UPPSALA
UNIVERSITET

UPTEC F11 064
Examensarbete 30 hp
December 2011

Controlling the Bootstrap Process

Firmware Alternatives for an x86 Embedded Platform

Svante Ekholm Lindahl

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Controlling the Bootstrap Process: Firmware Alternatives for an x86 Embedded Platform

Svante Ekholm Lindahl

The viability of firmware engineering on a lower-tier computer manufacturer (OEM) level, where the OEM receives processor and chipset components second hand, was investigated. It was believed that safer and more reliable operation of an embedded system would be achieved if system startup times were minimised. Theoretical knowledge of firmware engineering, methods and standards for the x86 platform was compiled and evaluated. The practical aspects of firmware engineering were investigated through the construction of an open source boot loader for a rugged, closed-box embedded x86 Intel system using Coreboot and Seabios. The boot loader was compared with the original firmware and the startup times were found to be reduced ninefold from entry vector to operating system handover.

Firmware engineering was found to be a complex field stretching from computer science to electrical engineering. Firmware development on a lower-tier OEM level was found to be possible, provided that the proper documentation could be obtained. To this end, the boot loader prototype was proof of concept. This allowed an alternative, open-source oriented model for firmware development to be proposed. Ultimately, each product use case needed to be individually evaluated in terms of requirements, cost and ideology.

Handledare: Per-Oskar Andersson
Ämnesgranskare: Justin Pearson
Examinator: Tomas Nyberg
ISSN: 1401-5757, UPTEC F11 064
Sponsor: CrossControl AB

Summary in Swedish – Sammanfattning på svenska

Fast programvara (eng: *firmware*) ligger i gränslandet mellan hårdvara och mjukvara. Det är den inbyggda samlingen instruktioner som initialiserar hårdvaran till ett fungerande tillstånd. Då inbyggda system och applikationer blir allt vanligare i industrin börjar kunder uttrycka allt fler preferenser och krav. Eftersom ett system är otillgängligt medan det startar om är förmågan att starta snabbt viktig för produktens säkerhet och tillförlitlighet. Datortillverkare har sällan någon nämnvärd kontroll över den fasta programvaran som levereras med processorn och moderkortskretsarna. Vissa privilegierade tillverkare kan köpa eller licensiera fast programvara från fristående leverantörer. Större kontroll över den fasta programvaran för en tillverkare som får kretsarna i andra hand skulle medföra en eftertraktad förmåga att vidare kunna anpassa den efter produkten.

Detta arbete undersökte huruvida egenhändig utveckling av fast programvara var möjlig för en sådan datortillverkare och i så fall under vilka förutsättningar. Dels insamlades och utvärderades kunskap om rådande och framträdande standarder för fast programvara på x86-plattformen. Alternativ med både öppen och stängd källkod undersöktes. Dels undersöktes praktiskt kring området fast programvaruteknik. Prototyper av starthanterare utvecklades och utvärderades.

Fast programvaruteknik hindras i modern tid av begränsningar som den 30 år gamla *Basic Input/Output System*-standarden (BIOS) på x86 plattformen medför. Flertalet lösningar har föreslagits och två av dessa undersöktes. *Unified extensible firmware interface* (UEFI) är en gränssnittsstandard som förväntas ersätta BIOS-standarden i moderna datorsystem. Å andra sidan är *Coreboot* ett projekt med öppen uppstartskod som kan kopplas ihop med paketlösningar. *Seabios* är en paketlösning som implementerar BIOS-standarden.

En starthanterare baserad på öppen källkod implementerades och konfigurerades för målsystemet *CCpilot XL*, ett robust datorsystem med låsta komponenter. Starthanteraren utnyttjade både Coreboot och Seabios. CCpilot XL utnyttjade ett dator-på-modulsystem (COM), där en Intel Core 2 Duo-processor, en Intel `945GME`-nordbrygga och en Intel `ICH7M`-sydbrygga fanns på själva COM-modulen. Dessa komponenter behövde, tillsammans med flertalet hårdvaru- och gränssnittskretsar, konfigureras av starthanteraren. Programmering av den integrerade minneskrets som innehöll den fasta programvaran utfördes genom trådlödning direkt mot kortet. Intels utvecklingsverktyg för starthanterare, *boot loader development kit* (BLDK), var ytterligare ett alternativ som var tänkt att utvärderas. För detta krävdes en annan modul, men programmering av minneskretsen på denna andra modul misslyckades. Den första modulens uppstartstid uppmättes. I experimentuppställningen styrde ett primärt system ett sekundärt system och noterade tiden för mottagna felsökningsmeddelanden. Den ursprungliga fasta programvaran jämfördes med starthanteraren baserad på Coreboot+Seabios. Resultaten indikerade en minskning av den genomsnittliga uppstartstiden på cirka nio gånger.

Undersökningen fastställde att fast programvaruutveckling var ett komplext vetenskapsområde som sträcker sig ända från teoretisk datavetenskap till praktisk elektroteknik. Utveckling av fast programvara vara ansågs vara möjlig för en datortillverkare som köpte moderkortskretsar i andra hand under om hårdvarudokumentation fanns tillgänglig. Coreboot+Seabios sågs som ett gångbart alternativ om effektiva starthanterare efterfrågades, medan UEFI sågs som en lovande teknologi för flexibel, fast programvara med stor funktionalitet. Författaren föreslog även en öppen källkodsorienterad modell för fast programvaruutveckling som skulle gynna både samtliga datortillverkare och tillämpad akademisk forskning inom området.

Slutligen måste varje fall där lokal utveckling av fast programvara hos en icke-privilegierad datortillverkare utvärderas enskilt. I investeringskalkylen kommer faktorer så som krav, uppskattade kostnader och besparingar såväl som tillgängliga resurser och mjukvaruideologi att spela in.

"The process of evolving a platform from an expensive objet d'art into a fully operational computer system ... often includes superstitious rituals, cynical prayers, and lots of cussing."

–Tim Hockin
Google, Inc.
"Linux, Open Source, and System Bring-Up Tools" [1]

Summary

Firmware lies in the realm between the hardware and the software. It is the resident and initial set of instructions which bootstraps the computer into a working state where it can perform its proper function. As embedded applications are becoming more prevalent in the industry, customers are expressing ever more preferences and requirements. Because a system is offline during a reboot, the ability to power up a device quickly makes a product more reliable. Computer manufacturers (OEMs) do not have much control over the firmware shipped with the processors and chipsets. Higher-tier OEMs purchase or license the firmware from independent BIOS vendors. Controlling the firmware would empower the OEM to further customise the product.

This text investigated whether or not x86 platform firmware development on a lower-tier OEM was feasible and, if so, what the prerequisites were. Two avenues of applied research were pursued in parallel. First, the author aimed to compile and evaluate knowledge of current and emerging x86 platform firmware standards. Open and closed-source, free and cost-associated firmware alternatives were identified and evaluated. Secondly, the field of firmware engineering was investigated with the construction and deployment of prototype boot loaders.

Modern firmware engineering was hindered by the shortcomings of the 30 year-old *legacy basic input/output system* (BIOS) de facto standard. A number of solutions have been proposed and two are investigated in this text. The *unified extensible firmware interface* (UEFI) is a standard aimed to replace the legacy BIOS in modern computers. On the other hand, *Coreboot* is an open source bootstrap code project which can be coupled with a variety of payloads. *Seabios* is such a payload which implements the legacy BIOS interface.

The selected target hardware for the open source boot loader was the *CCpilot XL*, a rugged, closed-box x86 computer-on-module (COM) system. A boot loader utilising Coreboot Seabios was then constructed and configured. The COM module featured an Intel Core 2 Duo processor, Intel `945GME` northbridge and Intel `ICH7M` southbridge. These components, along with various hardware controller interfaces, needed to be initialised and configured correctly by the boot loader. Programming of the boot ROM, the integrated circuit holding the firmware, was successfully performed through in-system programming, a process where wires are soldered directly to the board. An alternative, the Intel *boot loader development kit* (BLDK), was to be evaluated and used with another, newer COM module. However, programming of the boot ROM on this module was ultimately unsuccessful.

The module initialisation time was measured. In the experimental setup, a master system controlled a slave system and timed debug messages from it. The original firmware, a legacy BIOS, was compared to the *Coreboot+Seabios* boot loader. Results indicated a reduction of the mean module initialisation time on the order of nine times.

Firmware engineering was found to be a complex field that scaled from heavy, theoretical aspects of computer science down to the finer points of hands-on, electrical engineering. If the proper documentation of the hardware can be obtained, then firmware development on a lower-tier OEM level is possible. *Coreboot+Seabios* was found to be a viable option for efficient boot loaders, while *UEFI* is a promising technology standard for flexible, full-featured firmware. The author also proposed and discusssed an open source-oriented firmware engineering ecosystem model which would benefit both lower-tier OEM firmware development and applied firmware research in academia.

Ultimately, each situation where in-house firmware development on a lower-tier OEM level is considered must be individually evaluated. Product and firmware requirements, estimated costs and cost savings as well as available resources and the software ideology will factor into the investment appraisal equation.

Contents

List of Figures

List of Tables

Acknowledgements

Many people have supported the author and the project. Above all, the author would like to thank the CrossControl AB staff: Per-Oskar Andersson provided in-house support and guidance, and Olov Hisved and Johan Strandberg allocated many necessary resources. Alf Luong, Magnus PE Olsson and also Pierre Winberg lended their soldering skills on more than one occation. The author would also like to thank Justin Pearson, Uppsala University, for his unwavering engagement and good advice. Marnix Toornvliet and Michael Brunner, Kontron Embedded GmbH devoted some R&D resources to support the project and Lars Hallberg, Congatec AG showed a continued interest and provided helpful support. The Coreboot development team, with Peter Stuge and Patrick Georgi in particular, also provided superb support and helpful input.

Last, but not least, the author wishes to especially thank his fiancée and parents in law for their exceptional amount of babysitting. Without it, this project could never have bootstrapped into a thesis.

Typographical conventions

This text uses the following typographical conventions:

- Numbers mentioned are always in base ten unless otherwise specified. `0xFF` or `FFh` in typewriter font will denote a hexademical number; in this case $\mathrm{FF}_{16} = 255_{10}$.
- Product names are written with an initial capital letter followed only by lowercase letters, regardless of how their respective vendors market them: "Seabios" rather than "SeaBIOS". CrossControl products, abbreviations and only uppercase names are exceptions.
- Component names and numbers are written in typewriter font for easy recognition: "The `8259C` interrupt controller."
- Whenever an important keyword or term is introduced, it will be emphasised. If a common abbreviation exists, it will follow directly afterwards: "The *three-letter abbreviation* (TLA)".
- Binary prefixes are used: 1 kB means $2^{10} = 1024$ bytes and 1 MB means $2^{20} = 1024$ kB.

Abbreviations are frequent throughout the text. Please note that appendix B on page 59 contains a glossary of abbreviations.

Disclaimer

1 Introduction

Consider the hypothesis that a small or medium-sized x86 *original equipments manufacturer* (OEM) can control the entire bootstrap process of their own products. This text is an investigation that attempts to prove or disprove this hypothesis for a specific, real use case.

1.1 Background

Embedded applications are becoming ever more prevalent in industry [2]. As such, customers of embedded platforms are no longer simply requiring safe and reliable operation of the embedded products. Customers are also starting to express detailed preferences regarding the overall design and product finish. Even if the system is comprised of multiple of components, each with its own origin, it should present a coherent interface. Such an interface in turn increases the overall ease of use on the platform, which reduces the risk of human error in safety-critical environments [3]. The platform should not just operate efficiently and present a unified interface. The ability to power up quickly and integrate the start-up phase of the device is also important. Gupta and Palod [4] emphasise that a safety-critical system is offline during a reboot. A product that boots quickly therefore has the potential of becoming a product that is safer and more reliable.

This project was carried out in cooperation with CrossControl AB, an original equipment manufacturer that designed, constructed and sold rugged embedded computers and display interfaces, such as the one in figure 1. Application software could be delivered with the system or supplied by the customer, interfacing with the application framework provided and supported by the company.

Lower-tier OEMs often do not have control over the firmware in their own products as it is generally supplied with the system or component [5, 6]. It is seldom altered or replaced by the lower-tier OEMs as it does not have the know-how or licenses required to optimise the firmware. As this text will explain, developing and testing firmware requires specialised tools as well as access to source code, specifications and documentation. If customising the firmware is a design objective then it stands to reason that such knowledge and know-how must be obtained.

CrossControl had previously developed firmware for many of its ARM-, AVR- and FPGA-based board controllers. However, the company had not written any firmware for the x86-based computer-

Figure 1: An example closed-box, rugged x86 computer system: CCpilot XL. A key feature to firmware development is the fixed hardware configuration. The photograph is copyrighted by CrossControl AB, 2011 and used with permission.

on-modules (COMs) where the main processor was located. Firmware optimisation was limited by goodwill offered by or support services sold by the COM vendors. Having control of the firmware themselves would empower the OEM to customise the product further. Resulting abilities would range from small but important tweaks – such as easily setting factory default display settings – to implementation of fully customisable boot loaders. This text investigates whether or not firmware engineering is viable for such an OEM and, if so, under which conditions.

1.2 Goals and purpose

The project was initiated to investigate and evaluate the role of the x86 firmware (BIOS) in the company's product line. While the project focused on components used (or considered for use) in CrossControl's products, it should also be relevant to similar scenarios. Priorities originally included optimisation of the boot latencies but also to seek knowledge of and evaluate the emerging *unified extensible firmware interface* (UEFI) x86 firmware standard. It was also of concern what those choices implied and what potential problems and benefits were coupled with each solution.

It became clear early into the project that optimising the firmware currently being used in the products would not be practically possible (see section 4). The project goals were therefore revised. Focus shifted from optimisation of the firmware to evaluating current and future firmware alternatives. More specifically, the revised goals of the project were:

1. *To compile knowledge of BIOS- and UEFI-based technology, their specific applications and/or technology benefits.* Knowledge of the current hardware generation's firmware, the legacy BIOS, was required in order to suggest optimisations or evaluate alternatives.

2. *To evaluate the use of UEFI technology in CrossControl products.* The UEFI standard offered possible benefits relating to the target platform. Regardless if the use of UEFI-based firmware was deemed unnecessary, future hardware generations are likely to have UEFI-based firmware. Intel and the UEFI forum are making great efforts to push the standard onto the market. Therefore, knowledge of UEFI will be increasingly relevant in the coming years.

3. *To improve user experience in CrossControl embedded products, for example by showing a splash screen during system boot.* Vertical integration of the firmware with the software would provide a more coherent interface to the user.

4. *To implement a working version of the BIOS and / or UEFI that was measurably faster than the original implementation on the selected platform.* The new firmware was to be either an optimised version of the original firmware or an entirely different firmware that equivalently initialised the hardware in a more efficient manner.

5. *To investigate and implement further enhancements and features.* If time allowed, to investigate the available options on several platforms and highlight differences in the process and results.

As the project progressed and as the author gained an overview, two perspectives naturally emerged. On one hand, from a *business perspective*, the options were evaluated both from their technological properties and their business viability, as there is no point in recommending a superior solution if it cannot be used in practice. There is, however, always a possibility that a solution might become viable for business at some point in the future. On the other hand, from *an academic perspective*,

the application of open source solutions to real world problems were of special academic interest during the project. Comparisons between open source and commercial alternatives further explores this interesting avenue of research throughout the text.

1.3 Limitations and range of validity

Limitations were carefully selected to help ensure project completion within a 20 week (single term) period. Imposing these limitations naturally limits the range of validity of the results, as well as the applicability of the conclusions drawn. The imposed limitations were:

- The embedded systems industry offers a plethora of hardware. This project did not aim to be an exhaustive survey of every option available. Such an undertaking would likely have proved overwhelming. The text should instead be regarded as a description or a snapshot of a real-world problem where proprietary and open source alternatives were considered.
- Boot time is often taken as the time from power on until the user can start using the machine. In this project, no attention was given to shortening the operating system (OS) load time when making suggestions for boot time optimisation. Whether or not the firmware changes caused any changes in operating system behaviour would still be noted, though.
- Only a *select few* versions of alternative boot loaders were compared on similar hardware. Originally, the limitation was to only implement a *single* optimised version of the original firmware on the selected target hardware. When the project goals were revised, this limitation followed.
- This text mostly explores *non-commercial* alternatives. All closed source solutions described herein are provided free of charge as a service to customers. Figure 2 highlights the difference between commercial and non-commercial alternatives. Having control of the firmware configuration and build process or not is the general divider between the non-commercial and commercial solutions.
- *Security* and *trust* concerns were not primarily considered.

	Non-Commercial	Commercial
Proprietary	Intel Boot Loader Development Kit (BLDK)	Vendor BIOS solutions
Open source	Coreboot+Seabios, Coreboot+Filo	Open source consultants

Figure 2: Classification of some available commercial and non-commercial firmware alternatives.

1.4 Literature review

In order to base the text on a sound scientific foundation, a systematic literature search was conducted. The advantages of such a search were twofold. First, it provided a large number of relevant articles, proceedings, books, etc. Secondly, it provided an overview of much related work. For the search, five trusted, managed databases (IEEExplore, ACM Digital Library, Web of Knowledge, SpringerLink and ScienceDirect) were queried with specific keywords. Google Scholar was also used. Though not a database managed by editors by itself, it often returns hits from other managed databases.

Keywords were selected to encompass as many relevant hits as possible. A selection by headlines then preceded an elimination by abstract and/or contents. Statistics from the search is shown in table 1. Note that some of the articles appeared in multiple databases and/or searches. Duplicate entries were excluded during the elimination stages. Unfortunately, access to some of the relevant papers was also prohibited.

The systematic search complemented a less scientific approach. This second search was characterised by following interesting hyperlinks and found much unpublicised information and industrial standard documentation.

1.5 Related work

By now, firmware is a well-known subject. Recent research focuses on novel requirements or firmware capabilities, such as the works of Zheng et al. [7] on multi-bootpath firmware or Marchiori and Han [8] regarding firmware for wireless sensor networks. Efficient boot loader design for embedded systems is a subject to active research and hundreds of papers have published in recent years for a variety of (non-x86) architectures. For example, Xu and Piao [9] details the reliable development of a RISC-architecture boot loader. Wan et al. [10] and Qingtian et al. [11] are examples of the many ARM-architecture based boot loaders described.

On the x86 (Intel) architecture, firmware development is driven by large vendors (Intel, AMD, AMI, Phoenix, etc.). Material regarding firmware and boot loaders for the x86 platform therefore mostly consists of corporate white papers, industry standards, processor and chipset specifications – some of which are subject to industrial secrecy restrictions. Recent (and public) work relating to boot time optimisation include Doran et al. [12], Kartoz et al. [13] and Rothman [14].

Prior to Intel's push of the *extensible firmware interface* (EFI) around 2004 [15], few papers are relevant. Ahmed and Farook [16] (1988) are pioneers in high-level language firmware for the x86 platform, while Sibert, O. and Porras, P.A. and Lindell, R. [17] (1996) focus on legacy BIOS security. Post EFI, academicians have investigated how the EFI runtime environment can be leveraged in new ways. For instance, Jiang et al. [18] and Wang et al. [19] focus on graphical user interfaces, while Feng et al. [20] incorporate the EFI interface with another platform model on the Intel IA-64 processor. Much research has also been put into the concept of trusted computing in various firmware contexts, such as the works of Zhang et al. [21] and Fang et al. [22].

The third player in the arena is the open source community. Projects include Coreboot [23], Open Firmware [24] and Das U-Boot [25]. While open source solutions might be avant-garde, little material is publicised by conventional means. Knowledge is often pooled in repositories (wikis, mailing lists) but is seldom complete or peer-reviewed.

To close up this section, Gupta and Palod [4] is mentioned. In their 2009 master's thesis, they face similar problems as this text. They focus on GNU/Linux optimisation on the assumption that *"there is little we can do about* [the] *BIOS step"*. This text investigates the validity of such assumptions.

Query: **firmware** and **x86**

	Initial hits	*After headline elimination*	*After abstract elimination*
IEEExplore	4	1	0
Google Scholar	1840	10	6
ACM Digital Library	145	1	1
Web of Knowledge	13	3	2
SpringerLink	77	3	1
ScienceDirect	127	1	0
Total	2736	19	**10**

(a)

Query: **"boot loader"** or **bootloader**

	Initial hits	*After headline elimination*	*After abstract elimination*
IEEExplore	56	11	5
Google Scholar	1160	6	1
ACM Digital Library	577	4	1
Web of Knowledge	25	0	0
SpringerLink	11	1	1
ScienceDirect	220	1	0
Total	2049	23	**8**

(b)

Query: **Intel** and **firmware**

	Initial hits	*After headline elimination*	*After abstract elimination*
IEEExplore	23	2	1
Google Scholar	5630	12	5
ACM Digital Library	574	2	1
Web of Knowledge	12	2	1
SpringerLink	288	5	0
ScienceDirect	774	3	0
Total	7301	26	**8**

(c)

Table 1: (a) Systematic literature search and elimination for keywords firmware *and* x86*. The Google Scholar search was limited further by eliminating the words* patent *and* virtualization*. (b) Systematic literature search and elimination for keywords* boot loader *OR* bootloader*. The Google Scholar search was limited further by eliminating the words* arm*,* mips*,* patent risc*, and* virtualization*.(c) Systematic literature search and elimination for keywords* Intel *OR* firmware*. The Google Scholar search was limited further by eliminating the words* arm*,* mips*,* patent risc*, and* virtualization*.*

1.6 Disposition

This chapter has introduced the reader to the subject and problem, presented the project goals and limitations and described related work and literature. The remainder of this text is structured as follows:

- Chapter 2 is a large body of compiled background material and theory:
  - First, the purpose and operation of firmware on the x86 platform is described. The operation of common protocols, such as *peripheral component interconnect* (PCI) interrupt routing, the video driver and the purpose of *advanced control and power interface* (ACPI) are given special attention. De facto and new emerging standards, such as the legacy BIOS and the UEFI are explained in detail. An open source alternative (Coreboot+Seabios) to the prominent standards is presented and its function and properties are explained, as well as alternatives.
  - Secondly, a summary of some BIOS optimisation tips and tricks is given.
  - Last, some closed source tools for firmware development are described; the Intel *boot loader development kit* (BLDK) for developing minimalistic UEFI-based firmware, the Intel *embedded graphics driver* (IEGD) toolkit for developing video drivers for Intel chipsets, and the *Congatec utility* for modifying a Congatec legacy BIOS binary.
- Chapter 3 describes the set of hardware for which boot loader prototypes were considered. Some development boards used are also given short descriptions.
- Chapter 4 details the methods and processes adhered to during project planning, execution and finalisation. Design decisions are specified. It also details the steps carried out in order to develop, test and compare some different alternatives on the target hardware. This work was carried out in cooperation with CrossControl AB. The experimental procedures and setups for measuring boot times are also described in this section.
- The results of the investigation are presented in chapter 5. This includes both the presentation of measurements from experiments but also a summary of constructed and attempted prototypes and their properties.
- Interpretations and discussions of the results are contained within chapter 6. Design decisions are defended but the methodology is also subject to self-criticism. After concluding the discussion, conclusions are drawn. The final part consists of recommendations for CrossControl AB as well as recommendations for future research.

2 Theory

This section presents knowledge and know-how compiled and during the course of the project and becomes the foundation for subsequent chapters. Some of the firmware, hardware and software standards are presented here. Some available tools are also described. The list is not meant to be exhaustive. Focus is placed on the protocols and interfaces that are most relevant to firmware development on the x86 architecture and to the work detailed in chapter 4. It is assumed that the reader is already familiar with basic computer system structures and operation. If not, chapter 2 of Silberschatz et al. [26] is recommended.

2.1 Firmware basics

Firmware is the initial set of instructions in a computer or piece of hardware that *bootstraps* or boots the hardware. That is, it initialises the system into a working state where it can perform its proper function. During the late sixties and throughout the seventies, firmware was simply the microcode that implemented the instruction set for a processor architecture. Davidson and Shriver [27] describe firmware in their 1978 paper as *"software in the read-only control store"*. Currently, the definition encompasses a broad range of near-hardware software configurations. For a few examples, see Rosch [5], Tanembaum [6], Catsoulis [28], Tolentino [29], Sally [30] and Zimmer et al. [31].

When power is turned on to the hardware, instructions are executed starting at a predetermined memory address (the so-called *entry vector*). The entry vector usually contains a `jmp` (jump) instruction to the first code block. The firmware program might perform a *power on self test* (POST), where the basic functions of the hardware are discovered and tested. It will explore the hardware configuration and possibly initialise resources and services (device drivers). Ultimately, the firmware may hand over control of the hardware to software located in the newly initialised secondary memory, such as an operating system. Especially for lightweight embedded systems, the firmware itself instead assumes the role of operating system or default application. In figure 3, the author contrasts the layered structure of a general-purpose computer with an embedded system. This figure moves to show that firmware engineering is not the same as embedded software engineering, where the user writes applications for an embedded system, possibly with a *real-time operating system* (RTOS) underneath. Firmware engineering concerns development of software that interfaces directly with the hardware.

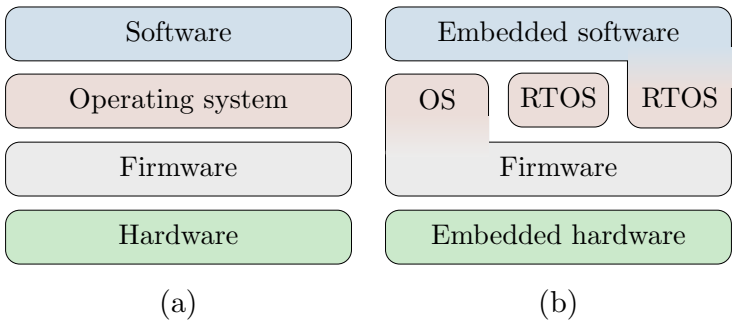


Figure 3: (a) Classical structured computer organisation. (b) In an embedded system, the layer stack might look slightly different. The firmware might assume the role of the operating system *(OS) or might be an independent layer. The embedded system might run an OS or* real-time OS *(RTOS), or the OS functionality is integrated into the embedded application.*

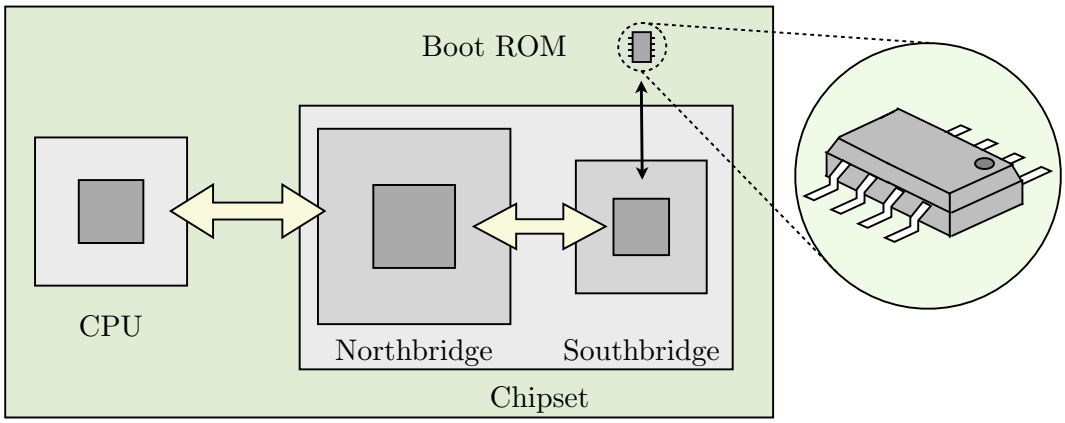


Figure 4: Simplified schematic of firmware stored in a flash memory small outline integrated circuit (SOIC) connected to the southbridge on the main circuit board. Arrows indicate interconnecting communication buses.

The firmware is stored in a non-volatile memory chip which simplified is located close to the *central processing unit* (CPU), as shown in figure 4 [30]. While firmware used to be programmed in *read-only memory* (ROM) chips, the advent of flash memory in *small outline integrated circuits* (SOIC) and similar form factors has enabled firmware reprogramming. Communication protocol standards like inter-integrated circuit (I^2C), *serial peripheral interface* bus (SPI) and *firmware hub* (FWH) are currently supported by SOIC manufacturers. The firmware can be then updated both by means of software and by means of external tools that are connected directly to the SOIC pins. Note that if the boot ROM (firmware image) is corrupt or otherwise fails to boot the device, an external tool must be used to restore the image to a working state.

2.1.1 Developing and debugging firmware

Firmware engineering is similar to embedded software development in two regards. First, the code deals with limited system resources and tends to be written in low-level languages such as C or assembly language. Secondly, the program is compiled on a *host* machine but executed on another *target* machine. Ganssle [32] emphasises that the code isn't magically transferred between these machines. For this reason, there is a number of hardware tools and peripherals the firmware developer must be familiar with. These tools include the external programming tool, the logic analyser, the voltmeter and the oscilloscope. They are not further explained in this text, but the reader should know of their existence.

Firmware code is inescapably hardware specific. That is, its code must be written to work with a specific configuration since the component setup and architecture instruction set differ between platforms. There is also a variety of chipsets and expansion buses, etc. Whenever new hardware components are constructed, the firmware must be ported or adapted to support the new configuration. If the new hardware is similar to the old then it is possible that the amount of work required is small. Zimmer et al. [33] note that approximately 95% of the source code of an x86 legacy BIOS can be reused for a new processor-chipset configuration. Part of the code is also universal for a specific architecture.

Rosch [5] notes that firmware traditionally is written in assembly language. With some basic chipset functionality and high memory initialised, the remaining firmware code can be written in a

higher level language. A special compiler can set up a stack and enable the code to run using only the registers in the CPU [34]. Furthermore, if a cache memory is available then it could be used as a scratchpad until high memory has been initialised [31].

Davidson and Shriver [27] note that debugging firmware code is difficult due to its very nature. Before the chipset initialisation sequence is complete, the programmer has no means of gaining information of the system state. This hurdle can partly be overcome by means of virtualisation – that is, by running the firmware on a simulated machine. Tools such as VirtualBox [35] and QEMU [36] can provide this environment. If breakpoints can be inserted into the firmware code, then the machine state, such as register values and memory contents, can be examined in detail.

Firmware often provides debugging information after the chipset has been initialised. For instance, legacy BIOS uses POST codes which are small messages that are output on *input/output* (I/O) adresses `0x80` and `0x81`. Some mainboards and add-on cards provide a translation function that outputs these messages on the serial port on I/O adress `0x3F8`, while others provide 7-segment LED displays for displaying POST codes. Some firmware types output more elaborate debug messages directly to the serial port. For example, Coreboot (described in section 2.1.6) outputs messages in clear text while Intel's UEFI implemention outputs messages compatible with the Microsoft Windows Debugger (WinDbg) [37]. This information enables for most of the real bootstrap process to be debugged and timed, albeit in a crude way.

2.1.2 Classification and terminology

Jensen and Hattaway [38] split firmware or boot ROMs into two important types; multi-purpose firmware (called *BIOS*) and *boot loaders*. A BIOS provides a feature rich environment with multiple boot paths. Such an implementation is more flexible than a boot loader; it supports more configurations but has a larger memory space footprint. Reconfiguring a BIOS can be done at runtime and a plethora of options might be available. On the other hand, a boot loader is scaled-down firmware, possibly optimised for a specific (often embedded) setup. Configuration is done at compile time and the task is to boot the target system as efficiently as possible.

The term *boot loader* has many definitions. Common to all definitions is that a boot loader is known as a small piece of software that loads an operating system. Sometimes, this piece of software is located in the secondary storage and enables multiple operating systems to be installed there. It could also help load the kernel directly into memory. Such a boot loader is called a *second stage* boot loader. *Grand unified bootloader* (GRUB) and *Linux loader* (LILO) are examples of common second stage boot loaders. This text is only concerned with *first stage* boot loaders, which initialise hardware prior to passing control to an operating system or a second stage boot loader.

2.1.3 Common x86 interfaces

The remainder of this text will concern only the Intel x86 architecture of the target hardware. The CISC-based x86 architecture is extremely popular and is described in detail in many works such as Tanembaum [6]. It will therefore not be explained in general here. However, short introductions to some common x86 standards and interfaces are required. These include *peripheral component interconnect* (PCI) interrupt routing, the *advanced control and power interface* (ACPI) and a short mentioning of host controller interfaces.

Protected mode x86 interrupt handling

The x86 CPU uses hardware interrupts to signal different system events. In the 32-bit protected mode, the CPU will keep an *interrupt descriptor table* (IDT) to keep track of all the installed *interrupt handlers*. The argument supplied with the `INT` instruction is the corresponding IDT entry. The IDT thus links the IDT entry (the *interrupt vector*) to the proper device driver. For example, the instruction `INT 10h` will cause a software interrupt. It invokes the handler linked to at entry `10h` in the IDT, the entry for the video driver. The `AX` register is used to specify the type of request and the `BX` and `CX` registers can be used to pass data parameters. There are many types of interrupts, including software interrupts, exceptions, and device interrupts caused by devices on the I/O bus.

PCI interrupt routing

The conventional *peripheral component interconnect* (PCI) I/O bus connect different parts of the modern x86 system into a working system. The standard does not just cover a communications protocol, is a widely used entire I/O signaling bus and describes the various physical and electrical characteristics of the PCI hardware. For a more detailed description of the PCI bus than the one given here, the textbook by Shanley and Anderson [39] is a detailed companion to the conventional PCI standard [40]. However, Baldwin [41] eminently explains the more select parts of x86 and PCI interrupt routing that are needed for firmware development. Interrupt management for *PCI express*, the successor to PCI, is mentioned on page 12.

To signal the CPU that a PCI peripheral (PCI device) needs attention, the device must utilise an *interrupt line*. The conventional PCI standard defines four interrupt lines, `INT A#` through `INT D#`. The first (and often only) function of a PCI device utilises the first line, `INT A#`. The destination for the *interrupt request* (IRQ) is the proper interrupt handler in a device driver, where an *interrupt service routine* (ISR) resolves the IRQ through some appropriate action.

All PCI device interrupt lines must be mapped to the interrupt handlers in some manner. *Interrupt controllers* exist between the devices and the CPU to facilitate this mapping. In addition, the PCI devices are physically mapped to the interrupt controllers in some configuration. This mapping can be very different from system to system. The key point here is that the firmware *must* know the corresponding logical mapping in order to correctly program the controllers [39, p. 231], or devices on the PCI bus might not function properly. There are two common interrupt controllers, called the PIC and the I/O APIC:

1. Dual `8259A` **PIC**s, or *programmable interrupt controllers*, were used in the original IBM PC with the *industrial standard architecture* (ISA) bus, a forerunner of PCI. Figure 5 (a) shows that each PIC holds eight interrupt inputs, but the output of the slave was chained to one of the inputs of the master. It is possible to chain eight slave PICs for a total of nine PICs and 64 interrupt lines [6, p. 189]. The single slave scheme became the de facto standard for interrupt routing, though, and persisted into the PCI era. As ISA did not allow IRQ sharing, most of the 15 inputs were used up by standard devices (printer, floppy, COM-ports, keyboard, etc.), leaving only 4 unused inputs for add-on cards to be used for PCI interrupt routing. Groups of PCI devices are routed through a *programmable interrupt router* (PIR)[1] to these inputs, where the members of a group share the same IRQ line. The number of inputs and outputs on PIRs varies among implementations.

[1] Note that PIR is an abbreviation for both *programmable interrupt router* and *PCI interrupt routing*.

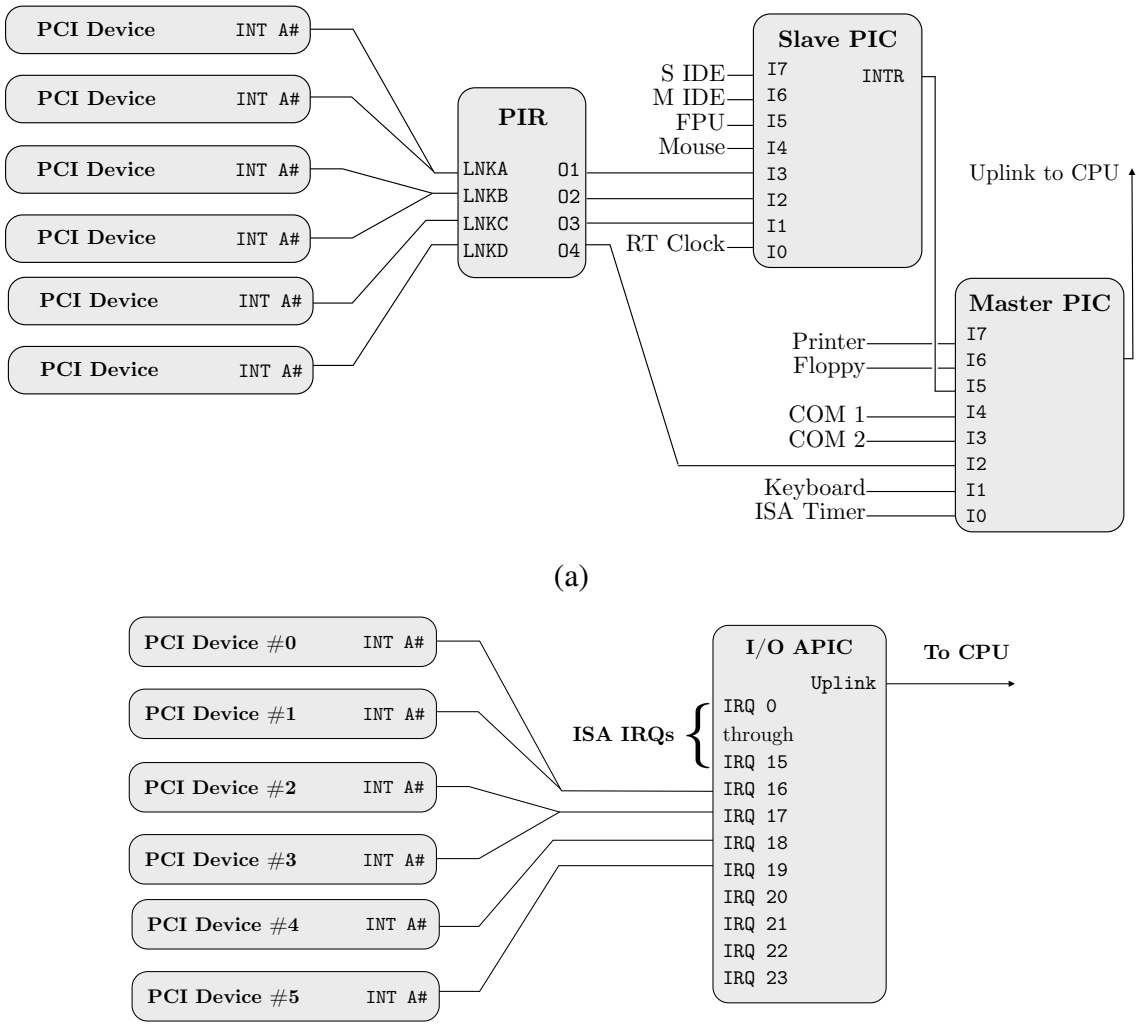


Figure 5: Example PCI interrupt routing. Real implementations vary between different hardware. (a) Routing with an older programmable interrupt router (PIR) and legacy `8259A` programmable interrupt controllers (PICs) [39]. (b) Routing with an advanced programmable interrupt controller (I/O APIC) [41].

2. An **I/O APIC** or *advanced* PIC moves the complexity into the controller implementation. The Intel APIC standard was developed to replace the legacy-encumbered PIC. An I/O APIC is often used in multi-processor systems and will interrupt only one of the processors. Figure 5 (b) shows that an I/O APIC is not restricted to 8 lines and it can hold an larger number (16, 24 or 32) of IRQ lines. Often the first 16 IRQ lines are programmed in the ISA manner and are then called ISA IRQs. The I/O APIC can also be used in conjunction with the legacy PICs.

The uplink to the CPU consists of several data lines, which the CPU not only uses to deassert interrupts after the ISR has finished, but also to update the status and registers of the interrupt controllers [6, p. 188]. In this way, the controllers can be programmed by the firmware, which can provide two tables.

			Pin:	Link:	Bitmap:
Bus:	Device	Type:	INT A#	0x60	0x1E39
1	0	Embedded	INT B#	0x61	0x1E39
			INT C#	0x62	0x1E39
			INT D#	0x63	0x1E39

Table 2: Example `$PIR` table entry. Here, each PCI `INTx#` is routed to a link and the bitmap shows the valid ISA IRQs for the interrupt – `0x1E39` corresponds to IRQs 3, 4, 5, 6, 9, 10, 11 and 15. The `$PIR` table contains an entry for each PCI device and slot attached to the dual PICs or *to the first 16 entries of an I/O APIC. [41]*

Type	Bus	Device	INT pin	Pin	APIC ID
INT	1	6	INTA#	15	0

Table 3: Simplified MP table entry for the fictional PCI device 1.6, whose `INTA#` pin is connected to the sixteenth pin of the first I/O APIC.

1. If dual PICs and a PIR are used, then a PCI interrupt routing (`$PIR`) table is required. The `$PIR` table is named after the ASCII signature in its header. If PICs are used in conjunction with an I/O APIC, the firmware might also need to supply a `$PIR` . The table then describes how the PCI devices are mapped to the input pins of the PIR. The `$PIR` table has one entry for each PCI device, specifying the bus and device identifiers, type (embedded or slot) and a subtable for the `INT A#-INT D#` pins. In the subtable, a link number and a bitmap of valid ISA IRQs is specified for each pin. The pin-link combinations contains the actual routing information. The bitmap could just be an opaque number, though, depending on which hardware (controllers, routers) are being used. An example table entry is shown in table 3.

2. For *multi-processor* (MP) systems or systems with an I/O APIC, the somewhat simpler MP table must be supplied. This table describes the I/O APICs in the system, among other things. Each input pin on the I/O APIC that is connected to a PCI device has an entry in the table. The interrupt entry (most importantly) specifies the bus and device numbers, the `INTx#` pin used on the device, the I/O APIC identifier, and the pin number.

Routing conventional PCI interrupts with the `$PIR` and MP tables can get quite complicated as devices often are wired in cascade. `INTA#` of the second device is normally wired to the (unused) `INTB#` of the first, and so on. This is done to avoid interrupt line sharing which, though supported, slows down a conventional PCI bus. Each new piece of hardware is also likely to be wired in new ways. The manufacturer either provides tables or schematics to the firmware vendor, or inputs tables into the firmware directly.

PCI express interrupt handling

The newer *PCI express* (PCI-e) standard uses *message-signaled interrupts* (MSI) rather than dedicated interrupt lines. With MSI, the device will write to a register to assert an interrupt and supply a short message. If the message is the actual interrupt vector that otherwise would be stored in the IDT, the CPU can then jump directly to the interrupt handler. This means that no routing information is required for PCI-e.

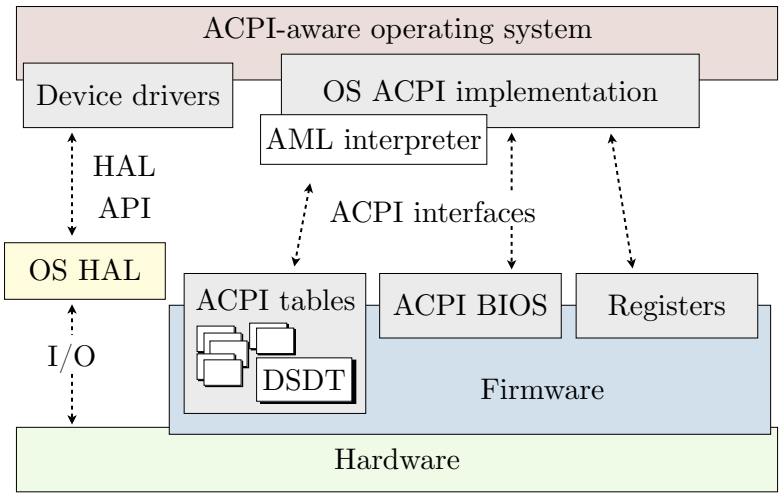


Figure 6: Simplified ACPI architecture diagram based on [42, fig. 1-1]. The interface enables transferal of system management complexity from the system firmware into the operating system. The hardware abstraction layer (HAL) and API is OS-specific.

Advanced control and power interface (ACPI)

As x86 computers grew ever more complex, the number of standards to keep track on grew larger and larger. The ACPI standard [42] unifies into a single standard a previously diverse range of standards for power management, device detection and recognition, thermal monitoring and multi-processor (MP) management, as well as PCI interrupt routing. As such, ACPI replaced both legacy Plug and Play (PnP) and the advanced power management (APM) standards.

The ACPI is an abstraction layer between the firmware and operating system that enables *operating system power management* (OSPM). It lets the operating system handle many of the things that previously needed to be handled by the firmware. The abstraction layer architecture is shown in figure 6. The *ACPI BIOS* is the part of the firmware that is compliant with the ACPI standard – note that the OS must be supplied with the ACPI tables to enable the ACPI interface. The *ACPI registers* are used for control and status purposes, as well as for events [42, p. 57]. ACPI also incorporates the `$PIR` and MP tables for convenient access by the operating system.

The ACPI tables in figure 6 enable the hardware manufacturer to describe the underlying hardware to the operating system in a coherent way across different hardware configurations. The tables are compiled from the *ACPI source language* (ASL) into *ACPI machine language* (AML). The AML is platform-independent and is parsed by an AML interpreter in the ACPI OS implementation. At minimum, two ACPI tables must be supplied by the OEM. The differentiated system description table (DSDT) is the most important table. Another important table is the *root system descriptor table* and its *pointer*, which together detail which tables exist. These are the tables the firmware must supply at minimum in the ACPI BIOS.

Host controller interfaces

A few x86-compatible host controller interfaces need to be mentioned. These controllers are often implemented as PCI devices and bridge two I/O buses. The *serial ATA* (SATA) interface for secondary storage commonly uses the *advanced host controller interface*, AHCI.

The *universal serial bus* (USB) peripheral I/O bus uses three types of controller interfaces. The *open host controller interface* (OHCI) is an open standard for USB 1.1, while the *universal host controller interface* (UHCI) is a proprietary USB 1.1 controller by Intel and often implemented in Intel chipsets. The *enhanced host controller interface* (EHCI) is a controller for the faster USB 2.0 bus.

2.1.4 Legacy BIOS

For the last 30 years, the 16-bit legacy BIOS has been the *de facto* standard firmware for the x86 computer architecture [15, 31]. The legacy BIOS has an interesting history of which Singh [43] provides a detailed, digestible account. The original *basic input/output system* (BIOS) [43] was 16-bit code written by IBM in 1981 for the Intel 8088 chip used in the IBM 5150. As IBM published the architecture and interfaces, other vendors were able to rewrite the BIOS and become the first *independent BIOS vendors* (IBVs). The legacy BIOS has remained the firmware base for the x86 platform ever since [31]. American Megatrends Inc. (AMI) and Phoenix Technologies Ltd. remain the two main suppliers of legacy BIOS code to OEMs [44].

The PC market dominance during the following three decades caused the word BIOS (incorrectly) to become descriptive of firmware in general. In this text, the sole word *BIOS* refers to an *unspecific, multi-purpose* firmware for the x86 or similar platform. The phrase *legacy BIOS* will refer to the 16-bit BIOS standard for the x86 platform.

Power on self test (POST) and option ROMs

The legacy BIOS default bootstrap procedure is described in Crouncher [44] and further specified in the BIOS Boot Specification [45]. When power is turned on to the system, the x86 CPU will start executing code at the entry vector `0xFFFFFFF0`, where a jump instruction moves the program counter to the POST code segment. This program initialises the hardware such as the memory controller, chipset and the I/O bus. The entire memory range might be also be checked. As executing code from the boot ROM directly is slow, the legacy BIOS is often copied to RAM once it has been initialised. The BIOS memory segment might be *shadowed*, that is mapped to the end of the address space and the address space artificially limited. The BIOS code memory range will then not be addressable by other applications.

The legacy BIOS then scans the entire memory range `0xC0000-0xEFFFF` in the boot ROM for the signature `0x55AA`, the trademark of an *option* or *expansion ROM* [39, p. 412]. These are are essentially basic *device drivers* that will initilise specific devices or components. Typically, if the system uses a PCI I/O bus, the POST program (or a PCI BIOS option ROM) will initialise the PCI bus by scanning for PCI devices. Some PCI devices have option ROMs which are stored in and executed from the devices themselves; a typical example is a PCI video expansion card.

The option ROMs execute in 16-bit real mode and have complete control of the system during execution. Because of this, integrating the option ROM functionality into the main BIOS code keeps the BIOS developer in control. Functionality implemented in option ROMs might therefore be integrated into the main BIOS code in later versions. Some way or the other, the legacy BIOS will initialise the required data structures such as the `$PIR` , MP table, ACPI tables, etc.

OS handover

Handing over control to the operating system is done through the *master boot record* (MBR). The search is initiated by asserting the `INT 19h` interrupt. The corresponding interrupt handler in legacy BIOS will conduct a search for bootable devices by investigating the first sector (512 kB) of all initialised secondary storage devices. The MBR table is exactly 512 kB long and located at the beginning of sector zero of the found bootable device, the so-called *boot sector* on the *boot device*. Address `0x0` in the MBR contains a small program that searches a fixed-size data structure for operating systems in the partitions of the boot device. When the operating system is found, control is turned over to it and the BIOS boostrap process is complete.

Interrupts in the legacy BIOS

After the BIOS has performed its task, it often resides in memory to handle legacy BIOS interrupt calls. The legacy BIOS uses hardware interrupts as means of providing services to the operating system. To add support for features or hardware, the legacy BIOS source code must be changed or an option ROM must be added to the firmware. Option ROMs may provide their own interrupt handlers to handle additional function calls (or they wouldn't be very useful device drivers). A linked list of interrupt handlers is then created – this is called *chaining* an interrupt (handler). The interrupt vector in the interrupt descriptor table (IDT) entry is the memory address of noted in memory by the option ROM, which places its own interrupt vector in its place. All unrecognised interrupts are then passed to the legacy BIOS handler, creating a chain of interrupt handlers. All the chained interrupt handlers are then *hooked* at the same interrupt vector (entry) in the interrupt descriptor table (IDT). The option ROM might hook its own interrupts.

While the old *disk operating system* (DOS) relied heavily on interrupt services provided by the legacy BIOS, few modern operating systems utilise these services except during booting. These operating systems replace the legacy BIOS with their own *hardware abstraction layer* (HAL). This allows the OS to have its own direct access to the hardware. Such an OS must also implement its own driver model, but is not limited to the obsolete 16-bit real mode. If the BIOS is shadowed in RAM, then the OS can usually "de-shadow" it as the default shadow BIOS memory location is known.

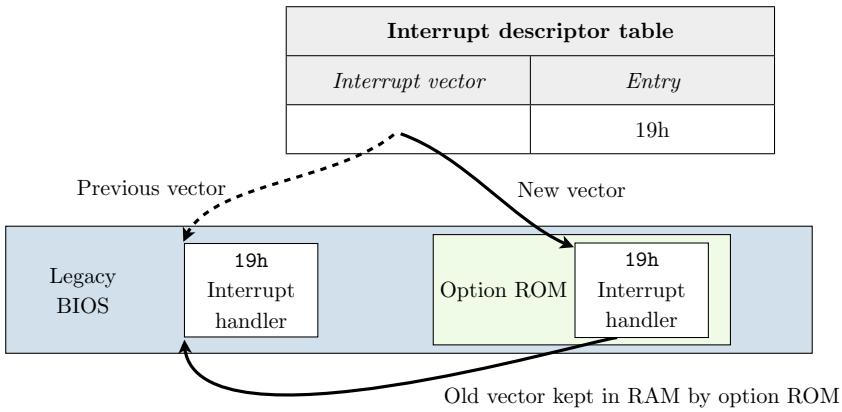


Figure 7: Expanding the legacy BIOS functionality by chaining interrupts. Here, the option ROM adds additional interrupt service routines (ISRs) to `INT 19h`, for example network device boot capabilities. If no bootable network device is found, the default ISR in the legacy BIOS is invoked.

Legacy BIOS video driver (VBIOS)

The most important option ROM in the legacy BIOS is the the video graphics array (VGA) option ROM or VBIOS. This is the video driver that enables the monitor during POST and is often run *before* POST so that the process can be followed by the user. Apart from the PCI option ROM signature `0x55AA`, the VBIOS also contains the phrase "IBM VGA COMPATIBLE BIOS" in standard ASCII encoding at offset `0x1e`. Bytes `0x44` through `0x47` in the file header tells the vendor ID and the device ID of the hardware to which the binary acts as a driver. If multiple devices are found, then the legacy BIOS determines which one it should run. As the VBIOS runs, it will hook a number of `INT 10h` interrupts, which allows the operating system to change the video settings or render graphics. However, as with the other legacy BIOS drivers, the VBIOS is often later replaced with another graphics driver provided by the operating system [44]. A modern VBIOS can initialise many different video interfaces, such as *digital visual interface* (DVI) and *low voltage differential signalling* (LVDS) – see Intel [46], for example.

Shortcomings of the legacy BIOS

Thirty years later, the 16-bit real mode legacy BIOS is still around. While it has been expanded and adapted to initialise ever more new hardware, it is facing the end of its era for several reasons.

First, the legacy BIOS consists of monolithic assembly code. Software and firmware engineering was in its cradle circa 1981, when the legacy BIOS was introduced. Davidson and Shriver [27] (1978) is an example of this. Writing long, monolithic programs in assembly language is simply put very inconvenient. The complexity of the code initialising the chipset and memory is ever increasing, making the use of assembly language impractical for the future [33, p. 50].

Secondly, the option ROMs are also restricted to 16-bit real mode. Deploying device drivers in assembly language is also inconvenient for OEMs. Giving total control of the system to the option ROM during its device or bus initialisation is bad enough, but option ROMs have also had problems regaining control at a later stage or allocating system memory safely.

Third, though assembly language has its advantages in performance over higher level languages, the 16-bit real mode severely slows down a multiprocessor 32- or 64-bit system.

Fourth, the interface to the legacy BIOS consists of real-mode interrupts and simple data structures stored at predetermined memory locations. While most operating systems provide hardware abstraction layers, this almost-static interface of the legacy BIOS prevents further vertical integration. Manageablity has also been a problem with the legacy BIOS. One would like to have better alternatives for reconfiguring or upgrading the firmware remotely, for example over a network connection.

Lastly, the legacy BIOS and its `INT 19h` OS handover call can only boot an operating system from partitions smaller than 2.2 TB [31, p. 3]. This restriction is due to the 32-bit entry for the partition size in the MBR. Disk drives are becoming ever larger and models breaking the 2.2 TB barrier now exist. This can be described as the firmware equivalent of running out of IPv4 addresses on the Internet.

2.1.5 Unified Extensible Firmware Interface (UEFI)

The UEFI standard [47] is a *firmware interface* – an abstraction layer – placed between the firmware and the operating system. It is designed to remedy most shortcomings of the legacy BIOS [15]. The interfaces defined also place requirements on how the BIOS is implemented; a UEFI compliant

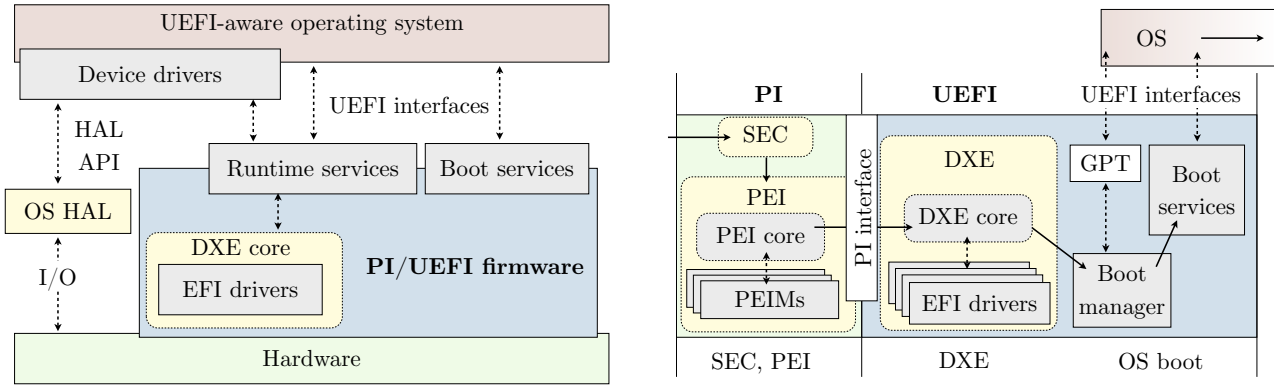


Figure 8: (a) Static UEFI architecture diagram based on [15, fig. 1]. UEFI is an interface *to which the UEFI BIOS (the implementation) conforms. The OS can interact with the UEFI BIOS runtime services or replace them with its own hardware abstraction layer (HAL). (b) Simplified dynamic UEFI architecture showing the phases during boot and order of events.*

BIOS offers an extensible, operating system-like environment with increased functionality, security, trust and portability. The original EFI 1.0 was developed by Intel Corporation in 1999 [31, p. 6]. It became the *unified* EFI or UEFI with version 2.0. The standard has since been maintained by the UEFI forum, a consortium of several leading corporations in the hardware market. The specification itself [47], currently at version 2.3.1, is open and publicly available under a distribution restriction license. It is rather lengthy. This section provides a short introduction to the UEFI architecture and concepts. Zimmer et al. [33] is a more narrative introction to EFI and Zimmer et al. [31] an in-depth description of the standard aimed at developers of device drivers – though both are written exlusively by the inventors and promoters of UEFI.

The general architecture of an UEFI-compliant x86 system is shown in figure 8 (a). Device drivers run in a multitasking environment called the driver execution environment (DXE) [33]. In this sense, an UEFI BIOS acts much like an operating system. The figure also moves to show that the OS can interact with the UEFI BIOS runtime services and utilise some firmware drivers to the hardware, or replace them with its own hardware abstraction layer. Another way to view the UEFI firmware system is from a dynamic event view, which is shown in figure 8 (b). The first two phases are described by the platform initialisation (PI) standard, while the remaining phases fall under the UEFI standard.

Platform initialisation (PI)

The platform initialisation (PI) is a companion standard to the UEFI. Whereas UEFI defines an interface between the firmware and the operating system, PI defines an interface between the basic hardware initialisation stage and the DXE environment.

The second phase in figure 8 (b) is described first. This *pre-EFI initialisation* (PEI) phase is reponsible for bringing up system resources to the extent required by the DXE runtime environment [31, p. 267]. The chipset (northbridge, southbridge, graphics controller, memory controllers) and main memory require initialisation. Each of these components are initialised by PEI modules (PEIMs), which run under a PEIM dispatcher. The dispatcher also provides multi-tasking services to the PEIMs through the PEI *services table* such as memory allocation, inter-PEIM communications

and a basic file system [31, p. 274]. Thus, a runtime environment is established very early in the boot process. This environment enables PEIMs to be written in a higher level language, such as C [31, p. 285]. Zimmer et al. [31] claim it makes firmware engineers better equipped to handle complex chipset and device functionality.

The first phase, the security phase (SEC), verifies the integrity of the various components of the following PEI phase. This phase works directly with the silicon of the CPU, similarly to a legacy BIOS (as chipset initialisation has not yet occured). The following PEI runtime environment requires a heap or stack of memory, but is also responsible for initialising the memory. Therefore, a small segment of assembly code reprograms the CPU during the SEC phase [31, p. 273]. The L1 cache memory can then be used as a scratchpad during PEI.

The DXE runtime environment

The third phase in figure 8 (b) is the *driver execution environment* phase, or DXE phase. In UEFI, the device drivers are called *EFI drivers* and *EFI applications* – or collectively as DXE drivers. DXE drivers are, like PEIMs, written in a higher level language (like C) [33, p. 53] and compiled to 32-bit x86, 64-bit x86 or 64-bit Itanium machine language. It could also be compiled into *EFI byte code* (EBC). Such programs are platform-independent and executed by an interpreter. The DXE environment is also the base for many common UEFI functions. These include but are not limited to input/output services (used for debugging) with the help of keyboards or serial ports and the UEFI network stack. With the chipset initialised, the DXE runtime environment can be (and is) more complex than the PEI environment. The DXE drivers can enable devices, buses and services [31, p. 141], including conventional PCI, PCI-e, USB, ACPI, SATA/AHCI etc. Some core services are defined as common to all UEFI systems, while other, optional drivers are added by OEMs, IBVs and hardware vendors. UEFI also allows for such programs to be run from locations *not* in the original firmware image, but also from other locations such as secondary storage or network connections. In many ways, an DXE driver is the UEFI equivalent of legacy BIOS option ROMs. The operating system-like environment also solves many of the practical problems with the legacy BIOS option ROMs. For example, the programming is done against an *application layer interface* (API) and not directly against the pure silicon of the x86 CPU. The DXE core can also be configured to dispatch only the required drivers for the boot services, leaving the rest to the operating system and reducing the time spent in firmware. Legacy option ROMs can still be run under UEFI to maintain support for legacy hardware within a UEFI BIOS, though doing so is not recommended [31, p. 312].

Internal workings of PEI/DXE handover

The internal workings of the PEI and DXE phase is shown in figure 9. Here, PI works as the interface between the bootstrap code and the following phases defined by the UEFI standard. During the PI stage, some data structures (*hand-off blocks* or HOBs) that describe the PEI modules are created. When the CPU and the on-board chips have been initialised, these contain information about the system state. The HOB list (which links to the individual HOBs) are required for passing control to the DXE phase and the initialising UEFI runtime environment.

The DXE drivers will install a number of EFI *protocols* into the UEFI *system table*. An EFI protocol is an EFI representation of a standard, such as ACPI. The representation describes the data structures and the API of the standard it implements. In fact, the DXE themselves work with a number of *architectural protocols*, the basic DXE driver API, which in turn are implemented in the

DXE. These implementations in turn work directly with the hardware. Every protocol is internally identified by its *global unique identifier* or GUID [31, p. 26] and stored in the UEFI system table. Through this model, UEFI aims to be extensible and to ease the implementation of future standards.

The Tianocore open source DXE implementation

Tianocore is an open source implementation of the DXE runtime environment [31]. The Tianocore source was originally written by Intel and then put into the open source domain. The code base is provided with development tools for EFI applications and drivers, known as the EFI development kit (EDK). For example, it is used as the default UEFI runtime environment in some virtualisation software, such as QEMU [36] and Oracle's *VirtualBox* [35]. However, for use on real hardware it requires companion bootstrap code compliant with the PI standard.

Boot manager and boot services

In the UEFI environment, DXE drivers come in two flavours; boot services and runtime services [31, p. 83]. Boot services run only during the DXE phase, while runtime services can persist and coexist with the operating system. The operating system can then interact with these remaining services through external application layer interfaces.

Before initialising the operating system, the boot target is selected by the *boot manager*. When the DXE core considers itself done about its business, the boot manager finds and boots from a target. The boot targets and boot orders can be entered into a variable and loaded from non-volatile memory (NVRAM), similarly to how a legacy BIOS setup utility works. The UEFI boot manager works with the *GUID partition table* (GPT) as well as the master boot record (MBR) of the legacy BIOS. The GPT is part of the UEFI standard. Most importantly, the 2 TB limitation of the MBR is remedied. The GPT supports drives up to 2^{64} bytes or 16 exbibytes[2] [48]. Note, though, that the operating system must be compliant with the GPT and that a legacy BIOS also could implement GPT support.

[2]This will last a while: $2^{64-32} \approx 4.3 \cdot 10^9$ times the 2.2 terrabyte MBR limit.

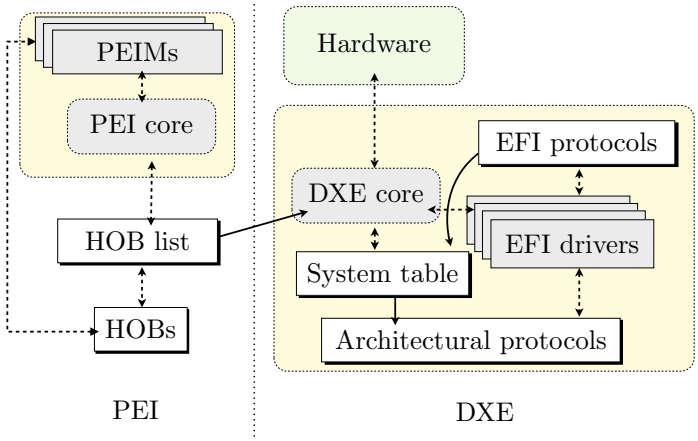


Figure 9: PEI/DXE handover diagram. The PEI dispatcher runs a number of PEI modules (PEIMs), which are described by the hand-over datablock *(HOB) structures. The HOB list is transfered into the DXE phase, where the DXE core is initialised. The latter loads the EFI drivers which install a number of EFI protocols in the UEFI system table.*

Though EFI and UEFI firmware has existed for nearly a decade, adoption of the standard has been slow. Recently, interest in the UEFI standard has been revitalised for several reasons. First, disk drives larger than 2.2 TB began shipping during 2010. Secondly, Microsoft announced its plans for extended UEFI support in Windows 8 in September 2011 [49]. Therefore, it will likely become the dominant x86 firmware type before the end of this decade.

2.1.6 Coreboot

Coreboot is an open source bootstrap code project for the x86 architecture [23]. It originates from the Advanced Computing Laboratory (ACL) at Los Alamos National Laboratory (LANL) in the United States. In 1999, Ron Minnich and other researchers there involved with high-speed data clusters were having trouble with the legacy BIOS installed on each node. In addition to the shortcomings of legacy BIOS outlined in section 2.1.4, Minnich found the closed source nature of the legacy BIOS problematic. Problems included the need to connect a keyboard to each of the 1024 network nodes to boot[3] or change the default firmware settings. The LANL team also required custom firmware that could be reconfigured and updated remotely. These requirements prompted the development of an open source boot loader. The resulting firmware, *LinuxBIOS*, utilised the Linux kernel as a boot loader [34]. After 8 years, the LinuxBIOS project was renamed to Coreboot as the Linux kernel was no mainly longer utilised as the boot loader. Currently, Coreboot is the bootstrap code segment which initialises the basic functionality of the hardware[4]. Combined with a *payload*, Coreboot becomes a complete, open-source firmware solution. The payload can for example be a Linux kernel, the *Seabios* open-source legacy BIOS (see page 23) or *FILO*, a boot loader which loads a Linux kernel from a file system without the help of legacy BIOS services.

Porting Coreboot

Before Coreboot can be used on a particular piece of hardware it must first be ported to it. Coreboot currently supports over 230 motherboards. The port effort to new hardware requires detailed documentation and data sheets of the processors, chipsets and other components. CPUs and chipsets are becoming ever more complex and some even have FPGA-like elements which are programmed by bytecode in the firmware. Without the support from hardware manufacturers, this is a difficult task. It has been the author's experience that Intel Corporation is particularly secretive regarding hardware

[3] "Keyboard not found. Press F1 to continue." No joke.

[4] The bootstrap code or *core boot code* performs the same basic function as the platform initalisation (PI) implementations of the UEFI standard. However, the DXE core is not compatible with the Coreboot interface.

Figure 10: The Coreboot logo. The logo is copyrighted 2008 by Konsult Stuge and coresystems GmbH and used with permission.

specifications. On the other hand, AMD is actively contributing to the Coreboot source code and has produced a wrapper for the *AMD Generic Encapsulated Software Architecture* (AGESA) interface. AGESA is a platform initialisation API created by AMD and used to bootstrap AMD processors and chipsets. On May 5th, 2011, AMD Embedded Solutions announced that *"AMD is now committed to support coreboot for all future products on the roadmap"* [50].

Simplified, the motherboard can be viewed as a combination of different components. To add support for a new board, code must be written to initialise the CPU (which sometimes require microcode updates), northbridge and the memory controllers, southbridge and the I/O buses and possibly the *super I/O* (SIO) for a serial connection. With little or no documentation available, this can be a very time consuming process. Interrupt routing tables such as the `$PIR` and MP tables must be also supplied. Much code can be reused if the board components are already supported. The Coreboot source tree contains directories for these different components. Therefore, in the simplest case, adding support for a new board is the process of combining the different pieces of the puzzle into working firmware. The file `devicetree.cb` describes the mainboard characteristics, such as which chips are on board and which I/O devices are installed. The Coreboot build system parses this file and puts all the pieces together at compile time.

Both Coreboot and most payloads (including Seabios and FILO) are compiled with the GCC toolchain in a Linux environment. A special cross-compiler (`xgcc`) is utilised to work around small issues with the compiler and to create the ELF executables. Debugging during the development process is done with the help of clear text messages on the COM1 serial port (I/O port `0x3F8`). If no SIO is available or the system has no serial port, it is possible to debug through a USB connection, however this requires basic initialisation of the EHCI controller.

Order of execution from entry vector to payload

As Coreboot is open source, the program flow from power on to payload handoff can be followed in detail. On the Intel architecture, the Coreboot bootstrap process works as follows [23, Developer manual]. The early initialisation is written in 16-bit assembly language and is followed by a mainboard initalisation stage (`mainboardinit`), written in 32-bit assembly and C. The final stage before payload handoff is called the main hardware stage, `hardwaremain`, and is written completely in C. The procedure is illustrated as a flowchart in figure **??**.

1. Execution begins at the entry vector `0xFFFFFFF0` in the 32-bit x86 architecture. This address translates to the top 16 bytes of the firmware image (`0xFFFF0`, assuming an 8 Mbit image). Here, an assembly jump instruction (`reset16.inc`) is placed. Execution continues in another assembly segment, `entry16.inc`, which performs initial CPU initialisation – the translation look-aside buffers (TLBs) are turned off and a global descriptor table (GDT) is configured so that the entire memory map can be access in protected mode. The CPU then exits the 16-bit real mode and enters the 32-bit protected mode. In `entry32.inc`, the processor is further initialised by specifying the CPU segment registers.

2. During the early mainboard initialisation (*mainboardinit*) stage, floating point units or instruction set additions might be initialised, depending on processor configuration. Some (well-documented) boards have support for cache-as-RAM, which is initialised at this point.

3. The late mainboard initialisation stage (the *romstage*) is written in 32-bit C. Unless the processor cache is used as RAM, it is compiled with a special register-only compiler, `romcc`. This is

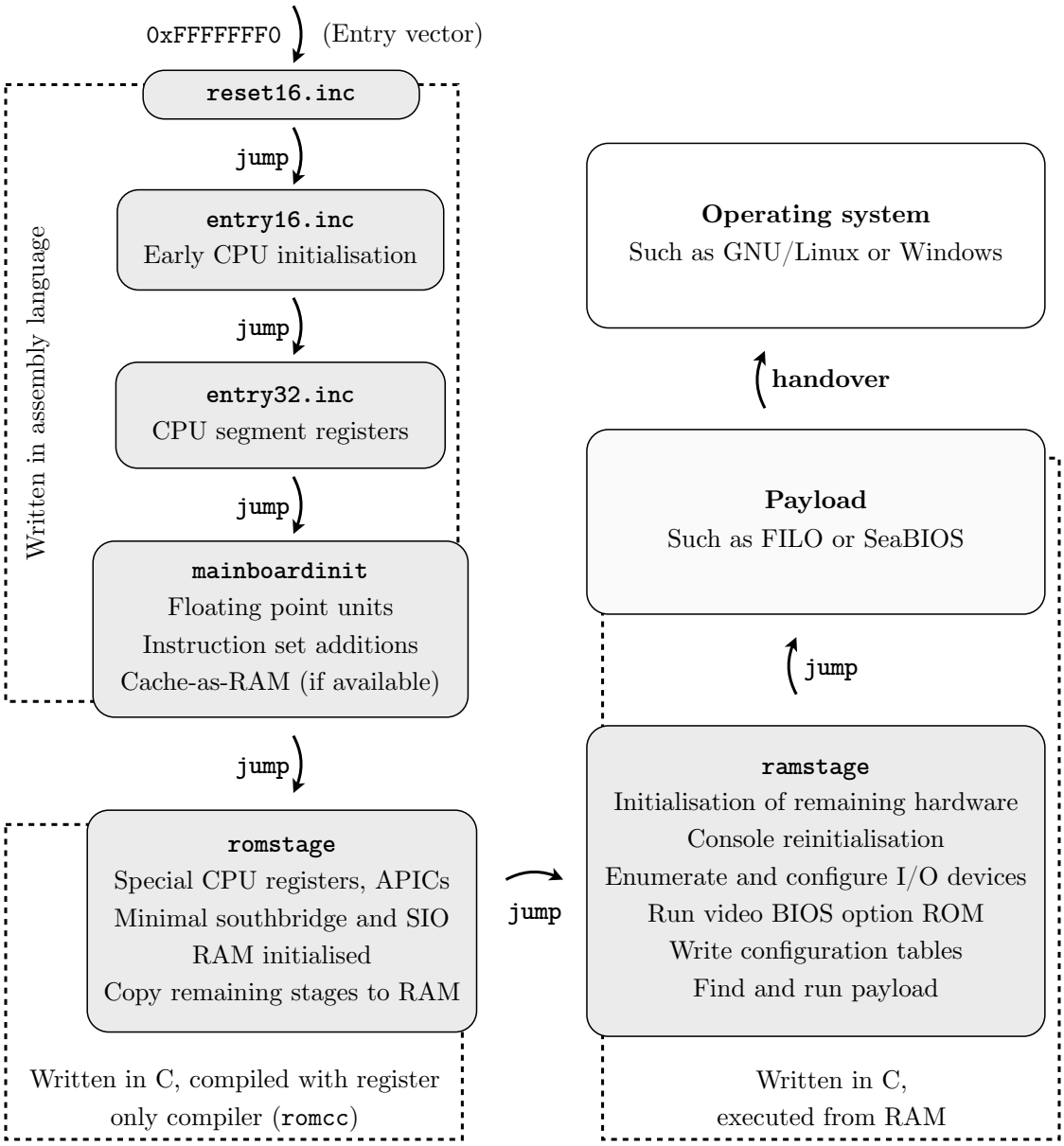


Figure 11: Flowchart of the program flow from power on to operating system handover using Coreboot.

necessary as no RAM controller yet has been initialised. During this stage, CPU-specific and memory type registers and the internal CPU interrupt controllers (APICs) are configured. The southbridge and SIO are minimally configured and a console initiated – at this point, debug messages start to appear on the serial connection. Through the southbridge, the RAM type is identified over the system management bus (SMB) and the memory controller is initialised in the northbridge. The RAM itself is then initialised.

4. At this point, the remaining stages are copied to RAM (possibly decompressed if compression is used). The interrupt descriptor table (IDT) is initialised and the last stage, the main hardware stage (`hardwaremain`) is entered.

5. The remaining code (the *ramstage*), is 32-bit C code with a full stack in RAM. The console is reinitialised and all the devices on the I/O bus are enumerated, configured and enabled. The list of found devices is compared to the predetermined `devicetree.cb` list. Optionally, Coreboot might run a VBIOS at this point to enable video output.

6. Finally, control of the system is transfered to the payload. The Coreboot image (that goes into the boot ROM) uses the (rather undocumented) *Coreboot file system* (CBFS). Payloads are stored as ELF executables and are called (jumped to) when Coreboot has initialised the basic system functionality. The payloads themselves are responsible for extracting the *CBFS tables* from memory, which contain the information regarding the system state.

The Seabios payload

The Seabios project is an open source implementation of basic legacy BIOS functionality, mostly written in 32-bit C by Kevin O'Connor [51]. It provides most services a modern operating system would expect from a legacy BIOS, including ACPI, SATA/AHCI (since version 1.6.2) and SMBIOS. Seabios was originally developed as the default firmware for the QEMU project [36], a virtual machine environment. It has since been converted into a payload for use on real hardware in conjunction with Coreboot. Seabios has not been extensively tested on real hardware due to its emulator origins but development is active. Seabios supports only the legacy MBR for operating system handoff, which also works with the LILO and GRUB software second stage boot loaders. Booting is supported from PATA, SATA, ATAPI CD-ROMs, USB hard drives, etc. Seabios can also be configured with a splash screen (showing either JPEG or BMP images) and a boot menu. If either is used the bootstrap process will stop while the image or boot menu text is displayed.

As a payload, Seabios gains control of the hardware and initialises the legacy BIOS services after Coreboot passes control to it. This implies that Seabios implements and hooks the corresponding interrupt service routines (ISRs) to enable an OS to boot and initialise its own hardware abstraction layer. Seabios will utilise the CBFS tables if it is compiled for Coreboot. Seabios will also locate and extract all option ROMs from the CBFS image and execute them. For more efficient execution, option ROMs can run in "parallel" with Seabios. The option ROM will execute while Seabios is waiting for I/O; it is then interrupted by a 1 KHz timer IRQ set by Seabios.

The Flashrom boot ROM programming utility

Flashrom is a cross-platform, open source utility for reading and writing to flash memory attached to an x86 computer system [52]. According to Borisov [53], Flashrom originated as an utility in the

Coreboot project and they are often used in conjunction with each other. The utility supports the common protocols *serial peripheral interface* (SPI) and *inter-integrated circuit* (I^2C), as well as the older *firmware hub* (FWH) and parallel programming (PP) protocols. Flashrom is often the software tool of choice to update the firmware on the x86 platform. This is partly because Flashrom enables the user to write anything to the non-volatile memory chip. The author has yet to find a free, commercial flash memory utility that will write arbitrary BIOS images.

2.2 BIOS optimisation schemes

In its broadest sense, boot time optimisation is all about streamlining the actions performed by the firmware. It is entirely possible that the firmware will perform unnecessary actions during the bootstrap process. Such actions must be identified and removed. If the firmware behaves procedurally, it might be enough to remove such actions. However, if the firmware behaves dynamically, such as the DXE runtime environment in UEFI, scheduling and dependency problems might also arise. The better the hardware configuration is known beforehand, the easier boot time optimisation becomes. Closed-box and embedded systems are therefore good targets for optimisation techniques. [13]

Recent work on BIOS optimisation schemes are presented by Doran et al. [12], Kartoz et al. [13], Rothman [14] and chapter 15 of Zimmer et al. [31]. All of these papers and chapters concern the UEFI environment, however some of the concepts should be applicable to all types of firmware:

1. BIOS boot time optimisation requires that the firmware source code is available or at least configurable. Even if source code is available, alternative firmware should always be considered. Depending on the design goals, it might or might not be a good idea to run a rather large dynamic runtime environment.

2. If a runtime environment is utilised and the drivers are ordered in a depedency tree, one must be sure to load the drivers in correct order [12]. The number of loaded drivers should also be minimised – which ones are *really* required to boot? There is, for example, no need to initialise the USB controllers unless USB keyboards during boot or USB booting are required. Video output during the boot phase might not be required; especially legacy VBIOS drivers are slow [12]. Drivers will be replaced by the OS hardware abstraction layer at a later stage [12].

3. The memory footprint of the BIOS should also be minimised. Reading the firmware from an SPI flash ROM is slow, especially if the the circuit is behind an embedded controller [12]. Compression of data could generate a performance boost if the flash ROM read operations are slow enough [13]. The overhead of decompression utilises the otherwise idle CPU time during SPI read operations. Code segments like the BIOS setup utility could be omitted to save space. As much code as possible should be executed from RAM rather than the SPI flash for faster execution speeds.

4. Code optimisations are always good. When development is complete, debugging should be turned off as the I/O operations can be extremely slow. If modules or drivers are written in a higher level language then compiler optimisations are helpful. [12]

5. The primary boot device can be set at compile time if the design requirements allow it. This eliminates the search time for a suitable boot device [13]. This idea can also be extended to memory type. The *system management bus* (SMB) on the southbridge is used to communicate

with the *serial protocol detect* (SPD) module in the DRAM circuits. If the memory properties are known beforehand there is no need to initialise the SMB at runtime.

2.3 Proprietary tools

This section describes some non-commercial, proprietary (freeware) tools investigated. Two are development tools offered by Intel Embedded, a subsidiary of Intel Corporation. The Intel BLDK toolkit is used by OEMs to develop firmware for Intel-based embedded systems, while the Intel IEGD toolkit is used to develop graphics drivers for the same. The third is a legacy BIOS modification utility provided by Congatec.

2.3.1 Intel boot loader development kit (BLDK)

The Intel BLDK is an enterprise software application that allows for rapid development of boot loaders. It is a relatively new alternative to vendors on the embedded systems market. Jensen and Hattaway [38] explain that the BLDK is based on Intel's own UEFI implementation and produces UEFI-compliant firmware. The developer application uses a graphical user interface (GUI) environment to allow construction of trimmed firmware images, which is depicted in figure 12. The interface is project-driven and allows simple configuration of the firmware feature set.

The BLDK creates firmware images using precompiled libraries (mostly Intel's core PI implementation for the chosen platform) and the Tianocore open source code base. The typical build path is based on the principles laid out by Zimmer et al. [31] to reduce boot latencies. The GUI enables the user to enable or disable features, much like a BIOS setup utility. However, settings are made prior to creating the firmware image and unselected functionality is never included. In this sense, the BLDK is more of a deployment kit comparable to a configurable, commercial off the shelf (COTS) firmware boot loader for the selected platform. The GUI also allows for a large portion of the Tianocore source

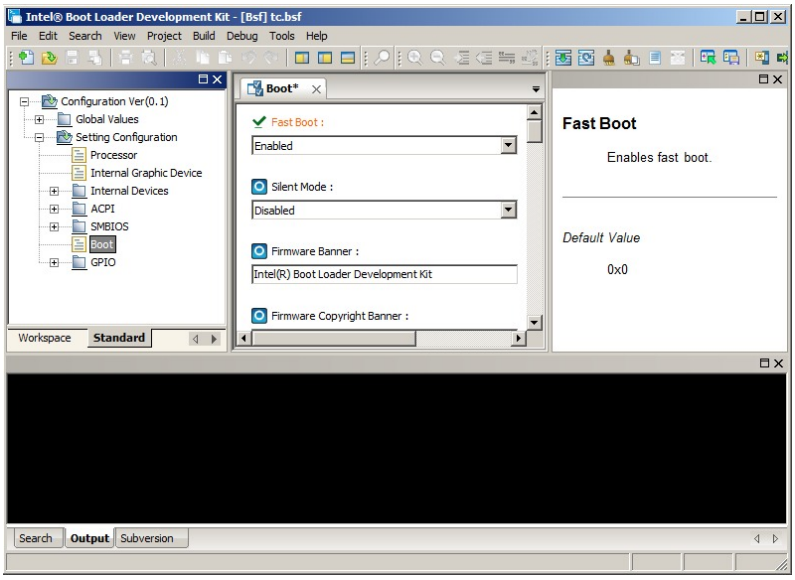


Figure 12: The graphical user interface of the boot loader development kit and the example configurations allows for quick deployment of firmware for select Intel architectures.

code to be edited. However, editing it must be done with care as it requires a deep understanding of the DXE environment and the UEFI standard.

The number of the platforms currently supported by the BLDK version 2.0 is very low. It currently only supports the *Intel Atom E600* series with upcoming support for the *Intel Atom E6x5C* series. As of July, 2011, there was no public roadmap for future support of processors or processor series [38]. The Intel BLDK version 1.0 also supports some older Atom platforms, including the Atom Z5xx-series (see section 3). However, Intel is only providing version 1.0 of the toolkit to partner companies in the embedded market.

Availability and dependencies

Intel currently provides the BLDK free of charge. Although the BLDK is free of charge, the application requires Microsoft Visual Studio 2008 to compile the images. Support for the GCC toolchain may be added at some point in the future, though. The BLDK also depends on having the Microsoft Windows Driver Development kit installed, as well as the Microsoft ACPI compiler. The public *Getting Started Guide* [37] describes the overall installation and setup process, which is straight-forward. Development and software support is only given to Intel premium partners, with lower-tier customers being refered to firmware vendors for support [38].

2.3.2 Intel Embedded Graphics Drivers (IEGD)

With recent Intel mobile and embedded chipsets, the graphics controller is integrated into the chipset. The IEGD is a development kit aimed at OEMs and BIOS vendors for configuring and building video drivers for the integrated graphics controllers. The utility can create drivers for Windows or Linux, a VBIOS for a legacy BIOS or DXE drivers. IEGD uses the open source *Open Watcom* C compiler to compile the video drivers. The GUI for IEGD is displayed in figure 13. Here, the developer creates the configurations and specifies the target chipset, display and port types, configuration handles and profile identifiers as well as detailed settings for the video modes.

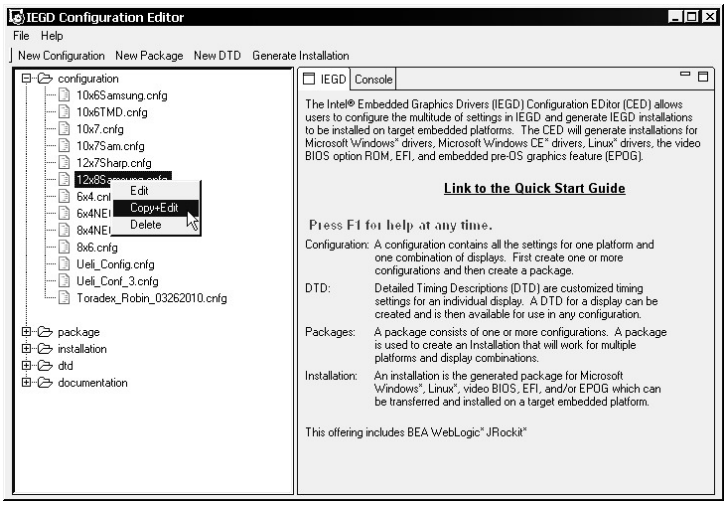


Figure 13: The graphical user interface of the IEGD allows for deployment of graphics drivers for Intel integrated graphics.

As with the BLDK, Intel is currently providing the IEGD free of charge but with little support. The program comes with a well documented *User's Guide* [46], but still requires detailed knowledge for correct display configurations, especially for *low voltage differential signaling* (LVDS) displays. Numerous power, synchronisation, width, height, polarity bits (and many more) options must be correctly specified or the image will not show properly, or not at all.

IEGD VBIOS and the 5F interrupt calls

During runtime, the VBIOS constructed by IEGD is associated with a number of proprietary system calls, the Intel 5F calls, which are documented in the IEGD *User's Guide* [46]. These calls all use interrupt `INT 15h` and are identified by `0x5F` in register AH. The legacy BIOS must supply an `INT 15h` ISR which manages calls made by the VBIOS. The VBIOS will chain its interrupt handler in the IDT to manage calls made by the legacy BIOS. Typically, the VBIOS will notify the legacy BIOS that the video driver is initialised and ready. In the ISR, the legacy BIOS can then request specific video settings by providing a configuration identifier. The configurations are fixed in the VBIOS at compile time, but some custom video settings may be set at runtime. The legacy BIOS can also request custom video modes by asserting interrupts that will be managed by the ISR chained by the VBIOS.

2.3.3 Congatec utility (CGUTIL)

The Congatec utility (CGUTIL) is a BIOS modification utility that can be used with the legacy BIOS. It is a good example of the type of applications used by BIOS vendors to modify a legacy BIOS binary directly. It enables the customer to add bootsplash logos and insert custom OEM code modules at various points. The default settings in the BIOS setup utility and the VBIOS can also be changed. The user works with a legacy BIOS binary image that is directly modified by the software. The utility has one graphical version for Windows (shown in figure 14) and one command line interface version for MS-DOS or GNU/Linux. They update the boot ROM on the supported boards. CGUTIL can only be used with select Congatec boards [54], so the number of supported boards is small.

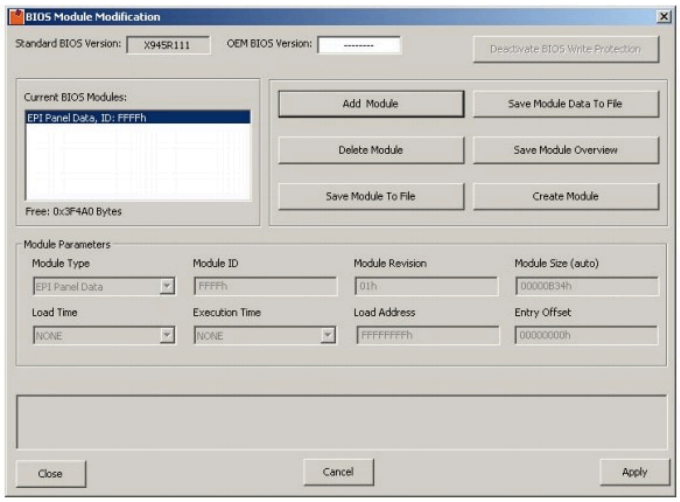


Figure 14: GUI of the Congatec utility, an example tool for modifying a legacy BIOS binary. The image is copyrighted by Congatec AG and used with permission.

3 Equipment and hardware

This section describes some of the equipment used and the target hardware, that is the platforms for which protoype solutions were considered.

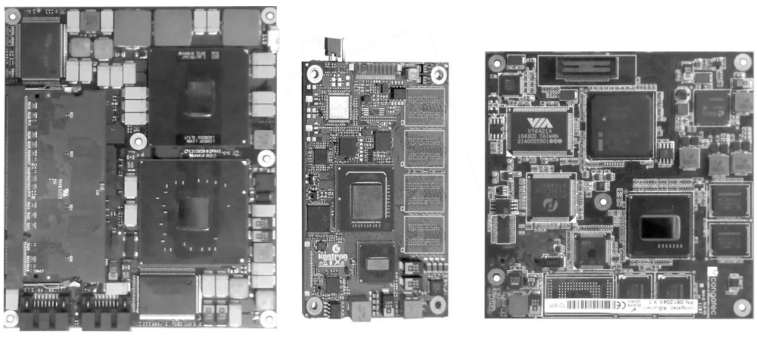
3.1 CCpilot platform

The *CCpilot* is a rugged embedded computer series manufactured and sold by CrossControl AB. Here, a limited description of the XL variant and the XM variant is given. The computers feature rugged connectors on the back and an optional touch interface panel on the front. Input and output ports may include the LVDS and/or VGA ports, an analogue video connector (XL only), USB 1.0 and 2.0 connectors, a serial communications port, an audio port as well as ethernet and CAN ports. The system is dependent on an external power source, such as a battery pack or a power adaptor. The default processors are either Intel Core 2 Duo or Intel Atom, a feature which currently makes them high-performance embedded systems. The CCpilot XL computer is depicted in figure 1 on page 1.

The CCpilot computers use ETX and COM Express form factor computer-on-module (COM) designs. This means that the processor, chipset and memory are located on the module and connected to the base board through sockets (COM connectors). When used in conjunction, the two work as a single mainboard. The base board has a small ARM or AVR embedded processor that initialises the base board before turning on power to the COM module. The flat panel display also has a microcontroller which enables it to be used both as an analogue (VGA) display or a low voltage differential signaling (LVDS) digital display. Because of the modular design, in theory any properly sized COM modules can be mounted. This enables an OEM using the COM system to consider a variety of module designs and vendors for use with a specific base board.

ETX-CD module

The CCpilot XL 3.0 ships with the Kontron ETX-CD COM module, depicted in 15 (a). The ETX-CD is a high performance module, with an Intel Core 2 Duo processor, the Intel `945GM` chipset



(a) (b) (c)

Figure 15: (a) The primary target hardware, a Kontron ETX-CD COM module used in conjunction with the CCpilot XL 3.0 base board. (b) Secondary target hardware, a Kontron Nanoetxexpress-SP COM express module used in conjunction with the CCpilot XM base board. The jumper on the top side was added by the author to enable TSOP hotswapping (see section 4.3.2). (c) Secondary target hardware, a Congatec Conga CA6 COM express module.

and Intel ICH7M Input/Output controller. The memory type is DDR2 SODIMM and PCI is only supported at legacy 32bit/33 MHz. The PCI routing is handled by both dual ISA PICs as well as an additional I/O APIC [55, p. 45]. Among module interfaces, which need to be physically implemented on the carrier board, the ETX-CD suppots VGA and LVDS (flat panel) video through the integrated graphics controller. There is also support for AHCI-compatible SATA hard disk interface, USB 1.0 and USB 2.0 interfaces as well as two serial ports through a Winbond W83627HG super input/output (SIO) chip. The main components and interconnecting buses of an ETX-CD module (connected to a CCpilot XL 3.0 base board) are shown in figure 16.

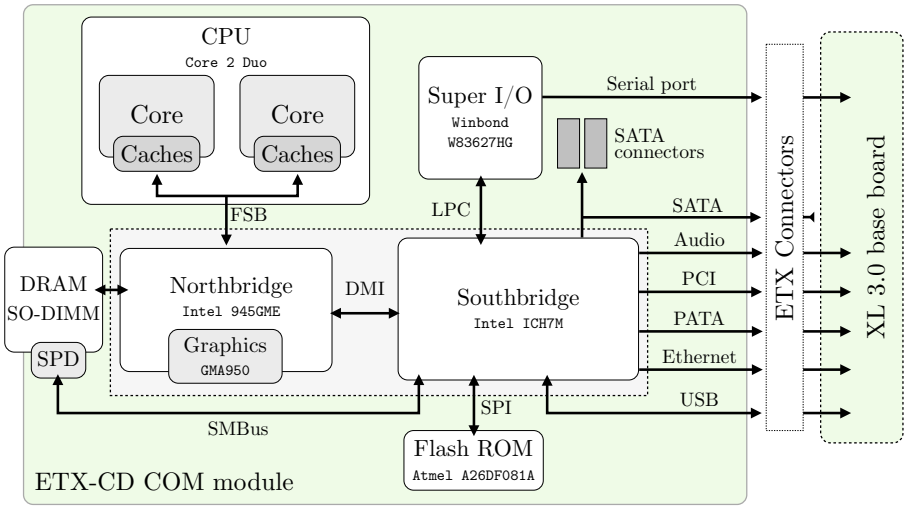


Figure 16: Simplified chipset layout of the ETX-CD COM module while connected to an XL 3.0 baseboard. Arrows indicate interconnecting buses.

Nanoetxexpress-SP module

The CCpilot XM ships with the Kontron Nanoetxexpress-SP COM express module, depicted in 15 (b). This module has a much smaller physical footprint and features an Intel Atom Z510 processor (codenamed *Silverthorne*), while the chipset is integrated into a single circuit; the US15W system controller hub (codenamed *Poulsbo*). The memory type is DDR2 SODIMM and PCI Express (PCI-e) is supported. The primary video interface is provided through LVDS through the COM Express connector. It also has a SATA to IDE bridge, which the CCpilot XM base board utilises to enable the compact flash slot. Note that the lack of an explicit SIO makes debugging difficult without extra tools (serial output is emulated by the chipset).

Conga-CA6 module

As the COM express standard connector allows flexibility in motherboard design, new COM express modules for the CCpilot platform were frequently evaluated. One of these candidates was the Conga-CA6, manufactured by Congatec AG. It featured the Intel Atom E6xx processor with the integrated controller hub (ICH) EG20T (codenamed *Tunnel Creek + Topcliff* by Intel), a newer model than the Z510+US15W. I/O was particularly modern, with PCI express and the latest SATA/AHCI interface and digital video output (sDVO and LVDS) only.

3.2 Development and debug boards

Some other boards were used in conjunction with the COM modules. A concept picture of a COM modules, a debug card and an development carrier board is shown in figure 17. The bottom layer is the development carrier board, a type of base board used for firmware development or product evaluation purposes. The middle layer is the debug card and the top layer is the COM module.

Three different *development carrier boards* were used. The Kontron *ETX Eval* for ETX modules and the Congatec *Ceval* for COM express modules resembled ordinary, ATX-size motherboards and were equipped with various outputs, including USB connectors, VGA-out, serial ports, PCI and PCI express expansion slots. The Kontron *Nanoetxexpress-HMI* was a miniature COM express evaluation board. It had a LVDS monitor attached as well as a super I/O and a serial port.

Debug cards have both female and male ETX or COM Express connectors and are inserted between the carrier board and the COM module. As the name suggests, they are used for debugging purposes and often come with a POST code display for displaying legacy BIOS POST codes (described on page 9). Two debug cards are mentioned here, namely the Kontron *ETX-port 80* ETX debug card and the Congatec *Cdebug* COM express debug card. The debug cards can be used on any base board and both were tested with the CCpilot XL and XM base boards, respectively.

The development and debug boards also often have extra Super I/O to enable a serial communication in case the COM module is lacking this capability. Note that the existance of an interface on the carrier board does not imply that the interface will be immediatly available. For example, the Nanoetxexpress-SP or Conga-CA6 did not have VGA out, so the VGA out connectors on the carrier boards were inoperable.

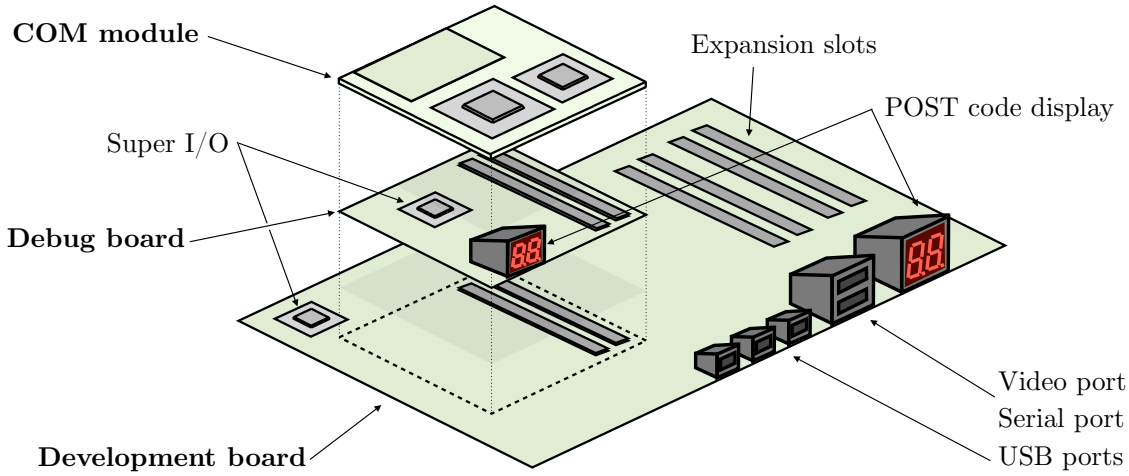


Figure 17: Development board and debug boards can facilitate firmware development. The development board is a normal carrier board but also offers slots for expansion cards, for example to enable an analogue video port for a COM-module without analogue video support. In case the COM module lacks a Super I/O chip, an additional chip on the development or debug board is necessary to establish a serial connection required for debugging. Both boards often have 7-LED displays to show the POST codes of a legacy BIOS.

4 Procedure

In this section, the methods and processes adhered to during the project, as well as design decisions, are explained and motivated. It also details the steps carried out in order to develop, test and compare some different alternatives on the target hardware. This work was carried out in cooperation with CrossControl AB on location in Alfta. The experimental procedures and setups for measuring boot times are also described in this section. Throughout this chapter, every statement of a performed action in passive voice should be read as if the author performed the action himself. For example, *"The module was examined"* should be read as *"The module was examined by the author"*.

4.1 Method

The methodology and procedure are important concepts to ensure that the project goals are reached. The PEP process [56], applied by CrossControl AB to master theses, enforced planning and put emphasis on producing artifacts at all stages, such as risk assessment, design requirements, documentation and prototypes.

Few circumstances around the given problem were originally known, as described in section 1.2. Relevant research into BIOS optimisation (see section 2.2) was based on the EFI and UEFI interfaces. The original plan was to utilise these papers and apply them to the real-world scenario, using *Tianocore* as code base (see section 2.1.5). However, for application on real hardware it required platform-specific bootstrap code compatible with the platform initalisation (PI) interface, a part of the UEFI standard. Such code was not available and compatible open source alternatives were non-existent. This prompted the change in project goals but also changed the direction of the work to be performed. The work performed by the author can therefore be characterised as a technical investigation into available firmware alternatives. Therefore, the PEP process could not directly be applied. However, some initial artifacts were produced:

- A *risk assessment* was made during the project startup phase. The greatest risk identified during this was *information shortage*. CrossControl AB had asked the author to investigate into x86 firmware development, hoping to expand their area of expertise. It was therefore difficult to evaluate whether or not the project goals were realistic or not. A feasibility assessment activity based on accumulated knowledge was therefore introduced. Another identified risk was the *dead end* risk. Continuous feedback to the author from the supervisor, reviewer and CrossControl management helped to avoid dead ends.

- The *goals and purpose* were also specified during planning. These are described in section 1.2, where points 3 and 4 on page 2 can be viewed as vague prototype design requirements.

The final methodology used by the author is shown in figure 18. This process was based loosely on the PEP process. Two avenues of applied research were pursued in parallel, dubbed *theory* and *practice*. The theory avenue addressed the goals to compile knowledge of BIOS- and UEFI-based technology and to evaluate the use of UEFI technology in CrossControl products. Current knowledge about the field of firmware engineering needed to be compiled. Section 1.4 explains that the literature review conducted did not reveal any directly applicable literature. The review did, however, reveal multiple sources of information, including industrial standards, other books and papers. The compilation of relevant information therefore became a top priority. This avenue of research ultimately resulted in chapter 2 of this thesis.

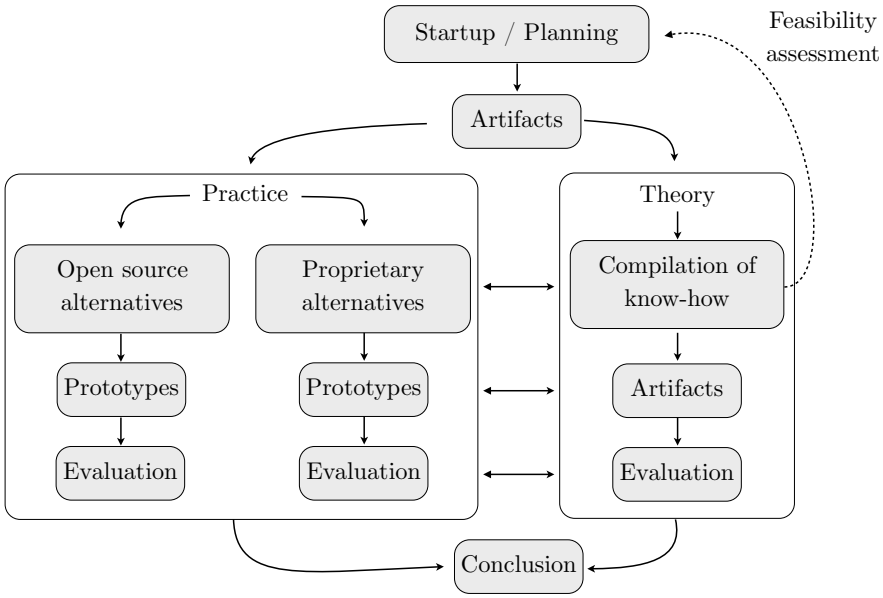


Figure 18: Action diagram of methodology and procedure.

The practice avenue of research addressed the goal of implementing new boot loader firmware on the target platforms. The process involved working directly with the hardware and tools, working with both open and closed source alternatives. The hands-on experiences were then leveraged to assess theory and observations. The work described in this chapter and the results described in chapter 5 mainly concern this practical part.

Selecting an open source alternative and target hardware

Open source firmware alternatives for the x86 platform were not abundant. The list presented here is exhaustive to the author's best knowledge. The following four alternatives were considered:

1. **Coreboot+Filo.** The Filo payload for Coreboot was a firmware variation of the second stage boot loader Lilo [57]. It would have provided efficient means to load a Linux kernel directly from the boot ROM, but only that specific Linux kernel.

2. The universal boot loader, **Das U-Boot** [11], could either be used as a payload for Coreboot or as a complete firmware solution. It could load the kernel of an embedded operating system, such as GNU/Linux, FreeBSD or Windows CE, directly into memory. However, the bootstrap code base for the x86 architecture was not as rich as that of Coreboot. Das U-Boot is mostly used in ARM embedded systems or as a second stage boot loader.

3. **Coreboot+Seabios.** The Seabios payload for Coreboot would provide legacy BIOS services which could boot both Windows XP, GNU/Linux and other operating systems.

4. **Coreboot+Openbios.** Another payload alternative put forth was *Openbios* [24], which was based on Openboot. Openboot was an open source *Open firmware* implementation developed by Sun Microsystems (now Oracle). Openbios was compliant with the now deprecated *IEEE-1275 Open Firmware Standard* and not a standard legacy BIOS in itself. As such, it could only boot certain SPARC- and PPC-based operating systems.

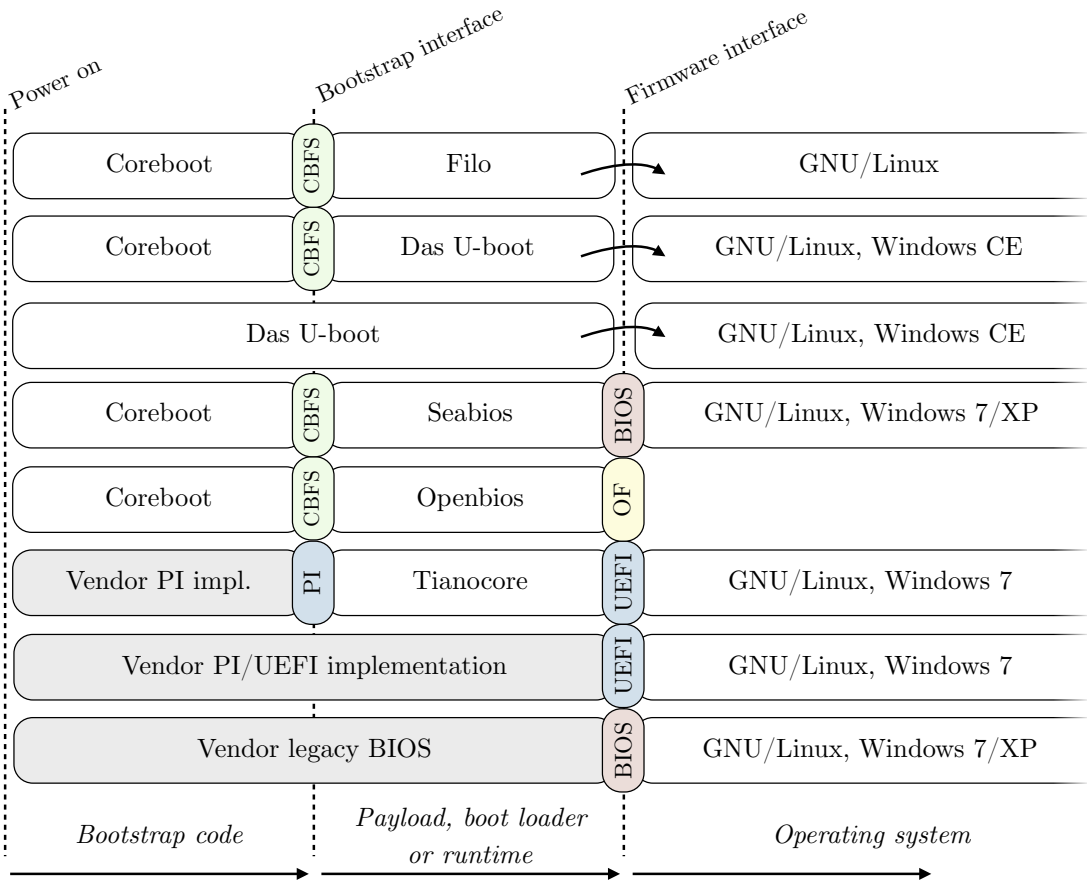


Figure 19: Comparison between bootstrap alternatives available as of 2011. White boxes in the first two stages indicate open source alternatives, while grey boxes indicate closed source. Between the different stages are clearly defined interfaces: The legacy BIOS interrupt services, the UEFI standard, the platform initalisation standard (PI), the Open Firmware standard (OF in this figure) and the CBFS (Coreboot file system). If a second stage boot loader is utilised directly in the boot ROM, no services are required as the OS initialises its own hardware abstraction layer (HAL).

These options, as well two proprietary legacy BIOS and UEFI options, are displayed for comparison in figure 19. The figure highlights the two phases during the initialisation process. Coreboot utilises payloads to load operating systems or provide firmware services and a PI implementation will utilise the UEFI runtime environment (here provided by Tianocore).

After careful consideration, Coreboot+Seabios was selected as the open source firmware platform. This was the option most similar to the legacy BIOS the new boot loader would replace. The Coreboot+Filo solution was thought to be too restrictive, as it would only load a specific Linux kernel. Locking the customer to a specific OS and version worked in some use cases, but not others. While U-Boot provided slightly more flexibility in terms of OS selection and location, it could still not boot Windows XP, required by some customers. Openbios was rejected because it could not boot any of the required operating systems.

The primary target hardware for the Coreboot+Seabios port consisted of a CCpilot XL 3.0 base board with a Kontron ETX-CD module. This configuration was selected for two reasons. First, it was used in the CCpilot XL computer. Secondly, the `945GME` northbridge, `ICH7M` southbridge and `W83627HG` SIO on the board were nearly identical to other components on boards already supported

by Coreboot. The firmware was located in an SOIC with an SPI interface, which could be programmed by an external SPI flash programmer (see section 4.2.1) available at CrossControl. This made the ETX-CD a logical choice to start off with.

The secondary target hardware for a Coreboot+Seabios port was the Kontron Nanoetxexpress-SP and the CCpilot XM base board. This platform was deemed difficult to develop for compared to the ETX-CD. While the *Silverthorne* + *Poulsbo* chipset on the Nanoetxexpress-SP module was also supported by Coreboot, the complexity of the chipset was greater and the code base had been unmaintained for nearly a year.

There was also concern regarding the video output, as the CCpilot XM board did not have a VGA port. This put greater requirements on the any VBIOS configuration, as it would have to initialise the LVDS display properly. Last, the boot ROM was located in a TSOP circuit with a FWH interface, so the available external programmer could not be used.

Selecting a proprietary alternative and target hardware

A comparison between open source and proprietary firmware solutions presented an interesting academic value proposition. The proprietary alternatives that were considered were:

1. **Intel BLDK**. The BLDK version 1.0 did support the *Silverthorne* + *Poulsbo* chipset on the Kontron Nanoetxexpress-SP module. Unfortunately, as described in section 2.3.1, this version was not publicly available from Intel. The author and CrossControl AB seeked access to this version for evaluation and production purposes through contacts at Congatec AG, a premium partner of Intel Embedded. This resulted in a non-disclosure agreement (NDA) between CrossControl AB and Intel Corporation being signed. While this agreement would have hindered describing the solution, the resulting firmware could still have been described. Despite the signed NDA, Intel did not provide access to the version 1.0 of the BLDK. BLDK 2.0 with support for the *Tunnel Creek* + *Topcliff* chipset would have to be used instead.

2. **Congatec utility.** The Congatec utility provided means to modify a legacy BIOS in certain Congatec COM modules. The author quickly had the tool running and successfully implemented a boot splash image on a Congatec conga-CS45 module with a Core 2 Duo processor. The tool also provided the option to develop OEM code modules, which would essentially run as option ROMs. Congatec also provided the necessary documentation, which stated that the OEM code modules potentially could be used in ways to optimise the BIOS.

The BLDK was considered an emerging and interesting technology. Even though the number of supported chipsets was low, BLDK was expected to gain support for upcoming Atom chipsets. As such, it would be increasingly relevant in the future. It also utilised the UEFI standard, which was expected to replace the legacy BIOS at some point in the future. On the other hand, the Congatec Utility could provide a unique insight into the workings of a legacy BIOS. However, the steep learning curve of writing OEM code modules and potential lack of detailed support and documentation was deemed a considerable risk.

After considering these options, the author and CrossControl AB agreed that the Intel BLDK 2.0 should be tested with the conga-CA6 module. The conga-CA6 was currently being evaluated for use in the CCpilot product series. It was the only card available at the time with a chipset to support the BLDK boot loaders. In the best of worlds, the proprietary and open source solutions would be tested on the same hardware. Unfortunately this was not possible.

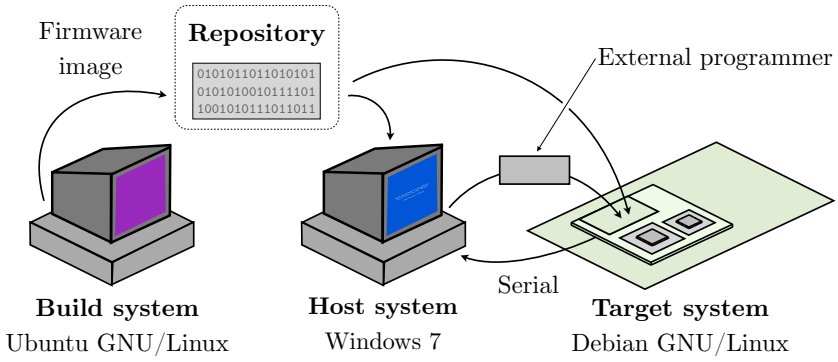


Figure 20: Build, host and target systems. The firmware was compiled on the build system and transfered to a repository. From there, the host system could program the target system using an external programmer, or the target system can update the firmware using an internal (software) programmer.

4.2 Coreboot+Seabios applied to Kontron ETX-CD module

The target system for the Coreboot+Seabios boot loader was the Kontron ETX-CD module, attached to a CCpilot XL 3.0 base board. Section 4.1 explained the rationale behind the choice of open source firmware and target platform. This section describes the work performed by the author to implement the source boot loader prototype.

4.2.1 Build system, host and target hardware

The build environment for Coreboot and Seabios was kept in a virtual machine running Ubuntu GNU/Linux (the *build system*). Both projects utilised the `gcc` toolchain, using the `xgcc` cross-compiler and the configuration system `make`. In Coreboot, the main configuration file (`.config`) specified the target mainboard and settings for the general functionality. In the mainboard directory, source code and configuration files for that particular mainboard was held. This mainboard-specific source code linked in other necessary source components, such as generic x86 code or chipset-specific code. Some of the Coreboot source code was not compatible with the default `gcc` compiler due to a bug in the so-called binary utilities (`binutils`) component of `gcc`. For this reason, the `xgcc` cross-compiler had been placed in the Coreboot source tree. It was compiled with `gcc` and the binaries were placed so that the `make` system could find them. The Seabios source code was generic for all x86 systems and configured through the `.config` in the Seabios directory.

The author needed to keep track of a history of various configuration files and to save log files from the build script. A build shell script was therefore written. It included options for configuring both Coreboot and Seabios, initiated the `make` build scripts if changes were made to the configurations, timestamped and backed up old logs, configurations and binaries. After the build process completed, the firmware image could optionally be sent over the network to a depository.

A Debian GNU/Linux installation on the target system was used to compile and run the Flashrom utility. A script was written that could download the latest binary image from the repository and write it to the boot ROM. The script could also restore the vendor BIOS image or revert to an older version of the firmware being developed. Alternatively, the *host system* could update the firmware using an external programmer. The host could also read firmware debug messages over a serial connection. The build, host and target systems are shown in figure 20.

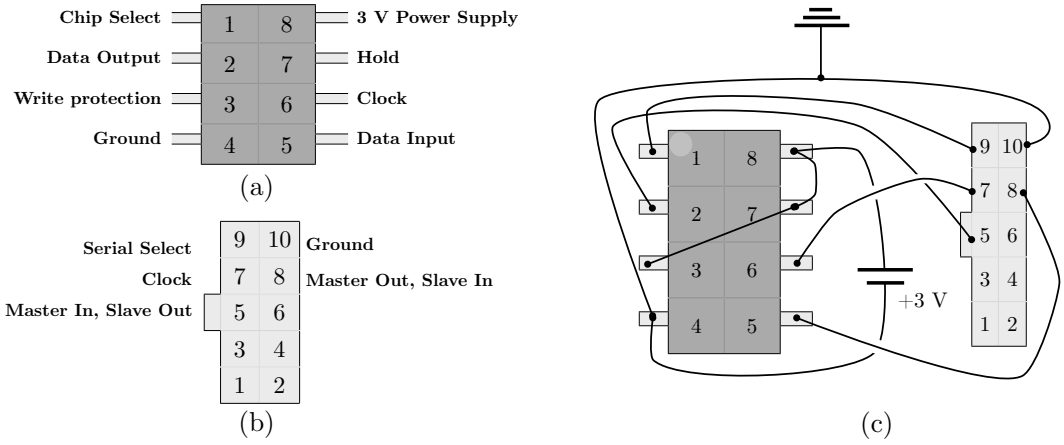


Figure 21: (a) Common pinout of SOIC flash ROMs. (b) Connector layout of the Aardvark SOIC programmer. Only SPI-relevant pins are specified. (c) Circuit diagram indicating how the SOIC connects to the programmer cable.

In-system programming of the boot ROM

When a firmware image failed to boot the target system, the host system was used to update the firmware on the target system. On the ETX-CD module, the firmware was stored in an Atmel `AT26DF081A` 1024 kB small-outline integrated circuit (SOIC). The *Aardvark* SPI/I^2C flash programmer, a USB peripheral, allowed the Aardvark software in host system to interface with such integrated circuits [58]. The process involved connecting the adapter to the circuit, either by placing the SOIC in a socket or soldering cables directly to the circuit. This procedure is called *in-system programming* (ISP) [28]. With ISP there is always a risk that parts of the chipset will be powered by the voltage supplied by the external programmer. This causes disruption and disturbances, so the procedure was not guaranteed to work. However, early attempts at programming the SOIC in a constructed socket failed, so ISP was attempted.

The pinout of the SOIC is shown in figure 21 (a), while the connector layout is shown in figure 21 (b). These were properly connected by soldering cables as shown by the circuit diagram in figure 21 (c). Twisted wire pairs were soldered directly to the SOIC pins on the ETX-CD board, the boot ROM could be successfully read. This wiring minimised interference in the wires. The original firmware, Phoenix TrustedCore BIOS version MCALR912 (the *vendor BIOS*), was read and saved to a file. Programming the SOIC was not successful until the connection between SOIC pins 3, 7 and 8 in figure 21 (c) was made. This wiring removed the write protection restriction. In this configuration, the chipset could also read the SOIC contents as normal. The soldering on the ETX-CD was performed by a CrossControl specialist.

4.2.2 Boot loader configuration

The Coreboot port for the ETX-CD was based on the source code for the `986lcd-m`, another mainboard manufactured by Kontron. It had a similar (but not the same) chipset to the ETX-CD. `986lcd-m` had Core 2 Duo processors, an Intel `945GM` northbridge and a Intel `ICH7R` southbridge, while the ETX-CD module has Core 2 Duo processors, an Intel `945GME` northbridge and an Intel `ICH7M` southbridge. The main difference was thus the chipset revision. The boards differed in other ways though; the `986lcd-m` was a mini-ITX-sized mainboard with a PCI express I/O bus, while the

and the ETX-CD was an ETX-sized COM module with conventional PCI.

Debugging was done with the help of the serial port. Coreboot and Seabios were configured to output debug messages on the COM1 port (I/O address `0x3F8`) by default. The messages were intercepted by the host system. If the firmware successfully initialised the basic functionality of the chipset and Super I/O, detailed messages were be logged on the host system.

Mainboard configuration

The mainboard configuration file, `devicetree.cb`, was parsed at compile time by the build system. It described the logical chipset structure and the I/O devices and buses in the target system. The configuration was hierarchical and corresponded to the ordering of these entities. Each entity in the device tree was described by a code block. If the entity was a chip, the block header contained the path to the source code corresponding to that chip. In this sense, the configuration file worked as a header file. Each block also contained a list of chips or devices logically attached to that entity. The outermost block was the northbridge block; the Super I/O block was in turn the innermost.

Properties such as device enablers, configuration space registers I/O port settings were also described inside each block. For example, the SATA/AHCI was enabled by setting two registers in the southbridge block – one to enable the interface and one bitmap to specify the number of ports. As another example; the COM port settings were described in the Super I/O block. By using the GNU/Linux command `lspci` in verbose mode, the PCI device tree could be enumerated. This output was then transplanted to the `devicetree.cb` file into the southbridge block and the location (bus and device number) for each found PCI device was specified.

IRQ routing

In the ICH7 southbridge of the ETX-CD, the IRQ routing is handled by an I/O APIC and two legacy PICs [55, p. 45]. Therefore, both the `$PIR` table and the MP table needed to be supplied. Two tools were provided by the Coreboot project to extract these tables from the target system, namely the `getpir` and `mptable`, two GNU/Linux command-line utilities. After booting the target system and GNU/Linux using the vendor BIOS, these utilies could search and extract the tables from memory. The `$PIR` table was extracted by the `getpir` tool contained an incorrect checksum. When used in the boot loader, this `$PIR` table produced IRQ routing errors and was discarded. A more correct `$PIR` table was instead supplied directly by Kontron Embedded. The author translated and installed this table into the Coreboot source code (`irqtables.c`) with greater success. This table contained 12 entries, as opposed to the 18 entries in the errorous table found by `getpir`. The `mptable` found and extracted an MP table as well as produced source code compatible with an older version of the Coreboot source tree. This code was rewritten to comply with the latest Coreboot version (v4).

ACPI and DSDT setup

A generic differentiated system descriptor table (DSDT) written in ACPI script language (ASL) was supplied with the source code branch. At compile time, this table was compiled using the Intel ASL compiler into AML bytecode and included in the Coreboot image. The table enabled a minimal implementation of ACPI but contained only generic macros for the northbridge and southbridge. An effort to improve the ACPI implementation and routing information into the DSDT table failed. For

this, the DSDT table written by the vendor BIOS was decompiled using the IASL compiler and the MP table then recompiled into the boot loader source code.

General configuration options for Coreboot

The general configuration menu (`make menuconfig`) for Coreboot and Seabios was a smorgasbord for the optimising firmware engineer. Coreboot and Seabios were configured to include a minimum set of features for booting the target hardware. Apart from the `$PIR` table, the MP table and ACPI tables, Coreboot was instructed to generate tables for GRUB2[5]. Since the table generation was done at compile time, it was believed not to impact performance of the boot loader. A generic PCI to SATA bridge driver was included (the Silicon Image `SIL3114`). Seabios was selected as the payload in Coreboot and compressed using LZMA. It was believed that the fast CPU would manage decompression in parallel with SPI read operations.

For Seabios, much functionality was included. Basic ATA controllers (for PATA/IDE drives) were included to interface with the main hard drive, but direct memory access for the ATA controllers did not function properly and was excluded. AHCI controllers were included in some builds (the SATA/ATAPI interface was required to install an operating system from secondary storage). OHCI, UHCI and EHCI controllers were added to enable USB support at boot time. USB support was deemed necessary to enable USB keyboards (for boot menus) and USB device booting. Among unnecessary interfaces initialised by the vendor BIOS, floppy support, PS/2 keyboard and mouse support and parallel port support was excluded. As seen in figure 22, a boot menu was configured and a splash screen was added, displaying a CrossControl logo and the phrase "Press F12 for boot menu". In the streamlined version of the boot loader tested in section 4.2.4, the bootsplash was *not* included as the boot menu option was removed. In this version, the USB controllers were also *not* included; the operating system later initialised the USB bus and allowed USB peripherals to function normally.

Serial port console output in both Coreboot and Seabios was set to COM1/`ttyS0` at I/O port `0x3F8` (the default, corresponding to the super I/O configuration space parameters). The debug level ranged from zero to eight, where zero meant no debug messages and eight was the maximum number of messages. During development this level was set to eight. During experiments (see section 4.2.4),

[5]GRUB2 or *grand unified boot loader* 2.0, is a second stage boot loader located at the MBR. Designed for Linux, it gives the user a choice between booting multiple operating systems.

Figure 22: Image added to the boot loader. If enabled, the image was shown for a few seconds to allow the user to select the target boot device.

a streamlined version of the boot loader was used. The debug level was set to minimum to minimise serial I/O delays and the baud rate was maximised (115200 baud) for this reason.

4.2.3 Installation and configuration the video driver

The Intel VBIOS was integrated into the CBFS image for execution under Seabios. The Intel embedded graphics driver (IEGD) OEM and end-user license [46] explicitly granted the rights to *" use and copy Software internally for Your own development and maintenance purposes"* provided the license applies to the copy or derivative work. Extracting the VBIOS binary from the vendor BIOS could therefore be done. In the CBFS, the video driver file was by convention named as "`pci`VendorID`,`DeviceID`.rom`", so in this case it was named "`pci8086,27a2.rom`".

The procedure was performed with an hex editor, though automated tool for option ROM extractions were also available. The BIOS image was uncompressed so the default PCI option ROM header could be identified (see section 2.1.4), with header fields for vendor and device identifiers. For Intel, the vendor ID was 8086 (`0x86,0x80`) and the device ID was 27a2 (`0xA2,0x27`) for the ICH7 graphics controller. The entire video driver file was found to be 64 kB in size.

Another method involved reading the VBIOS directly from memory after booting with the vendor BIOS. While the video driver was loaded in memory, it could be extracted and saved to a file. Under GNU/Linux, the command `dd` was be used for this purpose. If the vendor BIOS would have made any changes to the VBIOS, such as configuration by editing register values in memory, this method could have provided a mean of capturing these. The two methods produced identical VBIOS files for inclusion in the CBFS image.

Implementing system calls for VBIOS

At minimum, the correct boot display device and configuration identifier had to be supplied to the VBIOS by the firmware or the VBIOS would initialise the default settings. In the case of the ETX-CD, this restricted the signal output to the analog video port (VGA out). Therefore, efforts were made to instruct the VBIOS to output a correct video signal on the flat panel (LVDS) port.

The hooks and requests by the VBIOS were observed during boot and implemented in Seabios. `INT 15h` was hooked in the IDT by Seabios. All `INT 15h` calls were therefore routed to functions in `vgahooks.c`, an interrupt service routine. An example of such a function is shown in figure 23. Here, the VBIOS requests the configuration identifier. These identifiers were also fixed in Seabios at compile time and had to be changed to match the configuration.

While the interface was known, the data the vendor BIOS provided to the VBIOS was not. It was assumed that the VBIOS had been preconfigured at compile time and that only the correct configuration identifier and boot display device needed to be supplied. Therefore, an x86 assembly program was written to run under MS-DOS and query the VBIOS for settings. The method was straight-forward; the *Get Configuration ID* interrupt was asserted by writing registers AX=`0x5F61`, BX=`0x05` and executing `INT 15h`. If the VBIOS ISR handled the IRQ properly, then the AH (return) register would contain AH=`0x15` [46, p. 230]. Unfortunately, it was discovered through the assembly program that the VBIOS did not respond to any interrupt calls after the initalisation phase. Therefore, the VBIOS could not be configured properly.

The IEGD legacy VBIOS build environment, described in section 2.3.2, was used to construct a custom VBIOS. This approach allowed configuration of settings manually. As such, it required

detailed and complete information about the video configuration. No custom VBIOS tested with the boot loader initialised any of the video ports correctly.

Reset variants based on SDRAM initialisation status

Upon a cold boot, the SDRAM memory circuits attached to the ETX-CD module needed to be fully initialised before safe operation could continue. The status of the SDRAM was signaled by the `SLP S4#` (SDRAM assertion width) signal, a register in the northbridge. Figure 24 shows that if the `SLP S4#` is set wrong, the code would softly reset the COM module by writing `0x00,0x0E` to the reset control register (RSTC). It was noted through tests that this reset instruction was implemented with a five second delay in hardware, allowing for the SDRAM to initialise fully. If the reset was not invoked then the system would behave unreliably and occationally crash.

```
static void intel_155f40(struct bregs *regs) {
  dprintf(1,"Intel 5F40h call: Set Profile ID.");
  regs->ax = 0x005f; /* Supported */
  regs->cl = 0x01; /* Profile ID */
  set_success(regs);
}
```

Figure 23: Example implementation of interrupt `INT 15h` call `0x5F40`. Here, the VBIOS has requested the configuration identifier from the author's version of Seabios, which returns profile ID `0x01`. The register values are kept in the struct `regs`, while `dprintf` function prints out the debug message on the serial port. Seabios has previously hooked the interrupt `INT 15h` in the IDT, so that the CPU invokes the correct handler when a `15h` interrupt is asserted.

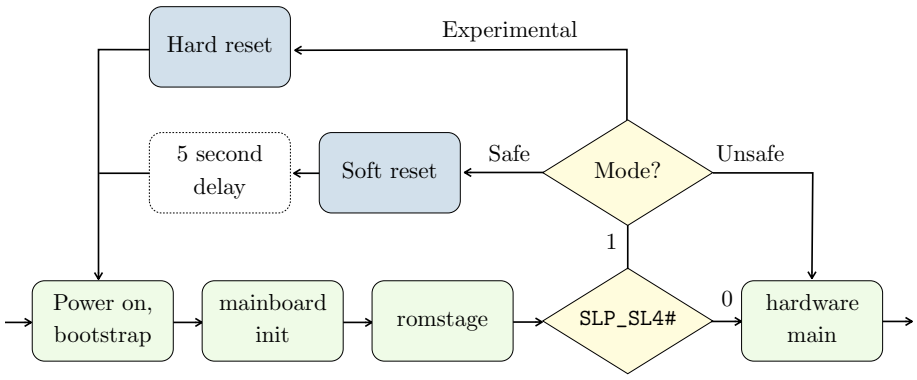


Figure 24: Flowchart of boot loader behaviour after SDRAM initialisation. The SDRAM error detection function and soft reset invocation were implemented in the RAM init initialisation code. The `SLP S4#` northbridge register corresponds to the memory circuit status. If this bit is set to one, then the SDRAM has not been successfully initialised. By default, a soft reset in invoked by writing `0x00, 0x0E` to the RSTC at I/O port `0xCF9`. However, by writing `0x02,0x06` instead, a hard reset is induced. Doing neither will result in an unstable system.

The first prototype implemented the soft reboot scheme. To optimise the boot time, another version of the boot loader prototype was constructed. In the other prototype, the author implemented a hard reboot scheme. This was forced by writing `0x02,0x06` to the RSTC instead. The first bit sets the *hard reset* option, while the second bit initiates the reset itself. This scheme caused the prototype to reboot immediatly multiple times in a loop. When `SLP S4#` bit was asserted, the system exited the loop and booted normally, often after a single reboot.

The `SLP S4#` signal and the RSTC are likely described in the Intel `945GM` specifications, which were not publicly available from Intel. The implementation was instead based on reviewing source code of Coreboot (as shown in figure 24) and the *FreeBSD* kernel `cpu_reset_real()` reset function [59], as well as the Intel `SCH` specifications [60, p. 69], a later chipset. Nevertheless, the procedure was successfully performed.

4.2.4 Experimental setup – measuring boot times

Following the construction of the boot loader, streamlined versions were developed. These boot loaders only implemented the necessary functionality to boot a GNU/Linux system from the ATA drive, but also included a VBIOS option ROM. The duration for the CCpilot XL hardware initialisation was measured for different types and COM module firmware versions. Because it is difficult at best for a computer to measure the duration it takes for itself to boot, the experimental setup instead involved two identical XL 3.0 computers. The setup is depicted in figure 25.

An experiment was conducted as follows: The first machine signaled the power supply to turn on power to the second machine via the USB serial control line. This command was logged by the system timer. As the second machine powered up, the firmware output short status messages or codes on the serial data line. These codes were logged by the first computer and its system timer. The process was controlled by the user via the keyboard and display. The second machine was also monitored via its flat panel display. In the vendor BIOS case, the Kontron *ETX-port 80* debug card was placed between the base board and the COM module on the second XL 3.0. The serial data line was then between the

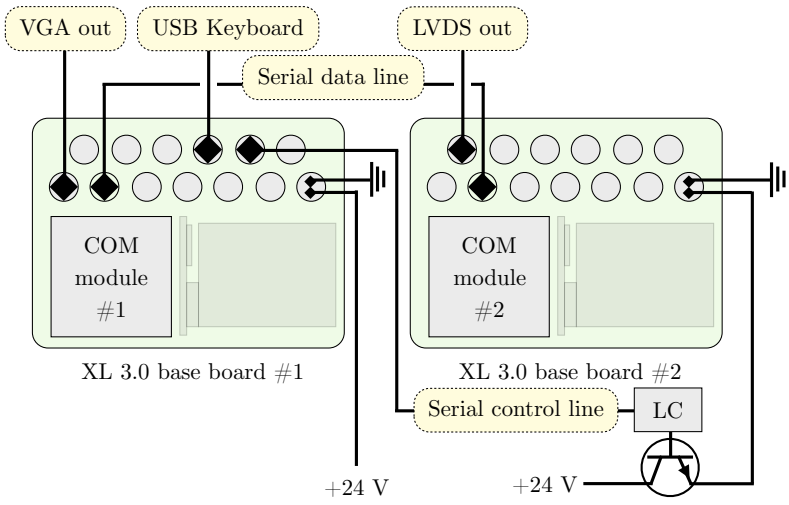


Figure 25: Experimental setup used to measure the XL 3.0, ETX-CD hardware initialisation durations. The second machine measures the bootstrap time of the first by controlling the power supply and reading debug messages on the serial data line.

first XL 3.0 and the debug card. The results of the experiment are described in section 5.

4.3 Programming attempts for other platforms

This section details the two attempts made to build and evaluate custom firmware for other platforms.

4.3.1 Intel BLDK on Conga-CA6 module

The Intel BLDK was intended to be evaluated on the Conga-CA6 module provided by Congatec. This module featured the specific Atom E6xx-series processor which was supported by the BLDK. A firmware candidate was built using the Intel BLDK. However, all attempts at establishing a reliable method for updating the firmware on the platform failed.

The firmware on the Conga-CA6 was stored on a `SST25VF032B` 4 MB SOIC ROM. The pinout and functionality of this SOIC was identical to the `AT26DF081A` (figure 21 (a), page 36). However, all attempts at in-system programming failed; all commands returned zero output from the SOIC. To rule out SOIC malfunction, it was exchanged for a ST `M25PE32` 4 MB SOIC flash ROM, a functionally equivalent circuit, but read operations still failed. It was concluded that nearby circuitry on the conga-CA6 board interfered with the in-system programmer.

The boot ROM was isolated through an external socket. The socket had short wires soldered onto the conga-CA6 board, but could also be disconnected. Reading and programming the ROM in the socket independently was successful when using the in-system programmer at 8 KHz . However, the conga-CA6 module could not read the firmware from the socket at 20 MHz. The clock signal stabilised and the module booted when observing the signals using an external oscilloscope. A theory to this effect is that the cables acted as bypass capacitors, causing a drop in signal-to-noise ratio. For this reason, the socket was equipped with bypass capacitors from VCC and CLK pins, as shown in figure 26. This enabled the module to *occationally* read the firmware and boot. However, the socket was unstable and considered unreliable due to the risk of damage to the hardware. The soldering of this socket was performed by the author, while the soldering on the conga-CA6 was performed by a CrossControl specialist.

Holding the chipset in reset was theorised to enable in-system programming. When the chipset reset signal pulled low, the pins that connect to the SOIC are high impedance inputs, in theory dis-

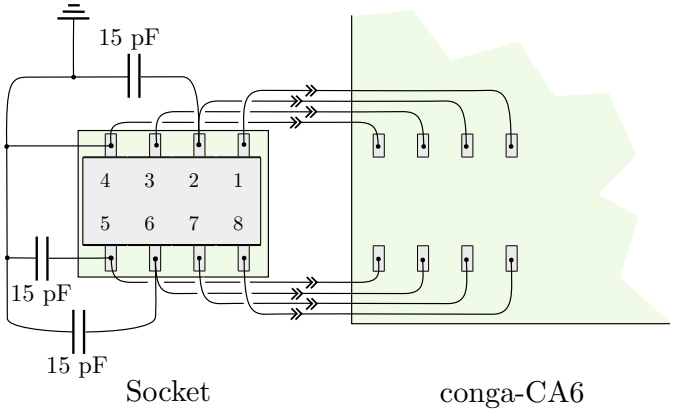


Figure 26: Schematic of the socket constructed by the author for reading and writing to the `M25PE32` SOIC. The bypass capacitors act as low-pass filters and clean up the signals.

connecting the circuit from the board. Pin B49 in the COM Express connector is such a signal – grounding it should have enabled for the SOIC to be reliably in-system programmed. However, this approach was also unsuccessful. Further investigation revealed that the clock signal generated by the board had become heavily distorted, rendering the board virtually inoperable. All programming attempts on this board were therefore aborted.

4.3.2 Coreboot+Seabios on the Kontron Nanoetxexpress-SP module

Utilising the experiences gained during the ETX-CD port of Coreboot, another Coreboot port was attempted for the secondary hardware, the Kontron Nanoetxexpress-SP (described on page 29). The source code for the *iWave RainboW-G6* port of Coreboot was utilised as a base. The G6 board had the same `US15W` chipset as the target (Intel Atom processor with Intel System Controller Hub (SCH) *Poulsbo*). Seabios was selected as payload and its configuration was identical to that of the ETX-CD port.

Chipset microcode (CMC)

The Intel SCH `US15W` chipet requires a 64 kB chipset microcode (CMC) segment to initialise the southbridge before the main processor starts executing instructions. According to the `US15W` specification [60], the location of this code segment is, in physical memory, `0xFFF*0000`, where `*` is either `B`, `C`, `D` (default) or `E`. The exact location is determined by a hardwired signal combination of the `GPIO3` and `GPIO0` pins to the `US15W` chip.

Upon a power on event, the 1024 kB BIOS image is copied to the upper bound of the addressable memory. For the `US15W`, this is the top 1024 kB of the 4 GB addressable space, or `0xFFF00000-0xFFFFFFFF`. Thus, assuming default configuration of the GPIO pins, the CMC segment is located in the vendor BIOS image at the range `0xD0000-0xDFFFF`. The segment was therefore extracted using a hex editor. The CBFS tool then included this segment in the Coreboot image at the exact same location.

Debugging

The Kontron Nanoetxexpress-SP was not equipped with a SIO chip, so debugging the boot loader via the serial port was impossible. To enable serial output, the module was placed on a Kontron Nanoetxexpress-HMI baseboard [61], a COM express carrier and test board. This board was equipped with a Winbond W83627HG SIO chip. Support for the this chip was added to the device tree configuration file (`devicetree.db`) and code to support it was added to the ROM stage source (`romstage.c`).

Overwriting the firmware

The boot ROM was found to be a TSOP circuit, the `SST 25LF008A` 8 Mbit Firmware Hub. This circuit has a larger footprint than the SOIC and a pin count of 32. In-system programming of the TSOP was not possible as the available tools did not support the FWH protocol.

The first attempt to write the Coreboot image to the TSOP by means of software (Flashrom) failed. The software failed to put the chip in write-enabled mode, causing the erase and write operations to fail. By connecting the write protection pin (`WP#`) to the voltage supply pin V_{DD}, write protection was

lifted. Erase operations were successful but writing the firmware back into the TSOP failed. Reading back the contents of the TSOP provided 1024 bytes of `0xFF`.

A second attempt was made with an older revision of the Nanoetxexpress-SP module. For this board a specific version of Flashrom that was known to be working with the board revision was used. A second TSOP circuit (*SST2*) was placed on top of the original TSOP (*SST1*) and a switching device constructed. The advantages of such a construction were twofold. First, a switch selected the active TSOP. This enabled the user to *hotswap* TSOP circuits while the system was running – effectively booting from one and writing to another through software. Secondly, accidental erasure and overwriting of the vendor BIOS was made impossible by pulling `SST1_WP#` low (connecting it to ground). This implied that there was no risk of ending up with a non-functional board.

Most of the pins on the top TSOP were connected to their counterparts on the lower TSOP. Notable exceptions include the chip selection pins `INIT#`, the voltage and ground pins `VDD` and `VSS` as well as the write protection pins `WP#` and `TBL#`. Three pins were also lifted from the board to prevent possible short-circuits of the chipset. The circuit diagram of the final construction is shown in figure 27. The soldering was instructed by the author and performed by a CrossControl specialist. The circuit worked as intended and enabled the author to reprogram the firmware. However, an electrostatic accident occured during development of the boot loader and rendered the hardware inoperable. Therefore the completed boot loader candidate program could not be properly evaluated. The accident occured despite ESD protection measures such as a grounded ESD-protection mat on the work table. An ESD-protection work floor was used with conductive ESD slippers to ground the author. Due to the time required for soldering the TSOP hotswap construction, the cost of the soldering specialist and the module, no further attempts were made.

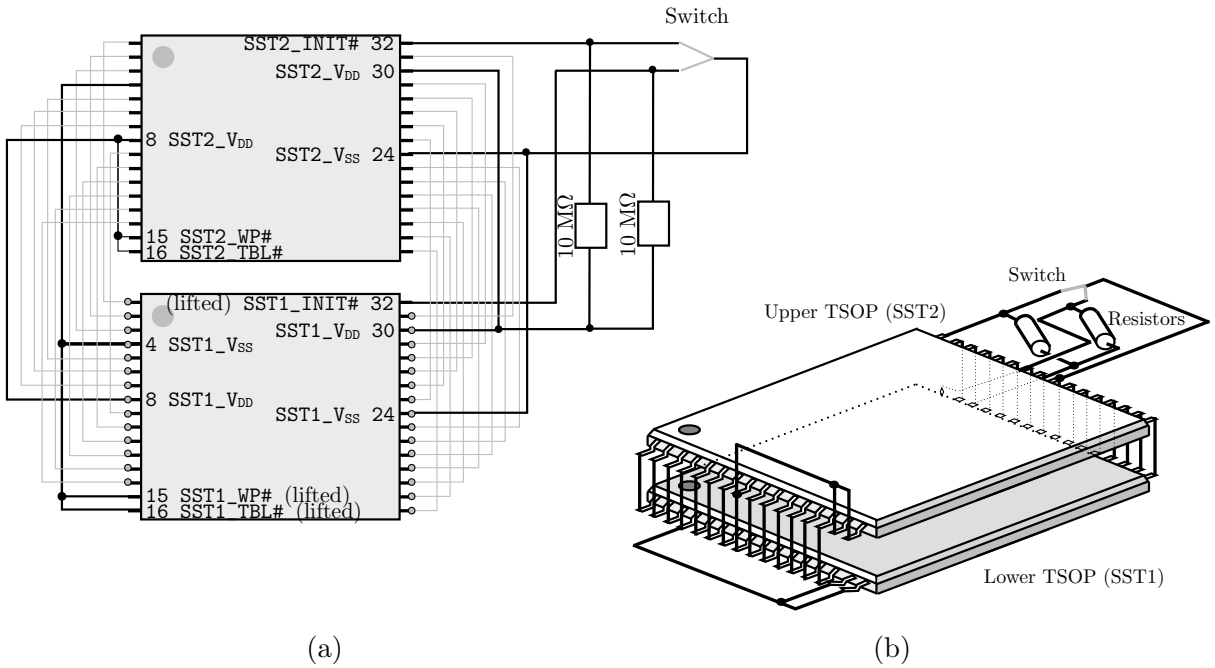


Figure 27: Construction for hotswapping of TSOP firmware circuits on the Kontron Nanoetxexpress-SP. (a) Circuit diagram. Only important pins are marked. Straight connections are marked with grey to ease visual reference. Grey dots on the lower SST1 circuit indicate which legs are soldered to the board. (b) Image that indicates how the TSOP circuits are placed on top of each other.

5 Results

This section presents the properties of the prototype boot loader that was constructed for the ETX-CD. Prototypes for the Kontron Nanoetxexpress-SP and as the Conga-CA6 modules were also constructed. However, due to the circumstances described in sections 4.3.2 and 4.3.1, these prototypes were never successfully tested on real hardware and their properties remain unknown.

CCpilot XL 3.0 boot loader prototype

Two Coreboot+Seabios boot loader prototypes was constructed for the CCpilot XL 3.0 platform (CCpilot XL 3.0 baseboard and Kontron ETX-CD module). Figure 24 on page 40 describes the difference between these variants. One variant implemented the safe reset scheme, while the other implemented the experimental reset scheme. The firmware had the default legacy BIOS interface and supported the default legacy BIOS interrupt calls. Both prototypes successfully initialised the hardware platform.

Initialisation time experiement results

This section presents the results of the firmware experiments described in section 4.2.4. Three different firmware types were tested on the Kontron ETX-CD module, namely:

1. Coreboot v4, with safe reboot scheme on `SLP_SL4#` signal, with Seabios as payload.
2. Coreboot v4, with experimental reboot scheme on `SLP_SL4#` signal, with Seabios as payload.
3. Phoenix TrustedCore BIOS by Kontron Embedded version MCALR912 (the *vendor BIOS*).

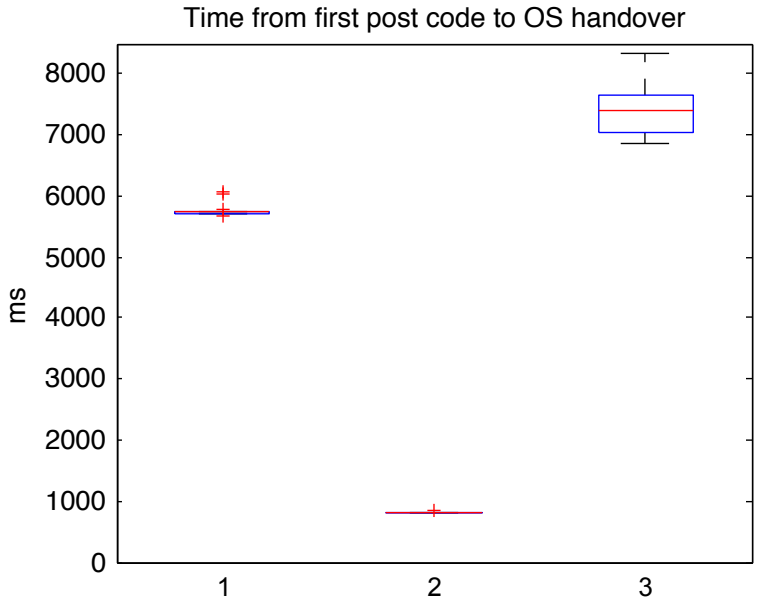


Figure 28: Box-and-whisker diagram of module initialisation time for the different firmware types on the Kontron ETX-CD. (1) Coreboot+Seabios in safe reboot scheme (2) Coreboot+Seabios in experimental reboot scheme (3) Phoenix TrustedCore BIOS by Kontron Embedded version MCALR912.

Firmware	*Mean time* (ms)	*Standard deviation* (ms)
1. Coreboot+Seabios (safe)	5751	92
2. Coreboot+Seabios (experimental)	818	20
3. Vendor BIOS	7396	394

Table 4: Mean module initialisation time for the different firmware types.

The duration between power on and the first received POST code is defined as the *base board initialisation time*. The following duration up until the firmware handover to the operating system is defined as *the module initialisation time*. The module initialisation durations for the three different firmware types are listed in table 4. The complete data set is provided in appendix A on page 58. Box plots generated using the MATLAB statistics package on the data set are shown in figure 28. Clearly, the Coreboot boot loaders obtain faster initialisation time.

The base board initialisation time was determined to be 3558 ± 110 ms when the flat panel display was connected and 7548 ± 97 ms otherwise. No dependency on COM module firmware type or version was suggested for this duration, but at the same time cannot be ruled out.

Noise sources in this experiment include scheduling latencies in the host machine operating system and physical processes in the electronics. The commands were sent via a serial connection, which had a very limited bandwidth and high latency. The mean duration is therefore preferred when presenting the data.

Effects on system operation

Changing the firmware effected both the operating system and the entire system operation. Different installations of Windows XP service pack 3 (SP3) and Debian GNU/Linux 5.0.8 were tested on the sysem before and after the Coreboot+Seabios boot loader was installed.

One installation of Windows XP with service pack 2 (SP2) and 3 (SP3) loaded normally but seemed to operate slower. After boot, the USB drivers would be reinstalled and peripherals would not work until the machine had rebooted at least once after the firmware change. Another installation of Windows XP SP3 on a different, but identical, XL 3.0 computer would not function after the firmware change. An new installation Windows XP (SP2) was made on a clean system with the boot loader. The computer would then boot without issues. However, after installing some board-specific Windows drivers, the system would hang at startup. This points to an unresolved conflict with the firmware configuration and one of the device drivers used.

Debian GNU/Linux loaded normally after the firmware change. No unwanted side effects were noted under GNU/Linux. However, no special drivers for GNU/Linux were tested. Regardless of the operating system being used, the colour depth on the display was in error, showing the default 24-bit depth rather than 18-bit depth. This caused colors to appear incorrectly. Neither operating system detected nor corrected this problem.

The machine could not reboot itself with the boot loader firmware installed. Instead, it would stop and freeze when the operating system attempted a reboot. The machine could power down as normal.

6 Discussion

This chapter contains an analysis by the author. First, firmware development in general is discussed in the lower-tier OEM context. The investigation has uncovered a number of firmware development options available to such OEMs. These alternatives and the necessary conditions for them are discussed in section 6.1. Section 6.2 discusses the boot loader prototype and on the conducted investigation itself, while final conclusions and recommendations are summarised in section 6.3.

6.1 Firmware development strategies

The investigation has made it clear that firmware engineering is a complex field that scales from theoretical aspects of computer science down to the finer points of hands-on electrical engineering. Firmware development requires know-how and experience in both these fields. Despite this, it has been shown that firmware engineering is possible even on lower-tier OEM levels under certain conditions. The constructed prototype boot loader is proof of concept thereof. However, the author cannot stress enough that the *proper documentation is required*. This includes data sheets and schematics of the chipsets, boards, standards, etc. If these resources are not found within the company, then good relations to the hardware and chipset vendors are required to gain access the necessary material.

Even when the knowledge and know-how is secured, practical complications during development will likely arise. Planning ahead and allocating the proper resources helps to avoid such difficulties. ESD protection measures as well as good soldering skills are required. Tools such as logic analysers and sockets for integrated circuits can help to alleviate any practical problems with the hardware.

To facilitate the discussion, note the relationships between the different parties in figure 29. The *chipset maker* sells chips and documentation to the *higher tier OEM*, which in turn sells boards to the *lower tier OEM*, which sells products to the end user. The traditional business model is depicted in the left side of the figure; the chipset maker sells documentation and source code to the *BIOS vendor*, which in turn sells the firmware or licenses the firmware source code to the higher level OEM. In contrast, an alternative ecosystem is proposed on the right side in the figure. Here, all tiers interact within the open source community. For example, this model is applicable to both the Coreboot and the Tianocore projects – AMD is supplying code and documentation to Coreboot while Intel is supplying both closed libraries and open source code to the Tianocore project. Note also that this model also allows academia to contribute and participate in applied firmware research.

This section contains a discussion of firmware strategies in a lower-tier OEM firmware development context. Depending on the design requirements, the properties of each solution must be valued differently. For example, flexibility and extensibility might be prefered to save costs in driver development or bootstrap speed might be much more important. Some commercial alternatives to in-house development are also mentioned as cost often must be considered.

Unified extensible firmware interface

The UEFI standard replaces the legacy BIOS with a modern, partially open source firmware foundation. As noted in section 2.1.6, it will likely become the dominant x86 firmware type before the end of this decade. While UEFI is mostly aimed at chipset manufacturers and independent BIOS vendors, UEFI could also provide some flexibility for OEMs.

Control of the firmware configuration is key to utilising the UEFI interface to its maximum potential. As explained in setion 2.1.5, UEFI runtime services can still persist when the UEFI boot manager

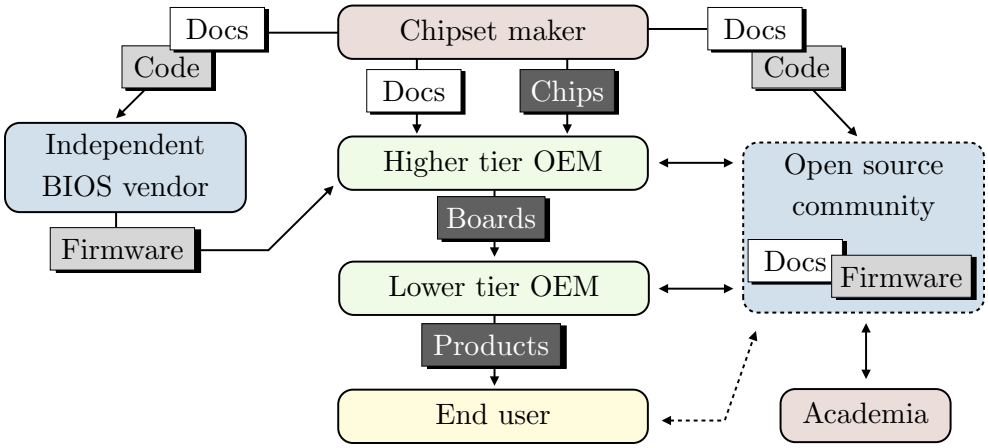


Figure 29: (Left side) Current firmware market model. In this model, the chipset maker and the BIOS vendor are controlling the firmware market. (Right side) Proposed open source-oriented firmware engineering ecosystem (Coreboot, Tianocore).

initiates the operating system loader. Suppose DXE drivers for specific components or devices used on OEM boards can be developed and added to the firmware. Then UEFI interfaces could facilitate runtime services as device drivers. Developing a single DXE driver might then be more cost-effective than developing the same device drivers for multiple operating systems. Note that the ability to add DXE drivers to the UEFI firmware is very important. For example, the firmware might be constructed on site using freely available tools such as the BLDK. If a vendor UEFI BIOS is used, it must be configurable to allow drivers to load from places other than the boot ROM, for example in NVRAM. Even though the UEFI interface allows for such mechanisms, they must also be implemented so that OEMs can make use of the them.

The PEI and DXE environments are primarily designed for extensibility and flexibility. This introduces an overhead which impacts performance negatively. For an embedded system, UEFI is simply put overkill. An UEFI solution is well suited when a full-fledged BIOS, rather than a simplistic boot loader, is prefered.

Intel boot loader development kit

Intel BLDK offers an interesting opportunity for OEMs to deploy UEFI boot loaders. Note the word *deploy*, because for all intents and purposes the development work has already been done by Intel. It is therefore relatively easy to construct a boot loader using the BLDK, but harder to customise. PCI and other I/O devices need to be configured, as well as any necessary IRQ routing tables, etc. This allows the lower-tier OEM to deploy boot loaders with much less information. Though the solution comes in a pretty package, it is a more complicated process to edit the precompiled packages and customise the solution further. Out of the box, the BLDK is clearly designed for tiny embedded platforms and not full-featured, closed-box computers.

The BLDK is still a young product. To date the number of supported boards and operating systems is still very small. As support for more chipsets is added, it may well become an serious contender. If the target hardware and design requirements match the BLDK capabilities, it can be a very powerful tool. It should also be noted that the possible boot times with BLDK firmware are in the range of sub-two second [38], compared to the sub-one second boot time acheived by Coreboot.

Coreboot with payload

Coreboot is clearly an elegant and powerful open source bootstrap solution. Results in chapter 5 clearly show that Coreboot and Seabios can initialise a mainboard in less than a second. This is about half the time of the UEFI boot loaders developed by Insyde and described in Doran et al. [12] (though it should be noted here that different platforms were used in this comparison). The fact that Coreboot is written in C also makes the source code manageable and maintainable. However, there is a serious lack of updated and approachable documentation which associates the Coreboot project with a rather steep learning curve.

While Coreboot supports a large number of boards, most of them tend to be of older design. It is reasonable to assume that an OEM would prefer to user newer chipsets in embedded designs. The next generation hardware usually offers better performance and power consumption efficiency. Naturally, chipset makers will continue to provide independent BIOS vendors with detailed documentation of new chipsets. The Coreboot community often lacks this prerogative, so adding support for new boards can be very tedious. Moving forward, the author predicts that chipset support in Coreboot will likely continue to lag behind. Developing boot loaders with Coreboot becomes a more viable option if the OEM is working with long time support (LTS) boards. These are boards supported and manufactured by the board vendor for an extended period of time. That time could be utilised for open source boot loader development. In fact, the Kontron ETX-CD was an LTS board. It should be noted, though, that the author had the luxury of selecting the target hardware based only on firmware requirements.

The choice of chipset vendor is also important. AMD Embedded Solutions provides documentation freely and actively contributes with AGESA code to the Coreboot project. On the other hand, Intel Embedded is extremely secretive about chipset documentation. If an OEM would pursue development of open source boot loaders then working with AMD chipsets would clearly be preferable. Working together with the hardware vendor is also an option. Unfortunately, the vendor might be opposed to the open source license of the Coreboot code. There is little incentive for a company to participate if competitors sell boards on which the developed boot loader can also be applied.

Coreboot only becomes a complete firmware solution when used in conjunction with a payload. Seabios is a capable and efficient implementation of the legacy BIOS standard. Though not tested explicitly by the author, the FILO payload introduces an interesting way to load a Linux kernel directly into memory. Clearly there are numerous ways to load the operating system if the hardware is supported by Coreboot. The difficulties lie in supplying the correct IRQ routing tables and ACPI tables – if these are incorrect it does not matter how robust the underlying firmware is.

Last, but not least, it should be noted that Coreboot is free firmware – both in terms of freedom and of cost. A commercial BIOS solution is often associated with a royalty fee. For high-volume products, the potential cost savings of utilising Coreboot could be rather large.

Coreboot and UEFI

There currently exists no complete open source UEFI solution. Figure 30 shows how an open source, UEFI-compliant firmware alternative could be architected with the help of Coreboot and Tianocore. Here, a glue code segment converts the CBFS tables and interface that describe the system state into a PI-compatible hand off blocks (HOBs). The DXE core would then assume computer control as if a closed source PI implementation had initialised the hardware. Some work has already been made in this direction, such as the works of Austin [62] and Schulz [63]. However, these projects are currently unfinished and have not been released to the public. Further research in this area is recommended.

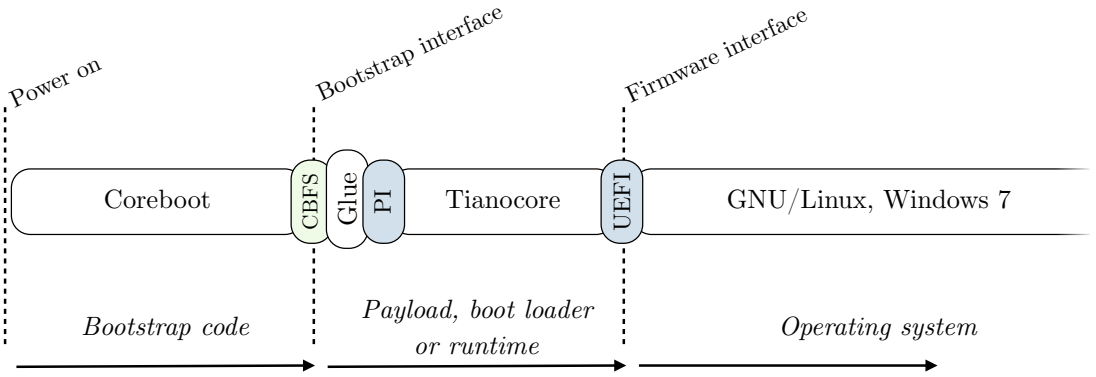


Figure 30: A theoretical open source UEFI-compliant firmware alternative: Coreboot and Tianocore combination using glue code. If the CBFS tables and interface could be made into PI-compatible HOBs, then Tianocore could run after hardware initialisation by Coreboot.

Commercial alternatives

Up until this point, this text concerns with non-commercial alternatives as defined in section 1.3 on page 3. To provide a complete frame of reference, commercial alternatives as a whole are briefly discussed in this section. Commercial alternatives can be considered by the lower tier OEM if in-house development of firmware have been ruled out for any number of reasons. There are two main types of commercial alternatives:

1. *Proprietary, commercial alternatives.* Keeping with the traditional model of the market, a firmware solution is purchased from another BIOS vendor. It is also conceivable that the lower tier OEM purchases an optimised version from the original BIOS vendor. Most likely the customer is supplied with a legacy BIOS or UEFI BIOS that is tweaked for the target hardware. The author can only speculate on the technical properties of such a solution. The downside of this approach is that the OEM will still be dependent on a BIOS vendor for support and maintenance.

2. *Open source, commercial alternatives.* In this scenario, consultants or companies develop open source firmware on behalf of the lower-tier OEM. Such a solution would likely be based on Coreboot or Tianocore. The advantages of this approach is that it supports the proposed open source firmware ecosystem. It also allows the OEM to perform firmware maintenance tasks in-house without the burden of firmware development. Note further that this is a self-improving process due to the feedback of source code, board support and know-how back into the open source community. However, the necessary chipset and board documentation must be procured for this approach to be realistic.

Which alternative (commercial or non-commercial, closed or open source) is the better choice will ultimately be a product many factors for the lower-tier OEM. These factors include (but are not limited to) the firmware design requirements, the properties of the target hardware, the cost associated with each solution, available documentation and resources and also the management ideology.

6.2 Comments on method and results

In this section, the practical aspects and the results of the performed work are discussed.

ETX-CD boot loader prototype

Chapter 5 shows that a working version of the firmware for the selected platform was successfully implemented. The firmware managed to bring up both GNU/Linux and a fresh install of Windows XP without any issues. The platform initialisation time experiment in section 4.2.4, with results in chapter 5, shows that Coreboot+Seabios is fast and efficient. It must be viewed as extraordinary that sub one-second COM module initialisation times were achieved. When the boot splash screen was included, it was displayed for a much longer duration. It must also be noted that the prototype could be constructed as open code packages for the `945GME` and `ICH7` components were available. The fact that open source components were utilisable and could be tailored for the specific board and system must also be viewed to hold academic value.

The fast boot times could be achieved as the BIOS optimisation schemes detailed in section 2.2 were indirectly utilised. First, an alternative firmware platform was selected with a more efficient code base. Secondly, the configuration was optimised to remove all actions performed by the boot loader not required to bring up the system. It can also be seen that further optimisations are possible. The video BIOS could be removed to save extra time; the OS could initialise video interface instead. The base board could be instructed to simply light up or show a splash picture on the display using LVDS signaling upon power on. The SDRAM type could also be preconfigured in advance; however this would restrict the SDRAM circuits used to a single type. The boot device could be also be set in advance, though Seabios boot device selection is already much more efficient than the examined vendor BIOS. However, the author must conclude that 0.8 seconds is *"fast enough"*.

There were, however, a few issues with the boot loader prototype. The hang-up problems encountered when evaluating the boot loader were probably caused by the DSDT table or the IRQ routing tables. The data supplied in one or some of these tables were likely not completely accurate. Specifically, the failure to reboot the system points to ACPI implementation errors. As the ACPI implementation is chipset specific, better documentation could be required to solve these problems.

The video driver was another source of headache as the flat panel display interface was never successfully initialised by the boot loader. When the operating system initialised it at a later stage in the bootstrap process the color depth was incorrectly configured. A better understanding of the video driver workings would be required to remedy the problems. During this project, the author did not anticipate or plan for in-house video driver development. It is likely that in-house LVDS video driver development could have resulted in a thesis by itself. Further research into the relationships of LVDS, the IEGD software and the video drivers for GNU/Linux and Windows is therefore recommended.

The practical aspects of working with the hardware must also be mentioned. Though in-system programming (ISP) of the SOIC boot ROM on the ETX-CD was successfully performed due to fortunate circumstances. Compared with the programming attempts of the Nanoetxexpress-SP and the Conga-CA6, one must conclude that one cannot always be so lucky. The possible complications are numerous and innovative solutions must be applied. The constructed TSOP hot-swap switch depicted in figure 27 on page 44 is an example of such an innovative solution.

Comments on experimental setup

Utilising the serial port was an interesting way of measuring the bootstrap times. However, I/O operations were slow. Outputting the messages on the serial port certainly influenced the initialisation time. When the debug level was set to maximum, several seconds would be added to the boot time. While the experiment was computer controlled, scheduling latencies in the operating system were also apparent. Noise sources in the experiment include these I/O and scheduling latencies in the host computer. Assume that the identical actions performed by the boot loader took exactly the same amount of time during each trial. Further assume that the combined measurement noise follows a stationary distribution. Then it follows that the the standard deviation caused by measurement noise is on the order of 20 ms.

There is, however, a rather large standard deviation in the vendor BIOS module inialisation times (394 ms). While these noise sources plausibly explain deviations in the Coreboot+Seabios case, they cannot explain the large deviation when booting with the vendor BIOS. The real cause for this deviation is most likely hidden in the vendor BIOS implementation.

A more accurate approach was also considered. The idea was to edit the `INT 19h` ISR in Seabios to output the value of the time stamp counter (TSC) register divided by the CPU clock speed. This would have provided a measurement accuracy on the order of nanoseconds. Unfortunately, the CPU frequency was not held constant during the bootstrap process and there was no apparent way to trace the CPU clock speed during the bootstrap process. Nevertheless, the trials conducted with the serial connection provided clear and statistically significant results.

Method self-critisism and defence

The methodology used during the project is described by figure 18 on page 32. The understanding of the problem grew as more information became available. Executing practice track in parallel with the theory track might not have been optimal. It is possible that the analysis should have completely preceeded the practical aspects of the work. Whether or not this alternative approach would have been more successful is purely speculative. The practical aspects of the work in parallel to the theory helped the author to better understand the concepts involved. However, specific knowledge was sometimes discovered which could have been successfully utilised in the practice track at an earlier point in time.

Parts of the practical work can also be attributed with a slight lack of focus. For instance, the author decided to attempt boot loaders implementations on several platforms. These attempts failed due to practical circumstances. Focus could instead have been given to finialising and perfecting the working prototype boot loader. In particular the IRQ routing and ACPI configuration tables were in need of attention. However, the conclusions were not only drawn from the successes but also from these failures. Although not perfect, the prototype boot loader was already proof of concept. More knowledge was thought to be gained through attempting implementation of other platforms with other tools.

6.3 Conclusions and recommendations

This text has shown that firmware engineering is a complex subject that stretches beyond mere concepts. Complexity arises due to realisation of these abstractions into standards. The field is characterised by a great mishmash of industrial standards of different scope and origin. The intricacy is further extended by the various implementations of the standards. Access to hardware documentation

is seldom given, yet access is crucial to facilitate efficient firmware engineering. If the available codebase does not support the desired target hardware, then the proper documentation must be obtained.

From the practical avenue it can be concluded that the prototype development was mostly successful. The prototype could be constructed as existing, open source code was available. This code was utilisable and could be tailored for the specific purpose. There were still issues with the configuration tables and video drivers that need attention. Still, the prototype is working proof of concept in its own and also proof that lower-tier OEM firmware development is possible.

Recommendations for CrossControl

CrossControl AB initiated this project to investigate and evaluate the role of the x86 firmware in their products. As CrossControl is a lower-tier OEM, the discussion in section 6.1 can be directly applied to the company. There are four main alternatives available:

1. **In-house firmware development.** This text has shown that in-house firmware development is a possibility under the right circumstances: know-how, documentation and practical resources.

   (a) *Open source development.* Open source boot loaders can be developed if the target platform is already supported by the Coreboot source tree. Support from the higher-tier OEM (in this case Kontron or Congatec) is likely required or at least very benefitial. If the chipset is not supported, then there *must* exist in-house knowledge and documentation of the chipsets.

   (b) *Closed or semi-open source development.* If the target platform is a newer Intel Atom processor supported by the BLDK, this is certainly a viable option. At present, no other realistic options exist in this category.

2. **Purchasing alternative firmware.** If the available in-house resources are not found to be sufficient, alternative firmware can be obtained from other sources.

   (a) *Open source development.* Open source consultants should be considered to allow CrossControl to maintain the developed source code *and* continue to build on firmware engineering practices and know-how.

   (b) *Closed source development.* If all the above options have been ruled out, commercial alternatives should be considered. Special care must be taken to ensure that the alternative is indeed a better one at a lower cost.

3. **Doing nothing.** This is the fallback position when the above four alternatives have been ruled out for some reason.

Ultimately, CrossControl management must decide which alternative is best suited for each product use case. Firmware requirements, estimated development costs, potential cost savings, license fees, available in-house resources and ideology will factor into this investment appraisal equation.

In the long run, lower-tier OEMs such as CrossControl would benefit from the alternative, open-source firmware engineering ecosystem seen in figure 29 on page 48. It is the author's opinion that CrossControl has much to gain by cooperating with the open source community, provided that circumstances allow it. If a fully featured UEFI BIOS is prefered, then further research into solutions

based on Tianocore is recommended. However, if boot speed is prefered and the legacy BIOS interface deemed sufficient, then working with AMD chipsets and Coreboot+Seabios firmware is recommended. AMD is the only major chipset maker which currently is providing code and documentation to the open source community. AMD processors and chipsets are therefore most likely to be supported by Coreboot moving forward.

Lastly, note that the CCpilot XL 3.0 base board initialisation duration was measured to be 3558 ± 110 ms (page 46). An effort to reduce this duration should also be made by CrossControl.

References

[1] Tim Hockin. Linux, Open Source, and System Bring-Up Tools. In *Proceedings of the Linux Symposium*, volume 1, 2008.

[2] N. Zhang, J. Ma, J. Chen, and T. Chen. SPM-Based Boot Loader. In *Embedded Software and Systems Symposia, 2008. ICESS Symposia '08. International Conference on*, pages 164 –168, july 2008.

[3] I. Sommerville. *Software Engineering*. Pearson Education, 9 edition, 2010.

[4] R. Gupta and A. Palod. Analysis of boot time reduction in Linux. Master's thesis, Malardalen University, 2009.

[5] W. L. Rosch. *Hardware Bible, Premier Edition*. Sams Publishing, 1997.

[6] A. S. Tanembaum. *Structured Computer Organization*. Pearson Prentice Hall, 2006.

[7] Jianwen Zheng, Xiaochao Li, and Donghui Guo. A novel multiboot framework for embedded system. In *Anti-counterfeiting, Security and Identification, 2008. ASID 2008. 2nd International Conference on*, pages 344 –347, aug. 2008.

[8] A. Marchiori and Qi Han. A Two-Stage Bootloader to Support Multi-application Deployment and Switching in Wireless Sensor Networks. In *Computational Science and Engineering, 2009. CSE '09. International Conference on*, volume 2, pages 71 –78, aug. 2009.

[9] Wei Xu and Yongjie Piao. Bootstrap loader design of aerospace payload controller based on TSC695F. In *Computational Intelligence and Natural Computing Proceedings (CINC), 2010 Second International Conference on*, volume 2, pages 60 –64, sept. 2010.

[10] Xiaofeng Wan, Hailin Hu, and Xiaojun Zhou. Analysis and Design of Boot Loader on Embedded Electric Power Steering System. In *Computing, Communication, Control, and Management, 2008. CCCM '08. ISECS International Colloquium on*, volume 2, pages 276 –280, aug. 2008.

[11] Geng Qingtian, Sun Zhanchen, Zhao Hongwei, and Gu Jianhao. The U-boot transplantation based on S3C2440. In *Mechatronic Science, Electric Engineering and Computer (MEC), 2011 International Conference on*, pages 2168 –2171, aug. 2011.

[12] M. Doran, K. D. Davis, and M Svancarek. UEFI Boot Time Optimization Under Windows 7. In *Intel Developer Forum 2009*, 2009.

[13] Mike Kartoz, Pete Dice, and Gabe Hattaway. Fastboot BIOS. White paper, Intel Corporation, September 2008.

[14] M. A. Rothman. Reducing Platform Boot Times – UEFI-based Performance Optimization. White paper, Intel Corporation, 2008.

[15] L. Dailey Paulson. New technology beefs up BIOS. *Computer*, 37(5):22, may 2004.

[16] A. Ahmed and O. Farook. Software-firmware design for 8088/8086 microprocessor based systems utilizing XT/AT compatible personal computers. In *Frontiers in Education Conference, 1988., Proceedings*, pages 448 –453, oct 1988.

[17] Sibert, O. and Porras, P.A. and Lindell, R. An analysis of the Intel 80x86 security architecture and implementations. *IEEE Transactions on Software Engineering*, 22(5):283 –293, may 1996.

[18] Y. Jiang, S. Zhang, and Y. Sun. Design and Study of GUI Based on EFI. *Jisuanji Gongcheng/ Computer Engineering*, 33(14):272–274, 2007.

[19] Haibo Wang, Shensheng Zhang, and Yuanhao Sun. NUWA: A new GUI solution for EFI/TIANO. *Jisuanji Yingyong yu Ruanjian / Computer Applications and Software.*, 25(6): 104–106, June 2008.

[20] Hua Feng, Wanqing Chi, and Yongpeng Liu. Designing and Developing Extensible Firmware Interface on IA-64 Platform. *Jisuanji Yingyong yu Ruanjian / Computer Applications and Software*, 28(1):167–169,215, January 2011.

[21] Rui Zhang, Jiqiang Liu, and Shuanghe Peng. A Trusted Bootstrap Scheme on EFI. In *Multimedia Information Networking and Security, 2009. MINES '09. International Conference on*, volume 1, pages 200 –204, nov. 2009.

[22] Weiwei Fang, Bingru Yang, Zheng Peng, and ZhiGang Tang. Research and Realization of Trusted Computing Platform Based on EFI. In *Electronic Commerce and Security, 2009. ISECS '09. Second International Symposium on*, volume 1, pages 43 –46, may 2009.

[23] The Coreboot project. *Coreboot*, retrieved August 30, 2011. URL `http://www.coreboot.org/`.

[24] Open Firmware Working Group. *Open Firmware Home Page*, retrieved October 26, 2011. URL `http://www.openfirmware.org`.

[25] DENX Software Engineering. *The Universal Bootloader*, retrieved October 26, 2011. URL `http://www.denx.de/wiki/U-Boot`.

[26] A. Silberschatz, G. Gagne, and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, 1995.

[27] S. Davidson and B.D. Shriver. An Overview of Firmware Engineering. *Computer*, 11(5):21–23, 1978.

[28] J. Catsoulis. *Designing embedded hardware*. O'Reilly Media, 2005.

[29] Matthew Tolentino. Linux in a Brave New Firmware Environment. In *Proceedings of the Linux Symposium*, volume 1, 2003.

[30] Gene Sally. *Pro Linux Embedded Systems*. Apress, 2010.

[31] V. Zimmer, M. Rothman, and S. Marisetty. *Beyond BIOS - Developing with the Unified Extensible Firmware Interface*. Intel Press, 2010.

[32] Jack G. Ganssle. *The Firmware Handbook*. Newnes, 2004.

[33] V. Zimmer, M. Rothman, and R. Hale. UEFI: From Reset Vector to Operating System. In *Hardware-Dependent Software: Principles and Practice*, 2009.

[34] D. Adhikary. Report on LinuxBIOS. Seminar, 2004.

[35] Oracle Corporation. *VirtualBox*, retrieved September 7, 2011. URL `http://www.virtualbox.org/`.

[36] Fabrice Bellard. *QEMU*, retrieved September 7, 2011. URL `http://wiki.qemu.org/Main_Page`.

[37] *Intel® Boot Loader Development Kit (Intel® BLDK) Version 2.0 – UEFI Standard Based – Getting Started Guide*. Intel Corporation, July 2011.

[38] D. Jensen and G. Hattaway. ABC's of the Intel® Boot Loader Development Kit. *EE Times*, July 2011.

[39] Tom Shanley and Don Anderson. *PCI System Architecture*. Addison Wesley, 1999.

[40] *PCI Local Bus specification*. PCI-SIG, December 1998.

[41] John Baldwin. PCI Interrupts for x86 Machines under FreeBSD. In *Proceedings of the BSDCan 2007*, May 2007.

[42] *Advanced Configuration and Power Interface Specification Rev. 2.0a*. Compaq Computer Corporation and Intel Corporation, Microsoft Corporation and Phoenix Technologies Ltd. and Toshiba Corporation, March 2002.

[43] G. Singh. The IBM PC: The Silicon Story. *Computer*, 44(8):40 –45, August 2011.

[44] Phil Crouncher. *The BIOS Companion*. Electrocution Technical Publishers, 2004.

[45] *BIOS Boot Specification*. Compaq Computer Corporation, Phoenix Technologies Ltd., Intel Corporation, January 1996.

[46] *Intel® Embedded Graphics Drivers, EFI Video Driver, and Video BIOS v10.3.1 – User's Guide*. Intel Corporation, 2010.

[47] *Unified Extensible Firmware Interface Specification, Version 2.3.1*. The UEFI forum, April 2011.

[48] *FAQ: Drive Partition Limits fact sheet*. The UEFI forum, 2010.

[49] Steven Sinofsky. *Reengineering the Windows boot experience*, retrieved November 20, 2011. URL http://blogs.msdn.com/b/b8/archive/2011/09/20/reengineering-the-windows-boot-experience.aspx.

[50] AMD Embedded Solutions. *An Update on Coreboot*, retrieved August 30, 2011. URL http://blogs.amd.com/work/2011/05/05/an-update-on-coreboot/.

[51] The SeaBIOS project. *SeaBIOS*, retrieved October 22, 2011. URL http://www.seabios.org/.

[52] The Flashrom project. *Flashrom*, retrieved October 21, 2011. URL http://www.flashrom.org/.

[53] Anton Borisov. Coreboot at your service! *Linux J.*, 2009, October 2009. ISSN 1075-3583.

[54] *congatec System Utility – Installation and usage of the congatec System Utility, User's Guide Revision 1.4,.* Congatec AG, May 2009.

[55] *Intel® ICH7 Family Datasheet*. Intel Corporation, April 2007.

[56] CrossControl AB. PEP – Process for ExamensarbetesProcess. Internal Documentation, 2004.

[57] The coreboot project. *FILO*, retrieved November 05, 2011. URL http://www.coreboot.org/FILO.

[58] *Aardvark I2C/SPI Embedded Systems Interface Data Sheet v5.13*. Total Phase, Inc., March 2011.

[59] The FreeBSD Foundation. *vm_machdep.c: cpu_reset_real()*, retrieved October 06, 2011. URL www.freebsd.org/cgi/cvsweb.cgi/src/sys/i386/i386/vm_machdep.c .

[60] *Intel® System Controller Hub (Intel® SCH) Datasheet*. Intel Corporation, May 2010.

[61] *Kontron User's Guide® nanoETXexpress-HMI Baseboard (ePDAnano)*. Kontron Embedded GmbH, 2007.

[62] Robert Austin. *TianoCore as a payload*, retrieved November 21, 2011. URL http://blogs.coreboot.org/blog/author/robertaustin/.

[63] Philip Shulz. *efiboot - An UEFI payload for coreboot*, retrieved November 21, 2011. URL http://www.phisch.org/website/efiboot/.

Appendix A: Data sets

This appendix contains data sets collected during the performed experiments described in section 4.

Coreboot/Seabios applied to Kontron ETX-CD module

Module initialisation time (ms)

Coreboot / Seabios configuration 1				Coreboot / Seabios configuration 2				Kontron / Phoenix Legacy BIOS			
5718.7	5718.7	5734.4	5750	812.5	859.3	796.8	811.5	8343.8	7000	7125	7046.9
5703.1	6074.1	5750	6034.4	828.1	828.1	812.5	812.5	8250	7218.7	7390.6	7640.6
5734.4	5703.1	5734.4	5734.4	796.8	859.4	812.5	796.9	7406.3	7656.3	7718.8	7734.3
5781.2	5734.3	5718.8	5734.3	812.5	828.1	812.5	812.5	6890.6	7703.1	7437.5	7546.8
5750	5734.3	5718.8	5718.8	812.5	812.5	812.5	796.9	6875	7421.9	7453.1	7390.6
5703.1	5687.5	5718.8		812.5	859.4	796.9		6875	6921.9	7515.6	
5734.4	5671.9	5724.8		812.5	859.4	812.5		6890.6	7656.2	7187.5	

Base board initialisation time (ms)

No display				XGA display			
7781.3	7515.7	7546.8	7546.8	3625	3500	3609.4	3593.7
7562.5	7500	7468.7	7500	3546.8	3593.7	3515.6	3546.8
7500	7468.8	7546.8	7546.9	3515.6	3078.2	3546.8	3609.3
7546.9	7468.7	7531.3	7515.6	3531.2	3609.4	3562.5	3592.9
7734.4	7500	7531.3	7531.2	3515.6	3546.9	3562.5	3562.5
7562.5	7593.8	7593.8	7531.3	3625	3531.3	3531.3	3562.5
7500	7500	7578.2	7484.4	3984.4	3546.8	3562.5	3546.9
7500	7546.9	7515.7	7484.4	3625	3546.9	3597.8	3593.7
7500	7562.5	7531.3	7515.7	3546.9	3578.1	3515.7	3531.3
7484.4	8000	7546.9		3515.6	3562.5	3500	

Appendix B: Abbreviations

`$PIR`	A type of PCI interrupt routing table.
ACPI	Advanced configuration and power interface, the standard for device configuration and power management.
AGESA	AMD generic encapsulated software architecture, the bootstrap interface for AMD chipsets.
AHCI	Advanced host controller interface, a standard for configuring SATA/ACHI adapters.
AMD	Advanced Micro Devices, Inc., a hardware vendor.
AMI	American Megatrends Inc, a firmware vendor.
AML	ACPI machine language (bytecode).
API	Application program interface.
APIC	Advanced programmable interrupt controller (advanced PIC).
APM	Advanced power management, an older standard for system power management.
ARM	Advanced RISC Machine, a processor architecture.
AVR	A microcontroller family developed by Atmel.
ASL	ACPI source language (scripting language).
ATA	see PATA.
ATAPI	Modification that allows SCSI packets to be sent over IDE or SATA, used to interface with CD/DVD drives over IDE or SATA.
BIOS	Basic input/output system, a firmware implementation configurable at runtime. (The default x86 BIOS is called the *legacy BIOS* in this text.)
BLDK	Intel boot loader development kit.
CBFS	Coreboot file system. CBFS works as the layout of the Coreboot boot ROM, and the interface to which payloads must comply.
CISC	Complex instruction set computing, the processor architecture paradigm of the x86 processor.
COM	Computer on module.
CPU	Central processing unit – the main processor.
DOS	Disk operating system. The original 16-bit OS for the IBM PC.
DMA	Direct memory access.
DMI	Direct media interface, an Intel bus connecting the northbridge and southbridge.
DVI	Digital visual interface, a video interface.
DRAM	Dynamic random access memory.
DSDT	Differentiated system descriptor table, part of the ACPI standard.
DXE	Driver execution environment, the runtime environment for EFI drivers.
EFI	Extensible firmware interface (superseded by UEFI).
EHCI	Enhanced host controller interface, a controller for USB 2.0.
ESD	Electrostatic discharge. Safety measures must be taken to ensure that sensitive electronics are not damaged by discharges caused by handling them.
ETX	Embedded technology extended, a COM form factor and connector standard.

FPGA Field programmable gate array.
FSB Front-side bus, the bus traditionally connecting the CPU to the chipset.
FWH Firmware hub, an IC protocol.
GCC GNU C compiler.
GNU GNU's not UNIX.
GPT Grand partition table, the EFI and UEFI equivalent of MBR.
GUI Graphical user interface.
GUID Global unique identifier, a UEFI object key.
HOB Hand-off block, a PEI data structure also used in the PI-UEFI transition.
I/O Input / output.
I^2C Inter-integrated circuit, an IC protocol.
IASL Intel ACPI script language compiler.
IBV Independent BIOS vendor.
IC Integrated circuit.
ICH Integrated controller hub, a type of southbridge.
IDE Integrated drive electronics, another name for PATA.
IDE Integrated development environment.
IDT Interrupt descriptor table.
IEGD Intel embedded graphics driver.
IRQ Interrupt request.
ISA Industrial standard architecture, an obsolete computer bus (superseded by PCI and PCI-e).
ISR Interrupt service routine.
ISP In-system programming, programming of an IC while still attached to a mainboard.
LVDS Low-voltage differential signalling.
LTS Long time support.
LZMA Lempel-Ziv-Markov chain-algorithm, a compression technique.
LPC Low pin count, an IC protocol.
MBR Master boot record.
MP Multi-processor.
MS-DOS Version of DOS supplied by Microsoft.
MSI Message signaled interrupt.
NVRAM Non-volatile RAM, a ROM where BIOS settings are stored.
OEM Original equipment manufacturer.
OS Operating system.
OSPM Operating system power management, a scheme where the OS handles power management (rather than the firmware).
OHCI Open host controller interface, an open controller standard for USB 1.1 (and others).
PATA (also IDE) Parallel ATA, an older interface for secondary storage.
PC-DOS Version of DOS supplied by IBM with the original IBM PC.
PCI Conventional peripheral component interconnect, an x86 standard I/O bus.
PCI-e PCI express, successor to conventional PCI.

PEI Pre-EFI initialisation (UEFI), a bootstrap phase during PI.

PEIM PEI module (UEFI), a device driver which runs during PI.

PI Platform initialisation, the bootstrap interface of the UEFI standard.

PIC Programmable interrupt controller.

PIR PCI interrupt routing.

PIR Programmable interrupt router.

PnP Legacy plug and play, an obsolete standard for device configuration using ISA and PCI, superseded by ACPI.

POST Power on self test.

PP Parallel programming, an older IC protocol.

PPC PowerPC, a RISC-based computer architecture.

RISC Reduced instruction set computing, a processor architecture paradigm.

ROM Read only memory.

RSTC Reset control register, a control register in contemporary Intel chipsets.

RTOS Real-time operating system.

SATA Serial ATA, a protocol for secondary storage.

SCH System controller hub. Intel term for an embedded system chipset.

sDVO Serial digital video out, a proprietary Intel video interface.

SEC Security phase (UEFI), a boot phase during PI.

SIO Super input/output, an IC providing extra I/O to the southbridge.

SMBIOS System management BIOS, an extended legacy BIOS interface for providing information about the legacy BIOS to the OS.

SMB (also SMBus) System management bus, an IC protocol.

SODIMM Small-Outline Dual In-Line Memory Module, a type of computer memory.

SOIC (also SOC) Small-Outline Integrated Circuit, an IC form factor.

SPD Serial protocol detect. Module on RAM component that contains information about the RAM.

SPARC Scalable processor architecture, a RISC-based computer architecture.

SPI Serial Peripheral Interface Bus, an IC protocol.

SST Silicon Storage Technology, an IC vendor.

TLA Three-letter abbreviation.

TLB Translation look-aside buffer.

TSOP Thin small-outline package, an IC form factor.

UEFI Unified extensible firmware interface.

UHCI Universal host controller interface, a proprietary USB controller developed by Intel.

USB Universal serial bus, a cable, connector, protocol and I/O bus standard for computer peripherals.

VBIOS (or VGA BIOS, Video BIOS) an x86 option ROM video driver.

VGA Video graphics array. An older graphics interface standard.

x86 (also Intel) PC architecture compatible with the Intel 8086 CPU.

XGA A screen resolution mode of 1024×768 pixels at 18 or 24 bits of colour depth.