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 1.0.

# Acknowledgments

First and foremost, I would like to express my deep gratitude to Marie-Minerve Louërat, without whom this work would not have come to fruition. Her efforts to assist me on legal matters and our enriching discussions on the philosophy of free software have been invaluable to me.

I am also thankful to Roselyne Chotin for agreeing to fund this work as without her support, none of this would have been possible, and Franck Wajsburt for his invaluable advice at the beginning of my work, which greatly helped me organize myself, as well as for his support and reviews throughout this period.

I wish to express my appreciation to the Free Software Foundation for funding the necessary equipment for this project, especially to the board members of the foundation, including Richard M. Stallman and Odile Bénassy, for making the funding for this project possible. Special thanks go to Zoë Kooyman and Ian Kelling for their exceptional support in managing the hardware order and for their kindness and assistance throughout the entire process.
I am deeply grateful to Denis Carikli (GNUtoo), my fellow GNU Boot co-maintainer, for his meticulous reviews, emotional support, and brilliant ideas that enriched this work, and to Richard M. Stallman for his advice and support throughout this journey.

A big thank you to Manuel Bouyer for his infinite patience with all my requests regarding network, software, and hardware configurations.

Also, I warmly thank my family and friends for their constant encouragement and for reviewing my work throughout this entire process.

And finally, I would like to thank the break room and its kettle, without which no tea would have been possible, thereby jeopardizing this work.

# 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.

# Contents

# List of Figures

# List of Listings

# 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 [45]. The term BIOS itself was coined by Gary Kildall, who developed the CP/M (Control Program for Microcomputers) operating system [100]. 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 [48][13], 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 [45].

### 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 [116]. 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 [13].

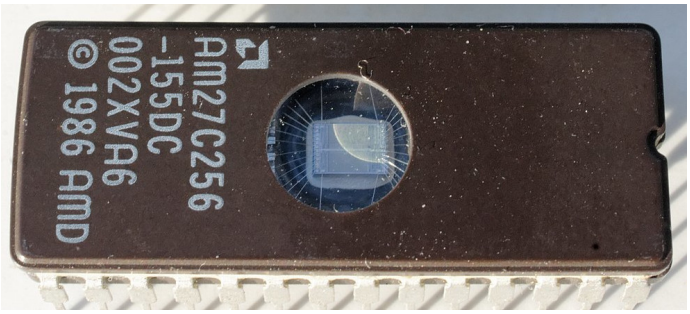


Figure 1.2: An AMI BIOS chip from a Dell 310, by Jud McCranie (CC BY-SA 4.0, 2018)

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 [85]. 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 [32]. 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 [40][97]. 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 [85][2].

## 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 [40]. 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 [32][99]. 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 [33][97].
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 [32]. 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 [2]. 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 [85][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 [40].

### 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[91]. 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. 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[91]. 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 [90].


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 [76]. For example, it provides a *verified boot* function, allowing to prevent rootkits and other attacks based on firmware modifications [89]. 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 [70], 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 only a distribution: it reuses existing software projects and is not very different from fully free GNU/Linux distributions like Trisquel or Guix, as GNU Boot is committed to use only free software for all aspects of firmware, making it a preferred choice for users and organizations that prioritize software freedom and

transparency. Its goal include to build the software and assemble it in something that can be installed, and also to test it and to provide installation and upgrade instructions [71].

## 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 [85].
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 [40][8]. Beyond memory management, firmware responsibilities have broadened to encompass a wide range of system-critical tasks, and even so by including runtime components in addition to its initialization 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 [40]. 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 [40]. 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 [85].

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

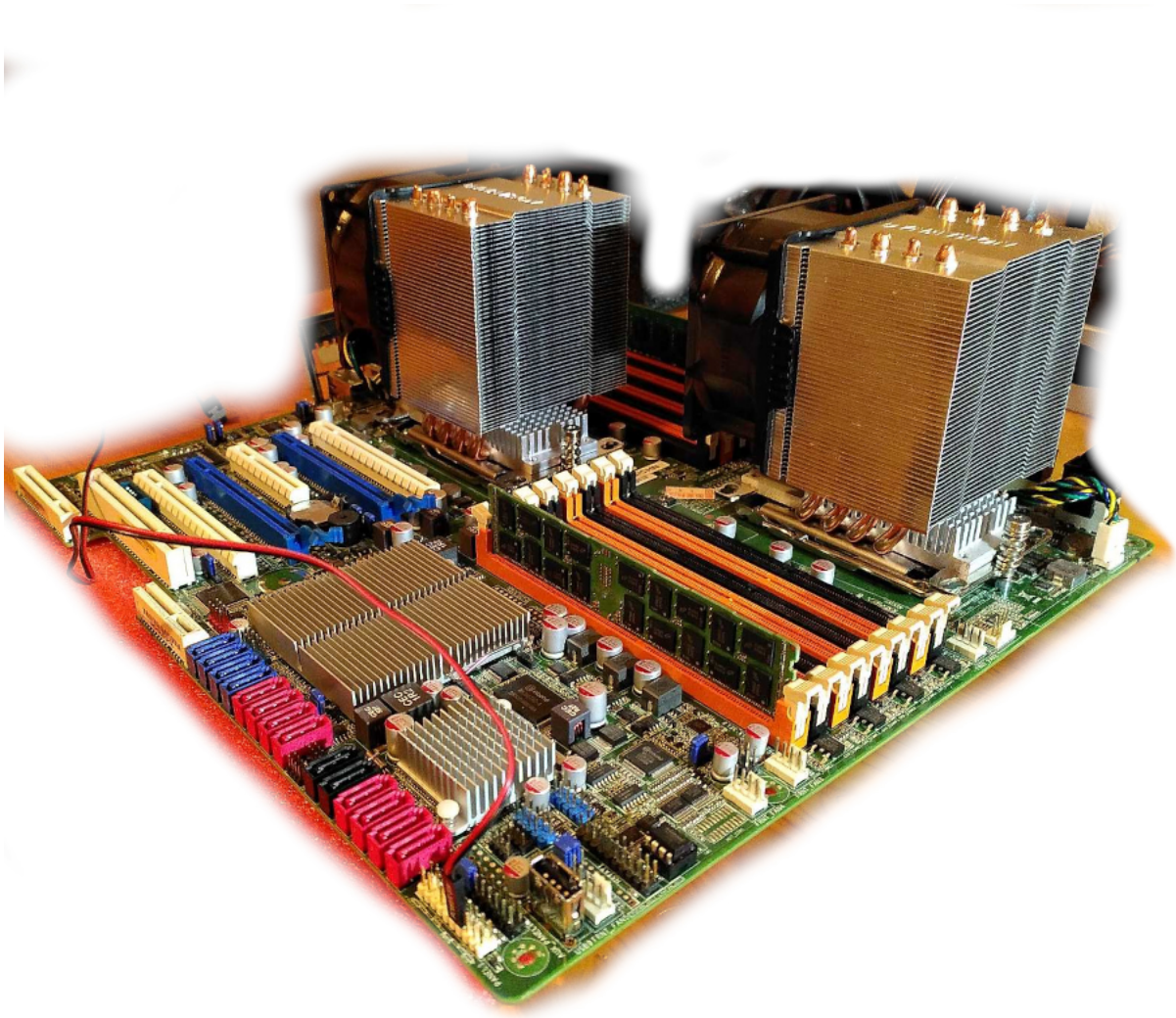

Figure 2.1: The KGPE-D16 (CC BY-SA 4.0, 2021)

## 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 [43]. Indeed, this mainboard can be operated with a fully free firmware such as GNU Boot [72].

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 [14]. 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.

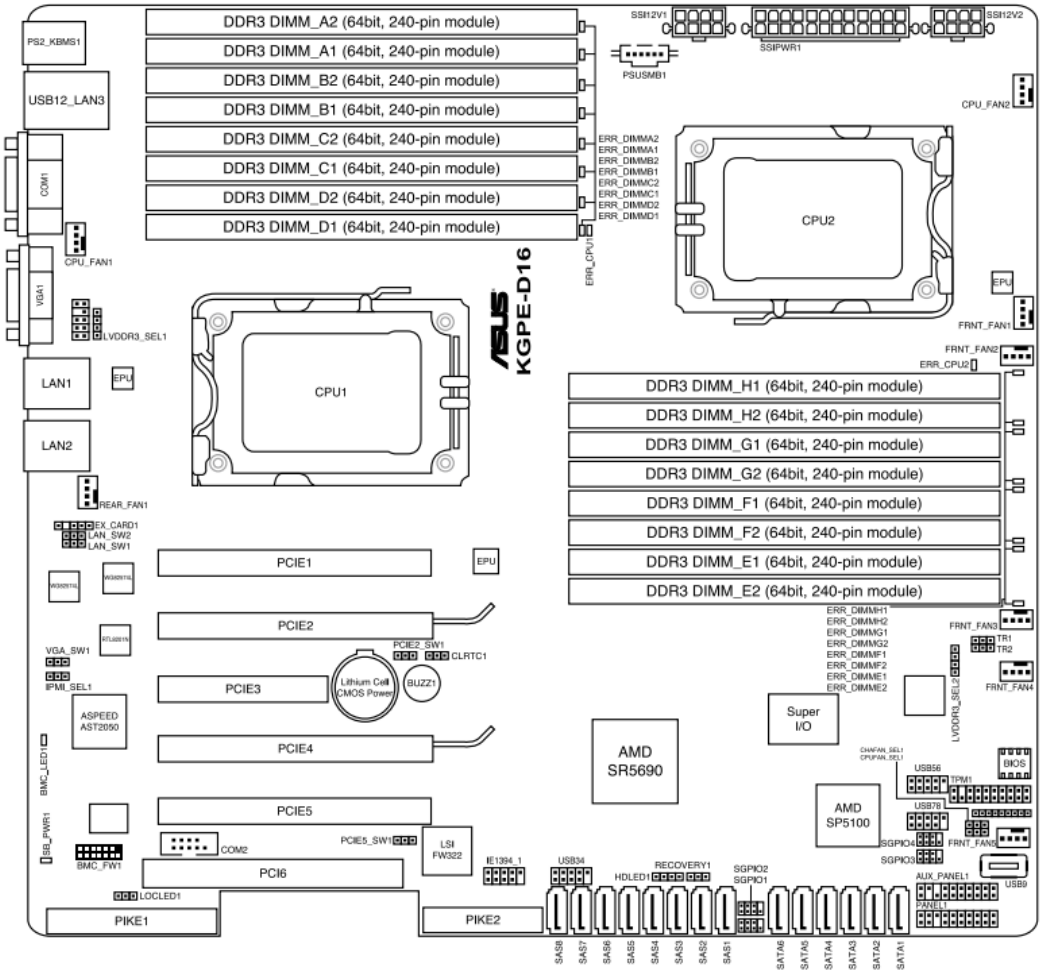


Figure 2.2: Basic schematics of the ASUS KGPE-D16 Mainboard, ASUS (2011)

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.

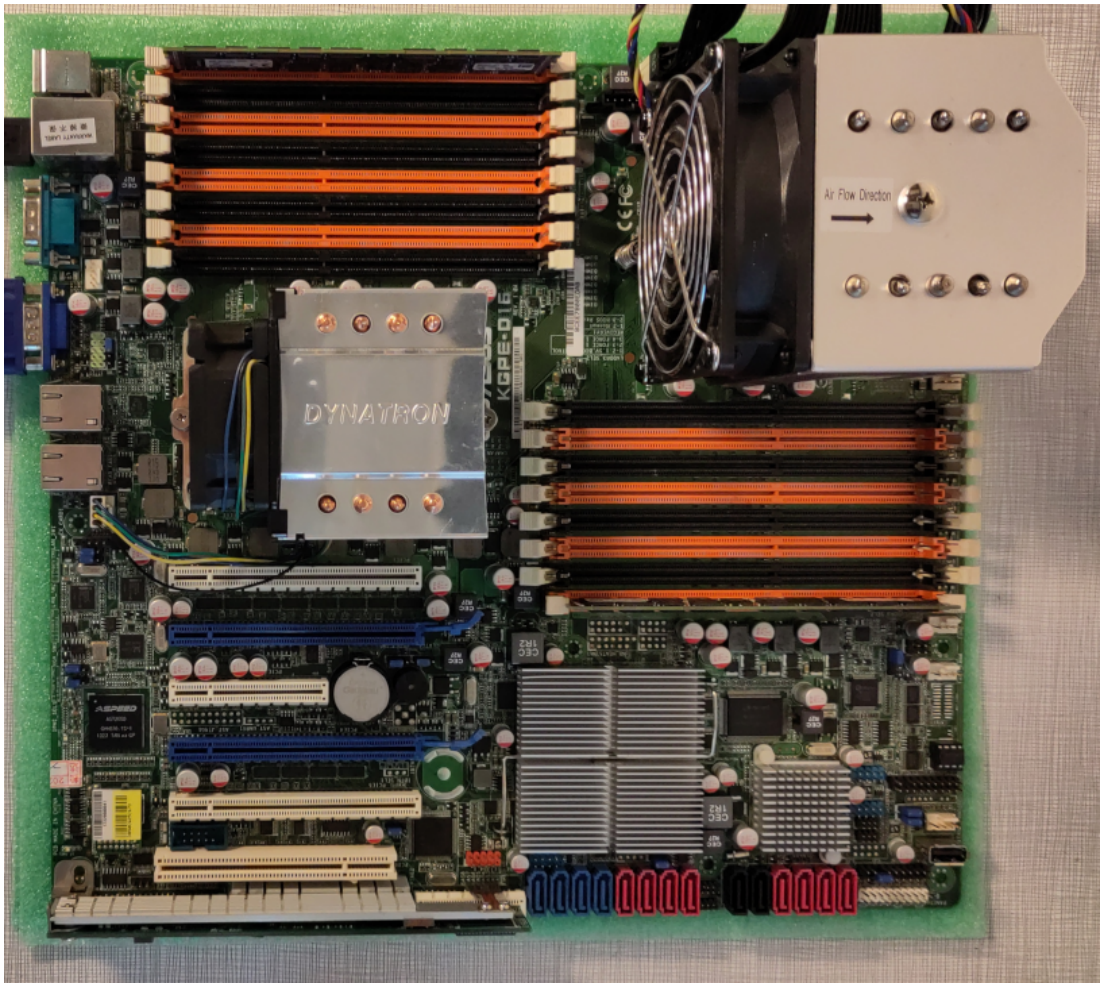


Figure 2.3: The KGPE-D16, viewed from the top (CC BY-SA 4.0, 2024)

## 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 [4].

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 [4][128].

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 [4][131]. 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].

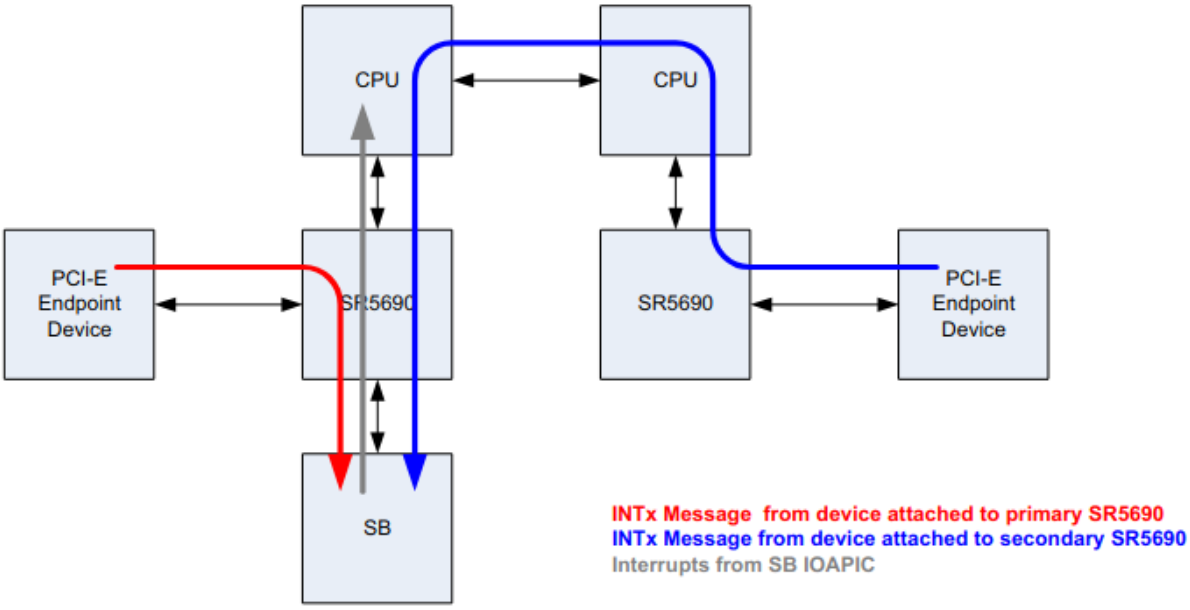


Figure 2.4: Functional diagram presenting the IOAPIC function of the SP5100, ASUS (2011)

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 (Serial Peripheral Interface, a synchronous serial communication protocol primarily used to communicate between microcontrollers and peripheral devices like sensors or memory devices) that contains the firmware [136]. 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 [80]. 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].

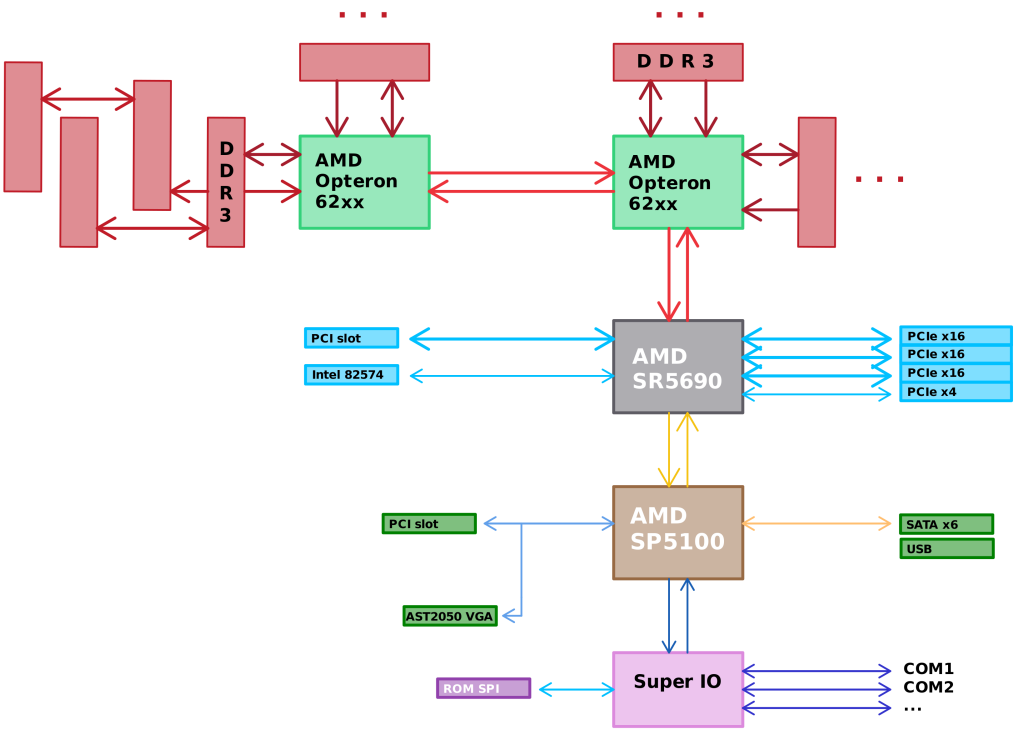


Figure 2.5: Functional diagram of the KGPE-D16 chipset (CC BY-SA 4.0, 2024)

## 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 [111].

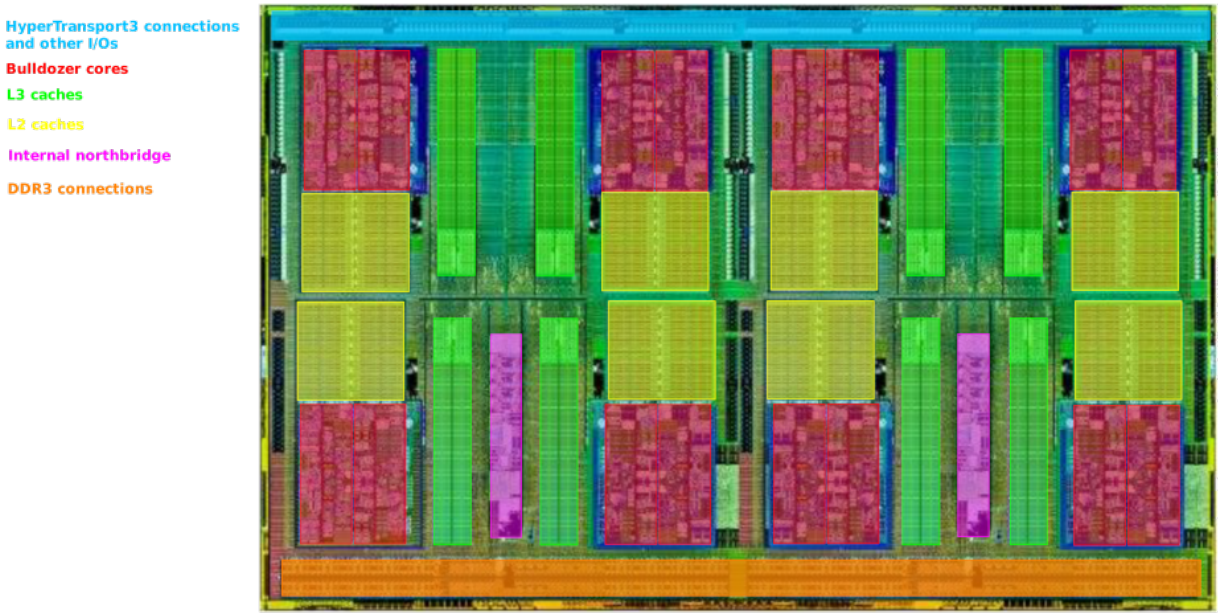


Figure 2.6: Annotated photography of an Opteron 6200 series CPU (2024), from a photography by AMD Inc. (2008)

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][98].
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][53].

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].

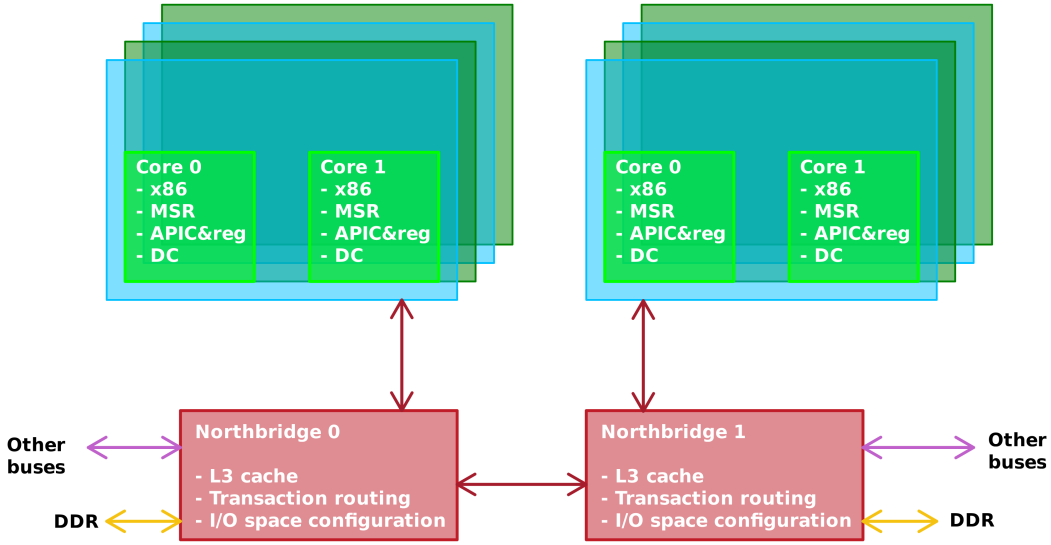


Figure 2.7: Functional diagram of an Opteron 6200 package (CC BY-SA 4.0, 2024)

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 [98]. 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 [3]. 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 occurred whether 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 [3][8].

## 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 [104]. 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 [102]. 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 [35]. 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 [105]. 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 [106]. 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 [129], a libre firmware that can be run on the AST2050 thanks to Raptor Engineering [93], 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 [59].

# 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. In this chapter and for the rest of this document, we're basing our study on the 4.11 version of *coreboot* [25], which is the last version that supported the ASUS KGPE-D16 mainboard.

For the initialization tasks to be done efficiently, *coreboot* is organized in different stages (fig. 3.1) [89].

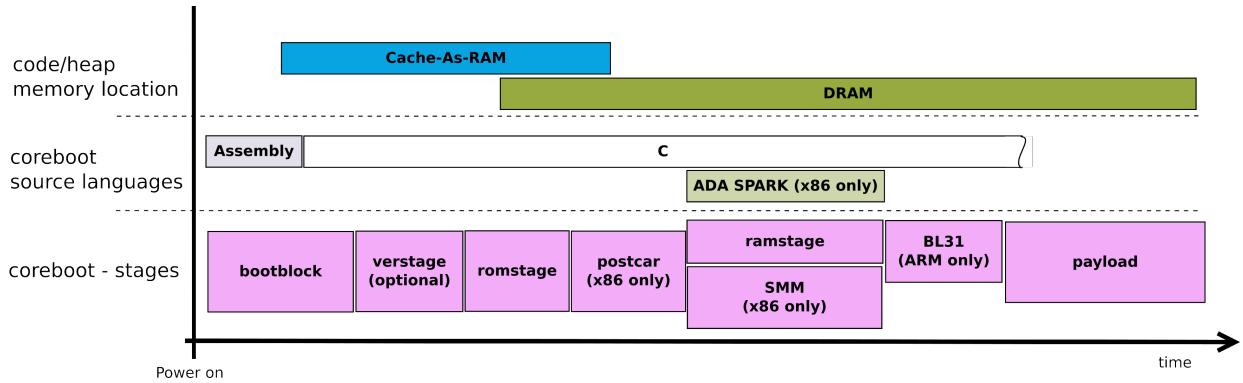


Figure 3.1: *coreboot*'s stages timeline, by *coreboot* project (CC BY-SA 4.0, 2009)

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-DEVICEMEMHIGH` | Reserved area for OS use. |

Table 3.1: *coreboot* memory map

### 3.1.1 Bootblock

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.

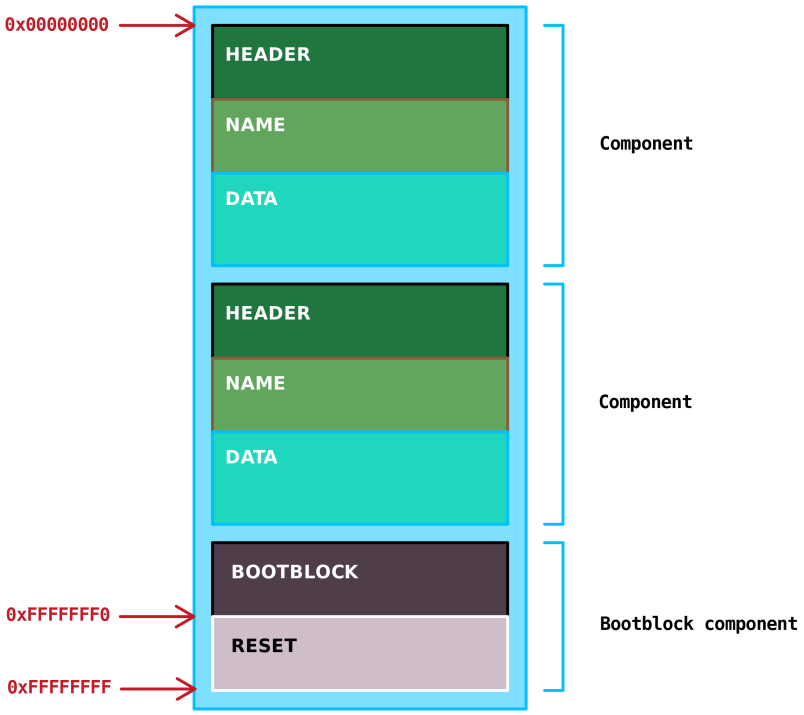


Figure 3.2: *coreboot* ROM architecture (CC BY-SA 4.0, 2024)

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 [8]. 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 (Serial Peripheral Interface, allowing to access the chip that contains 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. During this stage, microcode patches may be loaded into CPU and remain resident, settings related to memory controllers and CPU too.

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), which is a resident component in a protected area. [89]. CBMEM data structures (like coreboot tables, memory map, etc.) are populated during this stage and left resident for the OS or payload to access, same for SMBIOS tables, ACPI tables and eventually option ROMS.

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 [75][73]).

#### 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 [29]. 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 at runtime. It means that these are components from the firmware that remain resident while the OS runs. 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 [49]. 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 [88]. 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 [27]. 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 [49]. 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 [14].

#### 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 [62]. This is another firmware component that remains resident while the OS runs.

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 [51]. 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 [60]. 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 [36]. 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 [67]. 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[18]. 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 [94].
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 [79].

### 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 [89]. 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 [92]. 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 [84]. These are other examples of resident firmware components while the OS is running.
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 [52].
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 [52]. 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) [28]. 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 [44]. 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 [81]. 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 [107]. 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 standards dictated by third parties are allowed to access DRM-protected content, potentially limiting users' control over their own devices [17]. 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 [65][83]. 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 [44][57].

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 [47]. 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

## 4.1 Importance of DDR3 memory initialization

Memory modules are designed solely for storing data. The only valid operations on a memory device are reading data stored in the device, writing (or storing) data into 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 the integrity of the stored data. 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 DRAM memory cell is a capacitor that is charged to produce a 1 or a 0.

DDR3 (Double Data Rate Type 3) is a widely used type of SDRAM (Synchronous Dynamic Random-Access Memory) that offers significant performance improvements over its predecessors, DDR and DDR2. A DDR3 DIMM module contains 240 contacts. Key features of DDR3 include higher data rates, lower power consumption, and increased memory capacity, making it essential for high-performance computing environments [117]. One of the critical aspects of DDR3 is its internal architecture, which supports data rates ranging from 800 to 1600 Mbps and operates at a lower voltage of 1.5V. This enables faster data processing and more efficient power usage, crucial for modern applications that require high-speed memory access [69]. Additionally, DDR3 memory modules are available in larger capacities, allowing systems to handle larger datasets and more complex computing tasks [5]. However, the advanced features of DDR3 come with increased complexity in its initialization and operation. 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 memory read operation. Similarly, a DQS must be generated with its DQ information when writing to memory. The DQS differs between write and read operations. Specifically, the DQS generated by the system for a write operation is centered in the data bit period, while the DQS provided by the memory during a read operation is aligned with the edge of the data period [69].

Due to this edge alignment, the read DQS timing can be adjusted to meet the setup and hold requirements of the registers capturing the read data. To improve timing margins or reduce simultaneous switching noise in the system, the DDR3 memory interface also allows various other timing parameters to be adjusted. If the system uses dual-inline memory modules (DIMMs), as in our case, the interface provides write leveling: a timing adjustment that compensates for variations in signal travel time [55]. To reduce simultaneous switching noise, DIMM modules feature a fly-by architecture for routing the address, command, and clock signals, which causes command signals to reach the different memory devices with a delay. The fly-by topology has a "daisy-chain" structure with either very short stubs or no stubs at all. This structure results in fewer branches and point-to-point connections: everything originates from the controller, passing through each module on the node, thereby increasing the throughput. In this topology, signals are routed sequentially from the memory controller to each DRAM chip, reducing signal reflections and improving overall signal integrity. It means that routing is done in the order of byte lane numbers, and the data byte lanes are routed on the same layer. Routing can be simplified by swapping data bits within a byte lane if necessary. The fly-by topology contrasts with the dual-T topology (fig. 4.1). This design is essential for maintaining stability at the high speeds DDR3 operates at, but it also introduces timing challenges, such as timing skew, that must be carefully managed [55].

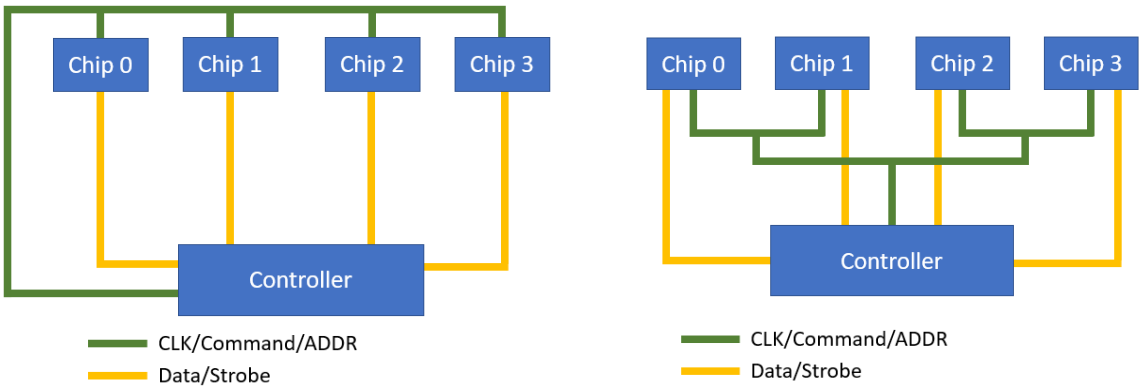


Figure 4.1: DDR3 fly-by *versus* T-topology (CC BY-SA 4.0, 2021)

Proper memory initialization ensures that the memory controller and the memory modules are correctly configured to work together, allowing for efficient data transfer and reliable operation. The initialization process involves setting various parameters, such as memory timings, voltages, and frequencies, which are critical for ensuring that the memory operates within its optimal range [69]. Failure to initialize DDR3 memory correctly can lead to several serious consequences, including system instability, data corruption, and reduced performance [103]. In the worst-case scenario, improper memory initialization can prevent the system from booting entirely, as the memory subsystem fails to function correctly. In the context of the ASUS KGPE-D16, a server motherboard designed for high-performance applications, proper DDR3 memory initialization is particularly important. The KGPE-D16 supports up to 256GB of DDR3 memory across 16 DIMM slots, and any issues during memory initialization, if non-fatal, could severely impact the system's ability to handle large datasets or maintain stable operation under heavy workloads [14]. Given the critical role that memory plays in the overall performance of the KGPE-D16, ensuring that DDR3 memory is correctly initialized is essential for achieving the desired balance of performance, reliability, and stability in demanding server environments.

## 4.2 General steps for DDR3 configuration

DDR3 memory initialization is a detailed and essential process that ensures both the stability and performance of the system. The process involves several critical steps: detection and identification of memory modules, initial configuration of the memory controller, adjustment of timing and voltage settings, and the execution of training and calibration procedures.

The initialization begins with the detection and identification of the installed memory modules. During the BIST, the firmware reads the Serial Presence Detect (SPD) data stored on each memory module. SPD data contains crucial information about the memory module's specifications, including size, speed, CAS latency (CL), RAS to CAS delay (tRCD), row precharge time (tRP), and row cycle time (tRC). This data allows to configure the memory controller for optimal compatibility and performance.

Indeed, once the memory modules have been identified, the firmware proceeds to the initial configuration of the memory controller. This controller is governed by a state machine that manages the sequence of operations required to initialize, maintain, and control memory access. This state machine consists of multiple states that represent various phases of memory operation, such as reset, initialization, calibration, and data transfer. The transitions between these states are either automatic or command-driven, depending on the specific requirements of each phase [69][55]. This state machine is presented in the fig. 4.2. Automatic transitions, depicted by thick arrows in the automaton, occur without external intervention. These typically include transitions that ensure the memory enters a stable state, such as the transition from power-on to initialization, or from calibration to idle states. These transitions are crucial for maintaining the integrity and stability of the memory system, as they ensure that the controller progresses through necessary stages like ZQ calibration and write leveling, which are essential for proper signal timing and impedance matching [69][55][20].

On the other hand, command-driven transitions, represented by normal arrows in the automaton, require specific commands issued by the memory controller or the CPU to advance to the next state. For instance, the transition from the idle state to the data transfer state requires explicit read or write commands. Similarly, transitioning

from the initialization state to the calibration state involves issuing mode register set (MRS) commands that configure the memorys operating parameters. These command-driven transitions are integral to the dynamic operation of the memory system, allowing the controller to respond to the system's operational needs and ensuring that memory accesses are performed efficiently and accurately [69][55].

The memory controller configuration involves setting up fundamental parameters such as the memory clock (MEMCLK) frequency and the memory channel configuration. The MEMCLK frequency is derived from the SPD data, while the memory channels are configured to operate in single, dual, or quad-channel modes, depending on the system architecture and the installed modules [20]. Proper configuration of the memory controller is vital to ensure synchronization with the memory modules, establishing a stable foundation for subsequent operations.

The first critical step, during the INIT phase involves the adjustment of timing and voltage settings. These settings are essential for ensuring that DDR3 memory operates efficiently and reliably. Key timing parameters include CAS Latency (CL), RAS to CAS Delay (tRCD), Row Precharge Time (tRP), and Row Cycle Time (tRC). These parameters are finely tuned to balance speed and stability [69]. The BIOS uses the SPD data to set these parameters and may also adjust them dynamically to achieve the best possible performance. Voltage settings, such as DRAM voltage (typically 1.5V for DDR3) and termination voltage (VTT), are also configured to maintain stable operation, especially under varying conditions such as temperature fluctuations [55].

Training and calibration are among the most complex and crucial stages of DDR3 memory initialization. The fly-by topology used for address, command, and clock signals in DDR3 modules enhances signal integrity by reducing the number of stubs and their lengths, but it also introduces skew between the clock (CK) and data strobe (DQS) signals [55]. This skew must be compensated to ensure that data is written and read correctly. The BIOS performs write leveling, which adjusts the timing of DQS relative to CK for each memory module. This process ensures that the memory controller can write data accurately across all modules, even when they exhibit slight variations in signal timing due to the physical layout [69].

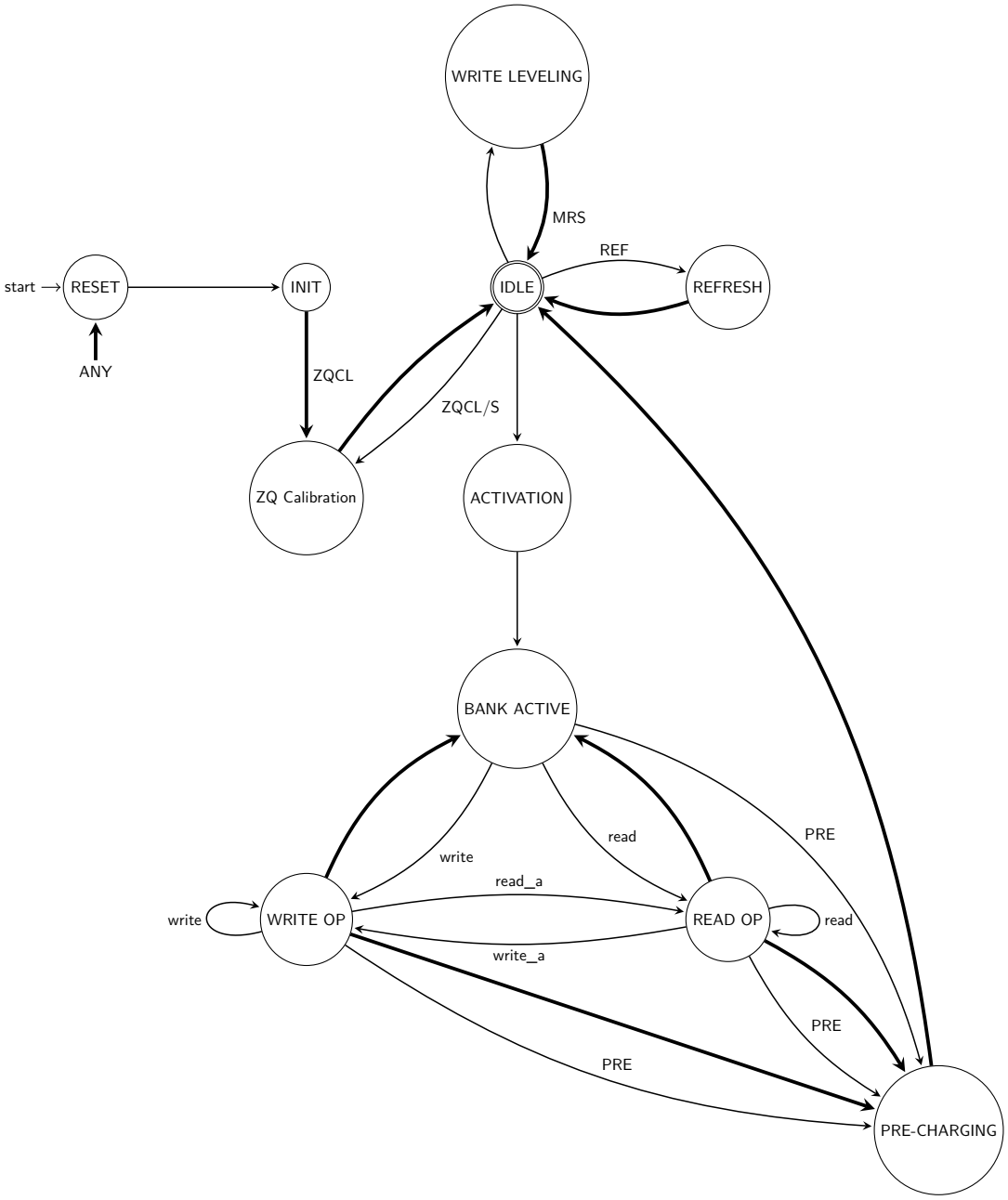


Figure 4.2: DDR3 controller state machine

ZQ calibration is another vital procedure that adjusts the output driver impedance and on-die termination (ODT) to match the systems characteristic impedance [55]. This calibration is critical for maintaining signal integrity under different operating conditions, such as voltage and temperature changes. During initialization, the memory controller issues a ZQCL command to the DRAM modules, triggering the calibration sequence that optimizes impedance settings. This ensures that the memory system can operate with tight timing tolerances, which is crucial for systems requiring high reliability. Read training is also essential to ensure that data read from the memory modules is interpreted correctly by the memory controller. This process involves adjusting the timing of the read data strobe (DQS) to align perfectly with the data being received. Proper read training is necessary for reliable data retrieval, which directly impacts system performance and stability.

ZQCS (ZQ Calibration Short) however is a procedure used to periodically adjust the DRAM's ODT and output driver impedance during normal operation. Unlike the full ZQCL (ZQ Calibration Long), which is performed during initial memory initialization, ZQCS is a quicker, less comprehensive calibration that fine-tunes the impedance settings in response to changes in temperature, voltage, or other environmental factors. This helps maintain optimal signal integrity and performance throughout the memory's operation without the need for a full recalibration.

In summary, the DDR3 memory initialization process in systems like the ASUS KGPE-D16 involves a series of

detailed and interdependent steps that are critical for ensuring system stability and performance. These include the detection and identification of memory modules, the initial configuration of the memory controller, precise adjustments of timing and voltage settings, and rigorous training and calibration procedures.

## 4.3 Memory initialization techniques

### 4.3.1 Memory training algorithms

Memory training algorithms are designed to fine-tune the operational parameters of memory modules, such as timing, voltage, and impedance. These algorithms play a crucial role in achieving the optimal performance of DDR3 memory systems, particularly in complex multi-core environments where synchronization and timing are challenging. The primary algorithms used in memory training include ZQ calibration and write leveling. Optimizing timing and voltage settings is a critical aspect of memory training. The memory controller adjusts parameters such as CAS latency, RAS to CAS delay, and other timing characteristics to ensure that data is read and written with minimal delay and maximum accuracy. Voltage adjustments are also crucial, as they help stabilize the operation of memory modules by ensuring that the power supplied is within the optimal range, compensating for any variations due to temperature or other factors [55][20][46].

ZQ calibration is a critical step in DDR3 memory initialization that ensures the proper impedance matching of the output driver and on-die termination (ODT) resistance. Impedance matching is crucial for maintaining signal integrity by minimizing reflections and ensuring reliable data transmission between the memory controller and the DRAM modules. It is initiated by sending ZQCL (ZQ Calibration Long) commands to the DDR3 DIMMs. Each ZQCL command triggers a long calibration cycle within the DRAM module. The purpose of this calibration is to adjust the output driver impedance and the ODT resistance to match the specified target impedance. This adjustment compensates for process variations, voltage fluctuations, and temperature changes that can affect the impedance characteristics of the DRAM module [46].

A bit in the DRAM Controller Timing register is set to 1 to send the ZQCL command, and an address bit is also set to 1 to indicate that the ZQCL command should be directed to the memory module. Upon receiving the ZQCL command, the DRAM module begins the calibration process. This involves a series of internal adjustments where the DRAM module measures its current impedance and compares it against the target impedance. The module then modifies its internal settings to reduce the difference between the current and target impedance values [46][69]. This process is iterative, meaning that it may require multiple adjustments to converge on the optimal impedance settings. The calibration is designed to ensure that the DRAM module's impedance remains within a tight tolerance, which is critical for high-speed data communication. The ZQ calibration process is time-sensitive. After issuing the ZQCL command, the system must wait for 512 memory clock cycles (MEMCLKs) to allow the calibration to complete. This delay is necessary because the calibration involves both measurement and adjustment phases, which require precise timing to ensure accuracy [46]. If the system does not wait the full 512 MEMCLKs, the calibration may be incomplete, leading to suboptimal impedance matching and potential signal integrity issues, such as reflections or noise on the data lines.

During the ZQ calibration, the DRAM module adjusts its output driver impedance, which controls the strength of the signals it sends out. The stronger the signal, the less susceptible it is to noise, but if the impedance is too high or too low, it can cause signal distortion or reflections. The ODT resistance is also calibrated to properly terminate signals that reach the end of a data line. Proper termination is essential to prevent signal reflections that could interfere with the integrity of the data being transmitted. The ZQCL command adjusts these settings by fine-tuning the resistance values based on the modules feedback, ensuring that the signal paths are optimized for both transmission and termination. Once the ZQ calibration is complete, the DCT register bit is reset to 0, indicating that the calibration command has been processed. The memory controller then verifies that the DRAM module has correctly adjusted its impedance settings. This verification process may involve additional test signals sent across the memory bus to confirm that signal integrity meets the required standards. If the calibration is successful, the memory subsystem is now properly calibrated and ready for normal operation. In systems with LRDIMMs or RDIMMs, additional steps may be necessary to ensure that all ranks and channels are calibrated correctly, particularly in multi-rank configurations where impedance matching can be more complex. However, in systems with complex memory configurations, such as those using multiple DIMMs per channel or operating at higher memory frequencies, the ZQ calibration process becomes even more critical. The calibration

may need to be repeated at different operating points to ensure that the memory subsystem remains stable across all conditions. This could involve performing multiple ZQCL calibrations at different memory frequencies, or under different thermal conditions, to account for the dynamic nature of memory operation in modern systems.

In seed-based algorithms, an initial "seed" value is used as a reference point for the calibration process. The memory controller iteratively adjusts the impedance based on feedback from the memory module, refining the calibration with each iteration. This method provides a more precise calibration, particularly in systems where fine-tuned impedance matching is critical for high-frequency operations [61]. Also, while seed-based methods can accelerate the convergence of calibration, they require careful selection of initial seed values to avoid suboptimal or even faulty impedance settings [46].

Write leveling is another critical aspect of memory training, particularly in DDR3 systems that use a fly-by topology. It involves using the physical layer (PHY) to detect the edge of the Data Strobe (DQS) signal in synchronization with the clock (CK) signal on the DIMM (Dual In-line Memory Module) during write access. The DQS signal is a timing signal generated by the memory controller that accompanies data (DQ) during read and write operations. For write operations, the DQS signal must be perfectly aligned with the CK signal to ensure that data is correctly written to memory cells. Indeed, in systems using a fly-by topology, the DQS signal might arrive at different times for different memory devices on the same module due to the signal traveling through different lengths of trace. Write leveling compensates for this skew by adjusting the timing of the DQS signal relative to the CK signal for each lane (a group of data lines) [20]. This training is performed on a per-channel and per-DIMM basis, ensuring that each memory module is correctly synchronized with the memory controller, minimizing timing mismatches that could lead to data corruption.

Write leveling implies to perform a DQS position training, a specific form of training focused on aligning the DQS signal with the data (DQ) signals during write operations. In this process, the memory controller adjusts the phase of the DQS signal to ensure that it is correctly aligned with the data signals across all data lanes, centering the DQS signal within the "data eye" for optimal timing. This ensures that all data bits are written correctly and consistently across the memory module, reducing the risk of timing errors and data corruption. Additionally, DQS receiver training is also needed to ensure that the memory controller can correctly capture the DQS signal during read operations [55]. The core operation is to make the MCT send out specific test patterns to the DRAM to determine the timing relationship between the DQS and data signals, then the MCT adjusts the delay or phase of the DQS signal relative to the clock signal (CK) and the data signals (DQ) while checking the integrity of the test data in the DRAM.

Using seed-based algorithms, the memory controller sets an initial delay value and then iteratively adjusts it based on the feedback received from the memory module. This process ensures that the DQS signal is correctly aligned with the CK signal at the memory module's pins, minimizing the risk of data corruption and ensuring reliable write operations [69][46]. Seed-based write leveling offers improved precision but must be finely tuned to account for the specific characteristics of the memory module and the overall system architecture [46].

In contrast to seed-based algorithms, seedless methods do not rely on an initial reference value. Instead, they dynamically adjust the impedance and timing parameters during the calibration process. Seedless ZQ calibration continuously monitors the impedance of the memory module and makes real-time adjustments to maintain optimal matching. This approach can be beneficial in environments where the operating conditions are highly variable, as it allows for more flexible and adaptive calibration [61]. Similarly, seedless write leveling dynamically adjusts the DQS timing based on real-time feedback from the memory module. This method is particularly useful in systems where the memory configuration is frequently changed or where the operating conditions vary significantly [55][46]. The traditional ZQ calibration methods, while effective, often struggle with matching impedance perfectly across all conditions. A master thesis by Gopikrishna [46] builds upon these traditional methods by proposing enhancements that involve more sophisticated calibration approaches, leading to better impedance matching and overall memory performance [46].

### 4.3.2 BIOS and Kernel Developer Guide (BKDG) recommendations

The BIOS and Kernel Developer Guide (BKDG from AMD [8]) is a technical manual aimed at BIOS developers and operating system kernel programmers. It provides in-depth documentation on the AMD processor architec-

ture, system initialization processes, and configuration guidelines. The document is essential for understanding the proper initialization sequences, including those for DDR3 memory, to ensure system stability and performance, particularly for AMD Family 15h processors.

The initialization of DDR3 memory begins with configuring the DDR supply voltage regulator, which ensures that the memory modules receive the correct power levels. Following this, the Northbridge (NB) P-state is forced to `NBP0`, a state that guarantees stable operation during the initial configuration phases. Once these preliminary steps are completed, the initialization of the DDR physical layer (PHY) begins, which is critical for setting up the communication interface between the memory controller and the DDR3 modules. PHY fence training deals with overall signal alignment at the physical interface, while ZQ calibration focuses on impedance matching, and write leveling addresses timing alignment during write operations. Each process involves different methods as PHY fence training uses iterative timing adjustments, ZQ calibration uses impedance adjustments via the ZQ pin, and write leveling adjusts DQS timing relative to CK during writes. These processes are critical for configuring DDR3 DIMMs and ensuring stable and reliable operation, especially when booting from an unpowered state such as ACPI S4 (hibernation), S5 (soft off), or G3 (mechanical off).

#### 4.3.2.1 DDR3 initialization procedure

DDR3 initialization is a multi-step process that prepares both the memory controllers and the DIMMs for operation. This initialization is essential to set up the memory configuration and to ensure that the memory subsystem operates correctly under various conditions.

- **Enable DRAM initialization**: The process begins by enabling DRAM initialization. This is done by setting the `EnDramInit` bit in the `D18F2x7C_dct` register to 1. The `D18F2x7C_dct` register is a specific configuration register within the memory controller that controls various aspects of the DRAM initialization process. Enabling this bit initiates the sequence of operations required to prepare the memory for use. After setting this bit, the system waits for 200 microseconds to allow the initialization command to propagate and stabilize.
- **Deassert memory reset**: Next, the memory reset signal, known as `MemRstX`, is deasserted by setting the `DeassertMemRstX` bit in the `D18F2x7C_dct` register to 1. Deasserting `MemRstX` effectively takes the memory components out of their reset state, allowing them to begin normal operation. The system then waits for an additional 500 microseconds to ensure that the memory reset is fully deasserted and the memory components are stable.
- **Assert clock enable (CKE)**: The next step involves asserting the clock enable signal, known as 'CKE', by setting the `AssertCke` bit in the `D18F2x7C_dct` register to 1. The `CKE` signal is critical because it enables the clocking of the DRAM modules, allowing them to synchronize with the memory controller. The system must wait for 360 nanoseconds after asserting `CKE` to ensure that the clocking is correctly established.
- **Registered DIMMs and LRDIMMs initialization**: For systems using registered DIMMs (RDIMMs) or Load Reduced DIMMs (LRDIMMs), additional initialization steps are necessary. RDIMMs and LRDIMMs have buffering mechanisms that reduce electrical loading and improve signal integrity, especially in systems with multiple memory modules. During initialization, the BIOS programs the `ParEn` bit in the `D18F2x90_dct` register based on whether the DIMM is buffered or unbuffered. For RDIMMs, specific Register Control (RC) commands, such as RC0 through RC7, are sent to initialize the memory module's control registers. Similarly, LRDIMMs require a series of Flexible Register Control (FRC) commands, such as F0RC and F1RC, to initialize their internal registers according to the manufacturers specifications.
- **Mode Register Set (MRS)**: The initialization process also involves sending Mode Register Set (MRS) commands. These commands are used to configure various operational parameters of the DDR3 memory modules, such as burst length, latency timings, and operating modes. Each MRS command targets a specific mode register within the memory module, and the exact sequence of commands is crucial for setting up the DIMMs according to the systems requirements and the DIMM manufacturers guidelines.

#### 4.3.2.2 ZQ calibration process

ZQ calibration is a key step in DDR3 initialization, responsible for calibrating the output driver impedance and on-die termination (ODT) resistance of the DDR3 modules. Proper impedance matching is essential for main-

taining signal integrity, reducing signal reflections, and ensuring reliable data communication between the memory controller and the memory modules. It is important to note that ZQ calibration is done directly by the memory controller, and that the firmware is simply triggering it.

- **Sending ZQCL commands**: The BIOS initiates ZQ calibration by sending two ZQCL (ZQ Calibration Long) commands to each DDR3 DIMM. ZQCL commands instruct the memory module to perform a long calibration cycle, during which the module adjusts its output driver impedance and ODT resistance to match the desired target impedance. This process compensates for variations due to manufacturing differences, voltage fluctuations, and temperature changes. To send a ZQCL command, the BIOS programs the `SendZQCmd` bit in the `D18F2x7C_dct` register to 1 and sets the `MrsAddress[10]` bit to 1, indicating that the ZQCL command should be sent to the memory module.
- **Calibration timing**: After sending the ZQCL command, the system must wait for 512 memory clock cycles (MEMCLKs) to allow the calibration process to complete. During this time, the memory module adjusts its internal impedance to ensure it matches the specified target impedance. This timing is critical, as inadequate wait times could result in incomplete or inaccurate calibration, leading to signal integrity issues and potential data errors.
- **Finalization of initialization**: Once the ZQ calibration is complete, the BIOS deactivates the DRAM initialization process by setting the `EnDramInit` bit in the `D18F2x7C_dct` register to 0. For LRDIMMs, additional configuration steps are required to finalize the initialization process. These steps include programming the DCT registers to monitor for errors and ensure that the LRDIMMs are operating correctly.

#### 4.3.2.3 Write leveling process

The BIOS and Kernel Developer Guide (BKDG) provides information on the write leveling process, which is essential for ensuring correct data alignment during write operations in DDR3 memory systems. Write leveling is particularly crucial in systems utilizing a fly-by topology, where timing skew between the clock and data signals can introduce significant challenges. This kind of algorithms were not necessary for DDR2, for example. If the target operating frequency is higher than the lowest supported MEMCLK frequency, the BIOS must perform multiple passes to achieve proper write leveling. The MEMCLK is the clock signal that synchronizes the communication between the memory controller and the memory modules.

During each pass, the memory subsystem is configured for a progressively higher operating frequency:

- **Pass 1:** The memory subsystem is configured for the lowest supported MEMCLK, ensuring that initial timing adjustments are made under the most stable conditions.
- **Pass 2:** The subsystem is then adjusted for the second-lowest MEMCLK, gradually increasing the operating frequency while fine-tuning the alignment of the DQS and CK signals.
- **Pass N:** This process continues until the highest MEMCLK supported by the system is reached, ensuring that the memory operates reliably at its maximum speed.

This step-wise configuration ensures that the memory system is stable across all supported operating frequencies, minimizing the risk of timing errors during write operations, especially as frequencies increase and timing margins become tighter. The configuration process varies depending on whether the DIMM is a Registered DIMM (RDIMM) or an Unregistered DIMM (UDIMM). RDIMMs include an additional buffer to improve signal integrity, which is particularly important in systems with multiple DIMMs. The steps common to both types include a preparation with the DDR3 Mode Register Commands (see fig. 4.2). For RDIMMs, a 4-rank module is treated as two separate DIMMs, where each rank is essentially a separate memory module within the same DIMM. The first two ranks are the primary target for the initial configuration. The remaining two ranks are treated as non-target and are configured separately.

Mode registers in DDR3 memory are used to configure various operational parameters such as latency settings, burst length, and write leveling. One of the key mode registers is `MR1_dct`, which is specific to DDR3 and controls certain features of the memory module, including write leveling. `MR1_dct` is used to enable or disable specific functions such as write leveling and output driver settings. The `dct` suffix refers to the Data Control

Timing that is specific to this register's function in managing the timing and control of data operations within the memory module.

Then, these steps are followed, still common to both RDIMMs and UDIMMs:

- **Step 1A: Output Driver and ODT configuration for target DIMM:**
  - For the first rank (target):
    - Set `MR1_dct[1:0][Level]=1` to enable write leveling.
    - Set `MR1_dct[1:0][Qoff]=0` to ensure the output drivers are active.
  - For other ranks:
    - Set `MR1_dct[1:0][Level]=1` to prepare for write leveling.
    - Set `MR1_dct[1:0][Qoff]=1` to deactivate the output drivers for ranks that are not currently being leveled.
  - If there are two or more DIMMs per channel, or if there is one DIMM per three channels:
    - Program the target ranks `RttNom` (nominal termination resistance value) for `RttWr` termination, which helps in managing signal integrity during the write process by ensuring the correct impedance matching.
- **Step 1B: Configure non-target RttNom to normal operation:**
  - After the initial configuration, the `RttNom` values for the non-target ranks are set to their normal operating states.
  - A wait time of 40 MEMCLKs is observed to ensure the configuration settings are stable before proceeding.
- **Step 3: PHY configuration:**
  - The PHY is then configured to measure and adjust the timing delays accurately for each data lane. The PHY layer is responsible for converting the signals from the memory controller into a form that can be transmitted over the physical connections to the memory modules.
- **Step 4: Perform write leveling:**
  - The actual write leveling process is executed, where the DQS signal timing is adjusted to ensure it aligns perfectly with the CK signal at the memory modules pins, ensuring that data is written accurately.
- **Step 5: Disable PHY configuration post-measurement:**
  - After completing the write leveling process, the PHY configuration is disabled to stop further timing measurements and adjustments, locking in the calibrated settings.
- **Step 6: Program the DIMM to normal operation:**
  - Finally, the DIMM is reprogrammed to its normal operational state, resetting `Qoff` and `Level` to `0` to conclude the write leveling process and return to standard operation.

For each DIMM, the BIOS must calculate the coarse and fine delays for each lane in the DQS Write Timing register:

- **Coarse Delay Calculation:** This involves setting a basic delay based on a seed value specific to the platform. The seed value is determined during initial system configuration and serves as a starting point for further delay adjustments.
- **Critical Delay Determination:** The minimum of the coarse delays for each lane and DIMM is considered the critical delay. This delay is crucial for ensuring that all data lanes are correctly synchronized.
- **Platform-Specific Seed:** The seed ranges between -1.20ns and +1.20ns, providing a small adjustment range to fine-tune the timing based on the specific characteristics of the platform. This seed value can differ for the first pass compared to subsequent passes, allowing for incremental adjustments as the system stabilizes.

## 4.4 Current implementation and potential improvements

### 4.4.1 Current implementation in coreboot on the KGPE-D16

In this part as for the rest of this document, we're basing our study on the 4.11 version of *coreboot* [25], which is the last version that supported the ASUS KGPE-D16 mainboard.

The process starts in `src/mainboard/asus/kgpe-d16/romstage.c`, in the `cache_as_ram_main` function by calling `fill_mem_ctrl` from `src/northbridge/amd/amdfam10/raminit_sysinfo_in_ram.c` (lst. 4.1). At this current step, only the BSC is running the firmware code. 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.

```c
void fill_mem_ctrl(u32 controllers,
                   struct mem_controller *ctrl_a,
                   const u8 *spd_addr)
{
    int i;
    int j;
    int index = 0;
    struct mem_controller *ctrl;
    for (i = 0; i < controllers; i++) {
        ctrl = &ctrl_a[i];
        ctrl->node_id = i;
        ctrl->f0 = NODE_PCI(i, 0);
        ctrl->f1 = NODE_PCI(i, 1);
        ctrl->f2 = NODE_PCI(i, 2);
        ctrl->f3 = NODE_PCI(i, 3);
        ctrl->f4 = NODE_PCI(i, 4);
        ctrl->f5 = NODE_PCI(i, 5);

        if (spd_addr == (void *)0) continue;

        ctrl->spd_switch_addr = spd_addr[index++];

        for (j = 0; j < 8; j++) {
            ctrl->spd_addr[j] = spd_addr[index++];

        }
    }
}

```

Listing 4.1: `fill_mem_ctrl()`, extract from `src/northbridge/amd/amdfam10/raminit_sysinfo_in_ram.c`

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` (lst. L.39) from `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`, which is responsible for the initial memory initialization, predominantly written by Raptor Engineering.

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 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.
- If we were using saved values, the NVRAM would have 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 (lst. L.40).
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`.

The memory modules must be initialized. All modules present on valid nodes are configured with 1.5V voltage (lst. L.41). The ZQ calibration is triggered at this stage.

Now, present memory modules are detected using `mct_initDCT` (lst. L.43). The memory modules existence is checked and the machine halts immediately after displaying a message if there is no memory. *coreboot* waits for all modules to be available using `SyncDCTsReady_D`.

The firmware maps the physical memory address ranges into the address space with `HTMemMapInit_D` as contiguously as possible while also constructing the physical memory map. If there is an area occupied by something else, it is ignored, and a memory hole is created.

Mapping the address ranges into the cache is done with `CPUMemTyping_D` either as WriteBack (cacheable) or Uncacheable, depending on whether the area corresponds to physical memory or a memory hole.

The external northbridge is notified of this new memory configuration.

The *coreboot* code compensates for the delay between DQS and DQ signals, as well as between CMD and DQ. This is handled by the `DQSTiming_D` function (lst. L.44). The initialization can be done again if needed after that, otherwise the channels and nodes are interleaved and ECC is enabled (if supported by every module).

After that being done, the DRAM can be mapped into the address space with cacheability, and the init process finishes with validation of every populated DCT node (lst. L.45).

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.

#### 4.4.1.1 Details on the DQS training function

The DQSTiming_D function is a critical part of the firmware responsible for initializing and training the system's memory. The function primarily handles the DQS timing, which is essential for ensuring data integrity and synchronization between the memory controller and the DRAM. Proper DQS training is crucial to align the data signals correctly with the clock signals.
The function begins by declaring local variables, which are used throughout the function for various operations. It also includes an early exit condition to bypass DQS training if a specific status flag (GSB_EnDIMMSpareNW) is set, indicating that a DIMM spare feature is enabled (lst. 4.2). These spare DIMMs are not used for normal memory operations but are kept in reserve for redundancy.

```
if (pMCTstat->GStatus & (1 << GSB_EnDIMMSpareNW)) {
    return;
}
```

Listing 4.2: Early exit check, extract from the DQSTiming_D function in src/northbridge/amd/amdmct/mct_ddr3/mct_d.c

Next, the function initializes the TCWL (CAS Write Latency) offset to zero for each node and DCT. This ensures that the memory write latency is properly aligned before the DQS training begins (lst. 4.3).

```
for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
    uint8_t dct;
    struct DCTStatStruc *pDCTstat;
    pDCTstat = pDCTstatA + Node;
    for (dct = 0; dct < 2; dct++)
        pDCTstat->tcwl_delay[dct] = 0;
}
```

Listing 4.3: Setting initial TCWL offset to zero for all nodes and DCTs, extract from the DQSTiming_D function in src/northbridge/amd/amdmct/mct_ddr3/mct_d.c

A retry mechanism is introduced to handle potential errors during DQS training and the pre-training function are called (lst. 4.4).

```
retry_dqs_training_and_levelization:
    nv_DQSTrainCTL = !allow_config_restore;

    mct_BeforeDQSTrain_D(pMCTstat, pDCTstatA);
    phyAssistedMemFnceTraining(pMCTstat, pDCTstatA, -1);
```

Listing 4.4: Retry mechanism initialization and pre-training operations, extract from the DQSTiming_D function in src/northbridge/amd/amdmct/mct_ddr3/mct_d.c

For AMD's Fam15h processors, additional PHY compensation is needed for each node and valid DCT (lst. 4.5). This is necessary to fine-tune the electrical characteristics of the memory interface. For more information about the PHY training, see the earlier sections about RAM training algorithm.

```
for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
    pDCTstat = pDCTstatA + Node;
    if (pDCTstat->NodePresent) {
        if (pDCTstat->DIMMValidDCT[0])
            InitPhyCompensation(pMCTstat, pDCTstat, 0);
        if (pDCTstat->DIMMValidDCT[1])
            InitPhyCompensation(pMCTstat, pDCTstat, 1);
    }
}
```

Listing 4.5: PHY compensation initialization, extract from the `DQSTiming_D` function in `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`

Before proceeding with the main DQS training, the function invokes a hook function that allows for additional configurations or custom operations: `mctHookBeforeAnyTraining`.

The `nv_DQSTrainCTL` variable is set based on the `allow_config_restore` parameter, determining whether to restore a previous configuration or proceed with fresh training. This is however not working on the current implementation of ASUS KGPE-D16 firmware (lst. L.42). If `nv_DQSTrainCTL` indicates that fresh training should proceed, the function performs the main DQS training in multiple passes, including receiver enable training with `TrainReceiverEn_D`, write leveling with `mct_WriteLevelization_HW`, DQS position training with `mct_TrainDQSPos_D` and the maximum read latency calculation with `TrainMaxRdLatency_En_D` (lst. 4.6). Write leveling is done in two passes, with a DQS receiver training between and another pass of receiver training after. After that, a DQS position training is done and the process finished with the maximum read latency, i.e the delay between the request for data and the delivery of that data by the DRAM.

```
if (nv_DQSTrainCTL) {
    mct_WriteLevelization_HW(pMCTstat, pDCTstatA, FirstPass);
    TrainReceiverEn_D(pMCTstat, pDCTstatA, FirstPass);
    mct_WriteLevelization_HW(pMCTstat, pDCTstatA, SecondPass);

    /* TODO: Determine why running TrainReceiverEn_D in SecondPass mode yields
    * less stable training values than when run in FirstPass mode as in the HACK
    * below.*/
    TrainReceiverEn_D(pMCTstat, pDCTstatA, FirstPass);
    mct_TrainDQSPos_D(pMCTstat, pDCTstatA);
     ...
    TrainMaxRdLatency_En_D(pMCTstat, pDCTstatA);
} else {
    mct_WriteLevelization_HW(pMCTstat, pDCTstatA, FirstPass);
    mct_WriteLevelization_HW(pMCTstat, pDCTstatA, SecondPass);
#if CONFIG(HAVE_ACPI_RESUME)
    printk(BIOS_DEBUG, "mctAutoInitMCT_D: Restoring DIMM training configuration"
                        "from NVRAM\n");
    if (restore_mct_information_from_nvram(1) != 0)
        printk(BIOS_CRIT, "%s: ERROR: Unable to restore DCT configuration from"
                            "NVRAM\n", __func__);
#endif
    exit_training_mode_fam15(pMCTstat, pDCTstatA);
    pMCTstat->GStatus |= 1 << GSB_ConfigRestored;"
}
```

Listing 4.6: Main DQS training process in multiple passes, extract from the `DQSTiming_D` function in `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`

The function checks for any errors during the DQS training. If errors are detected, it may request a retrain, reset certain parameters, and restart the training process and even restart the whole system if needed (lst. L.46). If the training process it to be restarted, the firmware sets the DIMMs frequencies to minimum and applies timing changes to DIMMs before jumping to the retry label (lst. 4.4).

Once the training is successfully completed without errors, the function finalizes the process by setting the maximum read latency and exiting the training mode. For systems with `allow_config_restore` enabled, it restores the previous configuration from NVRAM instead of performing a fresh training (lst. 4.6).

Finally, the function performs a cleanup operation specific to Fam15h processors, where it switches the DCT control register as required by a known erratum from AMD for the BKDG (Erratum 505) [10]. This is followed by a post-training hook that allows for any additional necessary actions (lst. 4.7).

```
for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
    pDCTstat = pDCTstatA + Node;
    if (pDCTstat->NodePresent) {
        fam15h_switch_dct(pDCTstat->dev_map, 0);
    }
}

/* FIXME - currently uses calculated value
 * TrainMaxReadLatency_D(pMCTstat, pDCTstatA); */
mctHookAfterAnyTraining();
```

Listing 4.7: Post-training cleanup and final hook execution, extract from the `DQSTiming_D` function in `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`

#### 4.4.1.2 Details on the write leveling implementation

The `WriteLevelization_HW` function is responsible for performing hardware-level write leveling on DRAM modules during the memory initialization process. Write leveling ensures that the DQS signals are correctly aligned with the clock signals, preventing timing mismatches during write operations.

The function begins by initializing pointers to key data structures, linking the memory controller (MCT) and DRAM controller timing (DCT) data for subsequent operations.

Auto-refresh and short ZQ calibration are temporarily disabled to prevent interference during the critical timing adjustments of write leveling. The memory controller is prepared for write leveling by configuring necessary parameters with `PrepareC_MCT`, then the main operation can begin.

In the first pass (lst. 4.8), the function repeatedly attempts to align the DQS signals with `PhyWLPass1`, retrying if invalid values are detected. This phase ensures basic alignment for further fine-tuning. The function retries up to 8 times if it detects invalid timing values.

During the second pass (lst. L.47), the function first checks if the target memory frequency (`TargetFreq`) is higher than the minimum memory clock frequency stored in the non-volatile bits (`NV_MIN_MEMCLK`). If so, the memory frequency is incrementally adjusted toward the final target f requency. This step-by-step approach is crucial, especially for AMD Fam15h processors, where the frequency must be gradually stepped up to avoid instability.

For each frequency step, the write leveling process is recalibrated by invoking the `PhyWLPass2` function. This function adjusts the DQS timing for each data channel (DCT) and validates the results. The function retries up to 8 times if it detects invalid timing values. The global status (`global_phy_training_status`) aggregates the results of each step, tracking any persistent issues.

The PhyWLPass1 and PhyWLPass2 function relyon AgesaHwWlPhase1, AgesaHwWlPhase2 and AgesaHwWlPhase3 for this.

Once the target frequency is reached and all write leveling adjustments are made, the final timing values are stored. The gross and fine delays from the previous passes are copied into the final pass structures. This ensures that the DQS timings are consistent and stable across all data channels.

If any issues persist after retries, the function logs a warning. This indicates that the system may continue to operate, but with a potential risk of instability due to imperfect write leveling calibration.

After leveling, the function re-enables auto-refresh and short ZQ calibration, ensuring the memory subsystem is correctly configured for normal operation.

```
if (Pass == FirstPass) {
    timeout = 0;
    do {
        status = 0;
        timeout++;
        status |= PhyWLPass1(pMCTstat, pDCTstat, 0);
        status |= PhyWLPass1(pMCTstat, pDCTstat, 1);
        if (status)
            printk(BIOS_INFO, "%s: Retrying write levelling due to invalid "
                              "value(s) detected in first phase\n", __func__);
    } while (status && (timeout < 8));
    if (status)
        printk(BIOS_INFO, "%s: Uncorrectable invalid value(s) detected in first "
                          "phase of write levelling\n", __func__);
}
```

Listing 4.8: Write leveling (first pass), extract from the WriteLevelization_HW function in src/northbridge/amd/amdmct/mct_ddr3/mcthwl.c

The detailled write leveling process is divided into three distinct phases, each managed by a specific function: AgesaHwWlPhase1, AgesaHwWlPhase2, and AgesaHwWlPhase3 from mcthwl.c. These phases work together to fine-tune the timing delays (gross and fine) for each byte lane, ensuring reliable data transmission.

The write leveling process begins by selecting the target DIMM. This is accomplished by programming the TrDimmSel register to ensure that the subsequent operations apply to the correct DIMM (lst. 4.9)

In the case of x4 DIMMs, which are common in high-density memory configurations, write leveling must be performed separately for each nibble (4-bit group). The function checks if x4 DIMMs are present and, if so, prepares to train both nibbles (lst. 4.10).

The DIMMs are prepared for write leveling by issuing Mode Register (MR) commands. These commands configure the DIMMs to enter a state where write leveling can be performed (lst. 4.11).

```
set_DCT_ADDR_Bits(pDCTData, dct, pDCTData->NodeId, FUN_DCT,
        DRAM_ADD_DCT_PHY_CONTROL_REG, TrDimmSelStart,
        TrDimmSelEnd, (u32)dimm);
```

Listing 4.9: Target DIMM selection for write leveling, extract from AgesaHwWlPhase1 function in src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c

```
train_both_nibbles = 0;
if (pDCTstat->Dimmx4Present)
    if (is_fam15h())
        train_both_nibbles = 1;
```

Listing 4.10: Handling of x4 DIMMs and nibble training, extract from `AgesaHwWlPhase1` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

```
prepareDimms(pMCTstat, pDCTstat, dct, dimm, TRUE);
```

Listing 4.11: Preparing DIMMs for write leveling, extract from `AgesaHwWlPhase1` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

The `procConfig` function is called to configure the processor's DDR PHY (Physical Layer) for write leveling. This configuration includes setting initial seed values for gross and fine delays, which are essential for the subsequent timing adjustments.

`procConfig` generates initial seed values (lst. L.48) for gross and fine delays. These seeds are calculated based on several factors:

- **Processor Type:** For Fam15h processors, specific tables from the Fam15h BKDG [8] are referenced to select appropriate seed values for different package types (e.g., Socket G34, Socket C32).
- **DIMM Type:** The seed values are adjusted based on whether the RDIMMs are registered or load-reduced, with different base values used for these configurations.
- **Memory Clock Frequency:** The seeds are further adjusted based on the current memory clock frequency (`MemClkFreq`), ensuring that the timing is correct for the operating speed of the memory.

The calculated seed values are then scaled to the minimum supported memory frequency and stored in the `WLSeedGrossDelay` and `WLSeedFineDelay` arrays for each byte lane.

Write leveling is initiated by enabling the `WrtLvTrEn` bit. This allows the DDR PHY to begin adjusting the DQS signals relative to the clock signals (lst. 4.13).

If the DIMM is not x4, the function skips the nibble training loop, as it is unnecessary (lst. 4.14).

After a delay to allow the leveling process to stabilize, the function reads the gross and fine delay values from the relevant registers and stores them (lst. 4.15). These values represent the initial timing adjustments necessary for correct DQS alignment.

#### 4.4.1.3 Details on the DQS position training function

The DQS position training is a crucial step in the memory initialization process, ensuring that both read and write operations are correctly aligned with the clock signal.

```
Seed_Total = (int32_t) (((((int64_t) Seed_Total) *
    fam15h_freq_tab[MemClkFreq] * 100) / (mctGet_NVbits(NV_MIN_MEMCLK) * 100)));

Seed_Gross = (Seed_Total >> 5) & 0x1f;
Seed_Fine = Seed_Total & 0x1f;
```

Listing 4.12: Seed generation, extract from `procConfig` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

```
set_DCT_ADDR_Bits(pDCTData, dct, pDCTData->NodeId, FUN_DCT,
        DRAM_ADD_DCT_PHY_CONTROL_REG, WrtLvTrEn, WrtLvTrEn, 1);
```

Listing 4.13: Initiating write leveling training, extract from `AgesaHwWlPhase1` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

```
if ((pDCTstat->Dimmx4Present & (1 << (dimm + dct))) == 0)
    break;
```

Listing 4.14: Exit for non-x4 DIMMs, extract from `AgesaHwWlPhase2` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

The function `TrainDQSRdWrPos_D_Fam15` orchestrates this process by iterating over memory lanes and adjusting timing parameters to find optimal settings (lst. 4.16). It is called by `mct_TrainDQSPos_D`.

The function begins by initializing several variables and settings necessary for the training process. These include:

- `Errors`: A variable to track any errors encountered during the training.
- `dual_rank`: A flag to indicate whether the current DIMM has two ranks.
- `passing_dqs_delay_found`: An array to track whether a passing DQS delay has been found for each lane.
- `dqs_results_array`: A multi-dimensional array to store the results of the DQS delay tests across different write and read steps.

The function then loops over each receiver (loosely associated with chip selects) to perform the training for each rank within each DIMM.

For each lane in the memory channel, the function iterates over possible write and read delay values to find the optimal configuration (lst. 4.17). This is done by:

1. Iterating over the write data delay values from the initial value to the initial value plus 1 UI (Unit Interval).
2. For each write data delay, iterating over possible read DQS delay values from 0 to 1 UI.
3. For each combination of write and read delays, testing the configuration by writing a training pattern to the memory and reading it back to check if it passes or fails.

During each iteration, the results are recorded in the `dqs_results_array`, which tracks whether the combination of write and read delays was successful (pass) or not (fail). The results are stored for both the primary rank and, if applicable, the secondary rank when dual rank DIMMs are used.

After iterating over all possible delay values, the function processes the results to determine the best DQS delay settings (lst. 4.18).

This is done by:

```
for (ByteLane = 0; ByteLane < lane_count; ByteLane++) {
    getWLByteDelay(pDCTstat, dct, ByteLane, dimm, pass, nibble, lane_count);
}
```

Listing 4.15: Reading and storing delay values after write leveling, extract from `AgesaHwWlPhase2` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

```
for (Receiver = receiver_start; Receiver < receiver_end; Receiver++) {
    dimm = (Receiver >> 1);
    ...
    if (!mct_RcvrRankEnabled_D(pMCTstat, pDCTstat, dct, Receiver)) {
        continue;
    }
```

Listing 4.16: Looping over each receiver, extract from `TrainDQSRdWrPos_D_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mctdqs_d.c`

```
for (current_write_data_delay[lane] = initial_write_dqs_delay[lane];
     current_write_data_delay[lane] < (initial_write_dqs_delay[lane] + 0x20);
     current_write_data_delay[lane]++) {
    ...
    for (current_read_dqs_delay[lane] = 0;
         current_read_dqs_delay[lane] < 0x20;
         current_read_dqs_delay[lane]++) {
        ...
        write_dqs_read_data_timing_registers(
            current_read_dqs_delay, dev, dct, dimm, index_reg);
        read_dram_dqs_training_pattern_fam15(
            pMCTstat, pDCTstat, dct, Receiver, lane, ((check_antiphase == 0)?1:0));
        ...
    }
}
```

Listing 4.17: Iteration over write and read delay values for each lane, extract from `TrainDQSRdWrPos_D_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mctdqs_d.c`

```
if (best_count > 2) {
    uint16_t region_center = (best_pos + (best_count / 2));
    if (region_center < 16) {
        printk(BIOS_WARNING,
            "TrainDQSRdWrPos: negative DQS recovery delay detected!");
        region_center = 0;
    } else {
        region_center -= 16;
    }
    ...
    current_read_dqs_delay[lane] = region_center;
    passing_dqs_delay_found[lane] = 1;
    write_dqs_read_data_timing_registers(
        current_read_dqs_delay, dev, dct, dimm, index_reg);
}
```

Listing 4.18: Processing the results to determine the best DQS delay settings, extract from `TrainDQSRdWrPos_D_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mctdqs_d.c`

```
for (lane = lane_start; lane < lane_end; lane++) {
    if (!passing_dqs_delay_found[lane]) {
        Errors |= 1 << SB_NODQSPOS;
    }
}
pDCTstat->TrainErrors |= Errors;
pDCTstat->ErrStatus |= Errors;
return !Errors;
```

Listing 4.19: Final error handling and return value, extract from `TrainDQSRdWrPos_D_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mctdqs_d.c`

- Finding the longest consecutive string of passing values for both read and write operations.
- Calculating the center of the passing region and using this as the optimal delay setting.
- If the center of the region is below a threshold, issuing a warning that a negative DQS recovery delay was detected, which could lead to instability.

Finally, the function checks if any lane did not find a valid passing region (lst. 4.19). If any lanes failed to find a passing DQS delay, the `Errors` flag is set, and this error is propagated through the `pDCTstat->TrainErrors` and `pDCTstat->ErrStatus` variables.

The function returns 1 if no errors were encountered, and 0 otherwise, which is unusual.

#### 4.4.1.4 Details on the DQS receiver training function

In AMD Fam15h G34 processors, the DQS receiver enable training is a critical step in ensuring that the memory subsystem operates correctly and reliably. This training aligns the DQS signal with the clock signal, ensuring proper data capture during memory reads.

The DQS receiver enable training algorithm is executed twice: first at the lowest supported MEMCLK frequency and then at the highest supported MEMCLK frequency. The purpose of this training is to fine-tune the timing parameters so that the memory controller can reliably read data from the memory modules. The algorithm is implemented in the function `dqsTrainRcvrEn_SW_Fam15` from `src/northbridge/amd/amdmct/mct_ddr3/mctsrc.c`, which orchestrates the entire process,

```
initial_seed = (uint16_t) (((((uint64_t) initial_seed) *
    fam15h_freq_tab[mem_clk] * 100) / (min_mem_clk * 100)));
```

Listing 4.20: Adjusting the seed values based on the operating frequency of the memory, extract from `dqsTrainRcvrEn_SW_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mctsrc.c`

```
for (lane = 0; lane < lane_count; lane++) {
    seed_gross[lane] = (seed[lane] >> 5) & 0x1f;
    seed_fine[lane] = seed[lane] & 0x1f;

    if (seed_gross[lane] & 0x1)
        seed_pre_gross[lane] = 1;
    else
        seed_pre_gross[lane] = 2;

    // Set the gross delay
    current_total_delay[lane] = ((seed_gross[lane] & 0x1f) << 5);
}
```

Listing 4.21: Setting initial delay values based on the generated seed values, extract from `dqsTrainRcvrEn_SW_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mctsrc.c`

called by the `mct_TrainRcvrEn_D` function, which has been called itself by `TrainReceiverEn_D` from `src/northbridge/amd/amdmct/mct_ddr3/mctdqs_d.c`.

Here, seeds are initial delay values used to set up the memory controller's timing parameters. These seeds are generated based on the specific characteristics of the memory configuration, such as the package type (e.g., G34, C32), the type of DIMMs installed (Registered, Load Reduced, etc.), and the maximum number of DIMMs that can be installed in a channel.

The seed generation is handled by the function `fam15_receiver_enable_training_seed`. This function generates a base seed value for each memory channel, based on predefined tables in the BKDG [8]. The base seed values are specific to the memory configuration and are adjusted based on the type of DIMM and the number of DIMMs in each channel.

The generated seed values are then adjusted (lst. 4.20) based on the operating frequency of the memory (MEMCLK). The adjustment scales the seed values to account for the difference between the current memory frequency and the minimum supported frequency. This ensures that the training can be accurately performed across different operating conditions.

Once the seeds are generated and adjusted, they are used to set the initial delay values for the DQS receiver enable training (lst. 4.21). The delay values are split into two components: gross delay and fine delay. The gross delay determines the overall timing offset, while the fine delay adjusts the timing with finer granularity.

These delay values are then written to the appropriate registers to configure the memory controller for the DQS receiver enable training. The training is performed in multiple steps, iteratively refining the delay values until the DQS signal is correctly aligned with the clock signal.

During the initialization phase, the memory controller is prepared for training. This includes enabling the training mode, configuring the memory channels, and disabling certain features such as ECC (Error-Correcting Code) to prevent interference during training (lst. 4.22).

The training phase is where the actual alignment of the DQS signal occurs. The memory controller iterates over each DIMM and each lane (lst. 4.23), applying the seed values and adjusting the delay registers accordingly.

```
fam15EnableTrainingMode(pMCTstat, pDCTstat, ch, 1);
_DisableDramECC = mct_DisableDimmEccEn_D(pMCTstat, pDCTstat);
```

Listing 4.22: Enabling training mode and disabling ECC, extract from `dqsTrainRcvrEn_SW_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mctsrc.c`

```
for (rank = 0; rank < (_2Ranks + 1); rank++) {
    for (nibble = 0; nibble < (train_both_nibbles + 1); nibble++) {
        ...
        write_dqs_receiver_enable_control_registers(
            current_total_delay, dev, Channel, dimm, index_reg);
        ...
    }
}
```

Listing 4.23: Iterating over ranks and nibbles to apply delay values, extract from `dqsTrainRcvrEn_SW_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mctsrc.c`

For each DIMM, the training is performed twice: once for the first nibble (lower 4 bits) and once for the second nibble (upper 4 bits) if the DIMM is x4.

During the training, the controller issues read requests to the memory to observe the timing of the DQS signal. The observed delays are then averaged and adjusted to ensure the DQS signal is correctly aligned across all lanes and ranks.

In the finalization phase, the memory controller exits the training mode (lst. 4.24), and the computed delay values are written back to the appropriate registers. This ensures that the DQS signal remains correctly aligned during normal operation.

### 4.4.2 Potential enhancements

#### 4.4.2.1 DQS receiver training

While the DQS receiver enable training implementation for AMD Fam15h G34 processors can perform its intended function in some cases, there are several areas where the code is either incomplete, suboptimal, or potentially problematic.

The presence of `TODO` comments in the code indicates areas where the implementation is either incomplete or lacks certain necessary functionality. These unaddressed tasks can lead to performance issues, potential bugs, or incomplete training, which could compromise the stability and reliability of the memory subsystem.

In the seed adjustment section for the second pass of training, the code includes a `TODO` comment regarding fetching the correct value from `RC2[0]` for the `addr_prelaunch` variable (lst. 4.25).

```
Calc_SetMaxRdLatency_D_Fam15(pMCTstat, pDCTstat, 0, 0);
Calc_SetMaxRdLatency_D_Fam15(pMCTstat, pDCTstat, 1, 0);
if (Pass == FirstPass) {
    mct_DisableDQSRcvEn_D(pDCTstat);
}
```

Listing 4.24: Exiting training mode and setting read latency, extract from `dqsTrainRcvrEn_SW_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mctsrc.c`

```c
uint8_t addr_prelaunch = 0; /* TODO: Fetch the correct value from RC2[0] */
```

Listing 4.25: An unimplemented feature in the seed adjustment logic, extract from `dqsTrainRcvrEn_SW_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mcrsrc.c`

```c
else if ((pDCTstat->Status & (1 << SB_LoadReduced))) {
    /* TODO
     * Load reduced DIMM support unimplemented
     */
    register_delay = 0x0;
}
```

Listing 4.26: LRDIMM support is unimplemented, extract from `dqsTrainRcvrEn_SW_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mcrsrc.c`

This unimplemented feature suggests that the training process may not be fully optimized, as the correct prelaunch address setting is not being applied. This could result in incorrect seed values being used during the training, leading to suboptimal alignment of the DQS signal. Also, the comment is unclear about what RC2[0] really means.

The code contains another `TODO` comment indicating that the support for Load Reduced DIMMs (LRDIMMs) is unimplemented (lst. 4.26).
This omission is significant because LRDIMMs are commonly used in server environments where high memory capacity is required. The lack of support for LRDIMMs could lead to incorrect training or even failures when such DIMMs are installed, severely impacting the reliability of the system.

`FIXME` comments in the code are often indicators of known issues or temporary workarounds that need to be addressed. In this implementation, there are several such comments that highlight critical areas where the current approach may be flawed or incomplete.

The first `FIXME` comment questions the usage of the `SSEDIS` setting during the training process (lst. 4.27).
The concern here is that disabling the `SSEDIS` (SSE Disable) bit could have unintended side effects, particularly in environments where SSE instructions are expected to be enabled. This could impact the performance of the system during training and potentially lead to instability.

The code also highlights a potential misprint in the BKDG regarding the `WrDqDqsEarly` value (lst. 4.28).
This indicates that the implementation may be based on incorrect or incomplete documentation, leading to potential errors in setting the delay values. If this is indeed a misprint in the BKDG, the correction should be verified with updated documentation, and the implementation should be adjusted accordingly.

```c
msr = HWCR_MSR;
_RDMSR(msr, &lo, &hi);
/* FIXME: Why use SSEDIS */
if (lo & (1 << 17)) {   /* save the old value */
    _Wrap32Dis = 1;
}
lo |= (1 << 17);        /* HWCR.wrap32dis */
lo &= ~(1 << 15);       /* SSEDIS */
_WRMSR(msr, lo, hi);    /* Setting wrap32dis allows 64-bit memory
                         * references in real mode                */
```

Listing 4.27: Questioning the use of `SSEDIS` in the MSR setting, extract from `dqsTrainRcvrEn_SW_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mcrsrc.c`

```c
/* NOTE: While the BKDG states to only program DqsRcvEnGrossDelay, this appears
 * to have been a misprint as DqsRcvEnFineDelay should be set to zero as well.
 */
```

Listing 4.28: A possible misprint in the BKDG regarding delay settings, extract from `dqsTrainRcvrEn_SW_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mcrsrc.c`

```c
initial_seed = (uint16_t) (((((uint64_t) initial_seed) *
    fam15h_freq_tab[mem_clk] * 100) / (min_mem_clk * 100)));
```

Listing 4.29: Complex seed adjustment logic, extract from `dqsTrainRcvrEn_SW_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mcrsrc.c`

In addition to the explicit `TODO` and `FIXME` comments, there are other aspects of the implementation that could impact performance and stability.

The logic for adjusting the seed values based on the memory frequency and the platform's minimum supported frequency is complex and prone to errors (lst. 4.29), especially when combined with the incomplete `TODO` features. The risk here is that incorrect seed values could be used, leading to timing mismatches during the training process.

Added to that, stock seeds from the BKDG are used (lst. 4.30). However, it seems that that seeds for used for DQS training should be extensively determined for each motherboard, and the BKDG [8] does not tell otherwise. Moreover, seeds can be configured uniquely for every possible socket, channel, DIMM module, and even byte lane combination. The current implementation is here only using the recommended seeds from the table 99 of the BKDG [8], which is not sufficient and absolutely not adapted to every DIMM module in the market.

The current implementation also has limited error handling and reporting. While some errors are detected during training, the code does not have robust mechanisms for recovering from or correcting these errors.

This approach might lead to further complications in high-load scenarios or when the memory configuration changes, as the underlying issues are not resolved.

#### 4.4.2.2 Write leveling

While the current implementation of write leveling on AMD Fam15h G34 processors with RDIMMs can be functional in some cases and provides the necessary steps to align DQS signals correctly during write operations, there are several areas where the implementation is either incomplete, relies on temporary workarounds, or may introduce stability and performance issues.

One of the most significant concerns with the current implementation is the presence of unresolved `TODO` and `FIXME` comments throughout the code. These comments indicate areas where the implementation is either incomplete or has known issues that have not been fully resolved.

In the `procConfig` function, a `TODO` comment mentions that the current implementation may not be using the correct or final value for this variable, once again because of a value from RC2[0] that isn't fetched, potentially leading to inaccuracies in the seed values used during write leveling (lst. 4.31). This inaccuracy can result in timing mismatches, which may cause data corruption or other stability issues.

Another `FIXME` in the code indicates that the `WrDqDqsEarly` parameter, which is critical for fine-tuning the DQS signals timing during write operations, is being ignored due to unresolved issues (lst. 4.32). This omission can result in less accurate timing adjustments, leading to potential marginal instability in systems where tight timing margins are critical.

```c
if (pDCTstat->Status & (1 << SB_Registered)) {
    if (package_type == PT_GR) {
        /* Socket G34: Fam15h BKDG v3.14 Table 99 */
        if (MaxDimmsInstallable == 1) {
            if (channel == 0)
                seed = 0x43;
            else if (channel == 1)
                seed = 0x3f;
            else if (channel == 2)
                seed = 0x3a;
            else if (channel == 3)
                seed = 0x35;
        } else if (MaxDimmsInstallable == 2) {
            if (channel == 0)
                seed = 0x54;
            else if (channel == 1)
                seed = 0x4d;
            else if (channel == 2)
                seed = 0x45;
            else if (channel == 3)
                seed = 0x40;
        } else if (MaxDimmsInstallable == 3) {
            if (channel == 0)
                seed = 0x6b;
            else if (channel == 1)
                seed = 0x5e;
            else if (channel == 2)
                seed = 0x4b;
            else if (channel == 3)
                seed = 0x3d;
        }
```

Listing 4.30: Seeds used for DQS Receiver training, extract from `dqsTrainRcvrEn_SW_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mcrsrc.c`

```c
uint8_t AddrCmdPrelaunch = 0;   /* TODO: Fetch the correct value from RC2[0] */
```

Listing 4.31: Incomplete seed generation implementation, extract from `procConfig` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

```c
/* FIXME: Ignore WrDqDqsEarly for now to work around training issues */
```

Listing 4.32: Omission of `WrDqDqsEarly` parameter, extract from `procConfig` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

```
/* FIXME: Implement mainboard-specific seed and WrDqsGrossDly base overrides.
 * 0x41 and 0x0 are the "stock" values */
```

Listing 4.33: Lack of mainboard-specific seed overrides, extract from `procConfig` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

```
/* FIXME: For now, disable CGD adjustment as it seems to interfere with
 * registered DIMM training */
```

Listing 4.34: Disabled CGD adjustment due to conflicts, extract from `AgesaHwWlPhase2` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

The current implementation uses generic or "stock" seed values for certain configurations, such as Socket G34 (lst. 4.33). Without mainboard-specific overrides, the memory initialization process might not be fully optimized for the particular motherboard in use. This could result in suboptimal performance or stability issues in specific environments, particularly in server applications where memory performance is critical.

In `AgesaHwWlPhase2`, there is a `FIXME` comment that suggests that the Critical Gross Delay adjustment has been temporarily disabled due to conflicts with RDIMM training (lst. 4.34). Disabling this adjustment can lead to less precise DQS alignment, especially in complex memory configurations like those using RDIMMs, potentially causing instability or degraded performance.

The function also bypasses (lst. 4.35) certain critical adjustments if the memory speed is being tuned (e.g., during frequency stepping). This bypass is noted as a temporary measure due to problems encountered during testing, where the first pass values were found to cause issues with PHY training on all Family 15h processors tested. This approach indicates a lack of robustness in the implementation, particularly in handling dynamic changes in memory frequency, which is essential for server environments where performance tuning is common.

The current implementation attempts to compensate for noise and instability by overriding faulty values with seed values in `AgesaHwWlPhase2` (lst. 4.36). However, this approach is somewhat blunt and reactive, addressing the symptoms rather than the underlying causes of instability. This method does not ensure that noise or instability is sufficiently mitigated, potentially leading to marginal or sporadic failures during normal operation.

The handling of x4 DIMMs, with separate training for each nibble, introduces additional complexity. While necessary for these configurations, the logic is fragmented, with several points where the function branches based on whether the DIMM is x4. This complexity increases the risk of bugs or missed conditions, particularly if future changes or enhancements are made to the code. The overcomplicated logic can also make the code more difficult to maintain and extend.

#### 4.4.2.3 DQS position training

While the DQS position training algorithm implemented in the `TrainDQSRdWrPos_D_Fam15` function may work in some cased to ensure optimal data strobe alignment, there are several critical flaws and issues within the implementation that could impact its effectiveness and reliability.

```
/* FIXME: Using the Pass 1 training values causes major phy training problems on
 * all Family 15h processors I tested (Pass 1 values are randomly too high,
 * and Pass 2 cannot lock). Figure out why this is and fix it, then remove
 * the bypass code below... */
```

Listing 4.35: Bypass of critical adjustments during speed tuning, extract from `AgesaHwWlPhase2` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

```
if (faulty_value_detected) {
    pDCTData->WLGrossDelay[index+ByteLane] =
        pDCTData->WLSeedGrossDelay[index+ByteLane];
    pDCTData->WLFineDelay[index+ByteLane] =
        pDCTData->WLSeedFineDelay[index+ByteLane];
    status = 1;
}
```

Listing 4.36: Blunt error handling to compensate for noise and instability, extract from `AgesaHwWlPhase2` function in `src/northbridge/amd/amdmct/mct_ddr3/mhwlc_d.c`

Throughout the function, there is an overreliance on hardcoded constants and magic numbers, such as:

- The use of `0x20` to represent 1 UI (Unit Interval) in multiple places.
- The constant `16` used in the adjustment of `region_center` during the processing of results.
- Magic numbers like 32 and 48 in the array dimensions for `dqs_results_array`.

These values should be replaced with named constants or variables that clearly indicate their purpose, improving code readability and maintainability. Additionally, using well-defined constants would allow easier adjustments if the algorithm needs to be adapted for different hardware configurations or future revisions of the architecture.

The error handling within the function is rudimentary, with errors being flagged primarily by setting bits in the `Errors` variable. However, the function does not provide detailed diagnostics or recovery strategies when an error occurs. For example:

- If no passing DQS delay is found for a lane, the function simply sets an error bit without attempting any corrective actions or providing detailed information on what went wrong.
- The early abort mechanism based on the value read from the `0x264` register does not offer a robust fallback or retry mechanism, which could lead to situations where minor, recoverable issues cause the entire training process to fail.

Improving the error handling to include detailed diagnostics, logging, and potentially corrective actions (such as retrying the training with adjusted parameters) would make the function more resilient and reliable.

The function contains several areas where the logic is more complex than necessary, which can lead to difficulties in understanding and maintaining the code. Examples include:

- The nested loops for iterating over write and read delays are deeply nested, making it challenging to follow the flow of the code and understand the interactions between different parts of the algorithm.
- The use of multiple copies of delay settings (e.g., `current_write_data_delay`, `initial_write_data_timing`, and `initial_write_dqs_delay`) introduces redundancy and increases the likelihood of errors or inconsistencies.

Refactoring the code to simplify the logic, reduce redundancy, and make the flow of operations clearer would improve both the readability and reliability of the implementation.

The current implementation does not adequately handle edge cases and boundary conditions, such as:

- The warning issued when a negative DQS recovery delay is detected suggests that the function continues despite recognizing a potentially critical issue, which could lead to system instability (lst. 4.37).
- The averaging of delay values for dual-rank DIMMs does not account for the possibility of significant discrepancies between the ranks, which could result in suboptimal or unstable settings.

```c
if (best_count > 2) {
    uint16_t region_center = (best_pos + (best_count / 2));
    if (region_center < 16) {
        printk(BIOS_WARNING,
            "TrainDQSRdWrPos: negative DQS recovery delay detected!");
        region_center = 0;
    } else {
        region_center -= 16;
    }
    ...
    current_read_dqs_delay[lane] = region_center;
    passing_dqs_delay_found[lane] = 1;
    write_dqs_read_data_timing_registers(current_read_dqs_delay, dev, dct, dimm, index_reg);
}
```

Listing 4.37: Allowing a negative DQS recovery delay measurement, extract from `TrainDQSRdWrPos_D_Fam15` function in `src/northbridge/amd/amdmct/mct_ddr3/mctdqs_d.c`

- The function does not include comprehensive checks for situations where the calculated delay settings might exceed hardware limitations or cause timing violations.

Improving the handling of edge cases and boundary conditions, possibly by incorporating more robust validation checks and conservative fallback mechanisms, would make the algorithm more reliable in a wider range of scenarios.

The code contains several `TODO` and `FIXME` comments that indicate incomplete or problematic parts of the implementation:

- The comment `TODO: Fetch the correct value from RC2[0]` suggests that critical configuration values are not correctly initialized, which could compromise the entire training process.
- The `FIXME` comments related to early abort checks and DQS recovery delay calculations indicate that there are known issues with the current approach that have not been resolved, potentially leading to incorrect or unstable results.
- The handling of antiphase results, particularly with respect to checking for early aborts, is incomplete and could lead to situations where incorrect results are accepted without proper validation.

The current implementation's approach to iterating over every possible combination of write and read delays is exhaustive but may be inefficient. The function performs multiple reads and writes to hardware registers for every iteration, which could be time-consuming, especially on systems with a large number of lanes or complex memory configurations.

Consideration should be given to optimizing the algorithm, possibly by narrowing the search space based on prior knowledge or implementing more efficient search techniques, to reduce the time required for DQS position training without compromising accuracy.

#### 4.4.2.4 On saving training values in NVRAM

The function `mctAutoInitMCT_D` is responsible for automatically initializing the memory controller training (MCT) process, which involves configuring various memory parameters and performing training routines to ensure stable and efficient memory operation. However, the fact that `mctAutoInitMCT\_D` does not allow for the restoration of training data from NVRAM (lst. L.42) poses several significant problems.

Memory training is a time-consuming process that involves multiple iterations of read/write operations, delay adjustments, and calibration steps. By not restoring previously saved training data from NVRAM, the system is

forced to re-run the full training sequence every time it boots up. This leads to longer boot times, which can be particularly problematic in environments where quick system restarts are critical, such as in servers or embedded systems.

Each time memory training is performed, it puts additional stress on the memory modules and the memory controller. Repeatedly executing the training process at every boot can contribute to the wear and tear of hardware components, potentially reducing their lifespan. This issue is especially concerning in systems that frequently power cycle or reboot.

Memory training is sensitive to various factors, such as temperature, voltage, and load conditions. As a result, the training results can vary slightly between different boot cycles. Without the ability to restore previously validated training data, there is a risk of inconsistency in memory performance across reboots. This could lead to instability or suboptimal memory operation, affecting the overall performance of the system.

If the memory training process fails during boot, the system may be unable to operate properly or may fail to boot entirely. By restoring validated training data from NVRAM, the system can bypass the training process altogether, reducing the risk of boot failures caused by training issues. Without this feature, any minor issue that affects training could result in system downtime.

Finally, modern memory controllers often include power-saving features that are fine-tuned during the training process. By reusing validated training data from NVRAM, the system can quickly return to an optimized state with lower power consumption. The inability to restore this data forces the system to operate at a potentially less efficient state until training is complete, leading to higher power consumption during the boot process.

#### 4.4.2.5 A seedless DQS position training algorithm

An algorithm to find the best timing for the DQS so that the memory controller can reliably read data from the memory could be done without relying on any pre-known starting values (seeds). This would allow for better reliability and wider support for different situations. The algorithm could be describe as follows.

- Prepare Memory Controller: The memory controller needs to be in a state where it can safely adjust the DQS timing without affecting the normal operation of the system. By blocking the DQS signal locking, we ensure that the adjustments made during training do not interfere with the controllers ability to capture data until the optimal settings are found.
- Initialize Variables: Set up variables to store the various timing settings and test results for each bytelane. This setup is crucial because each bytelane might require a different optimal timing, and keeping track of these values ensures that the algorithm can correctly determine the best delay settings later.

The main loop is the core of the algorithm, where different timing settings are systematically explored. By looping through possible delay settings, the algorithm ensures that it doesn't miss any potential optimal timings. The loop structure allows a methodical test of a range of delays to find the most reliable one.

The gross delay is here the coarse adjustment to the timing of the DQS signal. It shifts the timing window by a large amount, helping to broadly align the DQS with the data lines (DQ). The fine delay, which is the smaller, more precise change to the timing of the DQS signal once the coarse alignment (through gross delay) has been achieved, would then be computed.

To compute a delay, here would be the steps:

- Set a delay: Setting an initial delay allows the algorithm to start testing. The initial delay might be zero or another default value, providing a baseline from which to begin the search for the optimal timing.
- Test it: After setting the delay, it is essential to test whether the memory controller can read data correctly. This step is critical because it indicates whether the current delay setting is within the acceptable range for reliable data capture.

- Check the result: If the memory controller successfully reads data, it means the current delay setting is valid. This information is crucial because it helps define the range of acceptable timings. If the test fails, it indicates that the curren t delay setting is outside the range where the memory controller can reliably capture data.
- Increase/decrease delay: By incrementally adjusting the delay, either increasing or decreasing, the algorithm can explore different timing settings in a controlled manner. This ensures that the entire range of possible delays is covered without skipping over any potential good delays.
- Test again: Re-testing after each adjustment ensures that the exact point where the DQS timing goes from acceptable (pass) to unacceptable (fail) is caught. This step helps in identifying the transition point, which is often the optimal place to set the DQS delay.
- Look for a transition: The transition from pass to fail is where the DQS timing crosses the boundary of the valid timing window. This transition is crucial because it marks the end of the reliable range. The best timing is usually just before this transition.
- Record the best setting: Storing the best delay setting for each bytelane ensures that a reliable timing configuration is available when the training is complete.
- Confirm all bytelanes: Before finalizing the settings, it is important to ensure that the chosen delays work for all bytelanes. This step serves as a final safeguard against errors, ensuring that every part of the data bus is correctly aligned.

Each bytelane (8-bit segment of data) may require a different optimal delay setting. By repeating the process for all bytelanes, the algorithm ensures that the entire data bus is correctly timed. Misalignment in even one bytelane can lead to data errors, making it essential to tune every bytelane individually.

Once the best settings are confirmed, they need to be applied to the memory controller for use during normal operation. This step locks in the most reliable timing configuration found during the training process.

After the optimal settings are applied, it is necessary to allow the DQS signal locking mechanism to resume. This locks in the delay settings, ensuring stable operation going forward.

Finally, the algorithm needs to indicate whether it was successful in finding reliable timing settings for all bytelanes. This feedback is crucial for determining whether the memory system is correctly configured or if further adjustments or troubleshooting are needed.

# 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 [16], 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.

More importantly, the ACPI Component Architecture (ACPICA) is a critical component integrated into the Linux kernel, serving as the foundation for the system's ACPI implementation [26]. ACPICA provides the core ACPI functionalities, such as hardware configuration, power management, and thermal management, which are essential for modern computing platforms. However, its integration into the Linux kernel has brought significant complexity and code overhead, making Linux heavily dependent on ACPICA for managing ACPI-related tasks.
ACPICA is a large and complex project, with its codebase encompassing a wide range of functionalities required to implement ACPI standards. The integration of ACPICA into the Linux kernel significantly increases the kernel's overall code size. An example of that can easily be reproduced with a small experiment (lst. 5.38).

```
$ git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
$ cd linux/drivers/acpi
$ find . -name "*.c" -o -name "*.h" | xargs wc -l
...
 168970 total
```

Listing 5.38: How to estimate the impact of ACPICA in Linux

As of recent statistics, ACPICA comprises between 100,000 to 200,000 lines of code, making it one of the larger subsystems within the Linux kernel. This size is indicative of the extensive range of features and capabilities ACPICA must support, including but not limited to the ACPI interpreter, AML (ACPI Machine Language) parser, and various hardware-specific drivers. The ACPICA codebase is not monolithic; it is highly modular and consists of various components, each responsible for specific ACPI functions. For instance, ACPICA includes components

for managing ACPI tables, interpreting AML bytecode, handling events, and interacting with hardware. This modularity, while beneficial for isolating different functionalities, also contributes to the overall complexity of the system. The separation of ACPICA into multiple modules necessitates careful coordination and integration with the rest of the Linux kernel, adding to the kernel's complexity.

ACPICA's integration into the Linux kernel is designed to maintain a clear separation between the core ACPI functionalities and the kernel's other subsystems [26]. This separation is achieved through well-defined interfaces and abstraction layers, allowing the Linux kernel to interact with ACPICA without being tightly coupled to its internal implementation details. For example, ACPICA provides an API that the Linux kernel can use to interact with ACPI tables, execute ACPI methods, and manage power states. This API abstracts the underlying complexity of the ACPI implementation, making it easier for kernel developers to incorporate ACPI support without delving into the intricacies of ACPICA's internals. Moreover, ACPICA's role in interpreting AML bytecode, which is essentially a form of low-level programming language embedded in ACPI tables, adds a layer of abstraction. The Linux kernel relies on ACPICA to execute AML methods and manage hardware resources according to the ACPI specifications. This reliance further underscores the idea that ACPI acts as a virtualizing environment, shielding the kernel from the complexities of directly interfacing with hardware components.

## 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 [54] 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 [74] 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. Indeed, as we studied in a precedent chapter, UEFI is designed as a modular and extensible firmware interface that sits between the computer's hardware and the operating system. Unlike the monolithic BIOS, UEFI is composed of several layers and components, each responsible for different aspects of the system's boot and runtime processes. The core components of UEFI include the Pre-EFI Initialization (PEI), Driver Execution Environment (DXE), Boot Device Selection (BDS), and Runtime Services. Each of these components plays a critical role in initializing the hardware, managing drivers, selecting boot devices, and providing runtime services to the OS.

The PEI (Pre-EFI Initialization) phase is responsible for initializing the CPU, memory, and other essential hardware components. It ensures that the system is in a stable state before handing control to the DXE phase. In the DXE phase, the system loads and initializes various drivers required for the OS to interact with the hardware. The DXE phase also constructs the UEFI Boot Services, which provide the OS with interfaces to the hardware during the boot process. The BDS (Boot Device Selection) phase is responsible for selecting the device from which the OS will boot. It interacts with the UEFI Boot Manager to determine the correct boot path and load the OS. After the OS has booted, UEFI provides Runtime Services that remain accessible to the OS. These services include

interfaces for managing system variables, time, and hardware. UEFI also supports the execution of standalone applications, which can be used for system diagnostics, firmware updates, or other tasks. These applications operate independently of the OS, highlighting UEFI's capabilities as a minimalistic OS.

UEFI abstracts the underlying hardware from the OS, providing a standardized interface for the OS to interact with different hardware components. This abstraction simplifies the development of OSes and drivers, as they do not need to be tailored for specific hardware configurations. UEFI's hardware abstraction is one of the key features that enable it to act as a virtualizing environment for the OS [74].

### 5.3.1 Memory Management

UEFI provides a detailed memory map to the OS during the boot process, which includes information about available, reserved, and used memory regions. The OS uses this memory map to manage its own memory allocation and paging mechanisms. The overlap in memory management functions highlights UEFI's role in preparing the system for OS operation. This memory map includes all the memory regions in the system, categorized into different types, such as usable memory, reserved memory, and memory-mapped I/O. The OS relies on this map to understand the system's memory layout and avoid conflicts [86]. The OS extends UEFI's memory management by implementing its own memory allocation, paging, and virtual memory mechanisms. However, the OS's memory management is built on the foundation provided by UEFI, demonstrating the close relationship between the two.

### 5.3.2 File System Management

UEFI includes its own file system management capabilities, which overlap with those of the OS. The most notable example is the EFI System Partition (ESP), a special partition formatted with the FAT file system that UEFI uses to store bootloaders, drivers, and other critical files [40]. The ESP is a mandatory partition in UEFI systems, containing the bootloaders, firmware updates, and other files necessary for system initialization. UEFI accesses the ESP independently of the OS, but the OS can also access and manage files on the ESP, creating an overlap in file system management functions [62]. UEFI natively supports the FAT file system, allowing it to read and write files on the ESP. This support overlaps with the OS's file system management, as both UEFI and the OS can manipulate files on the ESP.

### 5.3.3 Device Drivers

As we studied in an earlier chapter, UEFI includes its own driver model, allowing it to load and execute drivers independently of the OS. This capability overlaps with the OS's driver management functions, as both UEFI and the OS manage hardware devices through drivers. UEFI drivers are typically used during the boot process to initialize and control hardware devices. These drivers provide the necessary interfaces for the OS to interact with the hardware once it has booted [62]. After the OS has booted, it loads its own drivers for hardware devices. However, the OS often relies on the initial hardware setup performed by UEFI drivers.

### 5.3.4 Power Management

UEFI provides power management services that overlap with the OS's power management functions. These services allow UEFI to manage power states and transitions independently of the OS [40]. These services ensure that the system conserves power during periods of inactivity and can quickly resume operation when needed The OS extends UEFI's power management by implementing its own power-saving mechanisms, such as CPU throttling and dynamic voltage scaling.

## 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 [19] 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 Processors microcode

Modern CPUs are incredibly complex, with their functionality relying heavily on microcode to interpret and execute instructions. Microcode acts as a translation layer between the high-level instructions that software provides and the lower-level operations that the hardware can execute. Microcode operates directly within the CPU.

CPU microcode is a set of low-level firmware instructions embedded within the processor. It translates complex machine instructions into simpler, executable sequences of operations that the CPU's hardware can directly perform [31]. This layer of abstraction allows CPU manufacturers to update or patch the behavior of the processor post-manufacturing, which is crucial for addressing bugs, optimizing performance, and applying security patches [135]. In a sense, microcode can be seen as an argument for the CPU running a form of low-level virtual machine. Just as a VM abstracts and manages hardware resources for a guest OS, microcode abstracts and manages the complexity of CPU hardware for machine-level instructions. This virtualization enables the CPU to support a wide variety of instructions and operational modes without needing to change the underlying hardware [1].

## 5.6 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 [101] 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 and even CPU microcode 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 [109]. 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] J.A. Abraham and M.A. Breuer. "A Study of Redundant Logic in Microprocessors". In: *Proceedings of the 10th Annual International Symposium on Computer Architecture*. ACM, 1983, pp. 116–126.

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

[3] 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.

[4] Advanced Micro Devices (AMD). *AMD SR5690 Databook*. Accessed: 2024-08-17. URL: https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/datasheets/43869.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. "BIOS and Kernel Developers Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors Rev 3.14". In: 42301. Jan. 2013.

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

[10] AMD. *Revision Guide for AMD Family 15h Models 00h-0Fh Rev 3.10*. Tech. rep. 48931. Advanced Micro Devices, Inc., May 2013.

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

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

[15] 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.

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

[17] 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.

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

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

[20] Jon Burnett. *DDR3 Design Considerations for PCB Applications (AN111)*. Presentation at Freescale Technology Forum. Freescale Semiconductor, Inc. July 2009.

[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 Developer Manual*. Accessed: 2024-08-17. URL: https://www.coreboot.org/Developer_Manual.

[25] Coreboot Project. *coreboot repository, tag 4.11*. Accessed: 2024-08-24. 2019. URL: https://review.coreboot.org/plugins/gitiles/coreboot/+/refs/tags/4.11.

[26] Intel Corporation. *ACPI Programming Reference*. Accessed: 2024-08-24. Apr. 2023. URL: https://cdrdv2.intel.com/v1/dl/getContent/772726.

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

[28] 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.

[29] Intel Corporation. *Introduction to ACPI*. Accessed: 2024-08-24. Apr. 2023. URL: https://cdrdv2.intel.com/v1/dl/getContent/772721.

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

[31] Intel Corporation. "Understanding Intel's Microcode". In: *Intel Developer's Manual* (2018). Available online: https://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

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

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

[34] 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.

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

[36] 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.

[37] 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.

[38] 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 informatĺąsiÿąionnykh tekhnologiÄ* 29.4 (2022), pp. 27–41. ISSN: 2074-7128.

[39] UEFI Forum. *ACPI Specification*. https://uefi.org/sites/default/files/resources/ACPI_Spec_6_5_Aug29.pdf. Accessed: 2024-07-05.

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

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

[42] 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.

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

[44] 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.

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

[46] Siddula Gopikrishna. "A Novel Impedance Calibration Method for Low Cost Memory Applications". Accessed: 2024-08-24. Master of Science Thesis. Hyderabad, India: International Institute of Information Technology, Hyderabad, Feb. 2021. URL: https://cdn.iiit.ac.in/cdn/web2py.iiit.ac.in/research_centres/publications/download/mastersthesis.pdf.a761b452d5c4ae57.476f70696b726973686e615f4d.pdf.

[47] 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.

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

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

[50] 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.

[51] 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.

[52] Maximilian Herrmann and Martin Niemietz. *Dissecting the AMD Platform Security Processor*. Conference presentation at the 34th Chaos Communication Congress (34C3). Accessed: 2024-08-27. 2017. URL: https://media.ccc.de/v/thms-38-dissecting-the-amd-platform-security-processor.

[53] 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.

[54] 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.

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

[56] Intel Corporation. *Intel Management Engine (Intel ME)*. https://www.intel.com/content/www/us/en/support/articles/000008927/software/chipset-software.html. Accessed: 2024-07-05.

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

[58] B.W. Johnson. *Fault-Tolerant Microprocessor Design*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

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

[60] 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.

[61] Dong-Seok Kim, Dong-Seok Oh, and Seok-Hoon Lee. "Design DDR3 ZQ Calibration having improved impedance matching". In: *Proceedings of the 2010 Fall Conference on Semiconductor and Display Technology*. Retrieved from https://koreascience.kr/article/CFKO200835536002505.pdf. Seoul, South Korea: Hanyang University, 2010, pp. 191–192.

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

[63] 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.

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

[65] 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.

[66] 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.

[67] 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.

[68] 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.

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

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

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

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

[73] 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.

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

[75] 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.

[76] 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.

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

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

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

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

[81] 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.

[82] 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.

[83] 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.

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

[85] OSDev Wiki contributors. *UEFI - OSDev Wiki*. [Online; accessed 25-August-2024]. 2024. URL: https://wiki.osdev.org/UEFI.

[86] OSDev Wiki contributors. *UEFI Memory - OSDev Wiki*. [Online; accessed 25-August-2024]. 2024. URL: https://wiki.osdev.org/UEFI#Memory.

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

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

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

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

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

[92] TianoCore Project. *TianoCore as a Coreboot Payload*. Accessed: 2024-08-17. 2019. URL: https://github.com/tianocore/tianocore.github.io/wiki/Coreboot_UEFI_payload.

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

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

[95] 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.

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

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

[98] 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.

[99] 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.

[100] 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/.

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

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

[103] 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.

[104] 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.

[105] 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.

[106] 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.

[107] 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.

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

[109] Ward Vandewege. "Coreboot: the view from the FSF". In: 2008.

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

[111] Vikings GmbH. *Vikings Hardware Recommendations for KGPE-D16*. Accessed: 2024-08-17. URL: https://wiki.vikings.net/hardware:kgpe-d16?s[]=kgpe&s[]=d16.

[112] 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.

[113] 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.

[114] 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.

[115] 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.

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

[117] 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.

[118] 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.

[119] 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.

[120] 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.

[121] 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.

[122] 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.

[123] 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.

[124] 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.

[125] 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.

[126] 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.

[127] 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.

[128] 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.

[129] 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.

[130] 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.

[131] 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.

[132] 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.

[133] 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.

[134] 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.

[135] J. Wilcox et al. "Understanding and Mitigating the Microcode Update Latency". In: *Proceedings of the 2018 IEEE International Symposium on High-Performance Computer Architecture* (2018), pp. 234–245.

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

[137] 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.

[138] 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.

# Appendix: Long code listings

```c
void mctAutoInitMCT_D(struct MCTStatStruc *pMCTstat,
                      struct DCTStatStruc *pDCTstatA)
{
    /*
     * Memory may be mapped contiguously all the way up to 4GB
     * (depending on setup options).  It is the responsibility of PCI
     * subsystem to create an uncacheable IO region below 4GB and to adjust
     * TOP_MEM downward prior to any IO mapping or accesses.  It is the same
     * responsibility of the CPU sub-system prior to accessing LAPIC.
     *
     * Slot Number is an external convention, and is determined by OEM with
     * accompanying silk screening.  OEM may choose to use Slot number
     * convention which is consistent with DIMM number conventions.
     * All AMD engineering platforms do.
     *
     * Build Requirements:
     * 1. MCT_SEG0_START and MCT_SEG0_END macros to begin and end the code
     * segment, defined in mcti.inc.
     *
     * Run-Time Requirements:
     * 1. Complete Hypertransport Bus Configuration
     * 2. SMBus Controller Initialized
     * 1. BSP in Big Real Mode
     * 2. Stack at SS:SP, located somewhere between A000:0000 and F000:FFFF
     * 3. Checksummed or Valid NVRAM bits
     * 4. MCG_CTL = -1, MC4_CTL_EN = 0 for all CPUs
     * 5. MCi_STS from shutdown/warm reset recorded (if desired) prior to entry
     * 6. All var MTRRs reset to zero
     * 7. State of NB_CFG.DisDatMsk set properly on all CPUs
     * 8. All CPUs at 2GHz Speed (unless DQS training is not installed).
     * 9. All cHT links at max Speed/Width (unless DQS training is not
     * installed).
     *
     * Global relationship between index values and item values:
     *
     * pDCTstat.CASL pDCTstat.Speed
     * j CL(j)       k   F(k)
     * --------------------------
     * 0 2.0         -   -
     * 1 3.0         1   200 MHz
     * 2 4.0         2   266 MHz
     * 3 5.0         3   333 MHz
     * 4 6.0         4   400 MHz
     * 5 7.0         5   533 MHz
     * 6 8.0         6   667 MHz
     * 7 9.0         7   800 MHz
     */
    ...
}
```

Listing L.39: Beginning of `mctAutoInitMCT_D()`, extract from `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`

```c
void mctAutoInitMCT_D(struct MCTStatStruc *pMCTstat,
                      struct DCTStatStruc *pDCTstatA)
{
    ...
restartinit:
    if (!mctGet_NVbits(NV_ECC_CAP) || !mctGet_NVbits(NV_ECC))
        pMCTstat->try_ecc = 0;
    else
        pMCTstat->try_ecc = 1;
    ...
    for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
        struct DCTStatStruc *pDCTstat;
        pDCTstat = pDCTstatA + Node;
        /* Zero out data structures to avoid false detection of DIMMs */
        memset(pDCTstat, 0, sizeof(struct DCTStatStruc));
        /* Initialize data structures */
        pDCTstat->Node_ID = Node;
        pDCTstat->dev_host = PA_HOST(Node);
        pDCTstat->dev_map = PA_MAP(Node);
        pDCTstat->dev_dct = PA_DCT(Node);
        pDCTstat->dev_nbmisc = PA_NBMISC(Node);
        pDCTstat->dev_link = PA_LINK(Node);
        pDCTstat->dev_nbctl = PA_NBCTL(Node);
        pDCTstat->NodeSysBase = node_sys_base;
        if (mctGet_NVbits(NV_PACK_TYPE) == PT_GR) {
            uint32_t dword;
            pDCTstat->Dual_Node_Package = 1;
            /* Get the internal node number */
            dword = Get_NB32(pDCTstat->dev_nbmisc, 0xe8);
            dword = (dword >> 30) & 0x3;
            pDCTstat->Internal_Node_ID = dword;
        } else {
            pDCTstat->Dual_Node_Package = 0;
        }
        printk(BIOS_DEBUG, "%s: mct_init Node %d\n", __func__, Node);
        mct_init(pMCTstat, pDCTstat);
        mctNodeIDDebugPort_D();
        pDCTstat->NodePresent = NodePresent_D(Node);
        if (pDCTstat->NodePresent) {
            pDCTstat->LogicalCPUID = mctGetLogicalCPUID_D(Node);
            printk(BIOS_DEBUG, "%s: mct_InitialMCT_D\n", __func__);
            mct_InitialMCT_D(pMCTstat, pDCTstat);
            printk(BIOS_DEBUG, "%s: mctSMBhub_Init\n", __func__);
            /* Switch SMBUS crossbar to proper node */
            mctSMBhub_Init(Node);
            printk(BIOS_DEBUG, "%s: mct_preInitDCT\n", __func__);
            mct_preInitDCT(pMCTstat, pDCTstat);
        }
        node_sys_base = pDCTstat->NodeSysBase;
        node_sys_base += (pDCTstat->NodeSysLimit + 2) & ~0x0F;
    }
    ...
}
```

Listing L.40: DIMM initialization in `mctAutoInitMCT_D()`, extract from `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`

```c
void mctAutoInitMCT_D(struct MCTStatStruc *pMCTstat,
                      struct DCTStatStruc *pDCTstatA)
{
    ...
    /* If the boot fails make sure training is attempted after reset */
    nvram = 0;
    set_option("allow_spd_nvram_cache_restore", &nvram);

#if CONFIG(DIMM_VOLTAGE_SET_SUPPORT)
    printk(BIOS_DEBUG, "%s: DIMMSetVoltage\n", __func__);
    /* Set the DIMM voltages (mainboard specific) */
    DIMMSetVoltages(pMCTstat, pDCTstatA);
#endif
    if (!CONFIG(DIMM_VOLTAGE_SET_SUPPORT)) {
        /* Assume 1.5V operation */
        for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
            struct DCTStatStruc *pDCTstat;
            pDCTstat = pDCTstatA + Node;
            if (!pDCTstat->NodePresent)
                continue;
            for (dimm = 0; dimm < MAX_DIMMS_SUPPORTED; dimm++) {
                if (pDCTstat->DIMMValid & (1 << dimm))
                    pDCTstat->DimmConfiguredVoltage[dimm] = 0x1;
            }
        }
    }
    ...
}


```

Listing L.41: Voltage control in `mctAutoInitMCT_D()`, extract from `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`

```c
void mctAutoInitMCT_D(struct MCTStatStruc *pMCTstat,
                      struct DCTStatStruc *pDCTstatA)
{
    ...
    /* If DIMM configuration has not changed since last boot restore
     *  training values */
    allow_config_restore = 1;
    for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
        struct DCTStatStruc *pDCTstat;
        pDCTstat = pDCTstatA + Node;

        if (pDCTstat->NodePresent)
            if (!pDCTstat->spd_data.nvram_spd_match)
                allow_config_restore = 0;
    }
    /* FIXME
     * Stability issues have arisen on multiple Family 15h systems
     * when configuration restoration is enabled.  In all cases these
     * stability issues resolved by allowing the RAM to go through a
     * full training cycle.
     *
     * Debug and reenable this!
     */
    allow_config_restore = 0;
    ...
}


```

Listing L.42: `mctAutoInitMCT_D()` does not allow restoring previous training values, extract from `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`

```c
void mctAutoInitMCT_D(struct MCTStatStruc *pMCTstat,
                      struct DCTStatStruc *pDCTstatA)
{
    ...
    for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
        struct DCTStatStruc *pDCTstat;
        pDCTstat = pDCTstatA + Node;
        if (pDCTstat->NodePresent) {
            printk(BIOS_DEBUG, "%s: mctSMBhub_Init\n", __func__);
            /* Switch SMBUS crossbar to proper node*/
            mctSMBhub_Init(Node);

            printk(BIOS_DEBUG, "%s: mct_initDCT\n", __func__);
            mct_initDCT(pMCTstat, pDCTstat);
            if (pDCTstat->ErrCode == SC_FatalErr) {
                goto fatalexit;                /* any fatal errors?*/
            } else if (pDCTstat->ErrCode < SC_StopError) {
                NodesWmem++;
            }
        }
    }
    if (NodesWmem == 0) {
        printk(BIOS_ALERT, "Unable to detect valid memory on any nodes.  Halting!\n");
        goto fatalexit;
    }
    printk(BIOS_DEBUG, "mctAutoInitMCT_D: SyncDCTsReady_D\n");
    /* Make sure DCTs are ready for accesses.*/
    SyncDCTsReady_D(pMCTstat, pDCTstatA);
    printk(BIOS_DEBUG, "mctAutoInitMCT_D: HTMemMapInit_D\n");
    /* Map local memory into system address space.*/
    HTMemMapInit_D(pMCTstat, pDCTstatA);
    mctHookAfterHTMap();
    printk(BIOS_DEBUG, "mctAutoInitMCT_D: mctHookAfterCPU\n");
    /* Setup external northbridge(s) */
    mctHookAfterCPU();
    ...
    return;
fatalexit:
    die("mct_d: fatalexit");
}
```

Listing L.43: Preparing SMBus, DCTs and NB in `mctAutoInitMCT_D()` from `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`

```c
void mctAutoInitMCT_D(struct MCTStatStruc *pMCTstat,
                      struct DCTStatStruc *pDCTstatA)
{
    ...
    /* FIXME
     * Previous training values should only be used if the current desired
     * speed is the same as the speed used in the previous boot.
     * How to get the desired speed at this point in the code?
     */
    printk(BIOS_DEBUG, "mctAutoInitMCT_D: DQSTiming_D\n");
    /* Get Receiver Enable and DQS signal timing*/
    DQSTiming_D(pMCTstat, pDCTstatA, allow_config_restore);
    if (!allow_config_restore) {
        printk(BIOS_DEBUG, "mctAutoInitMCT_D: :OtherTiming\n");
        mct_OtherTiming(pMCTstat, pDCTstatA);
    }
    /* RESET# if 1st pass of DIMM spare enabled*/
    if (ReconfigureDIMMspare_D(pMCTstat, pDCTstatA)) {
        goto restartinit;
    }
    InterleaveNodes_D(pMCTstat, pDCTstatA);
    InterleaveChannels_D(pMCTstat, pDCTstatA);
    ecc_enabled = 1;
    for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
        struct DCTStatStruc *pDCTstat;
        pDCTstat = pDCTstatA + Node;
        if (pDCTstat->NodePresent)
            if (!is_ecc_enabled(pMCTstat, pDCTstat))
                ecc_enabled = 0;
    }
    if (ecc_enabled) {
        printk(BIOS_DEBUG, "mctAutoInitMCT_D: ECCInit_D\n");
        /* Setup ECC control and ECC check-bits*/
        if (!ECCInit_D(pMCTstat, pDCTstatA)) {
            /* Memory was not cleared during ECC setup */
            /* mctDoWarmResetMemClr_D(); */
            printk(BIOS_DEBUG, "mctAutoInitMCT_D: MCTMemClr_D\n");
            MCTMemClr_D(pMCTstat,pDCTstatA);
        }
    }
    ...
    return;
fatalexit:
    die("mct_d: fatalexit");
}
```

Listing L.44: Get DQS, reset and activate ECC in `mctAutoInitMCT_D()` from `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`

```c
void mctAutoInitMCT_D(struct MCTStatStruc *pMCTstat,
                      struct DCTStatStruc *pDCTstatA)
{
    ...
    printk(BIOS_DEBUG, "mctAutoInitMCT_D: CPUMemTyping_D\n");
    /* Map dram into WB/UC CPU cacheability */
    CPUMemTyping_D(pMCTstat, pDCTstatA);
    printk(BIOS_DEBUG, "mctAutoInitMCT_D: UMAMemTyping_D\n");
    /* Fix up for UMA sizing */
    UMAMemTyping_D(pMCTstat, pDCTstatA);
    printk(BIOS_DEBUG, "mctAutoInitMCT_D: mct_ForceNBPState0_Dis_Fam15\n");
    for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
        struct DCTStatStruc *pDCTstat;
        pDCTstat = pDCTstatA + Node;
        mct_ForceNBPState0_Dis_Fam15(pMCTstat, pDCTstat);
    }
    enable_cc6 = 0;
    if (get_option(&nvram, "cpu_cc6_state") == CB_SUCCESS)
        enable_cc6 = !!nvram;
    if (enable_cc6) {
        uint8_t num_nodes;
        num_nodes = 0;
        for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
            struct DCTStatStruc *pDCTstat;
            pDCTstat = pDCTstatA + Node;
            if (pDCTstat->NodePresent)
                num_nodes++;
        }
        for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
            struct DCTStatStruc *pDCTstat;
            pDCTstat = pDCTstatA + Node;
            if (pDCTstat->NodePresent)
                set_up_cc6_storage_fam15(pMCTstat, pDCTstat, num_nodes);
        }
        for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
            struct DCTStatStruc *pDCTstat;
            pDCTstat = pDCTstatA + Node;
            if (pDCTstat->NodePresent) {
                set_cc6_save_enable(pMCTstat, pDCTstat, 1);
                lock_dram_config(pMCTstat, pDCTstat);
            }
        }
    }
    mct_FinalMCT_D(pMCTstat, pDCTstatA);
    printk(BIOS_DEBUG, "mctAutoInitMCT_D Done: Global Status: %x\n", pMCTstat->GStatus);
    return;
fatalexit:
    die("mct_d: fatalexit");
}
```

Listing L.45: Mapping DRAM with cache, validating DCT nodes and finishing the init process in `mctAutoInitMCT_D()` from `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`

```c
retry_requested = 0;
for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
    struct DCTStatStruc *pDCTstat;
    pDCTstat = pDCTstatA + Node;

    if (pDCTstat->NodePresent) {
        if (pDCTstat->TrainErrors & (1 << SB_FatalError)) {
            printk(BIOS_ERR, "DIMM training FAILED!  Restarting system...");
            soft_reset();
        }
        if (pDCTstat->TrainErrors & (1 << SB_RetryConfigTrain)) {
            retry_requested = 1;

            pDCTstat->TrainErrors &= ~(1 << SB_RetryConfigTrain);
            pDCTstat->TrainErrors &= ~(1 << SB_NODQSPOS);
            pDCTstat->ErrStatus &= ~(1 << SB_RetryConfigTrain);
            pDCTstat->ErrStatus &= ~(1 << SB_NODQSPOS);
        }
    }
}

if (retry_requested) {
    printk(BIOS_DEBUG, "%s: Restarting training on algorithm request\n",
                        __func__);
    /* Reset frequency to minimum */
    for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
        struct DCTStatStruc *pDCTstat;
        pDCTstat = pDCTstatA + Node;
        if (pDCTstat->NodePresent) {
            uint8_t original_target_freq = pDCTstat->TargetFreq;
            uint8_t original_auto_speed = pDCTstat->DIMMAutoSpeed;
            pDCTstat->TargetFreq = mhz_to_memclk_config(mctGet_NVbits(NV_MIN_MEMCLK));
            pDCTstat->Speed = pDCTstat->DIMMAutoSpeed = pDCTstat->TargetFreq;
            SetTargetFreq(pMCTstat, pDCTstatA, Node);
            pDCTstat->TargetFreq = original_target_freq;
            pDCTstat->DIMMAutoSpeed = original_auto_speed;
        }
    }
    /* Apply any DIMM timing changes */
    for (Node = 0; Node < MAX_NODES_SUPPORTED; Node++) {
        struct DCTStatStruc *pDCTstat;
        pDCTstat = pDCTstatA + Node;
        if (pDCTstat->NodePresent) {
            AutoCycTiming_D(pMCTstat, pDCTstat, 0);
            if (!pDCTstat->GangedMode)
                if (pDCTstat->DIMMValidDCT[1] > 0)
                    AutoCycTiming_D(pMCTstat, pDCTstat, 1);
        }
    }
    goto retry_dqs_training_and_levelization;
}
```

Listing L.46: Error detection and retry mechanism during DQS training, extract from the `DQSTiming_D` function in `src/northbridge/amd/amdmct/mct_ddr3/mct_d.c`

```
if (Pass == SecondPass) {
    if (pDCTstat->TargetFreq > mhz_to_memclk_config(mctGet_NVbits(NV_MIN_MEMCLK))) {
        uint8_t global_phy_training_status = 0;
        final_target_freq = pDCTstat->TargetFreq;

        while (pDCTstat->Speed != final_target_freq) {
            if (is_fam15h())
                pDCTstat->TargetFreq =
                    fam15h_next_highest_memclk_freq(pDCTstat->Speed);
            else
                pDCTstat->TargetFreq = final_target_freq;
            SetTargetFreq(pMCTstat, pDCTstatA, Node);
            timeout = 0;
            do {
                status = 0;
                timeout++;
                status |= PhyWLPass2(pMCTstat, pDCTstat, 0,
                                     (pDCTstat->TargetFreq == final_target_freq));
                status |= PhyWLPass2(pMCTstat, pDCTstat, 1,
                                     (pDCTstat->TargetFreq == final_target_freq));
                if (status)
                    printk(BIOS_INFO,
                        "%s: Retrying write levelling due to invalid value(s) "
                        "detected in last phase\n",
                        __func__);
            } while (status && (timeout < 8));
            global_phy_training_status |= status;
        }

        pDCTstat->TargetFreq = final_target_freq;

        if (global_phy_training_status)
            printk(BIOS_WARNING,
                "%s: Uncorrectable invalid value(s) detected in second phase of "
                "write levelling; "
                "continuing but system may be unstable!\n",
                __func__);

        uint8_t dct;
        for (dct = 0; dct < 2; dct++) {
            sDCTStruct *pDCTData = pDCTstat->C_DCTPtr[dct];
            memcpy(pDCTData->WLGrossDelayFinalPass,
                   pDCTData->WLGrossDelayPrevPass,
                   sizeof(pDCTData->WLGrossDelayPrevPass));
            memcpy(pDCTData->WLFineDelayFinalPass,
                   pDCTData->WLFineDelayPrevPass,
                   sizeof(pDCTData->WLFineDelayPrevPass));
            pDCTData->WLCriticalGrossDelayFinalPass =
                pDCTData->WLCriticalGrossDelayPrevPass;
        }
    }
}
```

Listing L.47: Write Leveling (second pass), extract from the `WriteLevelization_HW` function in `src/northbridge/amd/amdmct/mct_ddr3/mcthwl.c`.

```c
uint8_t MaxDimmsInstallable = mctGet_NVbits(NV_MAX_DIMMS_PER_CH);

if (pDCTstat->Status & (1 << SB_Registered)) {
    if (package_type == PT_GR) {
        // Socket G34: Fam15h BKDG v3.14 Table 99
        if (MaxDimmsInstallable == 1) {
            if (channel == 0)
                seed = 0x43;
            else if (channel == 1)
                seed = 0x3f;
            else if (channel == 2)
                seed = 0x3a;
            else if (channel == 3)
                seed = 0x35;
        }
        ...
    }
    ...
} else if (pDCTstat->Status & (1 << SB_LoadReduced)) {
    // Load Reduced DIMM configuration
    if (package_type == PT_GR) {
        // Socket G34: Fam15h BKDG v3.14 Table 99
        if (MaxDimmsInstallable == 1) {
            if (channel == 0)
                seed = 0x123;
            ...
        }
    }
}
```

Listing L.48: Seed generation for DQS receiver enable training based on DIMM type and configuration, extract from `fam15_receiver_enable_training_seed` function in `src/northbridge/amd/amdmct/mct_ddr3/mctsrc.c`

# GNU General Public License version 2



### Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

## Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public 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.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

### No Warranty

11. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

12. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

### End of Terms and Conditions

## Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.
To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> one line to give the program's name and a brief idea of what it does.
> Copyright (C) yyyy name of author
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Also add information on how to contact you by electronic and paper mail.
If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) yyyy name of author
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
> This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.
You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program
> 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> signature of Ty Coon, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

# GNU Free Documentation License

Version 1.3, 3 November 2008


<https://fsf.org/>



### 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.