Reducing Platform Boot Time

UDK 2010 Based Performance Optimization

January 2011

Authors

Michael Rothman
BIOS Architect
Intel Corporation

Genliu Xing
Senior BIOS Engineer
Intel Corporation

Yan Wang
Technical Marketing Engineer
Intel Corporation

Jiong Gong
Senior BIOS Engineer
Intel Corporation
Executive Summary

This document presents a series of methods that should enable a BIOS engineer to optimize the underlying platform BIOS so that it can reduce a platform’s boot time. However, it should be noted that the intent of this whitepaper is to illustrate how various, seemingly unrelated, product requirements can greatly affect the resulting platform boot performance. That being said, this whitepaper illustrates how the platform design based on marketing requirements, coupled with a properly constructed UEFI-compliant BIOS can greatly affect the performance characteristics of a platform.

Some of the key points in this white paper:

- How specific marketing requirements affect boot performance.
- Suggestions on what BIOS engineering choices can be made to optimize for a given platform requirement.
- Provide a realistic view of what performance enhancements can be done in a production BIOS.
- Establish viable next steps.
# Contents

**Executive Summary** .................................................................................................................. 2  
**Contents** ...................................................................................................................................... 3  
**Background** ................................................................................................................................. 5  
  - Overall UEFI architecture flow ........................................................................................................ 5  
**Marketing Requirements** ............................................................................................................. 9  
  - What are the design goals? .............................................................................................................. 9  
  - Is pre-OS user interaction always needed? ................................................................................... 9  
  - Can the system use system setting from last boot? .................................................................... 10  
  - What are the supported OS targets? .............................................................................................. 10  
  - Do we have to support legacy option ROMs? .............................................................................. 11  
  - Are we required to display an OEM splash screen? .................................................................... 12  
  - What type of boot media is supported? ......................................................................................... 12  
  - What is the BIOS recovery/update strategy? ............................................................................... 13  
**Proof of Concept** ......................................................................................................................... 14  
**Performance Measurement in UDK 2010** ..................................................................................... 16  
  - How does the performance measurement infrastructure work? ............................................... 16  
  - How to build in the performance measurement support? ....................................................... 16  
  - How to instrument code for performance profiling. .................................................................. 17  
  - How to interpret the performance measurement data? ........................................................... 19  
**The Guidelines of Performance Optimization** ............................................................................... 22  
  - [Guideline 1] Make good use of system cache ........................................................................... 22  
  - [Guideline 2] Organize the FLASH layout effectively .................................................................. 22  
  - [Guideline 3] Adjusting the BIOS to avoid unnecessary drivers ............................................... 23  
  - [Guideline 4] Avoid reset in drivers ........................................................................................... 23  
  - [Guideline 5] Use saved data ...................................................................................................... 23  
  - [Guideline 6] Do work in parallel ............................................................................................... 23  
**Practice of Performance Optimization in UDK2010** .................................................................... 25  
  - How to make good use of system cache .................................................................................... 25  
  - How to organize the FLASH layout effectively ........................................................................... 28  
  - How to adjust the BIOS to avoid unnecessary drivers ............................................................... 31  
  - How to avoid reset in drivers ....................................................................................................... 34  
  - How to use saved data .................................................................................................................. 35  
  - How to do work in parallel .......................................................................................................... 36  
**Conclusion** .................................................................................................................................... 37  
  - The Primary Adjustments ........................................................................................................... 37  
  - Suggested Next Steps .................................................................................................................. 38  
**Reference Documents** ................................................................................................................. 40  
**Acronyms** ...................................................................................................................................... 42
Reducing Platform Boot Time in UDK 2010

Figures:

Figure 1 – SEC Phase ................................................................. 6
Figure 2 – PEI Phase ................................................................. 7
Figure 3 – DXE and BDS Phase ................................................. 8
Figure 4 – Performance Comparison between Fast Boot and Normal Boot .... 15
Figure 5 – Performance of Enable/disable code cache in SEC phase ........ 26
Figure 6 – Performance of Enable/disable code cache in PEI (Pre-Memory) phase ........................................................................ 27
Figure 7 – Performance of Enable/disable code cache in PEI (Post-Memory) phase ........................................................................ 28
Figure 8 – The number of Enabled FVs impacts DXE phase time .......... 31
Figure 9 – Deconstructing the BDS launch of the Boot Target ............ 32
Figure 10 – Boot Time Comparison between Quick Boot Path and Full Configuration Path ................................................................. 34
Figure 11 – Boot Time Comparison between Disable and Enable Full Reset in PS2 Keyboard ........................................................................ 35
Figure 12 – How to achieve Fast Boot ............................................. 38
Background

This document will focus on specific aspects of a platform’s pre-O/S boot behavior and leverage concepts that are based on the UEFI BIOS architecture for which details can be found at www.uefi.org.

When examples are used to illustrate points being made in this whitepaper, the authors have used the UDK 2010 codebase, which is accessible through www.tianocore.org. This codebase is a UEFI compliant BIOS which members of the industry can use as a reference. Ultimately the problems and solutions encountered with performance optimization apply equally to all types of codebase implementations. This document allows for a solid how-to set of examples which can be applied to those who use UDK 2010 compliant codebases, but similar concepts and capabilities can equally well be applied to other codebases.

Overall UEFI architecture flow

Some of the fundamental things that need to be understood are different phases of platform initialization and how they are exercised as part of the platform boot process. The following flow diagrams illustrate the evolution of the platform initialization from the first moment that power is applied until the point where the BIOS hands-off to the target O/S:
Reducing Platform Boot Time in UDK 2010

Figure 1 – SEC Phase

Reset Vector
Flush cache and jump into main initialization routine in the ROM.

Switch to protected mode
Transition to a non-paged flat-model protected mode

Initialize MTRRs for BSP
Set cache states for various memory ranges to a known state.

Microcode Patch Update
Execute Microcode Patch Update for all of the present CPUs. (Common process, but an optional behavior in closed-box controlled configuration systems)

Initialize No-Eviction Mode (NEM)
Prior to the discovery of memory on the platform, a data area will be established within the CPU cache so that a stack-based programming language can be used early in the initialization.

Various early BSP/AP interactions
A series of standard steps which contain some fixed delay events such as:
Send INIT IPI to all APs
Send Start-up IPI (SIPI) to all Aps
Collect BIST data from the APs

Hand-off to PEI entry point
Figure 2 – PEI Phase

**Establish use of "memory"**
Transfer services from being ROM-based to data running from early memory (e.g. CPU cache). This includes the presence of PEI services such as memory, PEI module interfaces, and security.

**CPU PEIM**
Module which exposes a series of CPU-related functions. Some of these functions are the CPU Cache interface (Set/Reset), and CPU Frequency Select Interface.

**Miscellaneous Platform PEIM**
Executes a series of early hardware initialization such as memory controller hub (MCH) init, I/O controller hub (ICH) init, initialize built-in platform interfaces (e.g. Stall, SMBUS Policy, Reset, etc.). Also determines what the boot mode is we are currently booting with (e.g. Normal, Recovery, S3, etc.). This is also where the platform exposes the boot mode so that subsequent modules can potentially have boot mode based behavior.

**Memory Initialization PEIM**
Execute Memory Initialization for the platform. Assign memory for remainder of PEI and subsequent boot phases. In this case, some optimizations are enabled for performance such as eliminating memory test during S3 resume or re-programming captured memory reference code state in S3 resume mode.

**Are we in an S3 Boot mode?**

**Multiprocessor CPU PEIM For S3 Boot Mode**
Initializes a variety of components within the CPU domain with optimizations associated with S3. Basic initialization of CPU to establish various CPU-specific settings (e.g. VMX, SMRR, Thermal Throttling settings, MTRR Synchronization, etc.)

**S3 Boot Script Executor**
Executes the S3 Boot Script to re-establish hardware programming in a very low-overhead manner.

**O/S Resume Vector**

**Hand-off to DXE entry point**

---

Dashed Boxes or lines are informational.
Reducing Platform Boot Time in UDK 2010

Figure 3 – DXE and BDS Phase

Establish DXE infrastructure
The Driver eXecution Environment (DXE) is established based on the discovered resources described by the prior PEI phase of operations. This includes DXE core callable interfaces, event services, and the eventual launch of the DXE dispatcher.

DXE Dispatcher
The dispatcher is tasked with the job of discovering the FV (firmware volume) components that are available and processing them. Each of the discovered drivers within the FV is scheduled to be launched if and when their dependencies are met. Once a driver is scheduled to run, the dispatcher will proceed to launch the scheduled drivers and continue to do so until there are no more scheduled drivers.

Boot Device Select Phase
Based on the programmed boot variable, the Boot Device Select (BDS) phase ultimately will attempt to connect the boot devices required to load and invoke the selected boot target (e.g. O/S). This usually encompasses a recursive search for additional FVs and content to dispatch from them.

Architectural Protocols
Some of the key drivers needed for the core to operate. Some of these are the BDS, CPU, Timer, etc.

Hand-off from PEI to DXE Core

Dashed Boxes or lines are informational.

Discovering Components
During the search for FVs, various drivers can be discovered and potentially launched. Some of these drivers are components such as network drivers, I/O drivers (e.g. USB/PCI), and any OEM or platform specific drivers.

Can the boot target be loaded?
No

Have we made progress since last attempt?
Yes

Dispatch new DXE drivers

Load new boot option

Platform Policy
When no viable boot options exist, the platform will have some built-in boot behavior that is specific to the manufacturer of that platform.

Are there more boot options to try?
Yes

Hand-off to the Boot Target

Given the above information, the remainder of the document focuses on the important elements when considering how to best optimize some of the aforementioned behavior so a platform meets both its technical and marketing requirements yet achieves an optimal boot speed.
Marketing Requirements

Admittedly, marketing requirements are not the first thing that comes to mind when an engineer sits down to optimize a BIOS’s performance. However, the reality is that marketing requirements form the practical limits for how the technical solution can be adjusted.

The highlighted requirements are the pivot points in which an engineer can make decisions which will ultimately affect performance characteristics of the system. Since this section details the engineering responses to marketing-oriented requirements, it does not provide a vast array of code optimization “tricks”. Unless there is a serious set of implementation bugs in a given codebase, the majority of boot speed improvements are achieved from following the guidelines provided in this section. Not to worry though, there are code base independent “tricks” included which provide additional help.

What are the design goals?

Is the system designed to be an open one which allows for the hardware configuration to be easily changed (e.g., slots for new devices, or changing old devices)? Is there an intrusion detection mechanism or another method to know that the hardware has changed? Does the system support hot plug memory or processors?

How does the user need to use the platform? Is it a “closed box” system? Is it a traditional desktop? Is it a server? How the platform is thought of will ultimately affect what the user expects. Making conscious design choices to either enable or limit some of these expectations is where the platform policy can greatly affect the resulting performance characteristics.

Is pre-OS user interaction always needed?

Normally the overall goal is to boot the target OS as quickly as possible and the only expected user interaction is with the OS. That being said, the main reason for people today to interact with the BIOS is to launch the BIOS Setup. Admittedly, there are some settings within this environment which are unique and cannot be properly configured outside of the BIOS. However, it is possible that the system only allows user interaction in certain special cases instead of always showing a user-visible prompt and waiting for user interaction.
Can the system use system settings from the last boot?

Normally, scanning and setting the system hardware are a time consuming process if the system is not hot pluggable. This is especially true for memory and processor hot plug in servers. The ability to use the settings saved from the last boot can save a lot of time. For example, reading SPD data from SMBUS, or detect PCIe/PCI video devices, etc.

Platform Policy

One of the first considerations when looking at a BIOS and the corresponding requirements are whether or not an engineer can limit the number of variables associated with what the user can do to the system. For instance, it might be reasonable to presume that in a platform with no add-in slots, a user will not be able to boot from a RAID controller since the user cannot physically plug one in.

This is where a designer enters the zone of platform policy. Even though a platform may not expose a slot, the platform might expose a USB connection. A conscious decision needs to be made for how and when these components are used. A good general performance optimization statement would be:

“If you can shift something from a BIOS task to an OS task – do it!”

Since a user can connect anything from a record player to a RAID chassis via USB, the user might think that they would be able to boot from a USB-connected device. Though this is physically possible, it is within the purview of the platform design to enable or disable such a behavior.

Trick: In this particular platform, the decision was made not to support booting from USB media and not to support the user interrupting the boot process. This means that during the DXE/BDS phase, the BIOS was able to avoid initializing the USB infrastructure to get keystrokes and this resulted in a savings of nearly 0.5 second in boot time.

Note: Even though 0.5 second of boot time was saved by eliminating late BIOS USB initialization, upon launching the platform OS, the OS was able to interact with plugged-in USB devices without a problem.

Platform policy ultimately affects how an engineer responds to the remaining questions.

What are the supported OS targets?

Understanding the requirements of a particular platform-supported OS will greatly affect what optimization paths can be taken in the BIOS. Since
many “open” platforms have a wide variety of operating systems that they choose to support, this limits some of the choices available. In the case of the proof-of-concept platform, there were only two main operating systems that were required to be supported. This enabled the author to make a few choices that allowed the codebase to save roughly 400 ms of boot time by avoiding the reading of some of the DIMM SPD data for creating certain SMBIOS records since they weren’t used by the target operating systems.

**Note:** Changes in the BIOS codebase which avoided the unnecessary creation of certain tables saved roughly 400 ms in the boot time.

**Do we need to support legacy operating systems?**

The main consideration was whether a particular OS target was UEFI-compliant or not. If all the OS targets were UEFI-compliant, then the platform could have saved roughly 0.5 second in the underlying initialization of the video option ROM. In this case, we had conflicting requirements: one was UEFI-compliant and one was not). There are a variety of tricks that could have been achieved by the platform BIOS when booting the UEFI-compliant OS but for purposes of keeping fair measurement numbers, the overall boot speed numbers reflect the overhead of supporting legacy operating systems as well.

**Trick:** To save an additional 0.5 second or more of boot time when booting a UEFI-compliant OS, the BDS could analyze the target BOOT#### variable to determine if the target was associated with an OS loader – thus it is a UEFI target. The platform in this case at least has the option to avoid some of the overhead associated with the legacy compatibility support infrastructure.

**Do we have to support legacy option ROMs?**

Whether or not to launch a legacy option ROM depends on several possible variables:

- Does the motherboard have any devices built in which have a legacy option ROM?
- Does the platform support adding a device which requires the launch of a legacy option ROM?
- If any of the first two are true, does the platform need to initialize the device associated with that option ROM?

One reason why launching legacy option ROMs is dangerous for boot performance is that there are no rules associated with what a legacy option ROM will do while it has control of the system. In some cases, the option ROM may be rather innocuous regarding boot performance, but in other instances that is not the case. For example, the legacy option ROM is that it could attempt to interact with the user during launch. This normally involves advertising a hot-key or two for the user to press, which would
Reduce the BIOS in finishing its job for however long the option ROM pauses waiting for a keystroke.

**Trick:** For this particular situation, we avoided the launching of all of the drivers in a particular BIOS and instead opted to launch only the drivers necessary for reaching the boot target itself. Since the device we were booting from was a SATA device for which the BIOS had a native UEFI driver, there was no need to launch an option ROM. This action alone saved approximately 3 seconds on the platform. More details associated with this trick and others are in the Additional Details section.

**Are we required to display an OEM splash screen?**

This is often a crucial element for many platforms, especially from a marketing point of view. The display of the splash screen itself typically does not take that much time. Usually initializing the video device to enable such a display takes a sizable amount of time. On the proof-of-concept platform, it would typically take 300 ms. An important question is how long does marketing want the logo to be displayed? The answer to this question will focus on what is most important for the OEM delivering the platform. Sometimes speed is paramount (as it was with this proof of concept), and the splash screen can be eliminated completely. Other times, the display of the logo is deemed much more important and all things stop while the logo is displayed. An engineer’s hands are usually tied by the decisions of the marketing infrastructure.

**Trick:** One could leverage the UEFI event services to take advantage of the marketing-driven delay to accomplish other things, which effectively parallelizes some of the initialization.

**What type of boot media is supported?**

In the proof of concept platform description, one element was a bit unusual. There was a performance and a standard configuration associated with the drive attached to the system. Though it may not be obvious, the choice of boot media can be a significant element in the boot time when you consider that some drives require 1-5 seconds (or much more) to spin-up. The characteristics of the boot media are very important since, regardless of whatever else you might do to optimize the boot process, the platform still has to read from the boot media and there are some inherent tasks associated with doing that. Spin-up delays are one of those tasks that are unavoidable in today’s rotating magnetic media.

For the proof of concept, the boot media of choice was one which incurs no spin-up penalty; thus a solid state drive (SSD) was chosen. This saved about two seconds from the boot time.
What is the BIOS recovery/update strategy?

How a platform handles a BIOS update or recovery can affect the performance of a platform. Since there are many ways to accomplish this task, this may inevitably be one of those mechanisms which has lots of platform variability. There are a few common cases on how a BIOS update is achieved from a user’s perspective:

1. A user executes an OS application which they likely downloaded from the OEM’s web site. This will eventually cause the machine to re-boot.
2. A user downloads a special file from an OEM’s web site and puts it on a USB dongle and re-boots the platform with the USB dongle connected.
3. A user receives or creates a CD or Floppy with a special file and re-boots the platform to launch the BIOS update utility contained within that special file.

These user scenarios usually resolve into the BIOS, during the initialization caused by the re-boot, reading the update/recovery file from a particular location. Where that update/recovery file is stored and when it is processed is really what affects performance.

When processing things early

Frequently during recovery one cannot presume that the target OS is working. For a reasonable platform design, someone would need to design a means by which to update or recover the BIOS without the assistance of the OS. This would lead to user scenarios #2 or #3 listed above.

The question an engineer should ask themselves is – how do you notify the BIOS that the platform is in recovery mode? Depending on what the platform policy prescribes, this method can vary greatly. One option is to always probe a given set of possible data repositories (such as USB media, a CD, or maybe even the network) for recovery content. The act of always probing is typically a time-consuming effort and not conducive to quick boot times.

There is definitely the option of having a platform-specific action which is easy and quick to probe that “turns on” the recovery mode. How to turn on the recovery mode (if such a concept exists for the platform) is very platform-specific. Examples of this are holding down a particular key (maybe associated with a GPIO), flipping a switch (equivalent of moving a jumper) which can be probed for, etc. These methods are highly preferable since they allow a platform to run without much burden (no extensive probing for update/recovery).
Proof of Concept

In the proof of concept for this paper, total boot time is measured as the time between the CPU first having power applied and the transferring of control to the boot target (which is typically the OS). This particular white paper does not focus on the specifics of the hardware design itself since the steps that are described are intended to be platform agnostic. However, for those that absolutely must know from what type of platform some of the numbers are derived, they are:

- Intel® Core™ i5-520E Processor (3M Cache, 2.40 GHz) laptop design
- 1 GB DDR3 memory
- 4 MB Flash
- Intel® Solid State Drive X25-E (Intel® X25E SSD)

It should also be noted that this proof of concept was intended to emulate real-world expectations of a BIOS, meaning that nothing was done to achieve results which could not reasonably be expected in a mass-market product design. The steps that were taken for this effort should be easily portable to other designs and should largely be codebase independent.

Here are the performance numbers achieved while maintaining all of the various platform/marketing requirements for this particular system. The system boot time is reduced from 17.486s to 1.55s.

**Note:** Even in the same test platform with the same BIOS image, the boot time could be a little different in different boot cycles. And this kind of error could not be avoided. So all the data in this paper is an average value based on several iterations of the experiment. The data may have some variation of roughly 10 ms.
Reducing Platform Boot Time in UDK 2010

Figure 4 – Performance Comparison between Fast Boot and Normal Boot

Boot Time Comparison between Fast Boot and Normal Boot

<table>
<thead>
<tr>
<th></th>
<th>Fast Boot Time (ms)</th>
<th>Normal Boot Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDS</td>
<td>1056</td>
<td>9768</td>
</tr>
<tr>
<td>DXE</td>
<td>179</td>
<td>1423</td>
</tr>
<tr>
<td>PEI</td>
<td>275</td>
<td>6162</td>
</tr>
<tr>
<td>SEC</td>
<td>40</td>
<td>111</td>
</tr>
</tbody>
</table>

Normal Boot Time: 17.464s

Fast Boot Time: 1.55s
Performance Measurement in UDK 2010

Before we start to optimize the performance of a platform, we need to know how to measure the performance first. In UDK 2010, a performance measurement tool, DP (Dump Performance), is provided to measure the UDK 2010 code base firmware. This chapter gives a brief introduction of the tool and UDK 2010 performance measurement infrastructure.

How does the performance measurement infrastructure work?

The UDK 2010 firmware provides both tracing and measured profiling capabilities, each accessed via two macros; one is for starting measurement and the other is for ending the measurement. The start and end times, in timer ticks, for the region between the two macros is then saved in a list in memory which records the measurement’s name and time. Each invocation of the pair of trace macros creates a new record in the list, thus providing a chronological record of invocation and associated timing. Invocations of the profiling macros result in the cumulative time being updated in a single record associated with that measurement.

Then we can use the DP tool which is a Shell application to generate performance reports based upon the performance data recorded by the UDK 2010 performance infrastructure. It will dump all data contained in the performance measurement list, for all phases of execution from PEI through DXE and BDS. The performance infrastructure is responsible for preserving performance data across execution phase boundaries and ensuring that a single measurement record format is used for all records in the performance list.

How to build in the performance measurement support?

Profiling Instrumentation is added to the code by application of two performance measurement macros: PERF_START and PERF_END.

PERF_START is a macro that creates a record for the beginning of a performance measurement.

PERF_END is a macro that fills in the ending time of a performance measurement.

Both of the macros have four parameters, that are Handle, Token, Module and TimeStamp:
Reducing Platform Boot Time

in UDK 2010

- Handle is the Pointer to the environment specific context used to identify the component being measured.
- Token is the Pointer to a Null-terminated ASCII string that identifies the component being measured.
- Module is the pointer to a Null-terminated ASCII string that identifies the module being measured.
- TimeStamp is the 64-bit time stamp.

These two macros work as a pair with PERF_START starting a new measurement and PERF_END completing the measurement. If PERF_START is called for a new measurement, a new record is created. Subsequent PERF_END or PERF_START invocations which use the same Token will update the record.

For example:

```c
MyFunction( void )
{
    PERF_START (NULL," MyFunction ", NULL, 0);
    // Body of the function
    PERF_END   (NULL," MyFunction ", NULL, 0);
    return;   // The single point of return
}
```

The two macros which will count the number of times MyFunction is executed and accumulate the time spent in each invocation of the function.

How to instrument code for performance profiling

First, update the target platform DSC file.

The DSC file is the EDK II Platform Description file. Please refer to the EDK II DSC File Specification for more details. In the DSC file, edit entries within the appropriate [LibraryClasses] Performance Optimization sections to ensure that the correct timer and Performance libraries are used.

- Keep BasePerformanceLibNull for the PerformanceLib instance in [LibraryClasses.common]. This will take care of any cases where it is not explicitly overridden.
- Add instances of PeiPerformanceLib to [LibraryClasses.common.PEI_CORE] and [LibraryClasses.common.PEIM]
Reducing Platform Boot Time in UDK 2010

- Add an instance of DxeCorePerformanceLib to
  [LibraryClasses.common.DXE_CORE]
- Add instances of DxePerformanceLib to
  [LibraryClasses.common.DXE_DRIVER],
  [LibraryClasses.common.DXE_SMM_DRIVER],
  [LibraryClasses.common.UEFI_DRIVER] and to
  [LibraryClasses.common.DXE_RUNTIME_DRIVER] and
  [LibraryClasses.common.UEFI_APPLICATION] if either of those areas are
  applicable.
- Determine which area(s) you wish to profile. Ensure that the Timer
  Library for that area is appropriate. Not all Timer Libraries work for all
  phases. It is necessary to ensure that all performance measurements are
  made using the same Timer Library instance. The recommended Time
  Library instance which supports all phases is the one under
  PerformancePkg and the location is PerformancePkg\Library\TscTimerLib
- Edit the [LibraryClasses] sections again to select the correct ProfileLib
  instances: ProfileLibNull, PeiProfileLib, DxeCoreProfileLib and
  DxeProfileLib. Edit the [PcdsFixedAtBuild] sections to ensure that
  PcdMaxPeiPerformanceLogEntries is sufficient,
  PcdPerformanceLibraryPropertyMask is 1 and
  PcdProfileLibraryPropertyMask is 1.

Second, update the Module’s INF file.

The INF file is the EDK II build information file. Please refer to the EDK II
INF File Specification for more details. Edit the [Packages] section of the
module’s INF file. The trace facility, implemented in the PerformanceLib, is
entirely contained within the MdePkg and MdeModulePkg packages as
shown below. The profiling facility is provided by the PerformancePkg

The [LibraryClasses] section of the INF file may need to have entries added
for TimerLib, PerformanceLib, or ProfileLib.

[Packages]
  MdePkg/MdePkg.dec
  MdeModulePkg/MdeModulePkg.dec
  PerformancePkg/PerformancePkg.dec

Third, add the instrumentation

Surround the code to be measured with the appropriate START and END
macros as described before. One must take care that all code paths are
covered so that the measured section cannot be exited without
encountering an END macro.
An example of this can be found in DxeMain where the duration of the DXE CoreDispatcher is measured.

```c
//
// Invoke the DXE Dispatcher
//
PERF_START (NULL, "CoreDispatcher", "DxeMain", 0);
CoreDispatcher ();
PERF_END (NULL, "CoreDispatcher", "DxeMain", 0);
```

**How to interpret the performance measurement data?**

The DP tool produces three types of reports: Grouped, Sequential, and Raw. The Grouped report is the default type report. A Sequential report is selected with the `-A` command line option and the Raw report is selected with `-R`.

**Interpret the grouped report**

The Grouped report shows every major phase time, the PEIM and driver time and some other related information. There are many sections in the Grouped report. We choose two typical sections to interpret.

One sample is the Major Phases Section of Grouped report:

```plaintext
==[ Major Phases ]========
SEC Phase Duration: 111000 (us)
PEI Phase Duration: 6162 (ms)
DXE Phase Duration: 1423 (ms)
BDS Phase Duration: 9768 (ms)
Total Duration: 17464 (ms)
```

It shows the SEC, PEI, DXE and BDS total duration time is 17464ms, that is 17.464 s. And please note that the duration for the SEC phase is given in microseconds, us, while others are in milliseconds, ms. This is because the SEC phase may vary between less than and greater than one ms.

Another sample is Drivers by Handle section of Grouped report:

```plaintext
==[ Drivers by Handle ]========
Index: Handle Driver Name Description Time (us)
------------------------------------------------------------
1091: [AC] LegacyBiosDxe StartImage: 15850
2355: [B0] PciBusDxe DB:Start: 9955
```
Reducing Platform Boot Time in UDK 2010

3431: [B5] UsbKbDxe DB:Start: 10650

The third line of the section’s meaning is □

3431 - The Index of the matching measurement record.

[B5] - The measured driver handle.

UsbKbDxe - The name of the driver. Here is the USB Keyboard DXE driver.

DB:Start - A description which is the value of the Token parameter from the associated performance macros.

10650 - The driver took 10.65 milliseconds to execute.

**Interpret the sequential report**

Sequential trace reports are used when the temporal relationship between measurements is desired, each measurement displayed starts after measurements with a lower index number and before measurements with a higher index number. Only the starting relationship is shown, not the duration.

For example, the following report excerpt shows measurements during the PEI phase:

```plaintext
==[ Sequential Trace Records ]========
Index Handle Module Token ET (us)
--------------------------------------------------------------------------------
2: 0x00000000 PEI 2913812
3: 0x00000000 PreMem 2382889
4: 0x00000000 PEIM 1592
14: 0x00000000 PEIM 598
15: 0x00000000 PEIM 2343782
16: 0x00000000 PostMem 524321
17: 0x00000000 DisMem 26343
19: 0x00000000 PEIM 1064
```

**Interpret the raw report**

The raw trace Report is used to see exactly what is contained in the measurement records.

The following shows an excerpt example:

```plaintext
==[ RAW Trace ]========
Index Start Count End Count Module
--------------------------------------------------------------------------------
2: 0000000017992819 000000001A0EE3DE5 PEI
3: 0000000017992819 0000000158F89595 PreMem
```
16: 0000000158FB0CD1 00000001A0BFDD5 PostMem

From the report you can see the Start Count and the End Count of the PEI phase. It is 1895515CC (00000001A0EE3DE5 minus 0000000017992819). From the heading of every type report, we can get the frequency information.

The following is the heading of the report:

- DP Build Version: 2.2
- System Performance Timer Frequency: 2,266,956 (KHz)
- System Performance Timer counts UP from 0x0 to 0xFFFFFFFFFFFF

The frequency is 2,266,956 KHz. To get the PEI phase time 2910 ms, divide 1895515CC by the timer frequency.
The Guidelines of Performance Optimization

This chapter describes some guidelines to enhance the system boot performance. These guidelines are not specific for UDK2010, but could be treated as the generic guidelines in firmware performance tuning. There is a chapter which follows that shows some “how-to” examples on implementing these guidelines.

[Guideline 1] Make good use of system cache

CPU cache is the fastest memory bank in the system. Correctly using CPU cache can significantly improve the boot performance especially if the original code does not set the cache optimally.

Since reading data or code from FLASH is very slow, it is very important to cache the range of FLASH when executing code from FLASH in PEI phase.

[Guideline 2] Organize the FLASH layout effectively

In a BIOS which complies with the PI specification, there is a FLASH component concept known as an firmware volume (FV). This is typically an accumulation of BIOS drivers. It is a reasonable expectation that these FVs are organized into several logical collections that may or may not be associated with their phase of operations or functions. The access from FLASH is significantly slower than access from memory. Minimizing the FLASH access will significantly improve the boot performance. For some quick boot paths, FLASH access may take a very large percentage of the boot time.

Some drivers read data or binaries from an FV, such as CSM binary, Legacy Option Rom, BIOS ID, etc. As a common design, the BIOS is designed to scan all FVs. But for a specific platform, files can be well organized so that the code can avoid scanning all FVs.

The less space a BIOS occupies, the shorter the time is for routines within the BIOS to read content into faster areas of the platform (e.g., memory). This can be done by minimizing the drivers that are required by the platform, at least for a specific boot path. For example, if you want the splash screen to appear as early as possible, you can create a FV which only contains drivers required for graphic console. After the graphic console has connected, then it will dispatch other DXE drivers and boot.
[Guideline 3] Adjusting the BIOS to avoid unnecessary drivers

If the system is designed to boot the OS as fast as possible, you can enable a special boot path which only initializes the devices required for the boot.

There are two levels of hardware initialization. The first level is initializing the hardware to make it meet the industry standard infrastructure that the OS phase would expect. This also includes initializing any hardware necessary to discover and launch the OS. For example, set the SATA controller mode to AHCI, native or legacy so that the BIOS can launch the OS.

The second level of hardware initialization is to make certain devices (e.g., USB) which might not be necessary for the pre-OS phase to accessible in the pre-OS phase. Accessing files from USB mass storage devices requires a lot of driver connections in the BDS phase and can take quite a bit of time.

To boot the OS quickly, some of the second level initialization can be skipped. Those devices still can work correctly in OS. For example, if we want to boot OS from hard disk, we can skip USB, Network and CD-ROM initialization which can save a lot of time. When there is no initialization for these type of devices in the pre-OS phase, the OS has native drivers which can initialize these devices when the OS is running. In fact, the OS will typically reinitialize these devices regardless of whether the BIOS has initialized them.

[Guideline 4] Avoid reset in drivers

Some drivers reset the device and wait for the device ready. For example, a PS2 keyboard often-times needs more than 1 second to do a full reset. Even for an SSD device, it takes about 800ms to do reset of that device.

[Guideline 5] Use saved data

If the hardware has not changed, we can use saved data instead of access slow IO to get the data. For example, reading SPD data from SMBUS is much slower than reading it from NVRAM. Since it takes a long time to enumerate the PCI bus, we can use saved data from the last successful boot directly to reduce the overall boot time.

[Guideline 6] Do work in parallel

Firmware does not support multi-tasking like an OS, but we can do some parallel work according to hardware behaviors. For example, the hard disk needs seconds of time to spin up. So we can send the spin up signal to hard
disk as early as possible, then do other initialization work which can save the boot time.

In addition, where devices might provide DMA capabilities, there are methods which can be used to initiate a DMA transaction (i.e., pass a command to a controller to fill a buffer) while the main CPU is actively doing something else.
Practice of Performance Optimization in UDK2010

This chapter provides real “how-to” examples of performance tuning based on the performance optimization guidelines. UDK2010 is chosen as the practice code base. We will show some detailed source code examples in this chapter, which helps you to understand how to apply these guidelines into UDK2010.

How to make good use of system cache

Enable code cache for boot block in Pre-Memory phase

PEI phase is composed of two sub-phases: Pre-Memory (before memory is available) and Post-Memory (after memory is available).

In Pre-Memory phase, setting code cache is critical for boot performance. It is recommended to set the PEI FV to a cacheable state. But if the L2 cache size is not large enough to set the PEI FV cacheable, make sure all Pre-Memory PEIMs could be cacheable which is also helpful for performance.

UDK 2010 supports the dispatch of PEIMs from multiple FVs. A developer can make some adjustments in the FV layout to make sure all Pre-Memory PEIMs are cached during code execution.

Note: During pre-memory phase, the CPU cache is established as a temporary memory for stack and heap. This is often referred to as NEM (No Eviction Mode). Please follow the instructions in processor firmware writer’s guide to make sure CAR and code cache can work at the same time.

Below is the code sample to enable code caching in SEC. It is to set code cache for 128KB FvRecovery:

```
; Set the base address of the CodeRegion cache range
;
mov     eax, PcdGet32 (PcdFlashFvRecoveryBase)
or     eax, MTRR_MEMORY_TYPE_WP
xor     edx, edx             ; clear upper dword
mov     ecx, MTRR_PHYS_BASE_1 ; Load the MTRR index
wrmsr          ; the value in MTRR_PHYS_BASE_1
```
Reducing Platform Boot Time in UDK 2010

; Set the mask for the CodeRegion cache
; mov eax, CODE_REGION_SIZE_MASK OR MTRR_PHYS_MASK_VALID ; turn on the Valid flag
mov edx, 0xf ; edx <- MTRR_PHYS_MASK_HIGH
mov ecx, MTRR_PHYS_MASK_1 ; Load the MTRR index
wrmsr ; the value in MTRR_PHYS_BASE_1

In the figure below, we can see the performance difference between when we enable and disable code cache in SEC and PEI phase (Pre-Memory) by using the example code above. In SEC phase, the boot performance is improved from 111 ms to 40 ms. In PEI (Pre-Memory Phase), it is improved from 1526 ms to 94 ms, a 15x performance improvement.

**Figure 5 – Performance of Enable/disable code cache in SEC phase**

![Performance of Enable/Disable code cache in SEC phase](image-url)
Configure FLASH Area as WP (Write Protect) in Post-Memory phase

In Post-Memory phase, the cache setting will be reset according to system memory configuration. It’s recommended to set the entire FLASH area (>=Image size) cacheable (Write Protect Mode) which makes the FLASH access speed up.

Below is the code how to set the flash area cacheable in UDK2010. For most platforms, it is in memorycallback.c

```c
// Cache the flash area to improve the boot performance for
// both normal boot and S3

Status = MtrrSetMemoryAttribute ( PcdGet32 (PcdFlashAreaBaseAddress),
                                PcdGet32 (PcdFlashAreaSize),
                                CacheWriteProtected);```

The performance improvement of PEI (Post-Memory) is shown in Figure 7, the time of PEI (Post-Memory) phase is reduced from 4507 ms to 197 ms.
Reducing Platform Boot Time in UDK 2010

**Figure 7 – Performance of Enable/disable code cache in PEI (Post-Memory) phase**

### Performance of Enable/Disable code cache in PEI (Post-Memory)

- Flash Area Cached (ms): 197 ms
- Flash Area not Cached (ms): 4507 ms

**Note 1:** If the system memory is larger than 4G, MTRR register may not be enough to set the entire memory and the entire Flash area cacheable. One suggestion is to set memory (below 4G) and Flash area cacheable. And after PEI phase, we need to reset the MTRR to make sure the entire Memory cacheable in DXE core.

**Note 2:** Cache setting can also help to improve driver performance such as video and some other devices which have prefetching memory.

### How to organize the FLASH layout effectively

In DXE phase, minimizing the access of FVs in FLASH is very important for optimal boot speed. This can be achieved by optimizing the flash layout.

Here is how to organize the FLASH layout in UDK2010, which accesses only two FVs (FvMain and NvStorage FV) in DXE phase.

First we need to do some optimization for the FV Hand-Off Block (HOB). The FV HOBs are created in some platform PEIMs and transfer the discovered FV’s information to DXE phase. The DXE core then searches for DXE drivers in the FVs reported by the FV HOBs. Optimally, we would only put the absolutely necessary Flash-FV into FV HOB list or put memory-FV into FV HOB list.

Here are several optimizations on FV HOB:
Reducing Platform Boot Time in UDK 2010

- Only report FvMainCompact FV through EFI_PEI_FIRMWARE_VOLUME_INFO_PPI, then PEI core will decompress it and build decompressed FVMAIN in FV HOBS.
- Only build FV HOBS with FVs that contain DXE drivers. Mostly, the Recovery FV and NVStorage FV do not contain any DXE drivers, so we can remove FvRecovery & NVStorage from the HOBS.

```c
// Build HOB for DXE
switch (BootMode) {
  case BOOT_IN_RECOVERY_MODE:
    Status = InitializeRecovery (PeiServices);
    ASSERT_EFI_ERROR (Status);
    break;
  case BOOT_ASSUMING_NO_CONFIGURATION_CHANGES:
    // Install EFI_PEI_FIRMWARE_VOLUME_INFO_PPI to add this Fv into FvList.
    PeiServicesInstallFvInfoPpi (  
      NULL,  
      (VOID *) (UINTN) PcdGet32 (PcdFlashFvMainBase),  
      PcdGet32 (PcdFlashFvMainSize),  
      NULL,  
      NULL
    );
    break;
  default:
    BuildFvHob (  
      PcdGet32 (PcdFlashFvMainBase),  
      PcdGet32 (PcdFlashFvMainSize)
    );
    BuildFvHob (  
      PcdGet32 (PcdFlashNvStorageBase),  
      PcdGet32 (PcdFlashNvStorageSize)
    );
    BuildFvHob (  
      PcdGet32 (PcdFlashFvRecoveryBase),  
      PcdGet32 (PcdFlashFvRecoverySize)
    );
}
```

**Note:** Sometimes DXE drivers can be split to several FVs for some purpose, for example, fastest splash screen, developer can make decision on hide some FVs until graphic console is connected.
Besides FV HOB is used by DXE core, FvbService driver also needs FV information, which is mainly used for providing FirmwareVolumeBlockProtocol for write/erase flash capability. Normally, it will report Nv Storage FV, FvRecovery FV and FvMainCompact. For the quick boot path, only reporting NvStorage FV is enough in DXE phase to offer variable service.

In function FvbInitialize, adding some code for only reporting NvStorage.

```c
for (Idx = 0; Idx < sizeof (mPlatformFvBaseAddress)/sizeof (mPlatformFvBaseAddress[0]); Idx++){
    BaseAddress = mPlatformFvBaseAddress[Idx];
    // // for quick boot path, only handle NvStorage FV for best performance.
    // If ((BootMode == BOOT_ASSUMING_NO_CONFIGURATION_CHANGES)
    //     && (BaseAddress != FixedPcdGet32(PcdFlashNvStorageBase)){
    continue;
}
```

**Note:** For ECP solutions, the developer can make decision on only report NvStorage FVB through EfiFlashMapHob.

Figure 8 shows how the number of enabled FVs affects performance. By only enabling minimal FVs, we were able to reduce DXE phase time from 1325 ms to 178 ms.
How to adjust the BIOS to avoid unnecessary drivers

To make a quick boot, the BIOS does not need to initialize every device in the system. The basic idea is to initialize as few devices as possible in the quick boot path. For example, during a boot from the Hard Disk, we only need to initialize the Hard Disk, console input devices and console output devices, but not the USB, CDROM or other devices.

Since boot option enumeration is dependent on having connected devices, for the quick boot path, we can assume that the hardware configuration is not changed from the last boot. That is to say, all the boot options are still the same as before. The EFI boot option is defined by EFI device path. This EFI device path is a binary description of where the required boot target is physically located, which provides sufficient information to understand what components of the platform need to be initialized to launch the boot option.

Below is an example of just such a boot option:

```
Acpi(PNP0A03,0)/Pci(1F\1)/Ata(Primary,Master)/HD(Part3,Sig00110011)/"\EFI\Boot"/"OSLoader.efi"
```
By customizing the platform BDS, you can avoid calling routines which attempts to connect all drivers to all devices recursively (e.g., BdsConnectAll()), and instead only connect the devices directly associated with the boot target. Below is an example of that logic:

**Figure 9 – Deconstructing the BDS launch of the Boot Target**

![Diagram](image)

The code below shows how to deal with quick boot path and full configuration boot path (default boot path) in UDK2010.

```c
case BOOT_ASSUME_NO_CONFIGURATION_CHANGES:
    //
    // Perform some platform specific connect sequence
    //
    PlatformBdsConnectSequence();
    //
    // In no-configuration boot mode, we can connect the
    // console directly.
    //
    BdsLibConnectAllDefaultConsoles();
    // Dispatch driver before connect all to avoid connect all many times.
    gDS->Dispatch();
    PlatformBdsDiagnostics(IGNORE, FALSE);
    break;
```

```c
case BOOT_WITH_DEFAULT_SETTINGS:
    //
    // Connect platform console
```
Figure 10 shows that, in the quick boot path, it takes 2151 ms to connect devices. But in the full configuration path, it takes 7412 ms for all the devices in the system (no USB devices populated, no disk in CD-ROM driver). And in the quick boot path, we do not have BDS Front Page, which is mainly used for selecting the boot option. But in the full configuration path, 2272 ms is lost waiting for the user’s interaction to enter BDS Front Page. By customizing the test platform’s BDS, we reduce the device connect time from 7412 ms to 2151 ms. However, 2151 ms is not the final optimal BDS time. We can continue to do further optimization by applying the next guideline.
How to avoid reset in drivers

For Keyboard Driver Enhancement

Do not do a full reset when calling `EFI_SIMPLE_TEXT_IN_PROTOCOL.Reset()` by setting the second parameter as `FALSE`.

```c
FullReset = TRUE;
if (BootMode == BOOT_ASSUME_NO_CONFIGURATION_CHANGES){
    FullReset = FALSE;
}
//
// Reset the keyboard device
//
Status = ConsoleIn->ConIn.Reset (&ConsoleIn->ConIn, FullReset);
```

From Figure 11, we can see that BDS phase time is reduced by optimizing PS2 Keyboard time. And the reduced time is 1088ms.

*Note:* This trick might cause some keyboards to have an abnormal state after initialization. We have always seen that the keyboard works in the UEFI shell.
Reducing Platform Boot Time

with this methodology, but does not have predictable behavior in some Linux distributions if the keyboard driver does not do a full reset.

Figure 11 – Boot Time Comparison between Disable and Enable Full Reset in PS2 Keyboard

How to use saved data

On some platforms, it takes a long time to train hardware, such as PCI express bus, QPI link. We can save the last successful boot data into NVRAM and use the saved data to reduce the hardware training time. We can also avoid accessing slow IO devices by using the saved data. For example, the memory initialization driver currently needs the SPD data. If the system has the mechanism to save the SPD data into NVRAM, and we can directly access the SPD data from NVRAM. Hundreds of milliseconds to several seconds of time can be saved in a normal boot.

Below is an example of how to use a previously saved MRC time.

```c
if ( (BootMode == BOOT_ON_S3_RESUME) || (BootMode == BOOT_ON_FLASH_UPDATE) || (BootMode == BOOT_ASSUME_NO_CONFIGURATION_CHANGES)) {
    //
    // Load Memory configuration data saved in previous boot from variable
    //
    Status = LoadConfig();
```
Reducing Platform Boot Time in UDK 2010

This will save 30 ms per DIMM. In our test platform, it only has 1 DIMM. So it saves 30 ms.

How to do work in parallel

To minimize the time used to connect PATA/SATA hard disk, spinning up the hard disk as early as possible can save the overall boot time. Mostly, we do some hardware initialization to spin up the Hard Disk in PEI phase so that the time used in IDE bus driver can be reduced.

If you can’t avoid resetting hardware in a driver, it is better to separate it to two different stages and to initialize other hardware between them, which will improve overall boot performance.

In our test platform, we use an SSD hard disk, which did not show a big performance improvement. We did not list the experiment data here. But in some other platforms, there will be obvious performance improvement by using a common hard disk. The anticipated improvement is easy to predict since the “spin-up” time of conventional rotating media is very significant (multiple seconds). Since reading from the hard disk is typically one of the last acts of the BIOS initialization, the earlier in the boot process we can initiate the spin-up, the less time will be needed to wait for the spin-up associated with reading content from that media.
Conclusion

Ultimately, the level of performance optimization that is achievable is largely subject to the requirements associated with the platform. Given sufficient probing, there are almost always methods to achieve boot speed gains using some of the techniques highlighted in this white paper. Below, are some of the highlights of items to focus on and areas within each BIOS codebase that deserve further investigation.

The Primary Adjustments

Figure 12 shows the effect of every guideline when you do boot performance optimization. When you use [Guideline 3] to avoid unnecessary drivers, it takes up 47% of overall saved time (normal boot time minus fast boot time). And if you can make good use of cache as referred in [Guideline 1], cache optimization takes up 39% the saved time. So among all the guidelines, these two guidelines help much more than others. If you only have very limited time to do performance optimization, we suggest you to begin your optimization work from these two guidelines first.

Based on various conditions in a platform, the boot behavior can be adjusted to speed up the boot process. Much of this occurs in the BDS, but some areas of optimization may vary per each individual codebase.

- Focus on the marketing requirements.
- Based on the marketing requirements, many decisions which affect boot performance can be made. Open dialog between marketing and engineering helps with this.
- Minimize the use of slow media.
- Scanning for firmware component in a FLASH device can be very slow. Optimize routines which touch slow media.
- No need to poll for setup pages or even initialize a console in some cases.
- Polling for keys or user interaction can be minimized in the BDS.
- Not all hardware needs to be initialized. Often-times only the hardware directly associated with the valid boot target needs to be initialized.
- Tweaks
- Only initiate activity that the BIOS must do, the O/S is often-times going to repeat what the BIOS just did.
• If no hardware changes detected there is no need to reEnumerate various subcomponents.
• May not be a need to probe boot options if we cache the last known valid boot option.

Figure 12 – How to achieve Fast Boot

Suggested Next Steps

By using the five guidelines to optimize boot performance, we reduce the boot time from 17.46s to 1.55s. If you want to get better performance, some common procedures can be applied to all platforms:

• If the system has no requirement to boot legacy OS, pure EFI system is strongly suggested. In the final quick boot path, Legacy VBIOS takes almost half of the boot time. In the experiment, after replace legacy VBIOS with native EFI graphic driver which has better performance, another 750 ms can be saved, so the system can boot OS in 800 ms.
• Minimize the use of slow media.
• Optimize routines which touch slow media. For instance, the variable region is normally stored in flash and not cached due to some reasons; it is very time-consuming to variable search from uncached flash range. It would be a reasonable to maintain a memory copy of variable region so that read variable can directly from memory.

• Investigating behaviors of those devices which requires long time to initialize, if there are long time timeout in the driver, some kind of parallel work can be applied, for example, Spinning up harddisk in PEI phase or using an EFI event.
## Reference Documents

<table>
<thead>
<tr>
<th>Document</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beyond BIOS</td>
<td><a href="http://www.intel.com/intelpress/sum_efi2.htm">http://www.intel.com/intelpress/sum_efi2.htm</a></td>
</tr>
<tr>
<td>Harnessing the UEFI Shell</td>
<td><a href="http://www.intel.com/intelpress/sum_eshl.htm">http://www.intel.com/intelpress/sum_eshl.htm</a></td>
</tr>
<tr>
<td>PI (Platform Initialization) Specification</td>
<td><a href="http://www.uefi.org/specs/">http://www.uefi.org/specs/</a></td>
</tr>
<tr>
<td></td>
<td>This is where the terms SEC, PEI, DXE, and BDS are defined and referenced.</td>
</tr>
<tr>
<td>UEFI Specifications</td>
<td><a href="http://www.uefi.org/specs/">http://www.uefi.org/specs/</a></td>
</tr>
<tr>
<td></td>
<td>This is the OS interface and runtime EFI stuff.</td>
</tr>
<tr>
<td>Tiano Core Features you want to know about, to adopt</td>
<td>EDK II – open source framework portions of Tiano</td>
</tr>
<tr>
<td></td>
<td>EFI Shell – basics about building a shell</td>
</tr>
<tr>
<td></td>
<td>EFI SCT - self compliance testing for UEFI spec</td>
</tr>
<tr>
<td></td>
<td>EFI Toolkit - contains source code and documentation that enables rapid development of EFI base</td>
</tr>
<tr>
<td></td>
<td>ad applications, protocols, device drivers, EFI shells, and OS loaders.</td>
</tr>
<tr>
<td>UEFI and Windows</td>
<td><a href="http://www.microsoft.com/whdc/system/platform/firmware/UEFI_Windows.mspx">http://www.microsoft.com/whdc/system/platform/firmware/UEFI_Windows.mspx</a>;</td>
</tr>
<tr>
<td></td>
<td>Microsoft’s view of a BIOS, a bootloader and UEFI</td>
</tr>
<tr>
<td></td>
<td>This is a lot of miscellaneous stuff we shouldn’t have to do any more</td>
</tr>
<tr>
<td></td>
<td>This is a nice to have for any client controlled by IT</td>
</tr>
<tr>
<td>ACPI Specification</td>
<td><a href="http://www.acpi.info/">http://www.acpi.info/</a></td>
</tr>
</tbody>
</table>
## Reducing Platform Boot Time in UDK 2010

<table>
<thead>
<tr>
<th>Document</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI Specifications</td>
<td>[<a href="http://www.pcisig.com/specifications/">http://www.pcisig.com/specifications/</a>] (PCI conventional, IO Virtualization, PCI Express)</td>
</tr>
<tr>
<td>USB Specification</td>
<td>[<a href="http://www.usb.org/developers/docs/">http://www.usb.org/developers/docs/</a>]</td>
</tr>
<tr>
<td>Compact Flash Specification</td>
<td>[<a href="http://www.compactflash.org/spec_download.htm">http://www.compactflash.org/spec_download.htm</a>]</td>
</tr>
<tr>
<td>ONFI Nand Specifications</td>
<td>[<a href="http://www.onfi.org/documentation.html">http://www.onfi.org/documentation.html</a>]</td>
</tr>
</tbody>
</table>
Acronyms

AL: Afterlife phase. Power down phase

API: Application Program Interface. Programmatic interfaces for the firmware

BDS: Boot Device Selection phase

BFV: Boot Firmware Volume. Code (i.e., PEI and PEIM code) that appears in the memory address space of the system without prior firmware intervention. See also FV.

BIS: Boot Integrity Services

BIST: Built-in Self Test

Boot Device: The device handle that corresponds to the device from which the currently executing image was loaded

Boot Manager: The part of the firmware implementation that is responsible for implementing system boot policy. Although a particular boot manager implementation is not specified in this document, such code is generally expected to be able to enumerate and handle transfers of control to the available OS loaders as well as EFI applications and drivers on a given system. The boot manager would typically be responsible for interacting with the system user, where applicable, to determine what to load during system startup. In cases where user interaction is not indicated, the boot manager would determine what to load and, if multiple items are to be loaded, what the sequencing of such loads would be.

Boot Services: The collection of interfaces and protocols that are present in the boot environment. The services minimally provide an OS loader with access to platform capabilities required to complete OS boot. Services are also available to drivers and applications that need access to platform capability. Boot services are terminated once the OS takes control of the platform.

Capsule: A concept defined in the UEFI specification which is an encapsulated binary that is sent to the underlying UEFI BIOS and often-times processed across a platform reset. The binary is commonly stored in memory by the UEFI BIOS, but there is no architecturally defined requirement for the final resting spot of the encapsulated binary to be in memory.

Compatibility16: A traditional legacy BIOS with the POST and BIOS Setup code removed. Compatibility16 BIOS code executes in real mode

Compatibility BIOS: The combination of both EfiCompatibility and Compatibility16

CPU: Central Processing Unit (or Processor).
Reducing Platform Boot Time in UDK 2010


Depex: Dependency expression. Code associated with each Framework driver that describes the dependencies that must be satisfied in order for that driver to run. Controls order of execution in a Framework dispatch of PEIM and DXE drivers.

Dispatch Entry Point: The entry point that the dispatcher invokes.

Driver: Modular chunk of firmware code that supports chipset or platform features. Reusable in multiple system contexts.

DXE: Driver Execution Environment phase.

DXE Foundation: A set of intrinsic services and an execution mechanism for sequenced control of driver modules.

DXE Services: Services, such as security services and driver services, that are usable by DXE drivers.

EfCompatibiltiy: EFI code that corresponds to EFI compatibility drivers, code that generates data for compatibility interfaces, or code that invokes compatibility services.

EDK: EFI Developer Kit.

FAT: File Allocation Table.

FAT32: FAT32 File System Driver.

FD: Firmware Device. A persistent physical repository that contains firmware code and/or data and that may provide NVS. For the purposes of this architecture specification, the topology of FDs should be abstracted via FVs.

FFS: Firmware File System. A binary storage format that is well suited to firmware volumes. The abstracted model of the FFS is a flat file system.

Firmware Device: See FD.

Firmware Volume: See FV.

Foundation Code: The core interoperability interfaces between modules and in the Framework.

Framework: short for Intel® Platform Innovation Framework for EFI.

FV: Firmware volume. There are one or more FVs in the FS. The FV containing the "reset vector" is known as the Boot Firmware Volume (BFV).

GUID: Globally Unique Identifier. A 128-bit value used to differentiate services and structures in the boot services environment.
HII: Human Interface Infrastructure. A repository of configuration and translation information for localization. Typically used with boot manager and shell to provide a localized user interface.

HOB: Hand-Off Block. A structure used to pass information from one boot phase to another (i.e., from the PEI phase to the DXE phase)

IBV: Independent BIOS Vendor

IHV: Independent Hardware Vendor

NVS: Nonvolatile storage. Flash, EPROM, ROM, or other persistent store that will not go away once system power is removed

ODM: Original Device Manufacturer

OEM: Original Equipment Manufacturer

OpROM: Option ROM

OS: Operating System


PEI: Pre-EFI Initialization phase. Set of drivers usually designed to initialize memory and the cpu so that DXE phase can run. Usually the first set of code run starting from reset.

PEI Foundation: A set of intrinsic services and an execution mechanism for sequenced control of PEIMs

PEIM: Pre-EFI Initialization Module. Modular chunk of firmware code running in PEI that supports chipset or platform features. Reusable in multiple system contexts.

PEI Services: Common services that are usable by PEIMs

PHIT: Phase Handoff Information Table. A HOB that describes the physical memory used by the PEI phase and the boot mode discovered during the PEI phase.

PIC: Position-independent code. Code that can be executed at any address without relocation

POST: Power On Self Test

PPI: PEIM-to-PEIM Interface

RT: Runtime phase. For EFI and the Framework this is after exit boot services has executed and the OS is in control of the system.

Runtime Services: Interfaces that provide access to underlying platform-specific hardware that may be useful during OS runtime, such as time and date services. These services become active during the boot process but also persist after the OS loader terminates boot services.

SMI: System Management Interrupt
SMM: System Management Mode
SOR: Schedule on Request
Tiano: Codename for the Intel Project to develop the Framework
UNDI: Universal Network Driver Interface. Silicon specific driver in the preboot LAN stack that interfaces to SNP and PXEBC
USB: Universal Serial Bus. See http://www.usb.org/ for more information
XIP: Execute In Place. PEI code that is executed from its storage location in a firmware volume