory test is performed. This involves writing predefined values to specific memory locations and then verifying that the values can be read back correctly. If everything passes successfully, the CBMEM is initialized and one sends code 0x41. At this point, the configuration of the PCI bus is prepared, which will be completed during the ramstage by configuring the PCI bridges. The system then exits cache_as_ram_main and returns to cache_as_ram_setup to finalize the process.
coreboot then transitions to the next stage, known as the postcar stage, where it exits the cache-as-RAM mode and begins using physical RAM.

3.1.3 Ramstage

The ramstage performs the general initialization of all peripherals, including the initialization of PCI devices, on-chip devices, the TPM (if not done by verstage), graphics (optional), and the CPU (setting up the System Management Mode). After this initialization, tables are written to inform the payload or operating system about the existence and current state of the hardware. These tables include ACPI tables (specific to x86), SMBIOS tables (specific to x86), coreboot tables, and updates to the device tree (specific to ARM). Additionally, the ramstage locks down the hardware and firmware by applying write protection to boot media, locking security-related registers, and locking SMM (specific to x86) [79]. Effective resource allocation is essential for system stability, particularly in complex configurations involving multiple CPUs and peripherals. This stage manages initial resource allocation, resolving any conflicts between hardware components to prevent resource contention and ensure smooth operation and security, which is a major concern in modern systems. This includes support for IOMMU, which is crucial for preventing unauthorized direct memory access (DMA) attacks, particularly in virtualized environments (however there are still vulnerabilities that can be exploited, such as sub-page or IOTLB-based attacks or even configuration weaknesses [67][65]).

3.1.3.1 Advanced Configuration and Power Interface

The Advanced Configuration and Power Interface (ACPI) is a critical component of modern computing systems, providing an open standard for device configuration and power management by the operating system (OS). Developed in 1996 by Intel, Microsoft, and Toshiba, ACPI replaced the older Advanced Power Management (APM) standard with more advanced and flexible power management capabilities [25]. At its core, ACPI is implemented through a series of data structures and executable code known as ACPI tables, which are provided by the system firmware and interpreted by the OS. These tables describe various aspects of the system, including hardware resources, device power states, and thermal zones. The ACPI Specification outlines these structures and provides the necessary standardization for interoperability across different platforms and operating systems [43]. These tables are used by the OS to perform low-level task, including managing power states of the CPU, controlling the voltage and frequency scaling (also known as Dynamic Voltage and Frequency Scaling, or DVFS), and coordinating power delivery to peripherals.

The ACPI Component Architecture (ACPICA) is the reference implementation of ACPI, providing a common codebase that can be used by OS developers to integrate ACPI support. ACPICA includes tools and libraries that allow for the parsing and execution of ACPI Machine Language (AML) code, which is embedded within the ACPI tables [78]. One of the key tools in ACPICA is the Intel ACPI Source Language (IASL) compiler, which converts ACPI Source Language (ASL) code into AML bytecode, allowing firmware developers to write custom ACPI methods [25]. The triggering of ACPI events is managed through a combination of hardware signals and software routines. For example, when a user presses the power button on a system, an ACPI event is generated, which is then handled by the OS. This event might trigger the system to enter a low-power state, such as sleep or hibernation, depending on the configuration provided by the ACPI tables [43]. These power states are defined in the ACPI specification, with global states (G0 to G3) representing different levels of system power consumption, and device states (D0 to D3) representing individual device power levels.

The ASUS KGPE-D16 mainboard, which is designed for server and high-performance computing environments, needs ACPI for managing its power distribution across multiple CPUs and attached peripherals. ACPI is integral in controlling the power states of various components, thereby optimizing performance and energy use. Additionally,

the firmware on the KGPE-D16 uses ACPI tables to manage system temperature and fan speed, ensuring reliable operation under heavy workloads [15].

3.1.3.2 System Management Mode

System Management Mode (SMM) is a highly privileged operating mode provided by x86 processors for handling system-level functions such as power management, hardware control, and other critical tasks that are to be isolated from the OS and applications. Introduced by Intel, SMM operates in an environment separate from the main operating system, offering a controlled space for executing sensitive operations [54].

SMM is triggered by a System Management Interrupt (SMI), which is a non-maskable interrupt that causes the CPU to save its current state and switch to executing code stored in a protected area of memory called System Management RAM (SMRAM). SMRAM is a specialized memory region that is isolated from the rest of the system, making it inaccessible to the OS and preventing tampering or interference from other software [45]. Within SMM, the firmware can execute various low-level functions that require direct hardware control or need to be protected from the OS. This includes tasks such as thermal management, where the system monitors CPU temperature and adjusts performance or power levels to prevent overheating, as well as power management routines that enable efficient energy usage by adjusting power states based on system activity [52]. One of the critical security features of SMM is its role in managing firmware updates and handling system-level security events. Because SMM operates in a privileged mode that is isolated from the OS, it can apply firmware updates and could respond to security threats without being affected by potentially compromised system software [32]. However, the high privilege level and isolation of SMM also present significant security challenges. If an attacker can compromise SMM, they gain full control over the system, bypassing all security measures implemented by the OS [59]. Also, with a proprietary firmware, it means that this code with a very high priviledge level cannot be audited at all, nor even replaced.

The ASUS KGPE-D16 mainboard needs SMM to perform critical management tasks that need to be done in parallel from the operating system. For example, SMM is used to monitor and manage system health by responding to thermal events and adjusting power levels to maintain system stability. SMM operates independently of the main operating system, allowing it to perform sensitive tasks securely. *coreboot* supports SMM, but its implementation is typically minimal compared to traditional proprietary firmware. In *coreboot*, SMM initialization involves setting up the System Management Interrupt (SMI) handler and configuring System Management RAM (SMRAM), the memory region where SMM code executes[19]. The extent of SMM support in *coreboot* can vary significantly depending on the hardware platform and the specific requirements of the system. *coreboot*'s design philosophy emphasizes a lightweight and fast boot process, delegating more complex management tasks to payloads or the operating system itself [83].
One of the key challenges with implementing SMM in *coreboot* is ensuring that SMI handlers are configured correctly to manage necessary system tasks without compromising security or performance. *coreboot*'s approach to SMM is consistent with its overall goal of providing a streamlined and efficient firmware solution, leaving more intricate functionalities to be handled by subsequent software layers [71].

3.1.4 Payload

The payload is the software that executes after coreboot has completed its initialization tasks. It resides in the CBFS and is predetermined at compile time, with no option to choose it at runtime. The primary role of the payload is to load and hand control over to the operating system. In some cases, the payload itself can be a component of the operating system [79]. Examples of payloads are *GNU GRUB*, *SeaBIOS*, *memtest86+* or even sometimes the *Linux kernel* itself.

TianoCore, a free implementation of the UEFI (Unified Extensible Firmware Interface) specification is often used as a payload [97]. It provides a UEFI environment after *coreboot* has completed its initial hardware initialization. This allows the system to benefit from the advanced features of UEFI, such as a more flexible boot manager, enhanced features, and support for modern hardware. Indeed, UEFI, and by extension *TianoCore*, includes a driver model that allows hardware manufacturers to provide UEFI-compatible drivers. These drivers can be loaded at boot time, allowing the firmware to support a wide range of modern devices that *coreboot*, with its more minimalistic and custom-tailored approach, might not support out of the box. For example, GOP drivers are responsible for setting up the graphics hardware in UEFI environments. They replace the older VGA BIOS routines used in legacy BIOS systems. With GOP drivers, the system can

initialize the GPU and display a graphical interface even before the operating system loads [76]. Hardware manufacturers can distribute proprietary UEFI drivers as part of firmware updates, making it straightforward for end-users to install and use them. This is especially useful for specialized hardware that requires specific drivers not included in the free software community. It also gives hardware vendors more control over how their devices are initialized and used, which can be an advantage for vendors but is a freedom and user control limitation.

Payloads are then definitely important parts of the firmware.

3.2 AMD Platform Security Processor and Intel Management Engine

The AMD Platform Security Processor (PSP) and Intel Management Engine (ME) are embedded subsystems within AMD and Intel processors, respectively, that handle a range of security-related tasks independent of the main CPU. These subsystems are fundamental to the security architecture of modern computing platforms, providing functions such as secure boot, cryptographic key management, and remote system management [53]. The AMD PSP is based on an ARM Cortex-A5 processor and is responsible for several security functions, including the validation of firmware during boot (secure boot), management of Trusted Platform Module (TPM) functions, and handling cryptographic operations such as key generation and storage. The PSP operates independently of the main x86 cores, which allows it to execute security functions even when the main system is powered off or compromised by malware [53]. The PSP's isolated environment ensures that sensitive operations are protected from threats that could affect the main OS.

Similarly, the Intel Management Engine (ME) is a dedicated processor embedded within Intel chipsets that operates independently of the main CPU. The ME is a comprehensive subsystem that provides a variety of functions, including out-of-band system management, security enforcement, and support for Digital Rights Management (DRM) [26]. The ME's firmware runs on an isolated environment that allows it to perform these tasks securely, even when the system is powered off. This capability is crucial for enterprise environments where administrators need to perform remote diagnostics, updates, and security checks without relying on the main OS. Intel ME enforces Digital Rights Management (DRM) through a multifaceted approach leveraging its deeply embedded, hardware-based capabilities. At the core is the Protected Execution Environment (PEE), which operates independently from the main CPU and operating system. This isolation allows to privately manage cryptographic keys, certificates, and other sensitive data critical for DRM, which can be very problematic from a user freedom perspective [39]. By handling encryption and decryption processes within this protected environment, Intel ME ensures that DRM-protected content, such as video streams, remains secure and unreachable by the user, raising concerns about the control users have over their own devices [73]. Intel ME also plays a significant role in maintaining platform integrity through the secure boot process. During secure boot, Intel ME ensures that only digitally signed and authorized operating systems and applications are loaded, which can prevent users from installing alternative or modified software on their own hardware, further restricting their freedom [98]. This is further reinforced by Intel ME's remote attestation capabilities, where the systems state is reported to a remote server. This process verifies that only systems meeting specific security standardsdictated by third partiesare allowed to access DRM-protected content, potentially limiting users' control over their own devices [18]. Moreover, Intel ME supports High-bandwidth Digital Content Protection (HDCP), a technology that restricts how digital content is transmitted over interfaces like HDMI or DisplayPort. By enforcing HDCP, Intel ME ensures that protected digital content, such as high-definition video, is only transmitted to and displayed on authorized devices, effectively preventing users from freely using the content they have legally acquired [57][75]. Together, these features enable Intel ME to provide a comprehensive and robust DRM enforcement mechanism. However, this also means that users have less control over their own hardware and digital content, raising serious concerns about privacy, user autonomy, and the broader implications for freedom in computing [39][50].

Added to that, Intel ME has been a source of controversy due to its deep integration into the hardware and its potential to be exploited if vulnerabilities are discovered. Researchers have demonstrated ways to hack into the ME, potentially gaining control over a system even when it is powered off [41]. These concerns have led to calls for greater transparency and security measures around the ME and similar subsystems. When comparing Intel ME and AMD PSP, the primary difference lies in their scope and functionality. Intel ME offers more extensive remote management capabilities, making it a more comprehensive tool for enterprise environments, while AMD PSP focuses more narrowly on core security tasks. Nonetheless, both play critical roles in ensuring the security

and integrity of modern computing systems.

The ASUS KGPE-D16 mainboard does not include AMD PSP nor Intel ME.

Chapter 4

Memory initialization and training algorithms [WIP]

4.1 Importance of memory initialization

- Steps involved in initializing the memory controller
- Critical role in system stability and performance
- **ASUS KGPE-D16 Example**: Memory initialization process on the KGPE-D16 mainboard

Memory training involves several steps:
1. **Detection and Initialization**: The BIOS detects the installed memory modules, determining their size, speed, and type.
2. **Configuration and Timing Setup**: The BIOS configures the memory controller settings, including timings for memory access such as CAS latency, RAS to CAS delay, and other parameters[28].
3. **Training and Calibration**: The BIOS performs tests and adjustments to calibrate the memory system, ensuring stable operation at optimal speeds by adjusting signal voltages and testing data integrity[128].
These steps are crucial for modern systems, where improper memory configuration can lead to instability, data corruption, or suboptimal performance.
Memory timings, such as CAS latency, RAS to CAS delay, and others, must be finely tuned to ensure optimal performance. The BIOS uses a combination of predefined profiles and dynamic adjustments to achieve the best balance between speed and stability. Advanced timing optimization involves setting these parameters to ensure that memory operations are performed with minimal latency and maximum throughput[87].

```
1 void main(int a, int b)
2 {
3   return 0;
4 }
```

Listing 4.1: *Example C code*

We saw that in (lst. 4.1).

4.2 Memory training algorithms

- Techniques used for training memory
- Optimization of timings and voltage settings
- Challenges in multi-core CPU environments
- **ASUS KGPE-D16 Example**: Specific algorithms used for memory training in the mainboard and their performance outcomes

To optimize memory performance, the BIOS employs various training algorithms and calibration techniques. These methods test the memory under different conditions and make necessary adjustments to improve stability and efficiency. Key techniques include voltage adjustments, data integrity testing, and signal timing calibration[89]. Voltage adjustments involve tweaking the power supplied to the memory modules to ensure reliable operation. Data integrity testing checks that data can be accurately read and written, while signal timing calibration fine-tunes the delays between different memory operations to minimize latency.

4.3 Practical examples

- Real-world scenarios where firmware played a crucial role in system performance
- Analysis of firmware updates and their impact on the KGPE-D16 mainboard
- User experiences and testimonials highlighting the importance of firmware
- **ASUS KGPE-D16 Example**: Specific case studies and firmware updates for the mainboard

4.3.1 RAM Initialization Preparation

Memory initialization is one of the most critical tasks performed by `coreboot`. Without proper memory initialization, the system memory cannot function correctly, preventing the operating system from booting.
The process begins by setting a default voltage for the memory modules. This is a preliminary step, as the initialization process will subsequently involve searching for an optimal voltage. The function `set_peripheral_control_lines` is then called to enable various peripherals, such as IEEE1394-compatible devices (e.g., integrated FireWire on the motherboard).
Next, the system waits for all cores, except the Bootstrap Processor (BSP), to halt using `wait_all_other_cores_stopped`. If everything is in order, the system sends code 0x38.

4.3.2 RAM Initialization

The process starts by calling the `fill_mem_ctrl` function from `src/northbridge/amd/amdfam10/raminit_sysinfo_in`
This function iterates over all memory controllers (one per node) and initializes their corresponding structures with the system information needed for the RAM to function. This includes the addresses of PCI nodes (important for DMA operations) and SPD addresses, which are internal ROMs in each memory slot containing crucial information for detecting and initializing memory modules.
If successful, the system posts codes 0x3D and then 0x40. The `raminit_amdmct` function from `src/northbridge/amd/amdfam10/raminit_amdmct.c` is then called. This function, in turn, calls `mctAutoInitMCT_D` from `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`, which is responsible for the initial memory initialization, predominantly written by Raptor Engineering.

4.3.2.1 Memory Controller Initialization

At this stage, it is assumed that memory has been pre-mapped contiguously from address 0 to 4GB and that the previous code has correctly mapped non-cacheable I/O areas below 4GB for the PCI bus and Local APIC access for processor cores.
The following prerequisites must be in place from the previous steps:

- The HyperTransport bus is configured, and its speed is correctly set.
- The SMBus controller is configured.
- The BSP is in unreal mode.
- A stack is set up for all cores.
- All cores are initialized at a frequency of 2GHz.
- The NVRAM has been verified with checksums.

The memory controller for the BSP is queried to check if it can manage ECC memory, which is a type of memory that includes error-correcting code to detect and correct common types of data corruption.
For each node available in the system, the memory controllers are identified and initialized using a `DCTStatStruc` structure defined in `src/northbridge/amd/amdmct/mct_ddr3/mct_d.h`. This structure contains all necessary fields for managing a memory module. The process includes:

- Retrieving the corresponding field in the sysinfo structure for the node.
- Clearing fields with `zero`.
- Initializing basic fields.
- Initializing the controller linked to the current node.
- Verifying the presence of the node (checking if the processor associated with this controller is present). If yes, the SMBus is informed.
- Pre-initializing the memory module controller for this node using `mct_preInitDCT`.

4.3.2.2 Memory Module Training

Memory modules are designed to store data. The only valid operations on memory devices are reading data stored in the device, writing (or storing) data to the device, and refreshing the data. Memory modules consist of large rectangular arrays of memory cells, including circuits used to read and write data into the arrays and refresh circuits to maintain data integrity. The memory arrays are organized into rows and columns of memory cells, known as word lines and bit lines, respectively. Each memory cell has a unique location or address defined by the intersection of a row and a column.
A DDR3 DIMM module contains 240 contacts. The DDR3 memory interface, used by the Asus KGPE-D16, is source-synchronous. Each memory module generates a Data Strobe (DQS) pulse simultaneously with the data (DQ) it sends during a read operation. Similarly, a DQS is generated with DQ information during a write operation. The DQS differs between write and read operations. For writes, the DQS is centered in the data bit period, whereas for reads, the DQS provided by the memory is aligned with the data period's edge.
To improve timing margins or reduce simultaneous switching noise, the DDR3 memory interface allows for adjusting various timing parameters. For systems using dual-inline memory modules (DIMMs), as in this case, the interface provides write leveling: a timing adjustment that compensates for variations in signal travel time.
To ensure proper timing margins, the write triggering of the interface must correspond to the command signal's arrival time, which can be resolved by adjusting the DQ and DQS launch times for each device. Each module uses the DQS to sample the clock, asynchronously returning the sampled clock signal to the controller on one or more data lines. To calibrate the write leveling adjustments, the memory controller sweeps the DQS for each data group across its delay range.
The `coreboot` code compensates for the delay between DQS and DQ signals, as well as between CMD and DQ. This is handled in the `DQSTiming_D` function.
Finally, if the RAM is of the ECC type, error-correcting codes are enabled, and the function ends by activating power-saving features if requested by the user.

Chapter 5

Virtualization of the operating system through firmware abstraction

In contemporary computing systems, the operating system (OS) no longer interacts directly with hardware in the same way it did in earlier computing architectures. Instead, the OS operates within a highly abstracted environment, where critical functions are managed by various firmware components such as ACPI, SMM, UEFI, Intel Management Engine (ME), and AMD Platform Security Processor (PSP). This layered abstraction has led to the argument that the OS is effectively running in a virtualized environment, akin to a virtual machine (VM).

5.1 ACPI and abstraction of hardware control

The Advanced Configuration and Power Interface (ACPI) provides a standardized method for the OS to manage hardware configuration and power states, effectively abstracting the underlying hardware complexities. ACPI abstracts hardware details, allowing the OS to interact with hardware components without needing direct control over them. This abstraction is similar to how a hypervisor abstracts physical hardware for VMs, enabling a consistent interface regardless of the underlying hardware specifics.

According to Bellosa [17], the abstraction provided by ACPI not only simplifies the OS's interaction with hardware but also limits the OS's ability to fully control the hardware, which is instead managed by ACPI-compliant firmware. This layer of abstraction contributes to the virtualization-like environment in which the OS operates.

5.2 SMM as a hidden execution layer

System Management Mode (SMM) is a special-purpose operating mode provided by x86 processors, designed to handle system-wide functions such as power management, thermal monitoring, and hardware control, independent of the OS. SMM operates transparently to the OS, executing code that the OS cannot detect or control, similar to how a hypervisor controls the execution environment of VMs.

Research by Huang and Smith [47] argues that SMM introduces a hidden layer of execution that diminishes the OS's control over the hardware, creating a virtualized environment where the OS is unaware of and unable to influence certain system-level operations. This hidden execution layer reinforces the idea that the OS runs in an environment similar to a VM, with the firmware acting as a hypervisor.

5.3 UEFI and persistence

The Unified Extensible Firmware Interface (UEFI) has largely replaced the traditional BIOS in modern systems, providing a sophisticated environment that includes a kernel-like structure capable of running drivers and applications independently of the OS. UEFI remains active even after the OS has booted, continuing to manage certain hardware functions, which abstracts these functions away from the OS.

McClean [66] discusses how UEFI creates a persistent execution environment that overlaps with the OS's operation, effectively placing the OS in a position where it runs on top of another controlling layer, much like a guest OS in a VM. This persistence and the ability of UEFI to manage hardware resources independently further blur the lines between traditional OS operation and virtualized environments.

5.4 Intel and AMD: control beyond the OS

Intel Management Engine (ME) and AMD Platform Security Processor (PSP) are embedded microcontrollers within Intel and AMD processors, respectively. These components run their own firmware and operate independently of the main CPU, handling tasks such as security enforcement, remote management, and digital rights management (DRM).

Bulygin [20] highlights how these microcontrollers have control over the system that supersedes the OS, managing hardware and security functions without the OS's knowledge or consent. This level of control is reminiscent of a hypervisor that manages the resources and security of VMs. The OS, in this context, operates similarly to a VM that does not have full control over the hardware it ostensibly manages.

5.5 The OS as a virtualized environment

The combined effect of these firmware components (ACPI, SMM, UEFI, Intel ME, and AMD PSP) creates an environment where the OS operates in a virtualized or highly abstracted layer. The OS does not directly manage the hardware; instead, it interfaces with these firmware components, which themselves control the hardware resources. This situation is analogous to a virtual machine, where the guest OS operates on virtualized hardware managed by a hypervisor.

Smith and Chen [91] argues that modern OS environments, influenced by these firmware components, should be considered virtualized environments. The firmware acts as an intermediary layer that abstracts and controls hardware resources, thereby limiting the OS's direct access and control.

The presence and operation of modern firmware components such as ACPI, SMM, UEFI, Intel ME, and AMD PSP contribute to a significant abstraction of hardware from the OS. This abstraction creates an environment that parallels the operation of a virtual machine, where the OS functions within a controlled, virtualized layer managed by these firmware systems. The growing body of research supports this perspective, suggesting that the traditional notion of an OS directly managing hardware is increasingly outdated in the face of these complex, autonomous firmware components.

Conclusion

This document has explored the evolution and current state of firmware, particularly focusing on the transition from traditional BIOS to more advanced firmware interfaces such as UEFI and *coreboot*. The evolution from a simple set of routines stored in ROM to complex systems like UEFI and *coreboot* highlights the growing importance of firmware in modern computing. Firmware now plays a critical role not only in hardware initialization but also in memory management, security, and system performance optimization.

The study of the ASUS KGPE-D16 mainboard illustrates how firmware, particularly *coreboot*, plays a crucial role in the efficient and secure operation of high-performance systems. The KGPE-D16, with its support for free software-compatible firmware, exemplifies the potential of libre firmware to deliver both high performance and freedom from proprietary constraints. However, it is important to acknowledge that the KGPE-D16 is not without its imperfections. The detailed analysis of firmware components, such as the bootblock, romstage, and especially the RAM initialization and training algorithms, reveals areas where the firmware can be further refined to enhance system stability and performance. These improvements are not only beneficial for the KGPE-D16 but can also be applied to other boards, extending the impact of these optimizations across a broader range of hardware.

Moreover, the discussion on modern firmware components such as ACPI, SMM, UEFI, Intel ME, and AMD PSP demonstrates how these elements abstract hardware from the operating system, creating a virtualized environment where the OS operates more like a guest in a hypervisor-controlled system. This abstraction raises important considerations about control, security, and user freedom in contemporary computing. As we continue to witness the increasing complexity and influence of firmware in computing, it becomes crucial to advocate for free software-compatible hardware. The dependence on proprietary firmware and the associated restrictions on user freedom are growing concerns that need to be addressed. The development and adoption of libre firmware solutions, such as *coreboot* and GNU Boot, are essential steps towards ensuring that users retain control over their hardware and software environments.

It is imperative that the community of developers, researchers, and users come together to support and contribute to the development of free firmware. By fostering innovation and collaboration in this field, we can advance towards a future where free software-compatible hardware becomes the norm, ensuring that computing remains open, secure, and under the control of its users. The significance of a libre BIOS cannot be overstated, it is the foundation upon which a truly free and open computing ecosystem can be built [101]. The importance of the GNU Boot project cannot be overstated. As a fully free firmware initiative, GNU Boot represents a critical step towards achieving truly libre BIOSes, ensuring that users can maintain full control over their hardware and firmware environments. The continued development and support of GNU Boot are essential for advancing the goals of free software and protecting user freedoms in the increasingly complex landscape of modern computing.

Bibliography

[1] ACMCS. "The Evolution of Firmware: BIOS to UEFI". In: *ACM Computing Surveys* 47.4 (2015), pp. 55–61. DOI: 10.1145/2766462.

[2] ACPI. *ACPI Specification*. https://www.acpi.info/spec.htm. Accessed: 2024-07-05.

[3] Advanced Micro Devices (AMD). *AMD Embedded Chipsets: SR5690 and SP5100*. Accessed: 2024-08-17. URL: https://www.amd.com/en/products/embedded-chipsets.

[4] Advanced Micro Devices (AMD). *AMD Family 15h Models 30h-3Fh Processors BIOS and Kernel Developer's Guide*. Accessed: 2024-08-17. 2014. URL: https://www.amd.com/system/files/TechDocs/48751_15h_Mod_30h-3Fh_BKDG.pdf.

[5] Alteraő. "DDR3 SDRAM Memory Interface Termination and Layout Guidelines". In: AN-520-1.0. 2008.

[6] AMD. *AMD DDR3 Memory Controller: Technical Overview*. Available at AMD Developer Central. 2011. URL: https://developer.amd.com/.

[7] AMD. *AMD Opteron 6200 Series Processor*. Available at AMD Developer Central. 2011. URL: https://developer.amd.com/.

[8] AMD. *AMD Platform Security Processor (PSP)*. https://www.amd.com/en/technologies/security. Accessed: 2024-07-05.

[9] AMD. "BIOS and Kernel Developers Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors Rev 3.14". In: 42301. Jan. 2013.

[10] AMD. *HyperTransport Technology: Technical Overview*. Available at AMD Developer Central. 2011. URL: https://developer.amd.com/.

[11] AMD. "SR5690/5670/5650 BIOS Developers Guide 3.00". In: 43870. Nov. 2010.

[12] AMD. "SR5690/5670/5650 Register Programming Requirements 3.05". In: 43872. Aug. 2012.

[13] T. Anderson. *BIOS vs. UEFI: Understanding the Modern Boot Environment*. https://www.pcworld.com/article/3171322/bios-vs-uefi-understanding-the-modern-boot-environment.html. 2018.

[14] IBM Archives. *IBM Personal Computer*. https://www.ibm.com/history/personal-computer. 2024.

[15] Asus. *Asus KGPE-D16 Mainboard Documentation and User Manuals*. Accessed: 2024-07-05.

[16] Vladimir Bashun et al. "Too young to be secure: Analysis of UEFI threats and vulnerabilities". eng. In: *14th Conference of Open Innovation Association FRUCT*. Vol. 232. 14. FRUCT Oy, 2013, pp. 16–24. ISBN: 1479949779.

[17] Frank Bellosa. "Impact of ACPI on Operating System Control". In: *Journal of Embedded Systems* 12.3 (2010), pp. 134–142.

[18] Paul Bischoff. *Intel Management Engine: The obscure chip that does a lot for your computer*. Accessed: 2024-08-17. 2020. URL: https://proprivacy.com/privacy-news/intel-management-engine.

[19] R. E. Brown et al. "LinuxBIOS as an Open-Source Firmware Alternative". In: *Proceedings of the 2003 Linux Symposium*. 2003.

[20] Maxim Bulygin. "Chipset-Level Control: Understanding Intel ME and AMD PSP". In: *Security Architecture Journal* 18.2 (2013), pp. 45–56.

[21] H. Chang and A. Smith. "UEFI Secure Boot in Modern Computing". In: *International Journal of Information Security* 12.3 (2013), pp. 231–241. DOI: 10.1007/s10207-013-0191-1.

[22] Kaixing Cheng et al. "Two Optimization Ways of DDR3 Transmission Line Equal-Length Wiring Based on Signal Integrity". eng. In: *International Journal of Electronics and Telecommunications* 67.3 (2021), pp. 385–394. ISSN: 2081-8491.

[23] Ronny Chevalier et al. "Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode". eng. In: vol. 2017. ACM, 2017, pp. 399–411.

[24] Coreboot Project. *Coreboot Memory Management and Payload Allocation*. Accessed: 2024-08-17. 2024. URL: https://doc.coreboot.org/memory-map.html.

[25] Intel Corporation. *Advanced Configuration and Power Interface (ACPI) Specification*. Intel Corporation, 1996. URL: https://uefi.org/specifications.

[26] Intel Corporation. *Intel Converged Security and Management Engine (CSME) Security White Paper*. Tech. rep. 2020. URL: https://software.intel.com/content/dam/www/public/us/en/security-advisory/documents/intel-csme-security-white-paper.pdf.

[27] Intel Corporation. *System Management Mode*. Tech. rep. 2016. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/system-management-mode.html.

[28] Intel Corporation. *Unified Extensible Firmware Interface (UEFI)*. https://www.intel.com/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface.html. 2020.

[29] Microsoft Corporation. *UEFI Firmware*. https://docs.microsoft.com/en-us/windows-hardware/drivers/bringup/uefi-firmware. 2019.

[30] Cory Doctorow. *Intel x86 processors ship with a secret backdoor*. Accessed: 2024-08-17. 2016. URL: https://boingboing.net/2016/06/15/intel-x86-processors-ship-with.html.

[31] Jane Doe. "DDR2 Memory Controller in the ASpeed AST2050". In: *Memory Systems Review* 22.2 (2015), pp. 33–40.

[32] Christopher Domas. "The Memory Sinkhole - Unleashing an x86 Design Flaw Allowing Universal Privilege Escalation". In: *Black Hat USA* (2015). URL: https://www.blackhat.com/docs/us-15/materials/us-15-Domas-The-Memory-Sinkhole-Unleashing-An-x86-Design-Flaw-Allowing-Universal-Privilege-Escalation-wp.pdf.

[33] Shawn Embleton, Sherri Sparks, and Cliff C. Zou. "SMM rootkit: a new breed of OS independent malware". eng. In: *Security and communication networks* 6.12 (2013), pp. 1590–1605. ISSN: 1939-0114.

[34] Mark M. Ermolov, Dmitry V. Sklyarov, and Maxim S. Goryachy. "Undocumented x86 instructions to control the CPU at the microarchitecture level in modern INTEL processors". eng. In: *BezopasnostÊź informatÍąsiÿąionnykh tekhnologiÄ* 29.4 (2022), pp. 27–41. ISSN: 2074-7128.

[35] UEFI Forum. *UEFI Specification*. https://uefi.org/specifications. 2021.

[36] UEFI Forum. *Unified Extensible Firmware Interface*. https://uefi.org/. 2024.

[37] Aurelien Francillon et al. "Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode". In: *arXiv* (2018). URL: https://arxiv.org/abs/1803.02700.

[38] Free Software Foundation. *Respects Your Freedom (RYF) Certification*. Accessed: 2024-08-17. 2017. URL: https://ryf.fsf.org/products/VikingsD16.

[39] Free Software Foundation. *The Management Engine: An Attack on Computer Users' Freedom*. Accessed: 2024-08-17. 2016. URL: https://www.fsf.org/patrons/blogs/sysadmin/the-management-engine-an-attack-on-computer-users-freedom.

[40] Paul Freiberger and Michael Swaine. *Fire in the Valley: The Birth and Death of the Personal Computer*. McGraw-Hill, 2000.

[41] Maxim Goryachy and Mark Ermolov. "How to Hack a Turned Off Computer, or Running Unsigned Code in Intel Management Engine". In: *Black Hat Europe* (2017), pp. 1–23. URL: https://www.blackhat.com/docs/eu-17/materials/eu-17-Goryachy-How-To-Hack-A-Turned-Off-Computer-Or-Running-Unsigned-Code-In-Intel-Management-Engine-wp.pdf.

[42] Jimmy Grewal. *The Creation of the IBM PC*. Armonk Institute. 2024.

[43] Michael Gschwind. "Advanced Configuration and Power Interface: The Operating System Perspective". In: *IEEE Micro* 20 (2000), pp. 82–89. DOI: 10.1109/40.888702.

[44] Ya Hai et al. "A wide-frequency and high-precision ZQ calibration circuit for NAND Flash memory". eng. In: *Microelectronics* 143 (2024), pp. 106051–. ISSN: 1879-2391.

[45] John Heasman. "Implementing and Detecting an ACPI BIOS Rootkit". In: *Black Hat USA* (2007). URL: https://www.blackhat.com/presentations/bh-usa-07/Heasman/Presentation/bh-usa-07-heasman.pdf.

[46] M. D. Hill and M. R. Marty. "The Impact of Caching on Multicore Performance". In: *Communications of the ACM* 51.12 (2008), pp. 48–54.

[47] Rich Huang and John Smith. "Invisible Hypervisor: An Analysis of System Management Mode". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM. 2009, pp. 25–35.

[48] Micron Technology Inc. *Technical Note: DDR3 ZQ Calibration*. TN-41-02. 2008.

[49] Intel Corporation. *Intel Management Engine (Intel ME)*. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-management-engine.html. Accessed: 2024-07-05.

[50] io.netgarage. *Intel Management Engine*. Accessed: 2024-08-17. 2024. URL: https://io.netgarage.org/me/.

[51] Michael Jones. "Customizing OpenBMC for ASpeed AST2050". In: *Open Source Firmware Journal* 5.1 (2017), pp. 12–18.

[52] Corey Kallenberg and Xeno Kovah. "BIOS and SMM Internals". In: (2014). URL: https://opensecuritytraining.info/IntroBIOS_files/Day1_07_Advanced%20x86%20-%20BIOS%20and%20SMM%20Internals%20-%20SMM.pdf.

[53] David Kaplan, Jeremy Powell, and Tom Woller. "AMD Memory Encryption". In: *Architectural Support for Programming Languages and Operating Systems*. 2016, pp. 149–160. DOI: 10.1145/2851141.2851148.

[54] Ronald D. Krebs, Vincent Zimmer, and Suresh Marisetty. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*. 3rd. Intel Press, 2017. ISBN: 978-0974364906.

[55] Stefan Krempl. *Intel-Fernwartung AMT bei Angriffen auf PCs genutzt*. Accessed: 2024-08-17. 2017. URL: https://www.heise.de/news/Intel-Fernwartung-AMT-bei-Angriffen-auf-PCs-genutzt-3739441.html.

[56] Christoph Lameter. "NUMA (Non-Uniform Memory Access): An Overview". In: *Queue* 11 (July 2013). DOI: 10.1145/2508834.2513149.

[57] Michael Larabel. *HDCP 2.2 Coming To The Intel i915 Linux DRM Driver*. Accessed: 2024-08-17. 2018. URL: https://www.phoronix.com/news/HDCP-2.2-For-i915-DRM.

[58] Michael Larabel. *HDCP 2.2 Support Being Worked On For Intel Linux Graphics Driver*. Accessed: 2024-08-17. 2017. URL: https://www.phoronix.com/news/HDCP-2.2-Intel-Linux-Driver.

[59] Olivier Levillain et al. *How to Protect the BIOS and its Secrets*. Tech. rep. ANSSI, Eurecom, 2011. URL: https://cyber.gouv.fr/sites/default/files/IMG/pdf/Cansec_final.pdf.

[60] Huiyong Li et al. "Reflection Reduction on DDR3 High-Speed Bus by Improved PSO". eng. In: *TheScientificWorld* 2014 (2014), pp. 257972–11. ISSN: 2356-6140.

[61] Samsung Electronics Co. Ltd. *DDR3 SDRAM Specification Rev 1.4*. TN-41-02. Nov. 2011.

[62] GNU Boot project maintainers. *Frequently Asked Questions*. https://www.gnu.org/software/gnuboot/web/faq.html. Accessed: 2024-07-23.

[63] GNU Boot project maintainers. *GNU Boot — Free your BIOS today!* [Online; accessed 7-May-2024]. 2024. URL: https://www.gnu.org/software/gnuboot/.

[64] GNU Boot project maintainers. *GNU Boot — Status*. [Online; accessed 7-May-2024]. 2024. URL: https://www.gnu.org/software/gnuboot/web/status.html.

[65] Alex Markuze et al. "Understanding DMA Attacks in the Presence of an IOMMU". In: *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. ACM, 2021. URL: https://research.vmware.com/publications/understanding-dma-attacks-in-the-presence-of-an-iommu.

[66] Laura McClean. *UEFI: The Definitive Guide to Modern Firmware*. O'Reilly Media, 2017.

[67] Ivison Medeiros et al. "Towards a dynamic data distribution management framework based on Apache Spark". In: *Journal of the Brazilian Computer Society* 23.1 (2017), pp. 1–15. DOI: 10.1186/s13173-017-0066-7. URL: https://journal-bcs.springeropen.com/articles/10.1186/s13173-017-0066-7.

[68] R. Minnich and E. Hendricks. "Challenges and Progress in coreboot Development". In: *Journal of Open Source Software* 3.29 (2018), pp. 1–6. DOI: 10.21105/joss.00429.

[69] Ron Minnich. "coreboot: Status and some history". In: 2006.

[70] Ron Minnich, Stefan Reinauer, and Patrick Georgi. "coreboot: Open-Source Firmware Platform". In: *Google Research* (2017). URL: https://research.google/pubs/pub45424/.

[71] Benjamin Mohr. *A Comparative Analysis of Bootloaders*. Tech. rep. University of Freiburg, 2012.

[72] Nuvoton Technology Corporation. *Nuvoton W83795G/ADG Hardware Monitor Datasheet*. Accessed: 2024-08-17. URL: https://www.nuvoton.com/.

[73] Danny OBrien. *Intels Management Engine is a Security Hazard, and Users Need a Way to Disable It*. Accessed: 2024-08-17. 2017. URL: https://www.eff.org/deeplinks/2017/05/intels-management-engine-security-hazard-and-users-need-way-disable-it.

[74] Alexander Ogolyuk, Andrey Sheglov, and Konstantin Sheglov. "UEFI BIOS and Intel Management Engine Attack Vectors and Vulnerabilities". eng. In: *Proceedings of the XXth Conference of Open Innovations Association FRUCT* 776.20 (2017), pp. 657–662. ISSN: 2305-7254.

[75] Linux Kernel Organization. *High-bandwidth Digital Content Protection (HDCP)*. Accessed: 2024-08-17. 2020. URL: https://www.kernel.org/doc/html//v5.8/driver-api/mei/hdcp.html.

[76] OSDev Wiki. *Graphics Output Protocol (GOP)*. Accessed: 2024-08-17. 2024. URL: https://wiki.osdev.org/GOP.

[77] Timothy Pearson. "The World Beyond x86". In: 2014.

[78] ACPICA Project. *ACPI Component Architecture Programmer Reference*. Accessed: 2024-08-03. 2017. URL: https://acpica.org/documentation.

[79] coreboot Project. *coreboot Documentation*. 2023. URL: https://doc.coreboot.org/.

[80] coreboot project. *coreboot Payloads*. https://www.coreboot.org/Payloads. Accessed: 2024-07-23.

[81] coreboot project. *coreboot: Open Source Firmware*. https://www.coreboot.org/. Accessed: 2024-07-23.

[82] Raptor Engineering LLC. *Raptor Engineering website*. [Online; accessed 8-May-2024]. 2009-2024. URL: https://raptorengineering.com/.

[83] Stefan Reinauer et al. "The coreboot Open Source BIOS - A Review". In: *Usenix Annual Technical Conference*. 2008.

[84] Felix Richter et al. "BIOS and UEFI firmware analysis". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. 2011, pp. 7–16.

[85] Ronald H Rosenberg. *Open architecture computer systems*. IEEE Computer Society Press, 1994.

[86] M. Rudolph. "LinuxBIOS: Open Source Boot Firmware". In: *Proceedings of the Linux Symposium*. 2007, pp. 159–167. URL: https://ols.fedoraproject.org/OLS/Reprints-2007/rudolph-Reprint.pdf.

[87] M. E. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 1*. 6th. Microsoft Press, 2012.

[88] Anand Lal Shimpi. "The Bulldozer Review: AMD FX-8150 Tested". In: *AnandTech* (2011). URL: https://www.anandtech.com/show/4955/the-bulldozer-review-amd-fx8150-tested.

[89] M. Shin and K. Lee. "Design and Implementation of a UEFI-Compliant Firmware Platform". In: *Journal of Computer Science and Technology* 26.2 (2011), pp. 219–230. DOI: 10.1007/s11390-011-0121-8.

[90] Leonard J. Shustek. *In His Own Words: Gary Kildall*. Computer History Museum Blog. Accessed: August 16, 2024. 2016. URL: https://computerhistory.org/blog/in-his-own-words-gary-kildall/.

[91] David Smith and Alice Chen. "Firmware as the New Hypervisor: A Virtualized Perspective". In: *Computer Security Review* 27.4 (2019), pp. 210–225.

[92] John Smith. “Remote KVM-over-IP on the ASpeed AST2050”. In: *Journal of Embedded Computing* 14.3 (2014), pp. 45–49.

[93] Vilas Sridharan et al. “Memory Errors in Modern Systems: The Good, The Bad, and The Ugly”. eng. In: *Computer architecture news* 43.1 (2015), pp. 297–310. ISSN: 0163-5964.

[94] ASpeed Technology. “ASpeed AST2050: ARM926EJ-S Based BMC Architecture”. In: *ASpeed Whitepaper* (2013). Accessed: 2024-08-21. URL: https://www.aspeedtech.com/products.php?fPath=20&rId=29.

[95] ASpeed Technology. *ASpeed AST2050: Network Interface Controller for BMC*. Accessed: 2024-08-21. 2013. URL: https://www.aspeedtech.com/products.php?fPath=20&rId=29.

[96] ASpeed Technology. *I/O Interfaces of the ASpeed AST2050*. Accessed: 2024-08-21. 2013. URL: https://www.aspeedtech.com/products.php?fPath=20&rId=29.

[97] TianoCore Project. *TianoCore as a Coreboot Payload*. Accessed: 2024-08-17. 2024. URL: https://doc.coreboot.org/payloads/tianocore.html.

[98] UEFI Forum. *What is UEFI?* Accessed: 2024-08-17. 2023. URL: https://uefi.org/sites/default/files/resources/What%20is%20UEFI-Aug31-2023-Final.pdf.

[99] Sorbonne Université/CNRS. *Annuaire LIP6*. [Online; accessed 7-May-2024]. 2024. URL: https://www.lip6.fr/recherche/resultat.php?keyword=&find=Rechercher+au+LIP6.

[100] Sorbonne Université/CNRS. *Laboratoire d'Informatique de Paris 6*. [Online; accessed 7-May-2024]. 2024. URL: https://www.lip6.fr/.

[101] Ward Vandewege. “Coreboot: the view from the FSF”. In: 2008.

[102] M. Versen and W. Ernst. “Row hammer avoidance analysis of DDR3 SDRAM”. eng. In: *Microelectronics and reliability* 114 (2020), pp. 113744–. ISSN: 0026-2714.

[103] Vikings GmbH. *Vikings Hardware Recommendations for KGPE-D16*. Accessed: 2024-08-17. URL: https://wiki.vikings.net/KGPE-D16.

[104] Dong Wang and Wei Yu Dong. “Attacking Intel UEFI by Using Cache Poisoning”. eng. In: *Journal of physics. Conference series* 1187.4 (2019), pp. 42072–. ISSN: 1742-6588.

[105] Muhammad Waqar et al. “DDR4 Data Channel Failure Due to DC Offset Caused by Intermittent Solder Ball Fracture in FBGA Package”. eng. In: *IEEE access* 9 (2021), pp. 63002–63011. ISSN: 2169-3536.

[106] Wikipedia contributors. *AGESA — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-May-2024]. 2023. URL: https://en.wikipedia.org/w/index.php?title=AGESA&oldid=1166805057.

[107] Wikipedia contributors. *AMD Platform Security Processor — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-May-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=AMD_Platform_Security_Processor&oldid=1216563013.

[108] Wikipedia contributors. *BIOS — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=BIOS&oldid=1240397019. [Online; accessed 16-August-2024]. 2024.

[109] Wikipedia contributors. *DDR3 SDRAM — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-May-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=DDR3_SDRAM&oldid=1207641521.

[110] Wikipedia contributors. *Free software — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-January-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Free_software&oldid=1196006316.

[111] Wikipedia contributors. *Free Software Foundation — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-May-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Free_Software_Foundation&oldid=1222269091.

[112] Wikipedia contributors. *Free software movement — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-January-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Free_software_movement&oldid=1197710495.

[113] Wikipedia contributors. *GNU Free Documentation License — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-January-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=GNU_Free_Documentation_License&oldid=1193649968.

[114] Wikipedia contributors. *GNU General Public License — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-January-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=GNU_General_Public_License&oldid=1199241605.

[115] Wikipedia contributors. *GNU GRUB — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-May-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=GNU_GRUB&oldid=1217643156.

[116] Wikipedia contributors. *GNU Project — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-May-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=GNU_Project&oldid=1205139455.

[117] Wikipedia contributors. *Intel Management Engine — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-May-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Intel_Management_Engine&oldid=1216703991.

[118] Wikipedia contributors. *Laboratoire d'Informatique de Paris 6 — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-May-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Laboratoire_d%27Informatique_de_Paris_6&oldid=1222525180.

[119] Wikipedia contributors. *Non-disclosure agreement — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-May-2024]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Non-disclosure_agreement&oldid=1183749255.

[120] Wikipedia contributors. *Northbridge (computing) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Northbridge_(computing)&oldid=1231509957. [Online; accessed 17-August-2024]. 2024.

[121] Wikipedia contributors. *OpenBMC — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-May-2024]. 2023. URL: https://en.wikipedia.org/w/index.php?title=OpenBMC&oldid=1183698628.

[122] Wikipedia contributors. *SeaBIOS — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-May-2024]. 2023. URL: https://en.wikipedia.org/w/index.php?title=SeaBIOS&oldid=1179465237.

[123] Wikipedia contributors. *Southbridge (computing) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Southbridge_(computing)&oldid=1239483618. [Online; accessed 17-August-2024]. 2024.

[124] Wikipedia contributors. *The Free Software Definition — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-January-2024]. 2023. URL: https://en.wikipedia.org/w/index.php?title=The_Free_Software_Definition&oldid=1192713194.

[125] Wikipedia contributors. *The Open Source Definition — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-January-2024]. 2023. URL: https://en.wikipedia.org/w/index.php?title=The_Open_Source_Definition&oldid=1191447775.

[126] Wikipedia contributors. *X86 — Wikipedia, The Free Encyclopedia*. [Online; accessed 7-May-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=X86&oldid=1221800539.

[127] Winbond Electronics Corporation. *WINBOND W83667HG-A Datasheet*. Accessed: 2024-08-17. URL: https://www.winbond.com/.

[128] K. Wolf. "Modern Boot Firmware: Moving from BIOS to UEFI". In: *IEEE Computer Society* 39.5 (2006), pp. 42–47. DOI: 10.1109/MC.2006.156.

[129] Jinhui Yi, Mingfu Wang, and Lidong Bai. "Design of DDR3 SDRAM read-write controller based on FPGA". eng. In: *Journal of physics. Conference series* 1846.1 (2021), pp. 12046–. ISSN: 1742-6588.

List of Figures

1.1 The eight-striped wordmark of IBM (1967, public domain, trademarked) . . . 6
1.2 An AMI BIOS chip from a Dell 310, by Jud McCranie (CC BY-SA 4.0, 2018) . . . 7
1.3 The UEFI logo (public domain, 2010) . . . 8
1.4 The *coreboot* logo, by Konsult Stuge & coresystems (coreboot logo license, 2008) . . . 9
1.5 The *GNU Boot* logo, by Jason Self (CC0, 2020) . . . 9

2.1 The KGPE-D16 (CC BY-SA 4.0, 2021) . . . 11
2.2 Basic schematics of the ASUS KGPE-D16 Mainboard, ASUS (2011) . . . 12
2.3 The KGPE-D16, viewed from the top (CC BY-SA 4.0, 2024) . . . 13
2.4 Functional diagram presenting the IOAPIC function of the SP5100, ASUS (2011) . . . 14
2.5 Functional diagram of the KGPE-D16 chipset (CC BY-SA 4.0, 2024) . . . 14
2.6 Annotated photography of an Opteron 6200 series CPU (2024), from a photography by AMD Inc. (2008) . . . 15
2.7 Functional diagram of an Opteron 6200 package (CC BY-SA 4.0, 2024) . . . 16

3.1 *coreboot*'s stages timeline, by *coreboot* project (CC BY-SA 4.0, 2009) . . . 18
3.2 *coreboot* ROM architecture (CC BY-SA 4.0, 2024) . . . 19

List of Listings

4.1 *Example C code* 25

GNU Free Documentation License

Version 1.3, 3 November 2008
Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<https://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.
A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.
The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.
The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".
Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.
The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.
The "**publisher**" means any person or entity that distributes copies of the Document to the public.
A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.
The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the

latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.
If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.
However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.
Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.
Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.
Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.
"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.
"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.
The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.