Comparing BIOS and Boot Loader Features
An Insyde Software Corp. White Paper

Charles Marslett, US & EU FAE, Insyde Software Corp.
charles.marslett@insydesw.com
Copyright

Copyright © 2010 by Insyde Software Corp. All rights reserved.

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form, or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Insyde Software Corp.

Disclaimers

Insyde Software Corp. provides this document “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose.

This document could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in future revisions of this document. Insyde Software Corp. is under no obligation to notify any person of the changes.

Trademarks

Insyde is a registered trademark and InsydeH2O and InsydeDiY are trademarks of Insyde Software Corp.; Intel and Intel Atom are trademarks or registered trademarks of Intel Corporation in the U.S. and/or other countries; Microsoft, Windows and Visual Studio are trademarks or registered trademarks of Microsoft Corporation in the U.S. and/or other countries.

All other trademarks mentioned herein are property of their respective holders.
Contents
Comparison Specifics and Simplifications: .................................................................4
  Developer Considerations: ..........................................................................................5
  Tool Chains: ..............................................................................................................6
  Operating System Requirements: ..............................................................................6
  Target System Demands on the Development Environment: ...............................7
  Project Size: ..............................................................................................................7
    Small Code Footprint and Resulting Complexity: ..................................................8
    Larger Footprint but with Simple Structures and Easy Consistency: .....................8
How do the UEFI Standards fit into our Consideration? ..........................................9
  Some History: Where did the UEFI Specifications come from? .............................9
  InsydeH2O: A Modern BIOS, Compliant with the UEFI Specifications ...................10
  InsydeDIY: a Hybrid, call it a UEFI Boot Loader from Insyde ................................11
  The BLDK: a Legacy BIOS Boot Loader from Intel .................................................11
  Coreboot: an Open Source Legacy BIOS Replacement or Boot Loader .................12
Tradeoffs: Training, Time, Cost and Reuse ...............................................................13
  Development Time Tradeoffs ..................................................................................13
  Boot Time Tradeoffs ..............................................................................................14
  Cost Tradeoffs .......................................................................................................15
  Planning for Reuse .................................................................................................16
    The Need for a Structured Boot Firmware Design .................................................16
    The Impact on Future Projects ............................................................................16
For More Information ...............................................................................................19
Comparison Specifics and Simplifications:

There are a number of differences between BIOSes from different Independent BIOS Vendors (IBVs) and even between BIOSes from the same vendor. The most significant distinction is that there are two major categories of BIOS products: legacy BIOSes implement the interfaces that have grown up around the original IBM PC computers of the late 1980's while the more recently developed UEFI boot software implement the interfaces specified by the Unified Extensible Firmware Interface (UEFI) Forum.

A first simplification to limit the scope of this paper is to compare boot loaders only with a UEFI Framework based BIOS, specifically the InsydeH2O UEFI boot software. In almost all cases this material will apply to any Framework based UEFI boot software but where there might be differences, InsydeH2O is the UEFI BIOS that will be described specifically. Also, for brevity, the term BIOS will refer to InsydeH2O unless specifically called out otherwise.

To even further minimize the complexity of this paper the discussion will be limited to comparisons to the InsydeH2O BIOS for Intel's Crown Beach Customer Reference Board (CRB). The differences between this BIOS and other InsydeH2O BIOSes or even between it and UEFI BIOSes from other independent BIOS vendors (IBVs) will be relatively small compared to the differences between it and a boot loader, so this is again a reasonable simplification.

There are also a larger number of differences between boot loaders: they are in fact more significant than the differences between UEFI BIOS implementations. Most of this paper is going to be concerned with examples representing three classes of boot loaders. The InsydeDIY boot loader is a commercial product that differs from a BIOS from an IBV most significantly in the fact that it is not subject to royalties and it has a more limited support structure. The second class is represented by the Intel BLDK Core Version, a boot loader that provides the minimum legacy BIOS interfaces required to start up common embedded software kernels. The third class is represented by the open source Coreboot V4/V4.1 BIOS Replacement providing a set of interfaces that meet the requirements of many open source software kernels with a focus on Linux. Even so, the boot loader simplifications may be less justified than the BIOS simplifications.

Also, please note that in this paper, ‘boot loader’ should not be confused with the more commonly mentioned ‘boot loader’ that is not Flash based system firmware, but is an OS component often stored in the boot sector of a mass storage device. In most cases, that is an OS software component which acts to load and initialize the remainder of the OS. In this paper that particular ‘boot loader’ will be referred to as the ‘payload’ to more clearly make the distinction.
A final simplification is that we do not consider general purpose computing systems. Only one target OS or software environment is to be executed by the system. This is a very significant simplification: it permits us to ignore features that are not required by the specific target OS or software kernel. In many cases the effort involved in bringing up a new platform may be dominated by the writing of support code for the large number of programming interfaces the many boot targets and environments supported by a general purpose product such as a Single Board Computer (SBC) might require. In the more difficult cases, the specific implementation may be different from one environment to another, and may require trial and error experimentation to make all the target environments work well. Should a general purpose computing system be the goal of a boot software developer, I would strongly recommend the use of a standard BIOS. Otherwise the development resources required are likely to be overwhelming.

**Developer Considerations:**

There are three major environmental components involved in the creation of a BIOS or boot loader.

First, it is ultimately a software product, so it requires a tool set or tool chain to convert the source code and build description data into an image of some sort. This will involve compilers, assemblers, linkers, utilities and in most cases some custom programs that create the specific binary image that the target hardware expects.

Second, that tool chain may be limited to a particular operating system or other environmental configuration. For example, Microsoft Visual Studio can only conveniently be run from a Microsoft OS. Similarly, Eclipse and GCC can be run on BSD or Linux based systems, Windows systems or Mac OSX systems, but some of the tools a particular boot loader is designed around may only be available in one of those ecosystems.

The third consideration that affects the choices made in a BIOS or boot loader project is the target hardware instruction set and code constraints. Some target hardware may have significant cache line boundary requirements or address space limitations on early RAM or cache utilization and if the tool chain makes the assertion of those constraints simpler, that will be valuable to the boot software developer.

Ignoring these constraints can result in unacceptable boot times or complex functional workarounds that will impact the quality of the final product and its delivery schedule.
**Tool Chains:**

There are two major tools ecosystems used for writing BIOSes and boot loaders, as well as several less widely used ones.

Microsoft Visual Studio (MSVS) has been used for several years, and provides a simple source control system (Visual SourceSafe), a C compiler, an x86 assembler and some rudimentary build sequencing functionality. It is also commonly available in many engineering environments because of its use building web software as well as Windows device drivers and applications. It, along with some BIOS specific helper tools, is certainly the most commonly used tool chain for building a BIOS.

The Gnu Compiler Collection (GCC) is the other commonly used set of tools for writing low level pre-boot code. Often it is augmented with one or another of the open source assembly language tool kits (NASM for example) since they appear in the same ecosystem. This tool chain is less often used to build BIOSes and more commonly used for building boot loaders, because many of the users of boot loaders are most familiar with the Linux and BSD open source environments.

Other tool chains that are occasionally used include Watcom or OpenWatcom C++ Tools – usually but not always hosted in a Windows environment, Digital Mars C or D tool sets – providing an efficient light weight object oriented tool chain, and the Intel C++ Compiler (ICC) – arguably producing the most efficient executables on Intel processors (and functioning well on both Linux and Windows systems). But to limit the scope of this paper I will not further discuss these tool chains.

**Operating System Requirements:**

The GCC tool chain can be used in a development context hosted on a Windows computer using Mingw or Cygwin (two commonly used wrappers to provide a Unix style environment for porting software to a Windows computer). There are, however, limits to the extent that these wrapper environments emulate a Unix OS. So occasionally using the GCC tool chain may require some adaptation. Fortunately, since the source code is available, this can be done with only a reasonable amount of effort. Of course, GCC is almost always available on Linux, BSD and OSX systems and furthermore it is almost always the native tool chain, so adaptation is hardly ever needed in such a case.

The Windows OSes (Windows XP, Windows Vista and Windows 7, most commonly) are the native environment of the Microsoft Visual Studio tool chain. So in a similar fashion, it is rare that a tool in the Visual Studio tool chain needs any adaptation.
for a particular version or service pack level of Windows. It is possible to use many of the Visual Studio tools in a similar wrapper environment on a Linux or BSD system. It is more difficult, however, because of the vastly larger Windows API set compared to those of a Unix-style OS. So only rarely will anyone attempt to build a BIOS or boot loader designed with a Visual Studio build environment on a Linux, BSD or OSX computer.

**Target System Demands on the Development Environment:**

Because the initial code in a BIOS, and possibly all the code in a boot loader, is executed directly from the Flash boot device, it can be very useful to build code that is extremely optimized for size. Even optimizations that cause significant performance penalties in almost any x86 architecture may be beneficial if the predominate execution time contributor is the code fetch from a serial Flash part or a firmware hub. Having such an optimization available in the C compiler can make it possible to write more code in C and do less hand optimization in assembly language to achieve the same performance goals.

Similarly a linker and compiler that permit alignment of loops (an optimization of great value when cache accesses are much faster than main memory accesses or when the processor loads instructions one cache line at a time) can be of significant value both before and after the processor caches are turned on.

**Project Size:**

In deciding between a boot loader and a complete BIOS there are two different measures of the size of a project. One could be interested in the size of the code body, both from the point of view of how much code must be executed, how much code must be validated in system test, or how much code must be looked at to feel comfortable that the whole system is understood by the design engineers. In these cases, portable or reusable code may result in a larger project. If this is a deciding factor in the design a boot loader might be the best choice.

Or one could be interested in how much additional code must be written to adapt the starting code body to the hardware and software that form the target system. A large number of files decomposed from the major stages of the boot code may make this task smaller, even though the decomposition into a larger than absolutely necessary set of files will increase the total number of lines of code (or instructions) and will probably make understanding the system in detail more difficult.
For example, if well documented, the code partitioning can make customization a much simpler task: the primary activity might become removal of functions, services or tables rather than coding and modifying or adding functions, services or tables. Likewise, table driven execution structures make it very easy to partition tasks. For example, if one table of function pointers contains all the low level I/O functions and a second table contains the file system component functions, it becomes very easy to port that file system code to a new hardware device with a very different architecture.

**Small Code Footprint and Resulting Complexity:**

In the past many PC and PC-like systems booted from a very small boot ROM without any ROM compression (32KB to perhaps 128KB were typical sizes). This resulted in heavily hand optimized assembly code in order to fit the functions required into the space available. Simple code structures and understandability had to be sacrificed very early, faced with these very restrictive requirements.

Today very large Flash devices are used to hold the BIOS code (in some cases 8MB or more). With that much space, a sophisticated decompressor can be used resulting in the Flash device holding five times the code it could hold uncompressed. So even allowing for the fact that modern systems are much larger and more complex than earlier systems, it is possible to use some of that extra space to make interfaces more consistent and to make the code easier to understand. An Insyde UEFI BIOS is written almost entirely in higher level languages (C for the executable code and ASL for ACPI tables and interfaces, for example) and the resulting BIOS code can be 5 MB in size, yet by using and efficient compression algorithm, this can still be stored in under 1 MB of a Flash device’s storage.

**Larger Footprint but with Simple Structures and Easy Consistency:**

The consistent interfaces for UEFI PPIs and Protocols mean much of the low level design is consistent between one part of the BIOS and another (see Beyond BIOS by Vincent Zimmer, Michael Rothman, and Robert Hale for a complete description of those interfaces and why they were designed that way). So an engineer can spend almost all his time working on the problem or feature at hand, rather than learning the detailed designs of the individual BIOS components it interacts with, as he would with a legacy BIOS.

In addition to the improved readability of the code resulting from consistent interfaces, the reuse of pieces of code minimize duplication of blocks of code so errors need only be addressed and fixed one place, rather than in all the duplicate (or in most cases, in all the nearly duplicate) blocks of code. And of course if the structure is standardized, it simplifies the documentation required for the BIOS.
Comparing BIOS and Boot Loader Features

Insyde Software Corp.

and that in itself will reduce the amount of work involved in creating and especially in maintaining the BIOS.

**How do the UEFI Standards fit into our Consideration?**

Because the UEFI standards are some of the more successful attempts to address the problem of efficiently implementing boot or pre-boot firmware on modern hardware, taking advantage of the UEFI architecture might make the design process more efficient.

**Some History: Where did the UEFI Specifications come from?**

The BIOS and non-BIOS solutions to the problem of running more or less general purpose environments for real world software projects, what we might call system-specific boot firmware, have evolved in many directions from the first general purpose compute engines with replaceable parts that came about in the early and middle 1980s. The Apple, Commodore and Atari “operating system ROMs” were capable of hardware extension, but were neither intended to support extension to a flexible operating system nor were the software writers of the day interested in extending the system ROMs in most cases. The IBM PC and its successors had a comparably flexible basic I/O system ROM (BIOS ROM), but the PCs were seen as capable of supporting a much more flexible preboot environment, leading to operating systems that could start up from a wide variety of hardware.

At first, the variations boot devices, then different display devices, more legacy serial and parallel interfaces were added. For some battery powered systems tuning operation for battery life and controlling the charging of the battery were added.

All these enhancements were done with independent standards or in some cases with *ad hoc* specifications leading to a large number of basic calling procedures, resource utilization rules and other constraints on the system. Phoenix Technologies made some valiant efforts to document and standardize these interfaces with their Technical Reference Series: *System BIOS for IBM PCs, Compatibles, and EISA Computers*, *CBIOS for IBM PS/ 2 Computers and Compatibles: The Complete Guide to ROM-Based System Software for DOS*, *ABIOS for IBM PS/ 2 Computers and Compatibles: The Complete Guide to ROM-Based System Software for OS/ 2* and others. But at that time the need for some kind of standardization was just not recognized by the users of the boot firmware at large.

Experience is a good teacher, though, and by the time Intel released its original Itanium processor in 2001 after 5 years of public joint development with HP, the
pre-boot software was so complex something had to be done. What was done was the extensible firmware interface (EFI). Like the Itanium, it was not immediately adopted everywhere, but unlike Itanium the EFI (now the Unified EFI or UEFI) boot firmware did develop widespread support. Complex pre-boot code was much easier to write and maintain with standard interfaces and a simple, consistent dispatching algorithm. UEFI had this, and legacy BIOSes did not. Even the late move from nearly pure 16-bit assembly code to C and 32-bit flat model code was not able to compete with the engineering efficiency of the well defined protocols and PPIs that a Framework based UEFI BIOS was built upon.

Framework based UEFI pre-boot code is not perfect, of course, but it is sufficiently robust and easy to work with that it may demonstrate longevity comparable to that of the x86 architecture. That is an achievement the architects should certainly be proud of.

**InsydeH2O: A Modern BIOS, Compliant with the UEFI Specifications**

InsydeH2O (a modern BIOS, that is) will implement the full UEFI specification, so a knowledge of the UEFI specification can be directly applied to BIOS development, documentation and testing. For example the UEFI Self Certification Test (SCT) can be used to verify that the stock components of the BIOS are all functioning correctly and can be depended on as the basis for customized features or services.

With a full featured UEFI environment, EFI pre-boot firmware runtime environment modules (PEIMs) and UEFI driver execution environment modules (DXE drivers) simply drop in and work. So if reuse of such code is important, that factor may make your decision between writing or adapting a boot loader and porting a BIOS to a new hardware platform. For example, the Intel Graphics Output Protocol (GOP) driver is used in conjunction with a Fast Boot BIOS to provide graphics console output functionality early in the BIOS initialization. The Intel GOP driver, needs only very minor adaptation to integrate it into a standard (UEFI) BIOS. The GOP driver can initialize and run very quickly using the full services of the BIOS, running in a fully cached 32 or 64 bit protected mode environment (see the Intel white paper [Fast Boot GOP and EPOG Drivers Optimized for Speed](https://resources.intel.com/) by Sunil Cheruvu and Satyeshwar Singh).

In a like manner, UEFI compliant reference code for a new peripheral or for a new processor (CPU), memory controller (MCH or ‘uncore’) or I/O controller hub (which may be referred to by any of a number of acronyms: SCH, PCH, IOH or ICH) can often be fully functional in a matter of hours by simply dropping it into
the code tree replacing an existing device or adding it along with existing
devices.

**InsydeDIY: a Hybrid, call it a UEFI Boot Loader from Insyde**

One interesting recent development in the embedded space is InsydeDIY, a
commercial product from Insyde Software that is fully UEFI compliant while still
similar in terms of cost with comparably featured boot loader software.

Though not an open source boot loader, it shows many of the features that a rewrite
of the Coreboot loader to be UEFI compliant would demonstrate. It does not
include the complex user interface that a traditional BIOS would carry with it, it
does not easily support booting multiple OSes, it does not require per-unit
royalty payments, and the source code license is relatively inexpensive.

But it does share a number of features with the InsydeH2O BIOS products: drivers and
modules written for InsydeDIY can be used without rewriting in an InsydeH2O
BIOS and vice versa, and the Insyde debugger designed to support InsydeH2O
works equally well with InsydeDIY (this is an example of code being portable
between the two worlds). As a result, it can be considered a middle approach
and could be useful in some projects.

**The BLDK: a Legacy BIOS Boot Loader from Intel**

The Intel Boot Loader Development Kit (BLDK) is another recent development in boot
loaders. Unlike InsydeH2O and other standard BIOS products, but like
InsydeDIY, it requires no per-unit royalties. And like InsydeDIY it is not an open
source project.

Two major differences between the BLDK and InsydeDIY are the development
environment and the basic architecture. These differences can be traced to the
purposes and rationales of the two products.

The BLDK is aimed at providing a minimal boot firmware environment with a minimal
software developer learning curve for designers with extensive deep embedded
experience. That makes Linux the obvious development environment choice
since many of the developers of deep embedded systems have been working
with Linux for a number of years. It also leads directly to the use of a GUI
based development environment to limit the need to learn details of the code
structure when building a first project. Further, since many embedded
engineers have experience using Microsoft DOS tools to test higher end products,
they may have some familiarity with the legacy BIOS feature sets.
In contrast InsydeDIY is aimed at extending the UEFI preboot environment architecture to products where a full function UEFI BIOS may not be appropriate. As a result it uses as many of the tools and software components as possible from the standard Intel Framework for UEFI and is closely tied to the Microsoft Windows and Visual Studio development environment used to build a Framework based UEFI BIOS. It sits between the Sourceforge EDK project on one side and the full featured InsydeH2O BIOS on the other, both fully UEFI compliant. And because those two development environments are not GUI based, InsydeDIY is not, either.

The Intel BLDK has a simple structure. Unlike, a Framework based boot loader like InsydeDIY, the BLDK boot loader is a monolithic executable (like most but not all legacy BIOSes). So components are built into libraries and linkable object modules rather than into file systems holding independent executables. This has the advantage of making component interactions simpler, but it also makes isolation of components from one another virtually impossible. And it means porting code to or from a UEFI environment involves more structural adaptation that one might at first glance expect.

It would require a major redesign to adapt it so as to work like a Framework based UEFI preboot environment with calling sequences and interfaces comparable to the UEFI BIOS interfaces without large amounts of code duplication.

The BLDK as supplied by Intel is specific to the Fedora Linux distributions and the Red Hat RPM package manager. This is not as constricting as it might sound: the Intel C compiler tool set for Linux is similarly distributed, so if you can install and run the Intel C compiler, you can probably run the BLDK tools.

Beyond the RPM package manager and its dependencies, the tool chain requirements are limited to what you would expect for any Linux based open source project: gcc, as, ld, make, etc. and (if the target OS uses ACPI for any of its hardware description) iasl. And the preferred development OS environment is a Fedora Linux distribution.

Currently, with a fairly small user community, there is very limited technical support for users of the BLDK compared to users of IBV products or even the Sourceforge EDK and EDK-II tools. This situation can be expected to improve over the next few years, though.

**Coreboot: an Open Source Legacy BIOS Replacement or Boot Loader**

The open source Coreboot V4 (or V4.1) boot loader is completely free in both senses of the word: it has no royalty requirements, no source code licensing requirements and it can be modified in any way you want. It has also been
around for some time, so there are significant programming and engineering resources available should you not want to do everything yourself. But as always, hiring services or engineers will increase the one-time cost of the project. It is most valuable for users of single board computers that provide direct support for it on their hardware, but a large number of chipsets have support in the code, and of course support can be added for new platform components as time passes.

The tool chain is quite comparable to the standard open source project: gcc/g++, make, ncurses-dev (for make menuconfig), doxygen (for generating/viewing documentation), flex and bison (for regenerating parsers), gdb (or some other debugger) and (if the target OS uses ACPI for any of its hardware description) iasl. Thus it differs only slightly from the BLDK requirements. And the preferred development OS environment is one of the Linux or BSD based distributions or operating systems.

It has developed in much the same way that a legacy BIOS matured. And it is dependent on SeaBIOS (functionally similar to a UEFI CSM) and sgabios (a serial port implementation of a traditional VGA option ROM) to support the typical embedded OS used with a serial console. It would require a major rewrite to make its calling sequences and interfaces comparable to the Framework based UEFI interfaces so components could be easily replaced. In many ways, it can be considered to have the same relationship to a legacy BIOSes as FreeDOS has to legacy DOSes (DRDOS and Microsoft DOS, for example).

**Tradeoffs: Training, Time, Cost and Reuse**

There are almost always downside factors that force us to carefully consider all the aspects of even a great idea. A well known adage concisely expresses this as “There is no such thing as a free lunch”: TINSTAFL.

You want good documentation, you want efficient and effective training, you want tools and frameworks (note the small ‘f’) that make best use of your engineering time, you want preboot firmware that runs and gets out of the way fast, you want everything you do to be reusable in future projects, and you want it to be very, very inexpensive. That is not all that much to ask for is it? You just want a very inexpensive lunch, not a free one.

**Development Time Tradeoffs**

If costs are not an object, there are some serious reasons to consider only the standard BIOS products. They became standard in the personal computer industry for much this reason. The independent BIOS vendors (IBVs) have tools
and processes that provide for a reproducible BIOS image so the PC vendor can root cause issues and correct them quickly. This is a requirement where the end user is not technically expert and the product is not inexpensive. They usually have a lot of experience training their customers, the engineering staff of the PC vendors, as well. And they can usually provide most if not all the feature set your system software requires. So in many cases, the development tasks are reduced to removing code, not writing it.

Finally, since it is a product that the IBV sells, and in some cases is their principal product, they provide engineering services (non-recurring engineering charges or NRE) that permit you to do the customization of the BIOS code without devoting engineering resources of your own to perform that task.

Also, in many cases your design will be based on a standard customer reference board (CRB) from Intel. The BIOS vendor, as a result, can provide the starting point BIOS immediately and you can run it on that CRB. This means that your engineering team can start work on any custom preboot firmware and they can start profiling the preboot firmware to address your critical path issues (timing, boot target selection, etc.) before a single piece of production hardware is available.

All of these factors result in a much faster time to market because your BIOS engineer has less work to do adapting the code to your customized platform and he or she may be able to start working on the custom code significantly earlier.

The benefit of earlier market entry or more thorough testing of your customizations before entering the market may well be worth the higher initial cost and the per unit royalty. And of course, if the maintenance and continuing engineering support is worthwhile, that is also available from an IBV, but rarely available for boot loader code.

**Boot Time Tradeoffs**

There are often requirements that an embedded system boot up very quickly, or at least appear to do so. For example, an automotive display controller should turn on all the displays, either flashing a ‘self test pattern’, simulating one or simply going to its default display state in no more than a second or two. It may not need to accurately display all the information for another 2-4 seconds but it will not be a positive selling feature if it is not fully functional within 5 seconds or so.

Often such a requirement might be taken to imply that a small preboot firmware package written in assembly language is needed.
But experience has proven that it can be beneficial to write the code in C or some other higher level language so it can be experimented with extensively, each experiment profiled and evaluated in a real world environment, and then the best choice made for each detailed design component. Perhaps by saving the last known good boot settings and using them until something changes in the system, all the training time can be deferred until the system is shutting down, if needed at all. Often shutdown time is not as critical as start up time.

Likewise, if large complex algorithms need to be executed, perhaps to initialize memory, perhaps to calculate parameters to be passed to other processors in the system or even to determine if a self test passed or failed, then running this code may be done fastest by switching the processor to protected 32-bit or 64-bit mode and using the processor registers and cache most efficiently.

Even simple tasks like fetching code into the processor can be made more efficient by storing the code in a compressed form, expanding it into system memory and then executing it from there, because the flash device holding the compressed image is usually slow enough compared to the system memory that the decompression will take less time than the additional flash read cycles required if the code were to be executed directly from the flash memory device.

Of course, any experimentation and evaluation of profile data takes engineering time, but if the pre-boot execution time is critical, this is an important tradeoff to consider. The cost is difficult to estimate because the information needed to estimate the cost will not really be available until after the fact.

**Cost Tradeoffs**

There are many costs that go into a project. The material costs and any licenses or royalties are the easiest to work with, and the ones that are first in the minds of the project manager. Usually the labor costs and the management of those labor costs are included today with little effort. Occasionally the cost of service returns, loss of goodwill, and end of life factors will also get considered.

But some additional costs that should be considered are nonrecurring: the time spent on the project architecture and planning the engineering and preproduction schedules, the time spent engineering the project (either as an external NRE project with its management costs or as an allocation of internal engineering resources for the duration of the engineering phase of the project, organizing the actual manufacturing processes and their supervision, and finally those costs associated with mistakes, redesigns, and other difficult to estimate factors. For example, the cost of experimentation mentioned in the last topic can be easily controlled, but cutting off experimentation after a fixed number of hours will not result in the most profitable product. Unfortunately, more cost effective
approaches to control expenditures on experimentation tasks make precise development planning time estimates much less accurate.

Planning for Reuse

The Need for a Structured Boot Firmware Design

The Impact on Future Projects

Why does this matter to us?

If a first project involves a large amount of engineering effort, either purchased NRE or internal development, the reuse of the results of the consumed resources is very attractive. In this part of the paper, we will assume that there is a desire to reuse it. Reuse can have a wide range of costs: it can be trivially inexpensive if the changes to the product are very small or if the changes are sandboxed by standard interfaces or effective code partitioning. But if neither of these are the case, then the new platform's boot firmware may need to be developed again from its baseline unspecialized boot loader or BIOS code.

A limited new project, perhaps to reduce costs

Perhaps components were included in the original design that never proved worth using. Perhaps a new hardware interface is available that is (mostly) software compatible with previous hardware, but much less expensive. There are many such relatively small hardware modifications that require a small, localized change to the boot firmware.

These changes are not significantly easier or harder to implement in any design. Depending on the details of the change it could be easier to modify tables in a Framework based UEFI BIOS, or it could be easier to insert a small block of code in a legacy BIOS or a boot loader.

A significant new project, perhaps changing processors and chipsets

Probably the most common embedded system porting task is the need, every few years, to update the platform hardware and the boot firmware has to be updated to the new hardware. Sometimes the software demands on the system have grown to the point where a faster processor or more memory have become
critical. On the other hand, the system may still meet all its requirements, but the processor it was designed around just will not be available any more. Or perhaps the system can be made smaller, more energy efficient and less expensive to manufacture. Any of these events could force a redesign.

There are two kinds of code that need to be updated here. Work-around code that needs to be executed at particular stages of the boot process to handle errata or to permit reliable booting to the platform software that the new hardware does not require might need to be removed, and likewise new work around code may need to be added for the new processor's errata or to handle faster devices or the new faster processor's improved ability to recognize glitches on older interfaces that were not visible to the previous processor. These fragments may need to be translated from one language or pseudo code to another language, but the porting task is effectively the same no matter what boot firmware is being ported (a UEFI BIOS, a legacy BIOS, InsydeDIY, the BLDK or a custom written legacy boot loader).

A second kind of code is usually referred to as reference code. Unlike the first set of code fragments, the reference code will likely be provided as source or object code and be larger components. Reference code will also expose a significant interface to the boot firmware. If the exposed interface can be easily integrated into the firmware, as is usually the case for a UEFI BIOS or boot loader, then this task becomes almost trivial. If the exposed interface needs a large translation layer or if the reference code requires significant adaptation to run in the context provided by the boot firmware, the integration task is no longer a small task. It can overwhelm the rest of the firmware modification.

To provide long term support for the reference code modules that might appear in an embedded system's boot firmware, a wrapper that isolates the reference code from the rest of the system is a useful approach. This wrapper can remap the address space and reconfigure the cache systems or other parts of the platform hardware that are used differently in the boot firmware than in the reference design that the reference code was designed to work in.

A major new project, perhaps changing from a boot loader to a BIOS

In this case, foresight (or just serendipity) is very valuable. With the Intel Framework the basis both for the old code and for the new code the transition is not as much an effort as it could otherwise be. For example if a project involves changing from one Intel Framework based IBV code base to another, the code can just be moved from one directory to another, possibly editing the include and configuration files to some extent, but otherwise, it just works. Likewise, moving from an InsydeDIY boot loader to InsydeH2O or another IBV's
Framework based code will involve little or no editing to port the code to the new platform.

On the other hand, if one needs to move to or from a Coreboot or BLDK environment it should be expected that at least some adaptation of the interfaces of any custom developed code being moved to the new platform will be a requirement.

Even in this case, though, if the custom code does not interact with the rest of the boot firmware the adaptation may well involve very little editing: for example if it is self contained code that just needs to be executed at a particular stage of the boot process, a small edit to an include file and inserting a simple call at the right place may be all that is needed.

More complex scenarios, for example, changing the way USB devices are detected, or changing the configuration of a custom peripheral that might have memory mapped I/O base address registers that cannot always be located at the same address space (perhaps that might be located in the 32-bit address space sometimes and in the 64-bit address space in other cases), could require rewriting significant blocks of interface code.
For More Information

For more information on Insyde Software products, including Insyde H2O and InsydeDIY, please check out the Insyde Software website: [http://www.insydesw.com](http://www.insydesw.com).

For more information on Intel’s Boot Loader Development Kit (BLDK) as well as a vast amount of other information useful to embedded system developers interested in designing with Intel processors and other products, please check out the Intel Embedded Design Center: [http://edc.intel.com](http://edc.intel.com).

For more information on coreboot v4 as well as earlier versions, please check out the coreboot website: [http://www.coreboot.org](http://www.coreboot.org).

Other websites providing information useful to an engineer involved in preboot software, whether for embedded systems or not, are the Distributed Management Taskforce, DMTF, website ([http://www.dmtf.org](http://www.dmtf.org)) where information on the SMBIOS tables used to pass details of the system structure to operating systems are maintained, the website devoted to the Advanced Configuration and Power Interface ([http://www.acpi.info](http://www.acpi.info)), the Unified EFI Forum website ([http://www.uefi.org](http://www.uefi.org)) where the UEFI standards and test suites are maintained, and the Sourceforge project that encompasses the open source parts of the Tiano project that was the origin of the EFI Framework mentioned in this paper ([http://sourceforge.net/apps/mediawiki/tianocore](http://sourceforge.net/apps/mediawiki/tianocore)).

Two books from Intel Press are useful to understand the UEFI standards, reasons they were developed and how they are used today:

<table>
<thead>
<tr>
<th>Beyond BIOS</th>
<th>by Vincent Zimmer, Michael Rothman and Robert Hale</th>
<th>Covers how the UEFI standards and forum came to be, and how they structure preboot firmware today.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harnessing the UEFI Shell</td>
<td>by Michael Rothman, Tim Lewis, Vincent Zimmer and Robert Hale</td>
<td>Covers the operation and use of the UEFI shell, a feature often provided by a UEFI BIOS to support test, manufacturing process and other command line or script based activities.</td>
</tr>
</tbody>
</table>