Neither Seen Nor Heard: Alternative Views of the State of Firmware

Robert P. Hale and Vincent J. Zimmer, *Member, IEEE*

***Abstract*—This paper will give an overview of system firmware with an eye toward clarifying upon which side of the hardware/software boundary firmware exists. This overview will also counterpoint the perception of firmware in the industry against the reality of firmware development and its role in the market. This paper will then provide a case study of a class of firmware, namely the BIOS, to clarify some of these issues. The evolution of BIOS in the face of the challenges will then be described. Finally, the challenges of firmware development in the fact of ever-increasing hardware complexity will be presented.**

***Index Terms*—BIOS, Firmware, Computing, Validation**

I. Introduction

Firmware has become a ubiquitous part of our personal life. With even a cursory scan of a house, those aware of firmware can locate its presence in the microwave oven, the conventional oven, possibly the toaster, possibly the mixer in the kitchen. In the family room, firmware is present in the TV, the receiver / amplifier, the DVD player, and the cable box. If we have modern appliances, the clothes washer and dryer are firmware controlled. In the garage, if the car is average it has about 50 processors [1], all with their own firmware. On our person, we probably carry a cell phone or an MP3 player, both of which could not function without firmware. Even on vacation, we'll use a GPS to find our way out of the way and a digital camera to record the fact that we were there – both non-functional without firmware.

In the work environment we are equally surrounded by firmware. Most office workers use a computer, which must have firmware to boot. We use copiers and printers, which have firmware. The routers and bridges and phones with which we communicate all rely on firmware. Our building's heating and cooling system is almost certainly has its firmware. Our elevators are firmware controlled. The number of industrial applications only adds to the web of firmware that constitutes modern life.

Manuscript received April 14, 2010.

Robert P. Hale is a principal engineer with the Software and Services Group (SSG) of Intel Corporation. Hillsboro, OR 97124 USA (phone: 503 264 8793 e-mail: robert.p.hale@intel.com).

Vincent J. Zimmer is a principal engineer with the Software and Services Group (SSG) of Intel Corporation. DuPont, WA USA (vincent.zimmer@intel.com)

Even with this reliance on firmware, it remains all but invisible. The entire user interface for firmware might be one or two LEDs and a dial for the microcontroller in a toaster. Even the firmware in an MP3 player's user interface is a small portion of its overall content. Decoding MP3 and JPEG files is algorithmically, as well as code size, much more complex. Firmware is generally characterized as user-interface-poor and content rich.

When we do notice firmware, it is generally for the wrong reasons. The firmware may explicitly report an error such as "Maintenance required" at exactly 10,000 miles on most cars or the dreaded "Cannot find boot device" on a system whose hard drive worked fine yesterday. Alternatively, the firmware may be proclaimed as the cause of some famous recall. The good news, the press will say, is that the fix is a simple 30 minute visit to the automobile dealership. In general, our reaction to firmware failure is surprise. This tends to indicate a high level of expectation: firmware works invisibly and well the vast majority of the time.

This perception seems at odds with the perception in the industry is "the techniques for creating and validating these crucial product components remain relatively arcane and less-studied to the point of being labeled an "art"" [3]. How can firmware be so badly developed and yet be seen as so reliable to the end user?

II. Perception and Reality

There are several reasons that can be seen as leading to this dichotomy.

Firmware is software and thus subject to the same issues as other software projects. These issues have remained remarkably consistent over the years at least as far back as [5]. Firmware then imposes its special spin particularly with respect to limited size and unusually tight hardware binding.

The user could see quality while the internal view is lack of quality because the validation task, if done properly, stresses the hardware and firmware beyond what end users would normally subject a device to. Most users are simply unlikely to encounter most of the errors in firmware (as with software or hardware) because they use only the simpler features of devices.

The majority of firmware bugs could also be negligible enough and users adaptable enough that no one notices most firmware bugs. For example, if the firmware in an MP3 player miscalculated the tones in every 256th iteration during an MP3 music decode, most users would probably not notice.

Projects could also simply take 3 or 4 times estimates due to required debugging and excessive validation runs in order to reach a satisfactory quality level. The end user would not notice this because the user probably didn't know the original schedule.

There could be confusion between the root cause and the fix for issues. In particular, there is great pressure to resolve hardware issues in firmware due to the order-of-magnitudes difference in the cost of resolution. A "spin" of a chip may take many weeks and cost millions of dollars whereas a firmware fix may cost a few thousand dollars and a day or two of total work. In fact, many modern chips are designed so the firmware can configure the chips to work around issues in the field rather than having the hardware recalled. The public is simply told that there is a firmware issue when, in fact, the firmware resolved what was a hardware issue.

There could be confusion about product readiness. Firmware is generally produced in companies and organizations whose main expertise is hardware, be it integrated circuits or microwave ovens or remote controls. The overriding perception is that, when the hardware completes its validation, the device should be ready to produce and ship to the customer. The firmware developer is called upon to both support the hardware debug and ensure that his own bugs are resolved on time. The time required by the firmware developer is seen as dead time that is keeping the hardware from shipping. A more functional view, that the hardware and firmware go together to make a system, is easy to have in the abstract but as easily lost in the heat of scheduling.

The gap in the understanding of the complexity of the firmware task may also lead to perceptions that firmware validation is excessively long and arcane. Estimates are that "embedded software complexity doubles in size every 10 months" [2] , firmware being a subset of embedded firmware. The developer of the hardware for an MP3 player may not have the visibility into the complexity associated with playing an MP3 recording or managing the NAND flash or the complexity in dealing with user interface experts.

The central role firmware integrating hardware pieces into a coherent whole means the firmware has a large number of customers each with different needs. In the case of larger firmware development, several validation groups think the main purpose for the firmware is theirs whereas the teams that do sales or enabling or press think it is theirs. Different subsystem developers require firmware assistance. Firmware must meet all requirements and usually has little say in terms of intermediate or final delivery dates. The subsystem developers and validation teams, on the other hand, can be unaware of or are unsympathetic to other needs.

Hardware validation uses firmware as a tool for its validation. For products that are updated each generation (such as BIOS, or "Basic Input Output System"), hardware validation would prefer that the firmware not change at all so as to provide a consistent platform for hardware validation and debug. Meanwhile there is pressure from the product teams to change to have new features. The economics of product development mean that the same basic source must support all customers. Over time this necessitates poor design decisions which have the effect of leading to less consistency and lower stability during validation.

Unless the firmware developers and validation engineers have a long working relationship, it is often hard for validation engineers to gauge the complexity of a task or change to firmware. Many times the firmware's design can accommodate apparently large and apparently monumental changes while stumbling over apparently quite small changes. With different players, this is common throughout the software industry.

Firmware is a wide space. Each of these explanations, plus probably many more could make up the story that is hidden behind each piece of firmware and hardware that is created.

III. A Case Study: BIOS

In the early days of computing, the reset vector, and the locations following, had to be filled in by hand via front panel toggle switches so the computer would have something useful to execute at reset. As time proceeded, the toggle switches and the operator that entered them were replaced by instructions stored in non-volatile memory chips: Firmware. This code has had various names including the system loader, the autoloader, the bootstrap loader, and the boot ROM. Due to the popularity of the IBM PC, the code now has the generic name BIOS (Basic Input Output System) [4]. The BIOS can be used as a case study of firmware heavily involved in validation, required for production, and which has a long generational record.

The BIOS consists of two main pieces known as POST (Power On Self Test) and Run-time. POST starts at the reset vector and ends at the hand off to the operating system's boot loader. The run-time provides a series of software interfaces which are used by the boot loader until the operating system loads its own drivers and may also be used by the operating system once it is running.

As the architecture of the PC has evolved the function of POST has changed. Early on, POST did a considerable amount of testing and diagnosis to locate which parts of the system failed. As the number of system components decreased, the number of failures decreased and the number of failures that the BIOS could diagnose but which did not also cause the BIOS to fail also decreased.

On the other hand, the BIOS was given more responsibility for system initialization. In an effort to make PCs easier to use, upgradeable components including RAM and add-in cards were made self-describing. The descriptions typically contain data structures of resource requirements and component attributes. It was then up to the BIOS to sense the presence of the add-in devices, read those descriptions, and configure the on-board hardware properly. The initialization of RAM, a few hundred bytes in the PC jr, now takes around 100 Kilobytes.

The BIOS must have the firmware equivalent of device drivers to communicate with those required for booting: media devices such as hard disks, CD, USB drive, networks, keyboards, mice, and video cards. Networks and video cards typically arrive with their own add-in drivers known as option ROMs. Over the years the specifications for each of these has evolved considerably. Initially drives were addressed using cylinder, head, and sector format with a 540MB and now are addressed as logical blocks with a limitation of above 2^64 bytes.

A modern BIOS must initialize and communicate over several major buses including SMBUS, USB, and PCIe [10] as well as the bus connecting the processor to the other chips on the system board. The BIOS must access devices ranging from SATA and SCSI to USB Media and HID devices [11] to IPv6.

The BIOS has also important and complex roles to play in describing the hardware to the operating system (8]), interfaces to management applications [9], support for the BIOS update and security, power and thermal management [8]. The original desktop BIOS has been modified to support many-way servers down to hand held devices. The BIOS, a less than 64KB ROM, has evolved into an approximately 2MB (uncompressed) software package stored in NOR FLASH, a type of electrically alterable non-volatile memory that is addressable as ROM.

Although BIOS development was almost exclusively done in assembly language, much of the rest of BIOS development was relatively modern. Most BIOS were relatively modular. Relying heavily on source code control systems, reuse factors of 70 to 90% between chipset generations were not uncommon. These allowed the best of the typically relatively small BIOS groups to produce large amounts of high quality results.

IV. BIOS Industry Response to Complexity

By the late 1990s and early 2000s, the complexity faced by the BIOS was clearly increasing. At the same time the code bases used to create BIOS were almost entirely in assembly language and carried 20 years of increasingly archaic interfaces. This led, over time, to an industry-wide Forum to start definition of a replacement for the PC BIOS known as UEFI, the Unified Extensible Firmware Interface. The UEFI specification itself [6] provides an agreed upon set of interfaces between the system firmware (the traditional BIOS), option ROMs on add-in cards, and operating systems. A companion specification, the Platform Initialization (PI) Specification describes the underlying structures inside the system firmware.

The design of the PI specification used the knowledge hard won through twenty plus years of BIOS development combined with solid software engineering.

PI is broken into 5 phases:

- SEC ("security" Phase): Basic processor initialization. Initialization of "starter RAM" (usually cache) to be used for temporary storage until RAM is available.
- PEI: Continued system initialization with the goal of initializing RAM. PEI consists of a core and a number of PEI Modules (PEIMs) communicating via simple abstractions known as PEIM to PEIM interfaces (PPIs). PEIMs are executed directly out of firmware.
- DXE: Continued system initialization and peripheral initialization. DXE consists of a core and a number of DXE Drivers communicating via abstractions known as protocols. DXE Drivers are loaded into RAM and executed. DXE components are typically stored compressed.
- BDS: Cooperating with DXE, causes the initialization of the appropriate boot devices and performs the initial operating system load. Completes creation of UEFI interfaces.
- Run-time: Performs similar tasks to BIOS run-time but with improved interfaces.

Below in Figure 1 is a temporal view of a UEFI-based firmware stack, as described above.

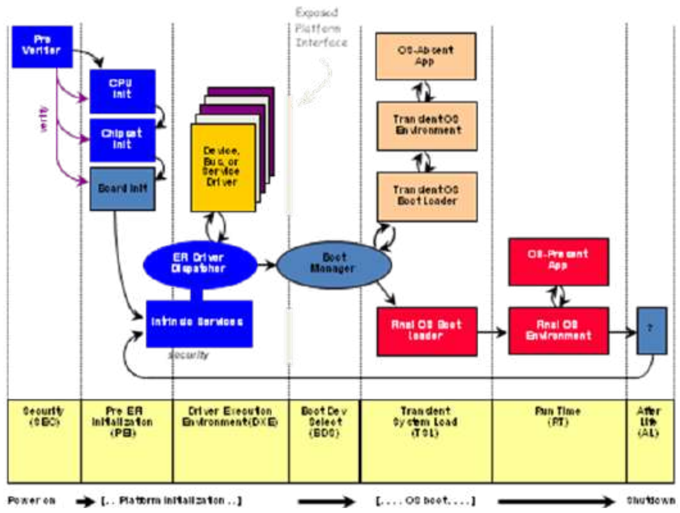

Figure 1 UEFI PI boot flow

In addition to the flow above, below in Figure 2 is a spatial view of the UEFI based firmware stack.

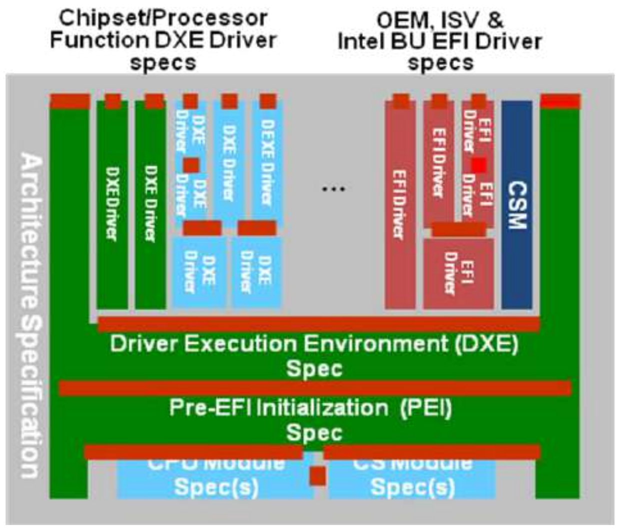
Chipset/Processor Function DXE Driver specs
OEM, ISV & Intel BU EFI Driver specs
Architecture Specification
DXE Driver
DXE Driver
DXE Driver
DXE Driver
DXE Driver
DEXE Driver
DXE Driver
DXE Driver
...
EFI Driver
EFI Driver
EFI Driver
EFI Driver
EFI Driver
CSM
Driver Execution Environment (DXE) Spec
Pre-EFI Initialization (PEI) Spec
CPU Module Spec(s)
CS Module Spec(s)

Figure 2 Spatial view of UEFI PI components

PEI and DXE modules describe their requirements for execution. The cores then execute the modules in the order their requirements become satisfied, ensuring a consistent execution order boot to boot. The modules become objects that form together to create stacks to implement e.g. file systems and networking from basic components. Typically only the basic components change for each generation since the related hardware changes.

On a typical notebook computer, PI is expected to run in less than 10 seconds.

V. Validation and Software Architecture

The initial implementation [7] started in 2002 and is now in its 9th generation. Deployment in any serious way started in 2003-2004 with only mixed reviews. Some teams flourished whereas others suffered greatly. Teams which did prototypes before plunging into production code were more successful. Teams which had more high level experience and teams with a wide range of levels of expertise seemed to do better as well. We found that, due to the software focus of the architecture, teams with more software engineers did better than teams that were dominated by hardware engineers. We also found that recent college graduates were unexpectedly successful, due to their training in more modern software skills.

The architecture is designed to encourage most development in the PEI phase and beyond, where modules are written in C. The architecture is also designed to isolate modules from each other enabling high levels of code reuse. In many cases 95% of modules are reused *without change* from chipset generation to chipset generation. This high level of reuse increases reliability.

The increased modularity has allowed some BIOS validation teams in Intel to change how they validate firmware. Instead of treating the entire BIOS as a black box, they do binary comparisons of modules from one release to the next to determine automatically which modules have changed. Most testing is then focused on the modules that have changed. The results have been encouraging, with test time cut by a factor of around 3 and escapes reduced.

The architecture has also included features which allow for improved validation automation. Consider, for example, BIOS Setup, a "program" within the BIOS which allows the user to change configurable options and view system information. In order to gain satisfactory test coverage, the various permutations of configurations that may be created via Setup are tested. Typically, this involves a human operator who has to manually enter Setup, change options, reset the system, and perform tests for each permutation. In order to improve this situation, the user interface management in UEFI and PI enables the ability to automate the manual steps associated with Setup testing. The testing becomes more reliable and more testing can be accomplished for a given amount of time.

UEFI / PI systems have, where applied well, allowed for improvements in development time, readiness, and validation. They have enabled the use of more modern software methodologies in the BIOS development community. In doing so, they provide a solid base for the next 20 to 30 years.

VI. Conclusion

Our experience with deploying what is a relatively large and complex piece of firmware for many years suggests that many of the validation issues with firmware arise from the different views the customers have of that firmware. Validation is simply one more viewer with a different view.

Hardware validation techniques are also not particularly appropriate to a piece of software. Firmware is software and software techniques are generally far more applicable to tests above the subsystem level. Automation is key to successful software testing. Designing validation hooks in during the architecture phase has proved useful.

Unlike applications, firmware is software delivered with the product itself. This means firmware is as much a part of the product as any piece of hardware is. Teams that remember that they are testing a system are much more successful than those that attempt to validate a single piece.

If firmware is software, it is not typical software. It requires a different mentality than net based development. The successful firmware developers probably did well in their operating system classes at school and were probably poor user interface designers. Hardware engineers who have been successful BIOS developers have generally focused on generational hardware changes or had considerable focus on software in school.

Acknowledgment

We would like to thank Mark Doran and Isaac Oram at Intel for their feedback on this paper. We would also like to thank Priyadarsan Patra for his guidance in this process.

References

[1] "How Car Computers Work," http://auto.howstuffworks.com/under-the-hood/trends-innovations/car-computer.htm
[2] Wolgang Ecker, Wolfgang Muller, Rainer Domer Eds, *Hardware-dependent Software: Principles and Practice*, Springer, 2009, ISBN 978-1-4020-9435-4.
[3] Priyadarsan Patra, *Description of the Special Session on Firmware Validation Challenges*, in HLDVT Workshop, 2010.
[4] IBM Corporation, *Technical Reference – Personal Computer PC Jr*, Boca Raton, Florida, 1983. Document number 1502293
[5] Barry W. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981, ISBN 0-13-822122-7.
[6] *UEFI 2.3 Specification*, www.uefi.org
[7] Vincent Zimmer, Michael Rothman, Robert Hale, *Beyond BIOS: Implementing the Unified Extensible Firmware Interface with Intel's Framework*, Intel Press, 2006, ISBN 0-9743649-0-8.
[8] Advanced Configuration And Power Interface Specification (ACPI) www.acpi.info
[9] Desktop Management Task Force (DTMF) www.dmtf.org
[10] Peripheral Component Interconnect (PCI) www.pci.org
[11] Universal Serial Bus (USB) www.usb.org