White Paper

A Tour Beyond BIOS Open Source IA Firmware Platform Design Guide in EFI Developer Kit II

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

May 2016

Executive Summary

This paper introduces a design guide for an EDKII open source IA firmware solution. In order to make an open IA firmware solution simple, we demonstrate a firmware design approach with minimal features. The only criteria are 1) It can boot to the OS, and 2) It is secure. We can remove many unnecessary silicon or platform features like Capsule update, Recovery, S3 resume, SMBIOS, EC, Super IO (SIO), I2C, and only enable ACPI & SMM to support booting.

Prerequisite

This paper assumes that the audience has EDKII/UEFI firmware development experience [UEFI][UEFI PI Specification] and FSP knowledge [FSP]. He or she should be familiar with the UEFI/PI firmware infrastructure (e.g., PEI/DXE) [UEFI Book] and know the Intel FSP flow [FSP EAS] [FSP Consumer].

Table of Contents

Overview

Introduction to open source Intel Architecture (IA) firmware

In order to make an IA platform boot, the IA firmware is needed to initialize the silicon and report necessary information to an operating system. An open source IA firmware is a firmware solution having public silicon code and public platform code based upon a public specification, such as [IA32 Manual] [Intel Graphic OpRegion] [Intel TXT] [Intel SGX] [Intel TraceHub] [Intel VT-d] [Baytrail data sheet] [Brasswell data sheet] [Quark data sheet] [Skylake SA data sheet] [Skylake PCH (Sunrise Point) data sheet]. The only binaries should be Microcode and the Intel Firmware Support Package (FSP) binary which contain IP sensitive codes. The benefit of an open source IA firmware solution is that everyone in the world can take the open source IA firmware as an example starting point, build a new platform, and subsequently create a new firmware solution.
Currently the open source IA firmware infrastructure includes [COREBOOT] and [EDK2]

Introduction to EDKII

EDKII is an open source implementation of UEFI PI-based firmware which can boot multiple UEFI-aware operating systems. The EDKII open source project includes several open source IA firmware such as MinnowBoard MAX and Quark. There will be more in the future.

Summary

This section provided an overview of an open platform firmware solution and EDKII.

Open Source IA Firmware Design

Problem statement

Before an open source IA firmware appears, there are many closed source IA firmware solutions in the world. We did research on the UEFI firmware examples of Intel ATOM based small core, Intel Core-i7 based big core client, and Intel XEON based big core server. We came across some common issues, including:

1) Developers need a way to turn on and off of a feature.

   For example, if the Trusted Platofrm Module (TPM) needs to be supported, or if UEFI Secure Boot needs to be supported. It is good to provide such capability, but the problem is too many configurations are provided. We observed one BIOS provides more than 100 configurations to let developers control. Some configurations of those various controls even do not work.

   Sometimes people just say: I want a minimal BIOS to boot, how to select the 100 configurations ?

2) Developers need a way to get the platform configuration data.

   For example, one configuration choice can include "is VT enabled by the end user?" Another control can include if the TSEG SMRAM size is 1M, 8M or 16M, or if there is an Embedded Controller (EC) or DOCK attached on the board. The EDKII BIOS provides many choices on the source of the configuration data. For example, the UEFI specification defined UEFI Variables; UEFI PI specification defined PCD; FSP defined VPD and UPD; silicon reference code defined policy Hob, policy PPI, and policy protocol; silicon specific signed static configuration data blob; and even legacy CMOS region.

   People may ask: which interface should I use in my platform code?

3) Developers need to do porting work from an existing board to a new board.

   There might be GPIO differences, SIO differences. However, some old platform code may use a "switch-case" mechanisms to check the board type, and such "switch-case" usages is scattered in many platform drivers, including AcpiPlatform, SmmPlatform, PlatformInit, EC, ASL code, VFR pages, etc. In order to add a new board on existing platform, a developer has to find out all the places.

   People may think: How can I know how many modules I need to port, and if I have finished updating all required modules?

4) Developers might need to work on a different board.

For example, there might be an ATOM based on a server, a Core-i7 based server, or a XEON based server. However, the BIOS from different segments are different. We once compared an ATOM based firwmare with a Core-i7 based firmware. There are ~20 directories under Platform. Only 2 are same, which are "Include", and "Library". People might need lots of time to ramp up again to get familiar with new platform structure.

Why can't the platform tree structures bear more similarity ?

Goal

Based on the above observation, people may feel that the existing IA firmware is complex and hard to port or enable for a new platform. The purpose of this whitepaper is trying to provide some guidance on how to design an IA firmware solution to meet the goals below:

- **Simple**. Code structure should be obvious and the firmware developer can easily turn on or turn off a big feature.

- **Portable**. Firmware developer can easily port and enable a new board.

- **Consistent**. Firmware code structure should be similar, no matter if it is an ATOM based embedded platform, a Core-i7 based mobile device, or a XEON based server.

Focus

In order to provide suggestions on the problem statement above, we would like to focus on the 4 areas below:

- **Feature**. How does a BIOS provide the feature selection option to a developer?
- **Configuration**. From which interface can a platform module get the configuration data?
- **Porting**. Where are the modules to be ported for a new board?
- **Tree Structure**. What does the EDKII platform package look like?

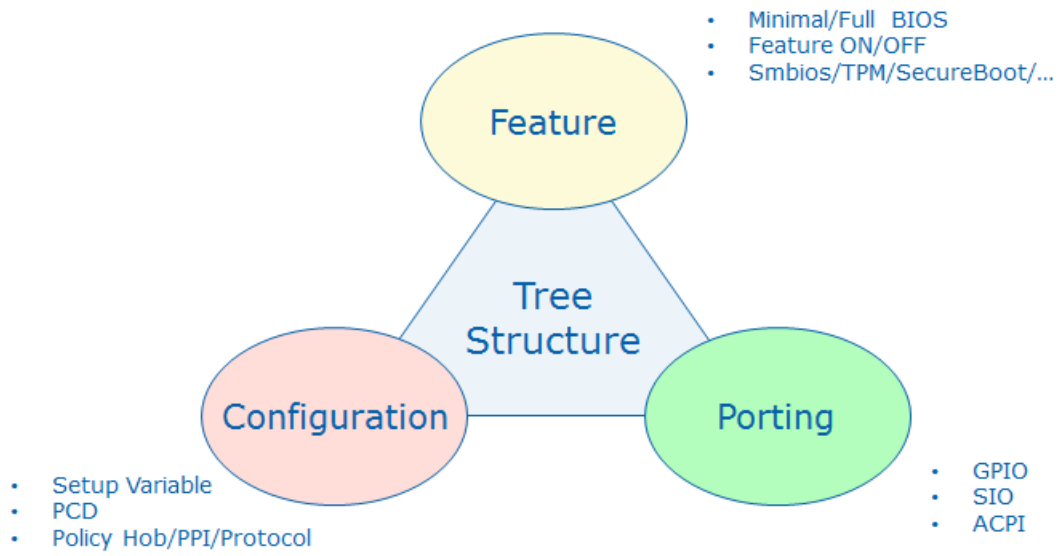


Figure 1 Focus Area

In the next several chapters, we will use some real examples to demonstrate these design ideas. We will use QuarkPlatform as example.

The original QuarkPlatform code is @ https://github.com/tianocore/edk2/tree/master/QuarkPlatformPkg, `(GIT-HASH: 33e0f9a7dfa536fc90c0d21be8ccf0483d751a48)`

The updated QuarkPlatform code is @ https://github.com/jyao1/OpenPlatform/tree/master/QuarkPlatformPkg.

Summary

This section introduced the open source IA firmware design goals.

Feature – BIOS module selection

This section will discuss the means by which to enable the modules.

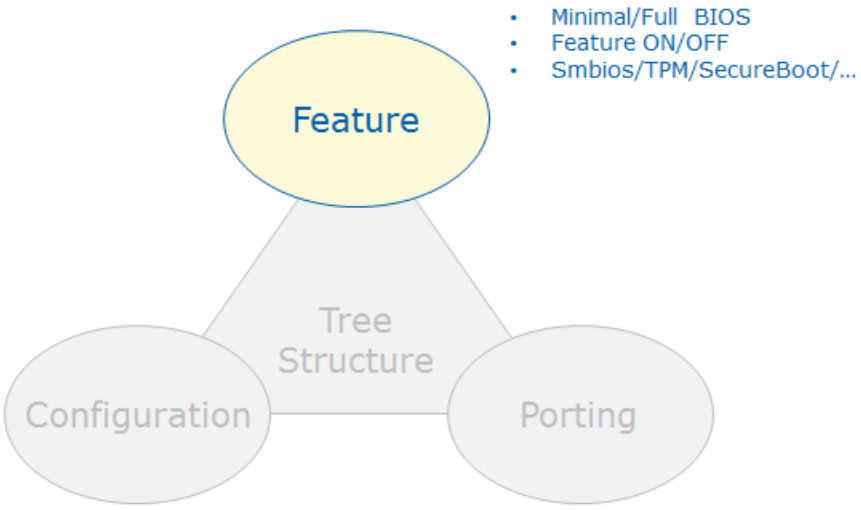


Figure 2 Focus Area - Feature

Category – minimal set V.S. full set

Different open source IA firmware solutions may have different feature sets, typically based upon different requirements. There are 2 possible categories:

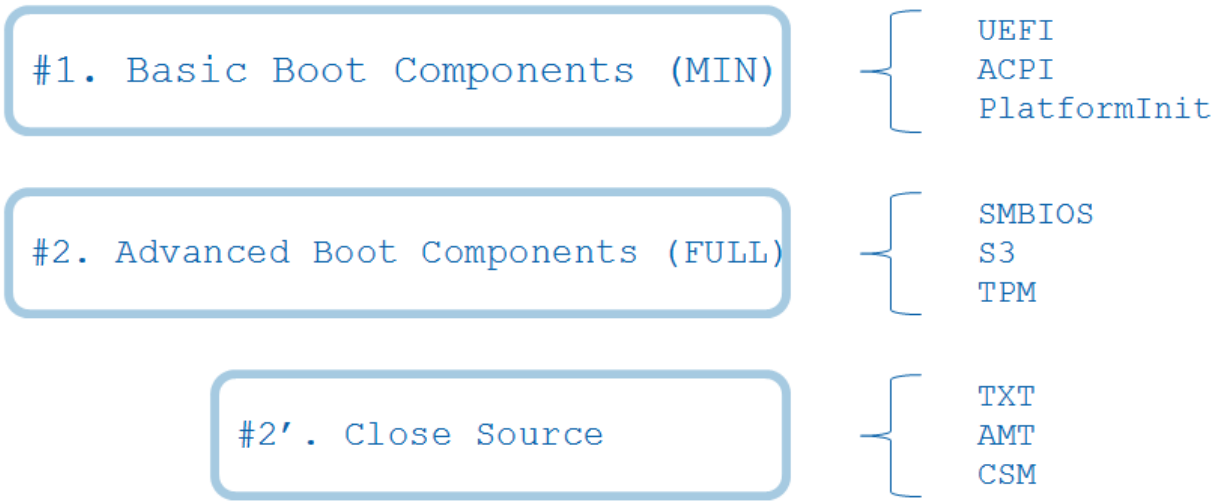


Figure 3 two categories

- Minimal set (basic boot component) – this includes the minimal components needed to boot to the UEFI Shell or to a UEFI OS. The feature set is limited, and may only include a basic ACPI table and some required platform initialization.
- Full set (advanced boot component) – this entails all of the components needed to make a production BIOS. For example, it may support S3, SMBIOS table, TPM, UEFI Secure Boot. Most adavenced modules can be open source, too. But there might be a small

portion of code that can not be open source, such as the binary elements used by TXT/AMT/CSM.

Basic Boot Components

Per our research on the ATOM, Core-i7, and XEON platform firmware, we found the basic boot components are almost same. In the below picture, the GREEN part means the generic EDKII core module. The YELLOW part means the silicon specific module. And finally, the RED part means the platform/board specific module.

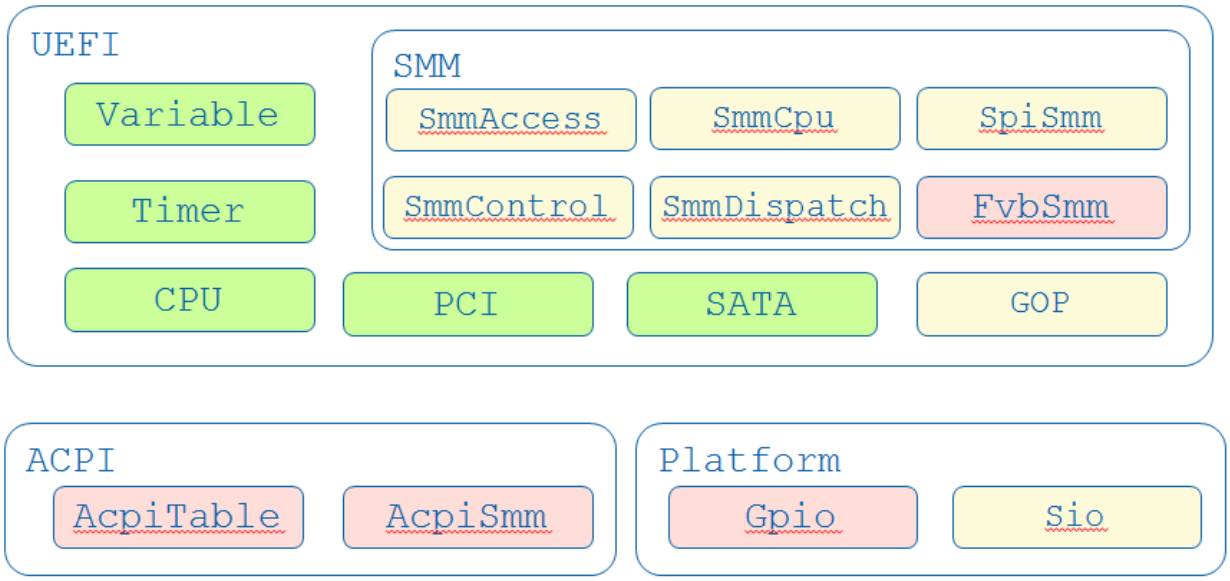


Figure 4 Basic Boot Components

In the UEFI scope, we need the variable, timer, CPU, PCI, either SATA or USB as storage, Graphic or terminal as console output, and finally, USB/PS2 Keyboard or terminal as console input. The SMM portion is required for most X86 platforms in order to support UEFI Authenticated Variable [AUTH VARIABLE].

Most UEFI OSes also require ACPI, so ACPI tables and an SMM driver to enable/disable ACPI are needed.

The platform may also need to initiliaze General Purpose Input/Ouput (GPIO) pins or a Super IO (SIO) to enable the basic boot functionality.

Guideline: Feature – table d'hôte V.S. à la carte

If I go to a new restaurant and do have not an idea on what to order, I will check table d'hôte menu at first. I trust the chef recommendation.

There is a similar approach for BIOS development wherein a platform firmware infrastructure may provide "table d'hôte" menu:

A) SET A: BOOT UEFI SHELL
B) SET B: BASIC OS BOOT
C) SET C: FULL PRODUCTION

This is good for a newcomer in order to give him or her a basic idea on what components are needed for a BIOS solution. It can be 3M full featured BIOS, or only 256K if just the basic boot is required in some cases.

This work can be done by defining some default configuration in PlatformConfig.dsc.
For example, `BOOT_SHELL_ONLY` can be used to configure a BIOS to support a boot to OS (with ACPI/SMM), or boot to shell only (without ACPI/SMM)
https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/QuarkPlatformConfig.dsc

```
====== QuarkPlatformConfig.dsc ======

#
# The basic configuration
#
DEFINE BOOT_SHELL_ONLY   = FALSE

==========================
```

At the same time, a platform firmware may provide "à la cart" menu so that an advanced user can configure an individual item. For example, `SECURE_BOOT_ENABLE` can be used to configure if a BIOS needs to support UEFI secure boot [SECURE BOOT]. `SOURCE_DEBUG_ENABLE` can be used to configure if a BIOS needs to support the source level debug tool.
https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/QuarkPlatformConfig.dsc

```
====== QuarkPlatformConfig.dsc ======

  #
  # Platform On/Off features are defined here
  #
  DEFINE SECURE_BOOT_ENABLE   = FALSE
  DEFINE MEASURED_BOOT_ENABLE = FALSE
  DEFINE SOURCE_DEBUG_ENABLE  = FALSE
  DEFINE PERFORMANCE_ENABLE   = FALSE

================================
```

We recomment limiting the number of features to a reasonable level. LESS IS BETTER THAN MORE.

Feature organization

Our experience for a minimal platform is that you should have:

- ~50 core modules
- <10 silicon modules
- <10 platform modules

We observed that the platform.dsc and platform.fdf might be very long on some platforms. We recommend defining CorePkgInclude.dsc and CorePkgInclude.fdf into which you can put all of the core modules.

The template for QuarkPlatform is shown in Appendix B.
NOTE: This template is just to serve as a reference. Different platforms may choose different modules based on the platform requirements.

Besides CorePkgInclude.dsc/fdf, we also recommend breaking down the big platform.dsc/fdf into smaller files and creating some other include files.

For example:
https://github.com/jyao1/OpenPlatform/tree/master/QuarkPlatformPkg/Include/Build includes the below items:

- BuildOption.dsc – The definition for build options.
- BuildRule.fdf – The build rule for FFS.
- CorePkgConfig.dsc – The PCD configuration.
- CorePkgInclude.dsc – The core module, library in PEI phase and DXE phase for build.
- CorePkgIncludeDxe.fdf – The core module in DXE phase for DXE FV.
- CorePkgIncludePei.fdf – The core module in PEI phase for PEI FV.
- QuarkFlashLayout.fdf – The flash layout in FD.

The final platform.dsc and platform.fdf might just need to have less than ten silicon modules and less than ten platform modules. At this point, people can have a clearer picture of which modules are needed to be considered as part of any porting work.

https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/Quark.fdf

```
====== Quark.fdf ======
[FV.FVRECOVERY]

#
# Core component
#
!include QuarkPlatformPkg/Include/Build/CorePkgIncludePei.fdf

FILE FREEFORM = PCD(gEfiQuarkNcSocIdTokenSpaceGuid.PcdQuarkMicrocodeFile) {
  SECTION RAW = QuarkSocBinPkg/QuarkNorthCluster/Binary/QuarkMicrocode/RMU.bin
}
INF  RuleOverride = NORELOC
QuarkSocPkg/QuarkNorthCluster/MemoryInit/Pei/MemoryInitPei.inf
!if $(BOOT_SHELL_ONLY) == FALSE
INF  RuleOverride = NORELOC
QuarkSocPkg/QuarkNorthCluster/Smm/Pei/SmmAccessPei/SmmAccessPei.inf
INF  RuleOverride = NORELOC
QuarkSocPkg/QuarkNorthCluster/Smm/Pei/SmmControlPei/SmmControlPei.inf
!endif
#!if $(GALILEO) == GEN1
INF  QuarkPlatformPkg/Board/Galileo/BoardEarlyInit/BoardEarlyInit.inf
#!endif
#!if $(GALILEO) == GEN2
```

```
INF  QuarkPlatformPkg/Board/GalileoGen2/BoardEarlyInit/BoardEarlyInit.inf
#!endif
INF  QuarkPlatformPkg/PlatformInit/PlatformInitPei/PlatformEarlyInit.inf



[FV.FVMAIN]

#
# Core component
#
!include QuarkPlatformPkg/Include/Build/CorePkgIncludeDxe.fdf

#
# Early SoC / Platform modules
#
INF  QuarkPlatformPkg/PlatformInit/PlatformInitDxe/PlatformInitDxe.inf
#!if $(GALILEO) == GEN1
INF  QuarkPlatformPkg/Board/Galileo/BoardInit/BoardInitDxe.inf
#!endif
#!if $(GALILEO) == GEN2
INF  QuarkPlatformPkg/Board/GalileoGen2/BoardInit/BoardInitDxe.inf
#!endif

!if $(BOOT_SHELL_ONLY) == FALSE
INF  QuarkPlatformPkg/Flash/SpiFvbServices/PlatformSpi.inf
INF  QuarkPlatformPkg/Flash/SpiFvbServices/PlatformSmmSpi.inf
!endif
INF  QuarkSocPkg/QuarkNorthCluster/QNCInit/Dxe/QNCInitDxe.inf
!if $(BOOT_SHELL_ONLY) == FALSE
INF  QuarkSocPkg/QuarkNorthCluster/Smm/Dxe/SmmAccessDxe/SmmAccess.inf
INF  QuarkSocPkg/QuarkNorthCluster/S3Support/Dxe/QncS3Support.inf
INF  QuarkSocPkg/QuarkNorthCluster/Spi/PchSpiRuntime.inf
INF  QuarkSocPkg/QuarkNorthCluster/Spi/PchSpiSmm.inf
!endif

!if $(BOOT_SHELL_ONLY) == FALSE
#
# ACPI
#
INF  RuleOverride = DRIVER_ACPITABLE QuarkPlatformPkg/Acpi/AcpiPlatform/AcpiPlatform.inf
#!if $(GALILEO) == GEN1
INF  RuleOverride = DRIVER_ACPITABLE
QuarkPlatformPkg/Board/Galileo/AcpiTables/AcpiBoard.inf
#!endif
#!if $(GALILEO) == GEN2
INF  RuleOverride = DRIVER_ACPITABLE
QuarkPlatformPkg/Board/GalileoGen2/AcpiTables/AcpiBoard.inf
#!endif
!endif

!if $(BOOT_SHELL_ONLY) == FALSE
#
# SMM
#
INF  QuarkSocPkg/QuarkNorthCluster/Smm/Dxe/SmmControlDxe/SmmControlDxe.inf
INF  QuarkSocPkg/QuarkNorthCluster/Smm/DxeSmm/QncSmmDispatcher/QNCSmmDispatcher.inf
```

```
INF  QuarkPlatformPkg/Acpi/AcpiSmm/AcpiSmmPlatform.inf
INF  QuarkPlatformPkg/Feature/PowerManagement/SleepSmm/SleepSmmPlatform.inf
INF  QuarkPlatformPkg/Feature/PowerManagement/CpuPowerManagement/SmmPowerManagement.inf
!endif

#
# PCI
#
INF  QuarkPlatformPkg/Pci/Dxe/PciPlatform/PciPlatform.inf
INF  QuarkSocPkg/QuarkSouthCluster/IohInit/Dxe/IohInitDxe.inf


=========================
```

Summary

This section introduces the feature - BIOS module selection.

Tree Structure – Platform Code Layout

This section discusses the tree structure.

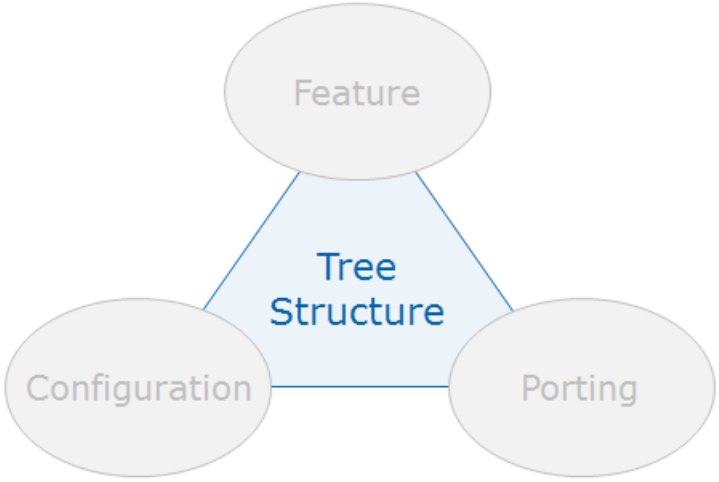


Figure 5 Focus Area – Tree Structure

Guideline: One feature, one directory

We observed some platforms use a flat mode directory layout and put most modules in the Platform directory. This brings trouble when the developer wants to find a driver. Below is an example for the open source MinnowMAX platform directory.

https://github.com/tianocore/edk2/tree/master/Vlv2TbltDevicePkg (GIT-HASH: 6b49f0e0d36e926042d91d2c78066b3d529c739f)

```
====== Vlv2TbltDevicePkg directory ======
Vlv2TbltDevicePkg
  AcpiPlatform
  Application
  BootScriptSaveDxe
  FspAzaliaConfigData
  FspSupport
  FvbRuntimeDxe
  FvInfoPei
  Include
  IntelGopDepex
  Library
  Logo
  Metronome
  MonoStatusCode
  Override
  PciPlatform
  PlatformCpuInfoDxe
  PlatformDxe
  PlatformGopPolicy
  PlatformInfoDxe
```

```
  PlatformInitPei
  PlatformPei
  PlatformSetupDxe
  PlatformSmm
  PpmPolicy
  SaveMemoryConfig
  SmBiosMiscDxe
  SmmSwDispatch2OnSmmSwDispatchThunk
  SmramSaveInfoHandlerSmm
  Stitch
  UiApp
  VlvPlatformInitDxe
  Wpce791
=======================================
```

We recommend using a hierarchial layout - only put the basic features into the root directory and put the advanced features into a "Feature" directory. For example:

```
====== XXXPlatformPkg directory ======
XXXPlatformPkg
  Acpi
      AcpiPlatform
      AcpiSmm
  Board
  Feature
      Amt
      Hsti
      I2c
      S3
      Smbios
  Flash
      SpiFvbService
  FspWrapper
  Include
  Library
  PlatformInit
      PlatformInitPei
      PlatformInitDxe
  Setup
  Tools
=======================================
```

Below is the QuarkPlatformPkg layout. The ACPI related features are in the Acpi directory. The board specific settings are in the Board directory. Board/Galileo designates the GalileoGen1 board settings. Board/GalileoGen2 designates the GalileoGen2 board settings. Feature directory contains advanced features, like SMBIOS, or Power Management support. Flash driver – SpiFvbService is under the Flash directory. Finally, PlatformInit directory includes PlatformInitPei and PlatformInitDxe.
If there are some other advanced features, they can be put into the Feature directory.
https://github.com/jyao1/OpenPlatform/tree/master/QuarkPlatformPkg

```
====== QuarkPlatformPkg directory ======
XXXPlatformPkg
  Acpi
      AcpiPlatform
      AcpiSmm
  Board
```

```
      Galileo
      GalileoGen2
  Feature
      PowerManagement
      Smbios
  Flash
      SpiFvbService
  Include
  Library
  PlatformInit
      PlatformInitPei
      PlatformInitDxe
=====================================
```

Summary

This section introduces the tree structure – platform code layout.

Configuration – platform policy data

This section describes platform policy data and configuration.

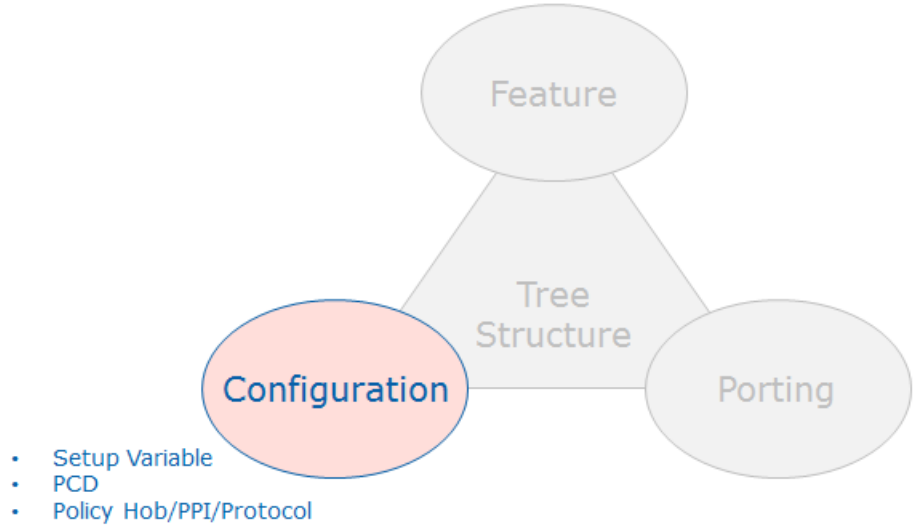


Figure 6 Focus Area – Configuration

Configuration options

As we mentioned before, there might be many sources of platform configuration data. Some general guidelines are defined below:

- **PI PCD** – The PI PCD could be static data fixed at build time or dynamic data updatable at runtime.
  - PcdsFeatureFlag: This type PCD only supports 1/0. Caller uses FeaturePcdGet() to retrieve the value. This type of PCD is mapped to be a MACRO so that a compiler optimization can remove the code scoped by "if(FALSE)". It is not allowed to set as a PcdsFeatureFlag.
  - PcdsFixedAtBuild: This type of PCD can be mapped to a global variable if the caller uses PcdGet(), or a MACRO if the caller uses FixedPcdGet(). As such, this type of PCD can be used in a data structure definition. It is not allowed to be set as PcdsFixedAtBuild.
  - PcdsPatchableInModule: This type of PCD is mapped to a global variable. It is allowed for use by both PcdGet and PcdSet. If PcdSet is called, it only changes the module-level PCD value instead of a system-level PCD value. Only the current module sees the PCD change. Other modules still see the original value.
  - PcdsDynamicDefault: PcdsDynamicDefault is mapped to a PPI or protocol. It is allowed for both PcdGet and PcdSet. PcdSet changes the system-level PCD value immediately. This type of PCD value is volatile. The changed value will not be saved in the next boot.
  - PcdsDynamicHii: PcdsDynamicHii is mapped to a UEFI variable. It is non-volatile. As such, the changed value can be saved in the next boot. However, the tricky thing is that this PCD value depends on the UEFI variable services

readiness. If PcdGet is called before UEFI variable services ready, the default PCD value will be returned instead of the updated PCD value. We suggest that the platform owner be very very careful of this trap. If DXE PcdGet is required before the UEFI variable services are ready, we suggest that the platform define PcdsDynamicDefault, and then use get variable data in the PEI phase to fill in this PCD value.

- PcdsDynamicVpd: PcdsDynamicVpd is to map configuration data to a static flash region so that a tool can modify the PcdsDynamicVpd after the flash image is generated. This is used by a BIOS that needs to support binary configuration after build. Intel FSP is an example of using PcdsDynamicVpd.
- SkuIds: SkuIds is a special usage of PCD. It can support multiple configurations generated at build time and support runtime selection to make one configuration take effect finally. The good point is that it is very straightforward for each board, if board configuration can be determined. However, current implementations just put all configuration data together without any size optimization. So even a one byte difference will cause full configurations to be duplicated. We can enhance the SkuIds PCD implementation. If there is any size concern in the SkuIds PCD, the alternative could be: define one PcdsDynamicDefault and let each platform update its own configuration there.

- **UEFI Variable** – The UEFI Variable can be non-volatile data or volatile data, and it is widely used by VFR.
  - In most cases, a non-volatile variable is used to store the user updatable configuration in a setup page. One example is VT enable/disable. This is purely a platform choice. We suggest that the platform map variable configuration to PCD, and use a PcdSet callback to set the variable data. The benefit is that if a new platform just wants to use a static setting, it can remove the variable easily.
  - A non-volatile variable may also be used to store the system configuration generated at runtime, for example, memory configuration data. In order to maintain security, we suggest that platform to lock the configuration variable before exiting PM auth/EndOfDxe event, by using the EDKII_VARIABLE_LOCK protocol.
  - A volatile variable is generated at runtime. A platform setup driver may use this information to control a VFR page to suppress or gray out a menu, or to display the system information, like CPU/SA/PCH stepping and features.
- **FSP UPD** – FSP UPD can be static default configuration, or a dynamic updatable UPD.
  - FSP UPD is used to pass configuration from the FSP wrapper into a FSP binary. A platform needs to convert the policy configuration in PCD to a FSP UPD before calling a FSP API, like FspMemoryInit, FspSiliconInit.
  - PcdsDynamicVpd.Upd: For a FSP binary, we use DynamicVpd.Upd to mark the configuration that needs to be in the UPD region. (Please be aware that UPD is not a standard PCD concept, it is an FSP extension)
- **Silicon Policy Hob/PPI/Protocol** – It is policy data constructed at runtime or it can be a hook for silicon code.
  - Policy data: Silicon Policy Hob/PPI/Protocol are useful in order to let one silicon code module support multiple boards. It is the interface between silicon code and

platform code. A platform needs to convert policy configuration in PCD into a Silicon Policy PPI/Protocol.
    - Silicon Hook: Sometimes, we observe that Silicon Policy PPI or Protocol provides a silicon hook for platform. This hook may perform some additional action based on a platform setting, or retrieve some system information. In most cases, we suggest to separate the hook function from policy data.
- **Configuration Block** – It is a data structure to put all policy data in a block without any C-language data pointer in a policy data.
    - The Configuration Block is a new idea to resolve data pointer issues objserved in earlier HOB usage. Previously, a silicon code module would define a root policy data object with some data pointers to sub-regions. For example, a PCH policy data may include a pointer to USB policy data, a pointer to SATA policy data, and a pointer to PCIExpress policy data. This HOB policy data pointer needs to be relocated after memory initialization, such as the Memory Reference Code (MRC), is done, because the PEI core needs to move data from temporary memory or Cache-As-RAM (CAR) to DRAM. If the platform code forgets to move the policy data and fix the pointer, the following code might retrieve the wrong policy data pointer. With the configuration block, the silicon and platform needn't worry about the invalid policy data pointer issue because the data pointer is eliminated. The PCH policy data block may include a USB policy data block, a SATA policy data and a PCIExpress policy data. Configuration Blocks can be mapped and used by either Hob/PPI/Protocol or PCD.
- **Global NVS** – It is an ACPI region to pass the configuration from the C code to ASL code.
    - Global NVS can be used for turning some feature on/off. An example includes returning different _STA values. 0x0 means the device does not exist. 0xF means the device exists.
    - It can be used as the policy data, for example CriticalTemperature value. A platform C code module may convert the configuration from PCD to Global NVS.
- **Platform signed data blob** – It is read only signed data at build time.
    - This signed data blob provides the configuration on a platform. An OEM may update the configuration for different boards. We suggest the platform map the signed data blob to PCDs so that a platform consumer can just use PcdGet to get the configuration without knowing the data source. The benefit is that all the platform code can be consistent, irrespective of whether the configuration data is from a signed data blob, a BIOS boot block static region, or a UEFI variable.
- **CMOS** – It is simple non-volatile storage, but it is not secure.
    - The most useful usage of CMOS is to use CMOS-CLEAR to determine if an end user wants to use the default variable configuration. This is a consistent user experience from old legacy BIOS. The new platform can use a special function key or a special GPIO as indicator of this logic.
    - The benefit of CMOS is that the CMOS can be accessed at early SEC phase without rich API requirements. Beyond that usage, though, we do not suggest a platform use CMOS to store configuration data.
- **MACRO** – C-language MACRO. It is fixed at build time.

- A MACRO can be used as static data configuration. It is useful if the MACRO is only used in one module and does not require user configuration. However, if the MACRO is used across many modules or is configurable, like PCIE_BASE, we suggest using PCD.
- A MACRO used in #IFDEF can be used to enable/disable features. If the consumer is in C code, it can be replaced by FeaturePcd. It will become if(0) or if(1) finally. The benefit of using PCD is that all the code in both path can be built.

Guideline: Use PCD in platform code.

PCD stands for "Platform Configuration Database". It is a platform database that contains a variety of current platform settings or directives that can be accessed by a driver or application.

We observed some platform just access the configuration data from a special source directly, such as a UEFI variable. For the latter, people then have to remember clearly on which configuration is stored in which direct location. Also if a platform decides to change the configuration source from one place (UEFI variable) to another (static region in boot block), all impacted platform modules are required to change. This is non-ideal for source code maintenance and development.

```
======old Platform.c======
//
// Get config from setup variable
//
VarDataSize = sizeof (SETUP_DATA);
Status = GetVariable (
                L"Setup",
                &gSetupVariableGuid,
                NULL,
                &VarDataSize,
                &mSystemConfiguration
                );

//
// Get platform info from Hob
//
HobList.Raw = GetNextGuidHob (&gPlatformInfoHobGuid, HobList.Raw);
PlatformInfo = (PLATFORM_INFO *) ((UINT8 *) (&HobList.Guid->Name) + sizeof (EFI_GUID));
=========================
```

We recommend using PCD only in platform code, no matter where a platform chooses to store configuration data based on production requirement, like below:

```
======new Platform.c======
//
// Get setup configuration from PCD
//
CopyMem (
  &mSystemConfiguration,
  PcdGetPtr (PcdSetupConfiguration),
  sizeof(mSystemConfiguration)
```

```
  );

//
// Get platform info from PCD
//
PlatformInfo = (PLATFORM_INFO *)PcdGetPtr (PcdPlatformInfo);
=========================
```

Then the code is consistent and easy to maintain, if the next generation platform decides to change the location.

Then question becomes: how does a platform fill in configuration data that is mapped to PCD?

We can use the below mechanism to update the existing platform code.

- **Configuration conversion**

In the current PI1.5 specification, a PCD driver can provide a callback function on PcdSet().

A platform may introduce a "ConfigConvert" module (GREEN box). It runs very early and calls PcdSet() to convert other storage data (variable, signed data blob, policy hob) to the PCD database.

Then the rest of PlatformInit code (YELLOW box) can just call PcdGet() to get the policy data.

If the Platform driver wants to update a PCD value by calling PcdSet() later, the "ConfigConvert" can register a PCD callback function to redirect setting to other source (for example, variable).

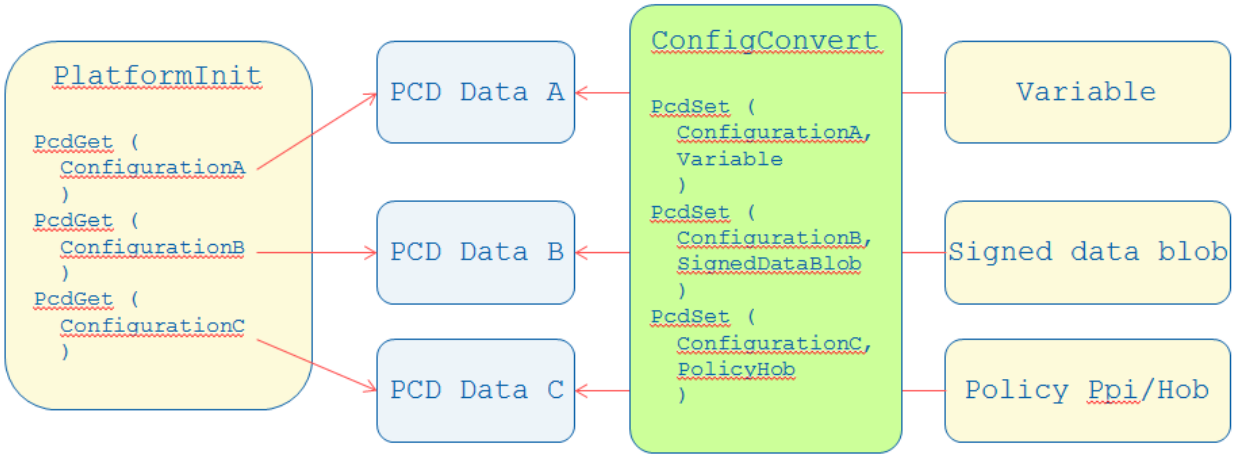


Figure 7 Configuration Conversion

As **Configuration conversion** is supported in the current implementation, we recommend platform to use it. The gist is that any other platform driver should use PcdGet() to retrieve policy data, and PcdSet() to update policy data.

QuarkPlatformPkg does not use a UEFI variable to save the configuration data. But this is used for other real platforms.

Summary
This section introduces configuration – platform policy data.

Porting – Board Specific initialization

This section talks about board porting.

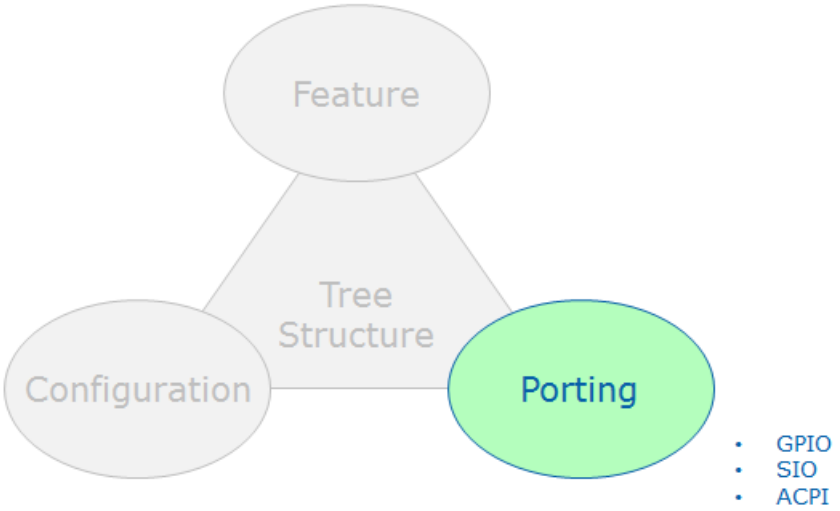


Figure 8 Focus Area – Porting

Multi-board support

There is the requirement that one BIOS should be able to boot the multiple boards. Different boards may have different configurations.

We observed many BIOS code examples use switch-case style board ID check in many platform drivers.
For example:

```
======old PlatformGpio.c======
switch (BoardId) {
case BoardIdBoard1:
    GpioPin = GPIO_BOARD1;
    Break;
case BoardIdBoard2:
    GpioPin = GPIO_BOARD2;
    Break;
case BoardIdBoard3:
    GpioPin = GPIO_BOARD3;
    Break;
}
============================

======old PlatformRecovery.c======
switch (BoardId) {
case BoardIdBoard1:
    IsRecovery = IsRecoveryBoard1 ();
    Break;
case BoardIdBoard2:
    IsRecovery = IsRecoveryBoard2 ();
    Break;
```

```
case BoardIdBoard3:
    IsRecovery = IsRecoveryBoard3 ();
    Break;
}
=================================

======old PlatformAcpi.c======
switch (BoardId) {
case BoardIdBoard1:
    AcpiConfig = ACPI_BOARD1;
    Break;
case BoardIdBoard2:
    AcpiConfig = ACPI_BOARD2;
    Break;
case BoardIdBoard3:
    AcpiConfig = ACPI_BOARD3;
    Break;
}
============================
```

A real example in Quark:
https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Platform/Pei/PlatformInit/PlatformEarlyInit.c

```
======old PlatformEarlyInit.c======
/**
  Initialize state of I2C GPIO expanders.

  @param  PlatformType  Platform type for GPIO expander init.

**/
EFI_STATUS
EarlyPlatformConfigGpioExpanders (
  IN CONST EFI_PLATFORM_TYPE              PlatformType
  )
{
  ......
  if (PlatformType == GalileoGen2) {
    //
    // Configure AMUX1_IN (EXP2.P1_4) as an output
    //
    PlatformPcal9555GpioSetDir (
      GALILEO_GEN2_IOEXP2_7BIT_SLAVE_ADDR,  // IO Expander 2.
      12,                                   // P1-4.
      FALSE                                 // Configure as output
      );
    ......
  }

  if (PlatformType == Galileo) {
    //
    // Detect the I2C Slave Address of the GPIO Expander
    //
    if (PlatformLegacyGpioGetLevel (R_QNC_GPIO_RGLVL_RESUME_WELL,
GALILEO_DETERMINE_IOEXP_SLA_RESUMEWELL_GPIO)) {
      I2CSlaveAddress.I2CDeviceAddress = GALILEO_IOEXP_J2HI_7BIT_SLAVE_ADDR;
    } else {
      I2CSlaveAddress.I2CDeviceAddress = GALILEO_IOEXP_J2LO_7BIT_SLAVE_ADDR;
    }
```

```
    ......
  }

}
============================
```

Similar code in https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Library/PlatformPcieHelperLib/PlatformPcieHelperLib.c and https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Library/PlatformSecureLib/PlatformSecureLib.c

The problem is that if a developer wants to add a new board, he/she must go through all the code to find out what needs to be changed in each driver. It is huge burden.

Guideline: One board, one directory

We recommend creating directory for each board and put all board specific settings in this board directory. For example:

```
====== XXXPlatformPkg directory ======
XXXPlatformPkg
  Acpi
  Features
  Board
      Library
      BoardX
      BoardY
  Flash
  FspWrapper
  Include
  Library
  Platform
  Setup
  Tools
=======================================
```

Below is the directory of QuarkPlatformPkg. We put Galileo and GalileoGen2 under the Board directory.
https://github.com/jyao1/OpenPlatform/tree/master/QuarkPlatformPkg

```
====== QuarkPlatformPkg directory ======
XXXPlatformPkg
  Acpi
  Board
      Galileo
      GalileoGen2
  Feature
  Flash
  Include
  Library
  PlatformInit
=======================================
```

If someone wants to add a new board, like GalileoGen3, he/she can copy GalileoGen2 to GalileoGen3, and then update all the modules in this GalileoGen3 directory.

Once we move the board specific code to the board specific directory, the generic platform code should not contain any board specific code.

The old Quark platform code violates this rule. It includes GalileoGen2 PCAL9555 specific functions.
https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Include/Library/PlatformHelperLib.h
https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Library/PlatformHelperLib/PlatformHelperLib.c

```
====== old PlatformHelperLib.c ======
VOID
Pcal9555SetPortRegBit (
  IN CONST UINT32                         Pcal9555SlaveAddr,
  IN CONST UINT32                         GpioNum,
  IN CONST UINT8                          RegBase,
  IN CONST BOOLEAN                        LogicOne
  )

VOID
EFIAPI
PlatformPcal9555GpioSetDir (
  IN CONST UINT32                         Pcal9555SlaveAddr,
  IN CONST UINT32                         GpioNum,
  IN CONST BOOLEAN                        CfgAsInput
  )

=======================================
```

The old Quark Platform code removes these functions and moves them to the Galileo Gen2 specific Pcal9555Lib.
https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/Board/GalileoGen2/Include/Library/Pcal9555Lib.h
https://github.com/jyao1/OpenPlatform/tree/master/QuarkPlatformPkg/Board/GalileoGen2/Library/Pcal9555Lib

Board detection

In order to detemine which board specific driver needs to run and which does not need to run, there must be some code to detect the board type.
The board detection code is board specific. It should be under the board specific directory, and it needs to register a BoardDetectionStartPpi callback.
After a PlatformInit PEIM does basic initialization, `gBoardDetectionStartPpiGuid` is installed. This is a NULL interface PPI. It is used to broadcast a message – start board detecting. Then each board detection function callback is run. Once a board detection function successfully recognizes the board, it installs `gBoardDetectedPpiGuid`. This is another NULL interface PPI. It is used as inicator that – board detection is successful and is finished, so that any other board detection function will just return immediately.

`XXXPlatformPkg\Board\BoardX\BoardInit\BoardInit.c, BoardXBoardDetectionCallback()` is the board detection callback for BoardX platform.

```
====== BoardInit.c ======
EFI_STATUS
EFIAPI
BoardXBoardDetectionCallback (
  IN CONST EFI_PEI_SERVICES     **PeiServices,
  IN EFI_PEI_NOTIFY_DESCRIPTOR  *NotifyDescriptor,
  IN VOID                       *Ppi
  )
{
  ...
  do {
    Status = InternalGetBoardId (&PlatformInfo, &BoardId);
  } while (Status != EFI_SUCCESS);

  if (BoardId == BoardIdBoardX) {
    // It is BoardX
    ...
    Status = PeiServicesInstallPpi (&mBoardXDetectedPpi);
  }
=======================================
```

The new Quark Galileo board detection is at https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/Board/Galileo/BoardEarlyInit/BoardEarlyInit.c
The new Quark GalileoGen2 board detection is at https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/Board/GalileoGen2/BoardEarlyInit/BoardEarlyInit.c
The detection callback `BoardDetectionCallback()` just checks the PCD value to show the concept. On other real platforms, the board detection function might check a GPIO or EC to get board ID data.

```
====== BoardEarlyInit.c ======
EFI_STATUS
EFIAPI
BoardDetectionCallback (
  IN CONST EFI_PEI_SERVICES     **PeiServices,
  IN EFI_PEI_NOTIFY_DESCRIPTOR  *NotifyDescriptor,
  IN VOID                       *Ppi
  )
{
  ......
  if (PcdGet16(PcdPlatformType) != Galileo) {
    return EFI_UNSUPPORTED;
  }
  DEBUG ((EFI_D_INFO, "Detected Galileo!\n"));
  ......
}
=======================================
```

Board initialization

Once Board detection is successful, the same function can set some board specific dynamic PCD value for board initialization. The PCD can be data, data pointer, or function pointer. Since a

PCD is quite flexible, it is a platform choice to define which PCD needs to be data, data pointer, or function pointer.

```
====== BoardInit.c ======
EFI_STATUS
EFIAPI
BoardXBoardDetectionCallback (
  IN CONST EFI_PEI_SERVICES     **PeiServices,
  IN EFI_PEI_NOTIFY_DESCRIPTOR  *NotifyDescriptor,
  IN VOID                       *Ppi
  )
{
  ...
  if (BoardId == BoardIdBoardX) {
    // It is BoardX

    // Function pointer
    PcdSet64 (PcdFuncConfigInit, (UINT64)(UINTN) BoardXConfigInit);
    PcdSet64 (PcdFuncBoardInit, (UINT64)(UINTN) BoardXBoardInit);
    PcdSet64 (PcdFuncPolicyOverride, (UINT64)(UINTN) BoardXPolicyOverride);

    // Data
    DataSize = sizeof(BoardXSpdAddressTable);
    PcdSetPtr(PcdSpdAddressTable, &DataSize, BoardXSpdAddressTable);

    // Data pointer
    PcdSet64 (PcdResistorPtr, (UINT64)(UINTN)BoardXResistor);
  }
=======================================
```

The PCD data/data pointer is used for silicon FSP policy initialization, and the PCD function is called by the PlatformInit driver as a platform hook. In the above case:
`PcdFuncConfigInit` is called to initialize platform configuration PCD. The data source can be dynamic configuration from a variable region, or static configuration from the boot block.
`PcdFuncBoardInit` is called to do board specific initialization. For example, GpioInit, SioInit.
`PcdFuncPolicyOverride` is called override default silicon policy set by platform. Example policies can include PcieConfig & UsbConfig.

The new Quark Galileo board initialization is at
https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/Board/Galileo/BoardEarlyInit/BoardEarlyInit.c
The new Quark GalileoGen2 board initialization is at
https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/Board/GalileoGen2/BoardEarlyInit/BoardEarlyInit.c
The initialization code is the detection callback `BoardDetectionCallback()`. It sets the PcdBoardInitPreMem function pointer and the PcdBoardInitPostMem function pointer. They are hook functions called by PlatformInit. It also sets the ResumeWell GPIO pin configuration based on different board.

```
====== BoardEarlyInit.c ======
EFI_STATUS
EFIAPI
BoardDetectionCallback (
  IN CONST EFI_PEI_SERVICES     **PeiServices,
  IN EFI_PEI_NOTIFY_DESCRIPTOR  *NotifyDescriptor,
```

```
  IN VOID                       *Ppi
  )
{
  ......
  Status = PcdSet64S (PcdBoardInitPreMem, (UINT64)(UINTN)BoarInitPreMem);
  ASSERT_EFI_ERROR(Status);
  Status = PcdSet64S (PcdBoardInitPostMem, (UINT64)(UINTN)BoarInitPostMem);
  ASSERT_EFI_ERROR(Status);

  Status = PcdSet32S (PcdPciExpPerstResumeWellGpio, PCIEXP_PERST_RESUMEWELL_GPIO);
  ASSERT_EFI_ERROR(Status);
  Status = PcdSet32S (PcdFlashUpdateLedResumeWellGpio,
GALILEO_FLASH_UPDATE_LED_RESUMEWELL_GPIO);
  ASSERT_EFI_ERROR(Status);
  ......
}
=======================================
```

Board specific driver

There might be a set of drivers under the BoardXXX directory, a set of drivers under the BoardYYY directory. How do we know which one will run and take effect finally?

```
====== XXXPlatformPkg directory ======
XXXPlatformPkg
  Board
      BoardX
          AcpiTables
      BoardY
          AcpiTables
=======================================
```

- **BoardId check**

The board specific module entry point may add a board ID check and run if and only if the board ID matches.

```
====== AcpiBoard.c======
EFI_STATUS
InstallAcpiBoard (
  IN EFI_HANDLE         ImageHandle,
  IN EFI_SYSTEM_TABLE   *SystemTable
  )
{
  EFI_STATUS                    Status;
  PLATFORM_INFO                 *PlatformInfo;

  PlatformInfo = PcdGetPtr (PcdPlatformInfo);
  if (PlatformInfo->BoardId != BoardIdBoardX) {
    return EFI_UNSUPPORTED;
  }
  //
  // Do initialization
  //
  ......
}
====================
```

This is an easy way to add the board specific code. If a platform does not need the multi-board support, this check can be skipped.

NOTE: The Board ID check should only happen in board specific drivers. The Board ID check is NOT allowed in any generic platform code.

For example, the new Quark Galileo ACPI board code at https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/Board/Galileo/AcpiTables/AcpiBoard.c

```
====== AcpiBoard.c======
EFI_STATUS
EFIAPI
AcpiBoardEntryPoint (
  IN EFI_HANDLE         ImageHandle,
  IN EFI_SYSTEM_TABLE   *SystemTable
  )
{
  EFI_STATUS                    Status;

  if (PcdGet16 (PcdPlatformType) != Galileo) {
    return EFI_UNSUPPORTED;
  }

  //
  // Board specific init
  //
  ......

}
====================
```

- **Dependency check**

Another way is to add BoardId specific GUID in INF dependency section.

```
====== BoardX.inf======
......
[Depex]
  gBoardIdBoardXGuid
======================
```

However it might be complicated because we have to define such a PPI/Protocol and need another driver to publish the PPI/Protocol based upon BoardId value.

Board specific ACPI

- **Board specific device selection**

We observed many BIOS code modules define BoardId in ACPI global NVS and use the board ID check in the ASL code. For example:

```
====== Old asl ======
```

```
  Device(DEV0)
  {
    Method(_STA,0)
    {
      If(LEqual(BID,BoardIdBoardX)
      {
        Return(0x0000)
      }
      Return(0x001F)
    }
======================
```

This approach is NOT portable if we want to add a new board.

One way to resolve this issue is to define a board-neutral name for the branch condition. For example: DEVP means Device Present.

```
====== recommended asl ======
  Device(DEV0)
  {
    Method(_STA,0)
    {
      If(LEqual(DEVP,0)
      {
        Return(0x0000)
      }
      Return(0x001F)
    }
======================
```

So that in the board specific AcpiBoard.c, each board can configure this value.

```
====== AcpiBoard.c======
EFI_STATUS
InstallAcpiBoard (
  IN EFI_HANDLE         ImageHandle,
  IN EFI_SYSTEM_TABLE   *SystemTable
  )
{
  ......
  mGlobalNvsArea->Area->DEVP = 0;

}
====================
```

If this device is a silicon device and it might be enabled/disabled by policy, we recommend using the above mechanism.

If this device is a board device and it exists only on boardX but does not exist on boardY, we recommend using the below mechanism – SSDT.

- **Board specific SSDT**

The other way to resolve above issue is to move the board specific ACPI Secondary System Description Table (SSDT) to the board specific directory and let it be installed by a board

specific ACPI driver. The basic platform ACPI driver should only handle generic ACPI tables, like FADT, MCFG, HPET, MCFG, and etc.

The old QuarkPlatform merged all device specific ASL code together into one big DSDT (AD7298/ CY8C9540A/ GpioClient/ ADC108S102/ CAT24C08/ PCA9685/ PCAL9555A) and used PlatformType to check if the device exists or not.
https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Acpi/AcpiTables/Dsdt/PCAL9555A.asi

```
======old PCAL9555A.asi======
Device(NIO1)
{
  ......
    Method(_STA, 0x0, NotSerialized)
    {
        //
        // Only Platform Type / Id 8 has this device.
        //
        If(LNotEqual(PTYP, 8))
        {
          return (0)
        }
        Return(0xf)
    }
}
============================
```

This code has same problem as PlatformInit: if a developer wants to add a new board, or wants to reuse the current device, he/she must go through all the code to find out what need to be changed in each driver. It is huge burden.

The new Galiloe Board specific ACPI driver produces the SSDT for AD7298/ CY8C9540A/ GpioClient device.
https://github.com/jyao1/OpenPlatform/tree/master/QuarkPlatformPkg/Board/Galileo/AcpiTables/SsdtBoard

The new GalileoGen2 Board specific ACPI driver produces the SSDT for ADC108S102/ CAT24C08/ PCA9685/ PCAL9555A device.
https://github.com/jyao1/OpenPlatform/tree/master/QuarkPlatformPkg/Board/GalileoGen2/AcpiTables/SsdtBoard

- **Board specific device configuration**

We also observed some BIOS code defines board specific configuration in the global NVS area. This is not good approach because a generic platform should not have the board specific knowledge. The better way is to move code to the board specific directory and define a board specific NVS, or just use a simple Name object under this device node.

For example:
The old QuarkPlatform has `AlternateSla` in global NVS area, but it is Galileo board specific:
https://github.com/tianocore/edk2/blob/master/QuarkPlatformPkg/Acpi/Dxe/AcpiPlatform/AcpiPlatform.c

```
======old AcpiPlatform.c======
```

```
EFI_STATUS
AcpiPlatformEntryPoint (
  IN EFI_HANDLE     ImageHandle,
  IN EFI_SYSTEM_TABLE   *SystemTable
  )
{
  ......
  if (mGlobalNvsArea.Area->PlatformType == Galileo) {
    if (PlatformLegacyGpioGetLevel (R_QNC_GPIO_RGLVL_RESUME_WELL,
GALILEO_DETERMINE_IOEXP_SLA_RESUMEWELL_GPIO)) {
      mGlobalNvsArea.Area->AlternateSla = FALSE;
    } else {
      mGlobalNvsArea.Area->AlternateSla = TRUE;
    }
  }
  ......
}
============================
```

The new Quark Galileo ACPI board code solution moves the code to the board specific directory at https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/Board/Galileo/AcpiTables/AcpiBoard.c

```
====== AcpiBoard.c======
EFI_STATUS
EFIAPI
AcpiBoardEntryPoint (
  IN EFI_HANDLE         ImageHandle,
  IN EFI_SYSTEM_TABLE   *SystemTable
  )
{
  EFI_STATUS                    Status;

  if (PcdGet16 (PcdPlatformType) != Galileo) {
    return EFI_UNSUPPORTED;
  }

  //
  // Configure platform IO expander I2C Slave Address.
  //
  if (PlatformLegacyGpioGetLevel (R_QNC_GPIO_RGLVL_RESUME_WELL,
GALILEO_DETERMINE_IOEXP_SLA_RESUMEWELL_GPIO)) {
    mAlternateSla = FALSE;
  } else {
    mAlternateSla = TRUE;
  }

  PublishAcpiTablesFromFv ();

  return EFI_SUCCESS;
}
====================
```

The Galileo Board uses a simple Name object to configure resource buffer content for CY8C9540A by ALTS.

https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/Board/Galileo/AcpiTables/SsdtBoard/CY8C9540A.asi

```
====== CY8C9540A.asi ======
    Name(_HID, "INT3490") // Cypress CY8C9540A Io Expander Function.
     Name(_CID, "INT3490")
     Name(ALTS, 0xFFFF0000)
     Method(_CRS, 0x0, NotSerialized)
     {
         CreateByteField(RBUF, 16, OB1)
         if (LEqual (ALTS, 0))
         {
             Store(0x20, OB1)
         }
         Else
         {
             Store(0x21, OB1)
         }
         Return(RBUF)
     }
====================
```

Then Galileo Board C code can patch this ALTS name object to configure the value.

```
====== AcpiBoard.c======
VOID
AcpiUpdateTable (
  IN OUT   EFI_ACPI_DESCRIPTION_HEADER  *TableHeader,
  IN OUT   EFI_ACPI_TABLE_VERSION       *Version
  )
{

  case EFI_ACPI_1_0_SECONDARY_SYSTEM_DESCRIPTION_TABLE_SIGNATURE:
    if (TableHeader->OemTableId == SIGNATURE_64('G', 'a', 'l', 'i', 'l', 'e', 'o', '1'))
{
      AML_NAME_DWORD_OBJ             *NameObj;
      //
      // Patch some pointers for the ASL code before loading the SSDT.
      //
      for (NameObj = (AML_NAME_DWORD_OBJ *)(TableHeader + 1);
           NameObj < (AML_NAME_DWORD_OBJ *)((UINT8 *)TableHeader + TableHeader->Length);
           NameObj = (AML_NAME_DWORD_OBJ *)((UINT8 *)NameObj + 1)) {
        if ((NameObj->NameOp == AML_NAME_OP) &&
            (NameObj->NameString == SIGNATURE_32('A', 'L', 'T', 'S')) &&
            (NameObj->DWordPrefix == AML_DWORD_PREFIX) ) {
          DEBUG((EFI_D_INFO, "Patch ALTS to 0x%x\n", mAlternateSla));
          NameObj->Value = mAlternateSla;
          break;
        }
      }
    }}
====================
```

Board specific VFR

We observed many BIOS code modules use the board ID check in VFR code. For example:

```
====== Old vfr ======
 oneof varid = SETUP_DATA.CameraType,
       prompt      = STRING_TOKEN(STR_CAMERA_TYPE),
       help        = STRING_TOKEN(STR_CAMERA_TYPE_HELP),
       default value = cond(ideqvallist SETUP_VOLATILE_DATA.PlatId == BoardIdBoardX
                                                                    ? 0x0:0x1),
defaultstore = MyStandardDefault,
       option text = STRING_TOKEN(STR_IVCAM_CAMERA), value = 0, flags = DEFAULT |
MANUFACTURING | RESET_REQUIRED;
       option text = STRING_TOKEN(STR_DS4_CAMERA),   value = 1, flags = RESET_REQUIRED;
 endoneof;
======================
```

This "default value" keyword is good for VFR because it provides a flexible way to determine a default value. However it is not good way for a board ID check. It is NOT portable if we want to add a new board. The better way is to add board specific override in `PcdFuncConfigInit`.

```
====== recommended vfr ======
       prompt      = STRING_TOKEN(STR_CAMERA_TYPE),
       help        = STRING_TOKEN(STR_CAMERA_TYPE_HELP),
       option text = STRING_TOKEN(STR_IVCAM_CAMERA), value = 0, flags = DEFAULT |
MANUFACTURING | RESET_REQUIRED;
       option text = STRING_TOKEN(STR_DS4_CAMERA),   value = 1, flags = RESET_REQUIRED;
 endoneof;
======================
====== recommended ConfigInit ======
EFI_STATUS
BoardXConfigInit (
  VOID
  )
{
  BOOLEAN     DefaultConfig;

  Status = PlatformConfigInit (&DefaultConfig);
  ASSERT_EFI_ERROR(Status);

  if (DefaultConfig) {
    //
    // Override setup configuration
    //
    Setup = PcdGetPtr (PcdSetupConfiguration);
    Setup->CameraType = 1;
    Size = sizeof(SETUP_DATA);
    PcdSetPtr (PcdSetupConfiguration, &Size, Setup);
======================
```

- **Board Default setup variable data**

If the BIOS uses different default setup variable values for different boards, some additional steps are needed. During the BIOS build phase, a FCE (Firmware Configuration Edit) tool (https://firmware.intel.com/sites/default/files/2015-WW48-FCE.31.zip) scans BIOS binary, extracts the setup IFR binary, and saves a set of configuration to the BIOS boot block. For example, if a BIOS supports 3 boards (Board1, Board2, Board3), there are 3 configuration data objects (Board1Config, Board2Config, Board3Config) saved in the BIOS boot block.

Because we do not recommend using board ID in the VFR page, we eliminate the multi board configuration. However the concept of default configuration is very important because it provides the system a way to recover if the user configuration causes system crash. How do we support that?

During BIOS boot time, the platform initialization module will call `PcdFuncConfigInit` to initialize the configuration data. As a default policy, the configuration data is stored in the UEFI variable region. However, in some special cases, like CMOS-CLEAR, Hardware Watchdog fired, recovery mode, or no variable because of first boot, the BIOS needs to use default configuration data. This is done by `PlatformConfigInit()`. If the above special condition is met, then the Multiplatform library (https://github.com/jyao1/OpenPlatform/blob/master/QuarkPlatformPkg/Include/Library/MultiPlatSupportLib.h , https://github.com/jyao1/OpenPlatform/tree/master/QuarkPlatformPkg/Library/PeiMultiPlatSupportLib) `CreateDefaultVariableHob()` function is invoked. `CreateDefaultVariableHob()` extracts data from the BIOS boot block and creates a HOb so that the variable driver knows the setup variable data information.

```
====== PeiBoardConfigLib.c ======
EFI_STATUS
EFIAPI
PlatformConfigInit (
  OUT BOOLEAN  *DefaultConfig
  )
{
  //
  // Check CMOS battery is ok.
  //
  if (IsCmosBad ()) {
    DEBUG ((DEBUG_INFO, "CMOS battery is bad. Reset the Setup variable.\n"));
    Status = CreateDefaultVariableHob (EFI_HII_DEFAULT_CLASS_STANDARD, 0);
    if (EFI_ERROR (Status)) {
      return Status;
    }
    *DefaultConfig = TRUE;
  }

  //
  // Check BootMode on Recovery boot or Boot with Default settings.
  //
  else if (BootMode == BOOT_IN_RECOVERY_MODE || BootMode == BOOT_WITH_DEFAULT_SETTINGS) {
    Status = CreateDefaultVariableHob (EFI_HII_DEFAULT_CLASS_STANDARD, 0);
    if (EFI_ERROR (Status)) {
      return Status;
    }
    *DefaultConfig = TRUE;
  }

  //
  // Check whether Setup Variable does exist to know the first boot or not.
  //
  DataSize = sizeof (SETUP_DATA);
  Status  = VariableServices->GetVariable (VariableServices, L"Setup",
&gSetupVariableGuid, NULL, &DataSize, &Setup);
```

```
  //
  // Setup variable is not found. So, set the default setting.
  //
  if (Status == EFI_NOT_FOUND) {
    Status = CreateDefaultVariableHob (EFI_HII_DEFAULT_CLASS_STANDARD, 0);
    if (EFI_ERROR (Status)) {
      return Status;
    }
    *DefaultConfig = TRUE;
    ......
=======================
```

After the board neutral default configuration is retrieved, each board can update the value and save it to a PCD again. This can ensure that different board constructs have different default configuration data without using board ID in setup. It can also save the BIOS build time because it takes long time to let the FCE tool generate multiplatform data for each board.

```
====== BoardInit.c ======
EFI_STATUS
BoardXConfigInit (
  VOID
  )
{
  BOOLEAN     DefaultConfig;

  Status = PlatformConfigInit (&DefaultConfig);
  ASSERT_EFI_ERROR(Status);

  if (DefaultConfig) {
    //
    // Override setup configuration
    //
    Setup = PcdGetPtr (PcdSetupConfiguration);
    Setup->CameraType = 1;
    Size = sizeof(SETUP_DATA);
    PcdSetPtr (PcdSetupConfiguration, &Size, Setup);
======================
```

The below figure shows the flow:

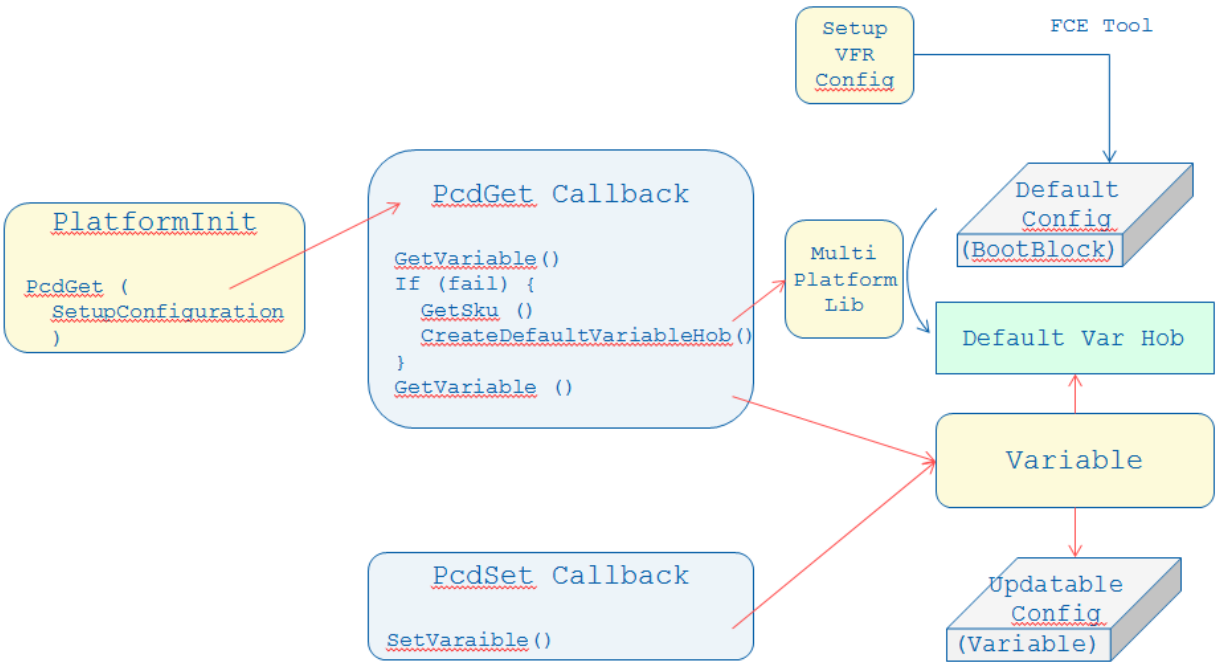


Figure 9 Multi-board support for setup configuration

Because the Quark platform does not have VFR, this example is not demonstrated on that platform.

- **Setup LoadDefault support**

Sometimes a setup driver can have a feature to load a default configuration based upon if the user updates some configuration by mistake. This default configuration might be different on each board.

The PCD PcdSetupConfiguration always contains the *current* setting, but not the *default* setting.

The DefaultConfig inserted by the FCE tool is only *common* default values, but not board specific default values.

In order to support "LoadDefault", we need another PcdSetupConfigurationDefault. The PCD PcdSetupConfigurationDefault contains board specific *default* settings. It should also be updated by BoardXConfigInit() according to different board types.

Then when the setup driver gets a "LoadDefault" request later, the setup driver uses the value in PcdSetupConfigurationDefault.

Because the Quark platform does not have VFR, this example is not demonstrated on Quark.

SKU-PCD

SkuIds is a special usage of PCD. It can support multiple configurations generated at build time, and it supports runtime selection to make one configuration take effect finally. Below is an example that shows how to use SKU-PCD.

```
====== dsc file ======
[SkuIds]
  0|DEFAULT              # The entry: 0|DEFAULT is reserved and always required.
  4|BoardX
  0x42|BoardY

[PcdsDynamicDefault.common.BoardX]
    gPlatformModuleTokenSpaceGuid.PcdGpioPin|0x8
    gPlatformModuleTokenSpaceGuid.PcdGpioInitValue|{0x00, 0x04, 0x02, 0x04, ...}

[PcdsDynamicDefault.common.BoardY]
    gPlatformModuleTokenSpaceGuid.PcdGpioPin|0x4
    gPlatformModuleTokenSpaceGuid.PcdGpioInitValue|{0x00, 0x02, 0x01, 0x02, ...}

============================
```

The SKU PCD is actually a dynamic PCD. The current implementation just puts all configuration data together without any size optimization, so even a one byte difference will cause a full SKU configuration to be duplicated. We can enhance the SkuIds PCD implementation. If there is any size concern in SkuIds PCD, the alternative could be: define one PcdsDynamicDefault, and let each platform update its own configuration there.

If a user finds it is hard to write PCD initialization in DSC file, the alternative is to define a DynamicDefaultPCD with all zero as the initialized value, and then let each platform update the value in BoardInit.c. If this solution is chosen, there is no need to define this configuration to be a SKU-PCD.

```
====== BoardInit.c ======
EFI_STATUS
EFIAPI
BoardXBoardDetectionCallback (
  IN CONST EFI_PEI_SERVICES     **PeiServices,
  IN EFI_PEI_NOTIFY_DESCRIPTOR  *NotifyDescriptor,
  IN VOID                       *Ppi
  )
{
  ...
  if (BoardId == BoardIdBoardX) {
    // It is boardX

    // Data
    DataSize = sizeof(BoardXSpdAddressTable);
    PcdSetPtr(PcdSpdAddressTable, &DataSize, BoardXSpdAddressTable);

    // Data pointer
    PcdSet64 (PcdResistorPtr, (UINT64)(UINTN)BoardXResistor);
  }
=======================================
```

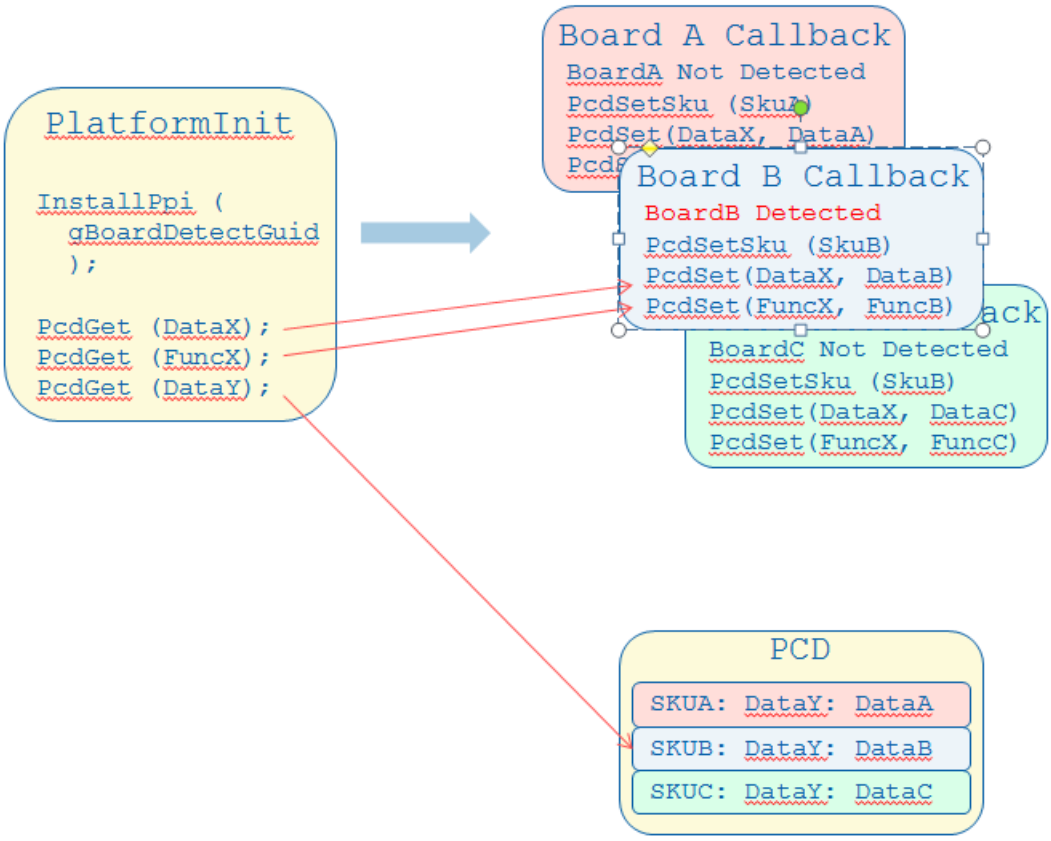


Figure 10 PCD usage summary

Summary

This section introduces porting – board specific initialization.

Security

Chipsec

CHIPSEC is a framework for security assessment of hardware and firmware components on the platform, enabling security research, testing, and forensics. Intel originally created it to help internal teams find and fix vulnerabilities in platform hardware and firmware.

We highly recommend a platform BIOS runs CHIPSEC before release.

HSTI

Microsoft Hardware Security Test Interface [HSTI] provides a set of requirement of Windows Hardware Certification Requirement.

EDKII provides a HSTI general definition at MdePkg/Include/IndustryStandard/Hsti.h and DXE phase library to construct HSTI table at MdePkg/Library/DxeHstiLib

A silicon HSTI driver should produce HSTI table as `PLATFORM_SECURITY_ROLE_PLATFORM_REFERENCE`. A platform HSTI driver should produce HSTI table as `PLATFORM_SECURITY_ROLE_PLATFORM_IBV`, `PLATFORM_SECURITY_ROLE_IMPLEMENTOR_OEM` or `PLATFORM_SECURITY_ROLE_IMPLEMENTOR_ODM`.

We highly recommend a platform BIOS reports HSTI and make sure no errors are reported in a HSTI table before release.

WSMT

In order to mitigate SMM communication buffer security issue on Microsoft Virtualization Based Security (VBS) in Windows 10, a platform BIOS need check all SMI handles to make sure all SMI handlers use fixed communication buffer and report Windows Security Mitigations Table (WSMT). [WSMT]

The latest EDKII core follows WSMT recommendation and enables fixed communication buffer. [SecureSmmComm]

We highly recommend a platform BIOS checks all SMI handlers and reports the WSMT table.

Summary

This section introduces assessing the security of a platform.

Core Module Selection

Mandatory V.S. Optional

EDKII source code has lots of modules (driver and library). Not all of them are needed in the final BIOS image.

We made an analysis on current EDKII core package usage in EDKII Quark platform.
In this analysis, EDKII "Core package" means `CryptoPkg, FatBinPkg, FatPkg, IntelFrameworkModulePkg, IntelFrameworkPkg, IntelFspPkg, IntelFspWrapperPkg, MdeModulePkg, MdePkg, NetworkPkg, PcAtChipsetPkg, PerformancePkg, SecurityPkg, ShellBinPkg, ShellPkg, SourceLevelDebugPkg, UefiCpuPkg`.
See Table 1 for the result.

	Quark (ShellOnly)	Quark	All
Module Count	35	70	244
Library Count	76	102	280
Include File Count	360	456	799
Module File Count	326	714	2247
Library File Count	955	1079	2159
All File Count	1678	2256	5297

Table 1 – Core module usage in EDKII BIOS.

For the final result, only 35 core module are used in the Quark (ShellOnly) platform. A full feature Quark needs 70 modules. More than 200 modules are not used in the Quark (ShellOnly) platform. These are optional.

This data can be obtained from a tool - CheckCodeBase.py (https://github.com/jyao1/OpenPlatform/blob/master/Tool/CheckCodeBase.py). It helps analyze how many files/modules are used in a tree.

We give a sample list of x86 min BIOS PEI/DXE module as Appendix B.

Deprecated module

Some EDKII core modules are marked as deprecated. If so, please try to not use them.
For example:

IntelFrameworkModulePkg\Universal\BdsDxe
SecurityPkg\Tcg\MemoryOverwriteRequestControlLock

Deprecated packages:

EdkCompatibilityPkg
EdkShellBinPkg
EdkShellPkg

Packages will be deprecated:

IntelFrameworkPkg
IntelFrameworkModulePkg

Deprecated API

The Deprecated API is quoted by `DISABLE_NEW_DEPRECATED_INTERFACES`

We suggest a platform enables this MACRO to make sure no deprecated APIs are used.

Core Module Override

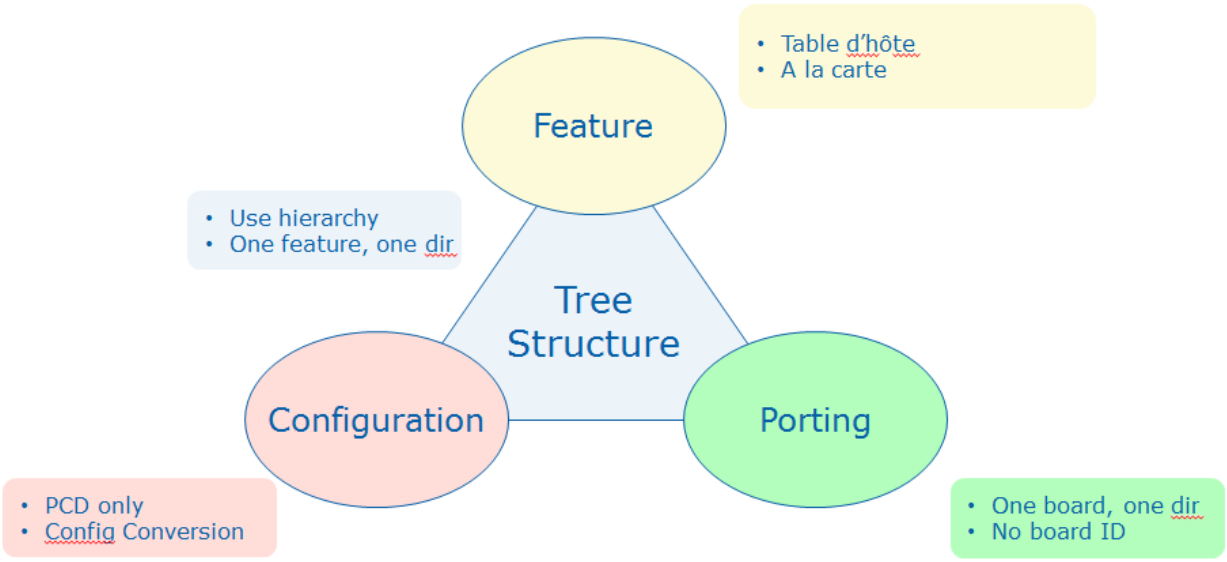
Some core modules might not meet a particular platform requirement. A platform may create an Override directory to create a duplicate set of core modules. This is only acceptable as a short term solution when a core module has design issues and the core module owner cannot resolve the issue quickly.

In general, we do not recommend a platform owner overriding the core module directly and putting them into a platform directory because it might bring core synchronization/update issues later. We recommend a platform owner reporting issue to a core module owner (see maintainer list on tianocore.org) to address the problem and work with core module owner to design a new solution. This not only benefits the platform owner by removing the extra overhead to maintain a new driver, but it also benefits the core module owner by enhancing the core solution to be applicable for more platforms.

Summary
This section talks about the distinction between core and platform modules.

Summary

Summarize recommended guideline below:



Appendix A – Open Platform Design Guideline

Platform Feature [F]

- [F1] Put one feature to one directory.
- [F2] Put advanced features to `$(PLATFORM_PACKAGE)/Features`.

Policy Configuration [C]

- [C1] Don't call GetVariable/SetVariable to get/set policy data.
- [C2] Use PCD to pass policy data in platform code.
- [C3] Use PCD callback to manage PCD on other storage, like variable.
- [C4] Use silicon interface (Hob/PPI/Protocol/FSP UPD) for silicon only.
- [C5] Expose one configuration source to user/developer.

Board Specific Code [B]

- [B1] Don't use a board specific check (BoardId/PlatformType) in any C file, except board specific driver.
- [B2] Don't use a board specific check (BoardId/PlatformType) in any ASL, except board specific driver.
- [B3] Don't use a board specific check (BoardId/PlatformType) in any VFR, except board specific driver.
- [B4] Put board specific configuration to `$(PLATFORM_PACKAGE)/Board/BoardX`.
- [B5] Put board specific initialization to `$(PLATFORM_PACKAGE)/Board/BoardX/BoardInit`.
- [B6] Put board specific ACPI table to `$(PLATFORM_PACKAGE)/Board/BoardX/AcpiTables`.
- [B7] Override board specific ACPI configuration in `$(PLATFORM_PACKAGE)/Board/BoardX/AcpiTables`.
- [B8] Override board specific VFR default value in `$(PLATFORM_PACKAGE)/Board/BoardX/BoardInit`.
- [B9] Platform code should not have board specific knowledge and not depend upon board code.

Secure By Default [S]

- [S1] Enable all silicon/platform security features by default. Especially, SMM Lock, Flash Lock, Chipset register lock, etc.
- [S2] Run CHIPSEC before release.
- [S3] Enable HSTI for Windows.
- [S4] Report WSMT for Windows Hypervisor.

Core module selection [M]

- [M1] Do not override EDKII core module.
- [M2] Do not use API deprecated by `DISABLE_NEW_DEPRECATED_INTERFACES`
- [M3] Do not use deprecated module in EDKII.

Appendix B – x86 Min BIOS Template

Many people asked how to create a minimal UEFI BIOS. Here is template for BASIC OS BOOT:

```
====== CoreBasicIncludePei.dsc ======

  # SEC
  UefiCpuPkg/SecCore/SecCore.inf

  # PEI Main
  MdeModulePkg/Core/Pei/PeiMain.inf

  # PCD
  MdeModulePkg/Universal/PCD/Pei/Pcd.inf

  # Status Code
  MdeModulePkg/Universal/ReportStatusCodeRouter/Pei/ReportStatusCodeRouterPei.inf
  MdeModulePkg/Universal/StatusCodeHandler/Pei/StatusCodeHandlerPei.inf

  # Variable
  MdeModulePkg/Universal/FaultTolerantWritePei/FaultTolerantWritePei.inf
  MdeModulePkg/Universal/Variable/Pei/VariablePei.inf

  # FSP (optional)
  IntelFsp2WrapperPkg/FspmWrapperPeim/FspmWrapperPeim.inf
  IntelFsp2WrapperPkg/FspsWrapperPeim/FspsWrapperPeim.inf

  # Capsule
  MdeModulePkg/Universal/CapsulePei

  # IPL
  MdeModulePkg/Core/DxeIplPeim/DxeIpl.inf

  # S3 (optional)
  UefiCpuPkg/PiSmmCommunication/PiSmmCommunicationPei.inf
  UefiCpuPkg/Universal/Acpi/S3Resume2Pei/S3Resume2Pei.inf

==========================

====== CoreBasicIncludeDxe.dsc ======
  # DXE Main
  MdeModulePkg/Core/Dxe/DxeMain.inf

  # PCD
  MdeModulePkg/Universal/PCD/Dxe/Pcd.inf

  # Arch
  UefiCpuPkg/CpuDxe/CpuDxe.inf
  PcAtChipsetPkg/HpetTimerDxe/HpetTimerDxe.inf
  MdeModulePkg/Universal/ResetSystemRuntimeDxe/ResetSystemRuntimeDxe.inf
  MdeModulePkg/Universal/Metronome/Metronome.inf
MdeModulePkg/Universal/ReportStatusCodeRouter/RuntimeDxe/ReportStatusCodeRouterRuntimeDxe
.inf
  MdeModulePkg/Universal/StatusCodeHandler/RuntimeDxe/StatusCodeHandlerRuntimeDxe.inf
  MdeModulePkg/Core/RuntimeDxe/RuntimeDxe.inf
```

```
MdeModulePkg/Universal/SecurityStubDxe/SecurityStubDxe.inf
MdeModulePkg/Universal/BdsDxe/BdsDxe.inf
MdeModulePkg/Universal/FaultTolerantWriteDxe/FaultTolerantWriteSmm.inf
MdeModulePkg/Universal/Variable/RuntimeDxe/VariableSmmRuntimeDxe.inf
MdeModulePkg/Universal/Variable/RuntimeDxe/VariableSmm.inf
MdeModulePkg/Universal/WatchdogTimerDxe/WatchdogTimer.inf
MdeModulePkg/Universal/MonotonicCounterRuntimeDxe/MonotonicCounterRuntimeDxe.inf
MdeModulePkg/Universal/CapsuleRuntimeDxe/CapsuleRuntimeDxe.inf
PcAtChipsetPkg/PcatRealTimeClockRuntimeDxe/PcatRealTimeClockRuntimeDxe.inf

# Misc
UefiCpuPkg/CpuIo2Dxe/CpuIo2Dxe.inf
PcAtChipsetPkg/8259InterruptControllerDxe/8259.inf
MdeModulePkg/Universal/DevicePathDxe/DevicePathDxe.inf

# ACPI
MdeModulePkg/Universal/Acpi/AcpiTableDxe/AcpiTableDxe.inf

# SMM
MdeModulePkg/Core/PiSmmCore/PiSmmIpl.inf
MdeModulePkg/Core/PiSmmCore/PiSmmCore.inf
UefiCpuPkg/PiSmmCpuDxeSmm/PiSmmCpuDxeSmm.inf
UefiCpuPkg/CpuIo2Smm/CpuIo2Smm.inf
MdeModulePkg/Universal/ReportStatusCodeRouter/Smm/ReportStatusCodeRouterSmm.inf
MdeModulePkg/Universal/StatusCodeHandler/Smm/StatusCodeHandlerSmm.inf
UefiCpuPkg/PiSmmCommunication/PiSmmCommunicationSmm.inf

# PCI
MdeModulePkg/Bus/Pci/PciHostBridge/PciHostBridge.inf
MdeModulePkg/Bus/Pci/PciBusDxe/PciBusDxe.inf

# ATA
MdeModulePkg/Bus/Ata/AtaBusDxe/AtaBusDxe.inf
MdeModulePkg/Bus/Ata/AtaAtapiPassThru/AtaAtapiPassThru.inf

# USB
MdeModulePkg/Bus/Pci/EhciDxe/EhciDxe.inf
MdeModulePkg/Bus/Pci/UhciDxe/UhciDxe.inf
MdeModulePkg/Bus/Usb/UsbBusDxe/UsbBusDxe.inf
MdeModulePkg/Bus/Pci/XhciDxe/XhciDxe.inf
MdeModulePkg/Bus/Usb/UsbKbDxe/UsbKbDxe.inf
MdeModulePkg/Bus/Usb/UsbMassStorageDxe/UsbMassStorageDxe.inf

# FS
MdeModulePkg/Universal/Disk/DiskIoDxe/DiskIoDxe.inf
MdeModulePkg/Universal/Disk/PartitionDxe/PartitionDxe.inf
MdeModulePkg/Universal/Disk/UnicodeCollation/EnglishDxe/EnglishDxe.inf
FatBinPkg/EnhancedFatDxe/Fat.inf

# Console
MdeModulePkg/Universal/Console/ConPlatformDxe/ConPlatformDxe.inf
MdeModulePkg/Universal/Console/ConSplitterDxe/ConSplitterDxe.inf
MdeModulePkg/Universal/Console/GraphicsConsoleDxe/GraphicsConsoleDxe.inf

# FSP (optional)
IntelFsp2WrapperPkg/FspWrapperNotifyDxe/FspWrapperNotifyDxe.inf

# UI (optional)
```

```
  MdeModulePkg/Universal/HiiDatabaseDxe/HiiDatabaseDxe.inf
  MdeModulePkg/Universal/SetupBrowserDxe/SetupBrowserDxe.inf
  MdeModulePkg/Universal/DisplayEngineDxe/DisplayEngineDxe.inf
  MdeModulePkg/Application/UiApp/UiApp.inf
  MdeModulePkg/Application/BootManagerMenuApp/BootManagerMenuApp.inf

  # Shell (optional)
  ShellBinPkg/UefiShell/UefiShell.inf

  # S3 (optional)
  MdeModulePkg/Universal/Acpi/S3SaveStateDxe/S3SaveStateDxe.inf
  MdeModulePkg/Universal/Acpi/SmmS3SaveState/SmmS3SaveState.inf
  MdeModulePkg/Universal/Acpi/BootScriptExecutorDxe/BootScriptExecutorDxe.inf
  MdeModulePkg/Universal/LockBox/SmmLockBox/SmmLockBox.inf

==========================
```

Conclusion

The previous IA firmware examples are sometimes complex and inconsistent. We are trying to provide some general guidelines for an open source IA firmware design to create a simple and consistent platform code set, and to scale the solution to all Intel platforms. These platforms include ATOM, Core-i7, and XEON generation.

Glossary

CMOS - Complementary Metal Oxide Semiconductor. A storage for legacy BIOS.

HSTI – Hardware Security Test Interface.

IPL – Initial program loader.

PCD – Platform Configuration Database. See [UEFI PI Specification].

PI – Platform Initialization. Volume 1-5 of the UEFI PI specifications.

UEFI – Unified Extensible Firmware Interface. Firmware interface between the platform and the operating system.

UPD – Updatable production data. See [FSP EAS].

VPD – Vital production data. See [FSP EAS].

References

[ACPI] Advanced Configuration and Power Interface, vesion 6.0, www.uefi.org

[AUTH VARIABLE] Yao, Zimmer, Zeng, "A Tour Beyond BIOS Implementing UEFI Authenticated Variables in SMM with EDKII – Verion 2," September 2015 https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Implementing_UEFI_Authenticated_Variables_in_SMM_with_EDKII_V2.pdf

[Baytrail Data sheet] Intel® Atom™ Processor E3800: Datasheet, http://www.intel.com/content/www/us/en/embedded/products/bay-trail/atom-e3800-family-datasheet.html

[Braswell Data sheet] N-series Pentium® and Celeron® Processors Datasheet, http://www.intel.com/content/www/us/en/processors/pentium/pentium-celeron-n-series-datasheet-vol-1.html, http://www.intel.com/content/www/us/en/processors/pentium/pentium-celeron-n-series-datasheet-vol-2.html, http://www.intel.com/content/www/us/en/processors/pentium/pentium-celeron-n-series-datasheet-vol-3.html

[CHIPSEC] CHIPSEC: Platform Security Assessment Framework http://www.intelsecurity.com/advanced-threat-research/chipsec.html

[COREBOOT] coreboot firmware www.coreboot.org

[EDK2] UEFI Developer Kit www.tianocore.org

[EDKII specification] A set of specification describe EDKII DEC/INF/DSC/FDF file format, as well as EDKII BUILD. http://tianocore.sourceforge.net/wiki/EDK_II_Specifications

[EMBED] Sun, Jones, Reinauer, Zimmer, "Embedded Firmware Solutions: Development Best Practices for the Internet of Things," Apress, January 2015, ISBN 978-1-4842-0071-1

[FSP] Intel Firmware Support Package http://www.intel.com/content/www/us/en/intelligent-systems/intel-firmware-support-package/intel-fsp-overview.html

[FSP EAS] FSP External Architecture Specification http://www.intel.com/content/www/us/en/embedded/software/fsp/fsp-architecture-spec-v1-1.html

[FSP Consumer] Yao, Zimmer, Rangarajan, Ma, Estrada, Mudusuru, "A_Tour_Beyond_BIOS_Using_the_Intel_Firmware_Support_Package_with_the_EFI_Developer_Kit_II_(FSP1.1)" http://firmware.intel.com

[HSTI] Hardware Security Testability Specification http://msdn.microsoft.com/en-us/library/windows/hardware/dn879006.aspx

[Intel Graphic OpRegion] Intel® Integrated Graphics Device - OpRegion Specification https://01.org/linuxgraphics/documentation/intel%C2%AE-integrated-graphics-device-opregion-specification

[IA32 Manual] Intel® 64 and IA-32 Architectures Software Developer Manuals http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[Intel DCMI] DCMI Host Interface Specification http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/dcmi-hi-1-0-spec.pdf

[Intel SGX] Intel® Software Guard Extensions Programming Reference https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf

[Intel TraceHub] Intel® Trace Hub Developer's Manual https://software.intel.com/sites/default/files/managed/d3/3c/intel-th-developer-manual.pdf

[Intel TXT] Intel® Trusted Execution Technology (Intel® TXT) Software Development Guide http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf

[Intel VT-d] Intel® Virtualization Technology for Directed I/O specification, Rev 2.3 http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/vt-directed-io-spec.html

[OSTS1] Vincent Zimmer, "Open Source IA Firmware Directions", Open Source Technology Summit, 2015

[Quark data sheet] Intel® Quark™ SoC X1000: Datasheet, http://www.intel.com/content/www/us/en/embedded/products/quark/quark-x1000-datasheet.html

[SECURE BOOT] Nystrom, Nicoles, Zimmer, "UEFI Networking and Pre-OS Security," Intel Technology Journal, October 2011

[Skylake SA data sheet] 6th Generation Intel® Core™ Processor Family Datasheet http://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-datasheet-vol-1.html?wapkw=6th+generation+intel+core+datasheet and http://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-datasheet-vol-2.html?wapkw=6th+generation+intel+core+datasheet

[Skylake PCH (Sunrise Point) data sheet] Intel® 100 Series Chipset Family PCH Datasheet http://www.intel.com/content/www/us/en/chipsets/100-series-chipset-datasheet-vol-1.html?wapkw=6th+generation+intel+core+datasheet and

http://www.intel.com/content/www/us/en/chipsets/100-series-chipset-datasheet-vol-2.html?wapkw=6th+generation+intel+core+datasheet

[SecureSmmComm] Yao, Zimmer, Zeng, A tour beyond BIOS secure SMM communciation, https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Secure_SMM_Communication.pdf

[UEFI Plugfest1] Leif Lindholm, "A Common Platforms Tree", UEFI Plugfest, 2015

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.5 www.uefi.org

[UEFI Book] Zimmer, et al, "Beyond BIOS: Developing with the Unified Extensible Firmware Interface," 2nd edition, Intel Press, January 2011

[UEFI PI Specification] UEFI Platform Initialization (PI) Specifications, volumes 1-5, Version 1.4 www.uefi.org

[WHCK System] Windows Hardware Certification Requirements for Client and Server Systems http://msdn.microsoft.com/en-us/library/windows/hardware/jj128256.aspx

[WSMT] Windows SMM Security Table, https://msdn.microsoft.com/en-us/library/windows/hardware/dn495660(v=vs.85).aspx#wsmt http://download.microsoft.com/download/1/8/A/18A21244-EB67-4538-BAA2-1A54E0E490B6/WSMT.docx

Authors

Jiewen Yao (jiewen.yao@intel.com) is an EDKII BIOS architect, EDKII TPM2 module maintainer, ACPI/S3 module maintainer, and FSP package owner with Software and Services Group (SSG) at Intel Corporation.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer with the Software and Services Group (SSG) at Intel Corporation. Vincent chairs the UEFI Security and Networking Subteams in the UEFI Forum.