Minimum Platform Specification

TABLE OF CONTENTS

EDK II Minimum Platform Specification

DRAFT FOR REVIEW

12/01/2020 06:42:40

Acknowledgements

Revision History

Revision	Revision History	Date
0.7	Initial release	May 2019

Tables

Figures

1 INTRODUCTION

This specification details the required and optional elements for an EDK II based platform design with the following objectives:

1. Define a structure that enables developers to consistently navigate source code, execution flow, and the functional results of bootstrapping a system.
2. Enable a minimal platform where minimal is defined as the minimal firmware implementation required to produce a basic solution that can be further extended to meet a multitude of client, server, and embedded market needs.
3. Minimize coupling between common, silicon, platform, and board packages.
4. Enable large granularity binary solutions.

A key aspect of these objectives is to improve the transparency and security quality across the client, server, and embedded ecosystems.

This document assumes a working knowledge of the EDK II and UEFI Specifications. The minimal platform defined supports the use of Intel® Firmware Support Package (FSP), but does not require usage of the Intel® FSP API. The minimal platform is binary component oriented, but designed to enable a highly optimized form for embedded boot loaders.

1.1 Audience / Document scope

The audience for this document includes UEFI firmware architecture, development, validation, and enabling engineers.

The UEFI Forum and the TianoCore.org EDK II specifications provide tremendous flexibility and extensibility. The Minimum Platform Architecture is intended to introduce interfaces and structure so that platforms are consistent and thus approachable by engineers from across the ecosystem. The minimal platform specifically refers to a platform layer within a multi-layer solution; its scope and therefore this specification defines this layer and its dependencies. The minimal platform is a single code package used as-is from open source similar to MDE Module package usage. By using this platform as a base, the fundamental boot flow is consistent, well-documented, and visible across the UEFI community.

This approach does not rule out innovation and customization. The platform calls two primary sets of external APIs throughout the boot, for board and chipset initialization. These APIs are considered dependencies, and therefore are defined in this specification. The implementation of these APIs is expected to vary based on unique board and chipset requirements. Furthermore, the minimal platform can be arbitrarily extended through a simple and modular advanced feature design.

The Minimum Platform Architecture enables scalability from pre-silicon validation activities, to final product shipment, to derivative product use. The Minimum Platform Architecture should enable engineering activities from all segments: from high-touch Intel supported OEMs to individual makers with previous UEFI experience but no direct support from Intel.

1.2 Document Flow

The document introduces the Minimum Platform Architecture, and then details the "boot stages" through the following subsections.

`Overview`	**An overview of the stage**
`Firmware Volumes`	The binary containers needed for each stage
`Modules`	The EDK II component binaries and static libraries required
`Required Functions`	The architecturally defined functions for a given stage
`Configuration`	The defined configurable parameters for a given stage
`Data Flows`	The architecturally defined data structures and flows for a given stage
`Control Flows`	Key control flows for a given stage
`Build Files`	The DSC/FDF for a given stage
`Test Point Results`	The test that can verify porting is complete for a given stage
`Functional Exit Criteria`	The testable functionality for the stage
`Stage Enabling Checklist`	The required activities to achieve desired functionality for a given stage

Table 1 Document Flow

1.3 Terminology

Term	Definition
ACM	Authenticated Code Module
ACPI	Advanced Configuration and Power Interface
BCT	Intel Binary Configuration Tool
BFV	Boot Firmware Volume
BoardPkg	The EDK II package a developer creates to port the Minimum Platform for their motherboard or family of motherboards
BSF	Boot Setting File
CAR	Cache-As-RAM
Component	An executable binary. Typically UEFI defined, e.g. PEIM, DXE driver, SMM driver, or UEFI application. Also used to refer to other system binaries. Not appropriate for statically linked libraries.
DXE	Driver execution environment. Role is to load drivers for system devices. Finds and executes boot code. After OS loads, it handles OS to UEFI calls.
DSDT	Differentiated System Description Table
EC	Embedded Controller
EDK	EFI Development Kit
FACS	Firmware ACPI Control Structure
FADT	Firmware ACPI Description Table
FFS	EFI Firmware File System Specification
FRU	Field Replaceable Unit, the minimal silicon that can be added or removed from a system, e.g.
	an SoC, a MCP, a standalone processor or PCH.
FSP	Intel® Firmware Support Package
Full Platform	A platform implementation that includes the minimal features, as well as some number of advanced features. (Stage I-VII). Note: most advanced features may not be described in this document.
FV	Firmware Volume, a UEFI Forum defined firmware storage container
GPIO	General Purpose Input/Output
GUID	Globally Unique Identifier(s)
HOB	Hand Off Blocks(s)
Hybrid EDKII	Any Module that contains both EDKII compliant wrapper code, and non EDK payloads (e.g., CSM-bin or FSP-bin)
IBB	Initial Boot Block
IFWI	Integrated Firmware Image, includes things like UEFI firmware, microcode, microcontroller and firmware, configuration data.
Term	Definition
IPL	Initial Program Load
MASM	Microsoft Macro Assembler

Minimum Platform	A platform implementation that only includes the minimal features. (Stage I-VII)
MinPlatformPkg	The EDK II package that contains common elements of the platform architecture.
Module	Typically any EDK II independently buildable item, includes static libraries and executables.
MOR	Memory Overwrite Request. See Trusted Computing Group documentation.
MTRR	Memory Type Range Register
NASM	Netwide Assembler
Native EDKII	All modules build with only EDKII compliant source code, and no non-EDK payloads (e.g., CSM-bin, LegacyOpRom, or FSP-bin)
NVRAM	Non-Volatile Random Access Memory
OBB	OEM Boot Block
OPROM	Option ROM
PCD	Platform Configuration Database
PEI	Pre EFI Initialization. Role is to initialize memory, and also initialize enough of the system to run DXE.
PEIM	Pre-EFI Initialization Module
PI	Platform Initialization
PPI	PEIM-to-PEIM Interface
RSDP	Root System Description Pointer
RSDT	Root System Description Table
SEC	Security phase. Role is to initialize the system far enough to find, validate, install and run PEI.
SiliconPkg	The EDK II Package that contains silicon support for a system.
SIO	Super I/O is a type of I/O controller IP. Typical functionality provided are one or more serial port UARTs, keyboard controller, and many others.
SMBIOS	System Management BIOS
SMM	System Management Mode
SSDT	Secondary System Description Table
T-RAM	Temporary RAM (memory used before permanent memory is initialized such as CAR)
TPM	Trusted Platform Module
UEFI	Unified Extensible Firmware Interface
UPD	Updatable Product Data
XSDT	Extended System Description Table

Table 2 Terminology

1.4 Reference documents

The following documents are referenced in this specification.

Abbreviation	Document	Version
ACPI_SPEC	Advanced Configuration and Power Interface (ACPI) Specification	Version 6.3 January 2019
BSF_SPEC	Boot Setting File (BSF) Specification	Version 1.0 March 2016
FSP_EAS	FSP 2.0 External Architecture Specification (EAS)	Version 2.0 May 2016
OpenPlatform_WP	Intel® Open Platform White Paper	May 2017
PI_SPECS	Platform Initialization (PI) Specification	Version 1.7 January 2019
	Volume I: PEI	
	Volume II: DXE CIS	
	Volume III: Shared Architecture Elements	
	Volume IV: SMM	
	Volume V: Standards	
UEFI_SPEC	Unified Extensible Firmware Interface (UEFI) Specification	Version 2.8 March 2019

Table 3 Reference Documents

2 ARCHITECTURE

The Minimum Platform Architecture is structured around required functionality over time. As such, the key elements of architecture and design (flows, interfaces, etc) are organized into a staged architecture, where each stage will have requirements and functionality that lead to specific uses. Stages build upon prior stages with extensibility to meet silicon, platform, or board requirements.

Early in a development cycle, engineering is focused on creating the platform and acquiring basic functionality. This can be pursued within simulation and emulation environments, or on real hardware. This often includes creating a new set of silicon and platform source code that handles the basic differences between the new target and prior solutions. Often this entails reuse of the prior generation silicon support and existing feature sets.

Later development engineering effort is focused on enabling the full range of functionality, supporting all deltas in the new platform - typically in the form of reference designs and lead products. Next, platform development is focused on scaling. This involves customer enabling and aligning products for time-to-market and silicon roadmaps. Finally, there is sustaining, maintenance, and derivatives activity. These are characterized by smaller changes to existing production-worthy solutions, repurposing them opportunistically.

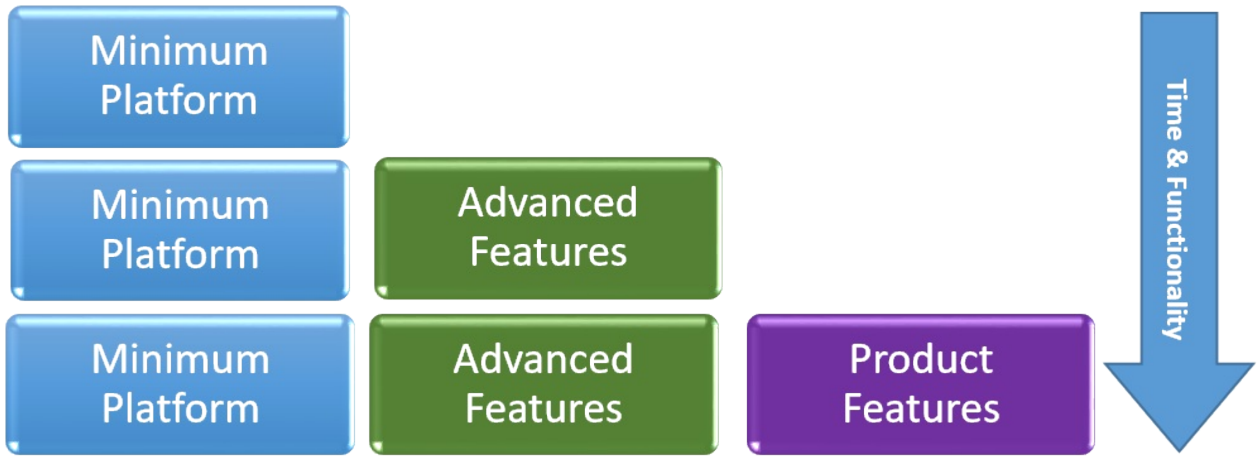


Figure 1 Minimum Platform Architecture Overview

Figure 1 shows the basic Minimum Platform Architecture. The enabling steps for a Minimum Platform solution should occur in the following order to add complexity over time, and only where it is necessary. This progression from minimum required to more full-featured permeates the design of the boot flow, modules implemented, and collection of components into firmware volumes.

1. **Develop a board solution around the minimum platform.** This involves implementing the essential board information and initialization defined in the Minimum Platform board API.
2. **Add silicon initialization support.** For many silicon vendors this will be accomplished through the use of binary blobs with well-defined interfaces. In any case, the silicon initialization invocation is performed from the board code.
3. **Add advanced features,** which are typically implemented in the form of source code designed to be generically plugged into a large number of diverse system types.
4. **Add product-specific features** that are required for product initialization. This support is not part of essential board organization or maintained as a generic advanced feature. It is enabled in the advanced feature stage.

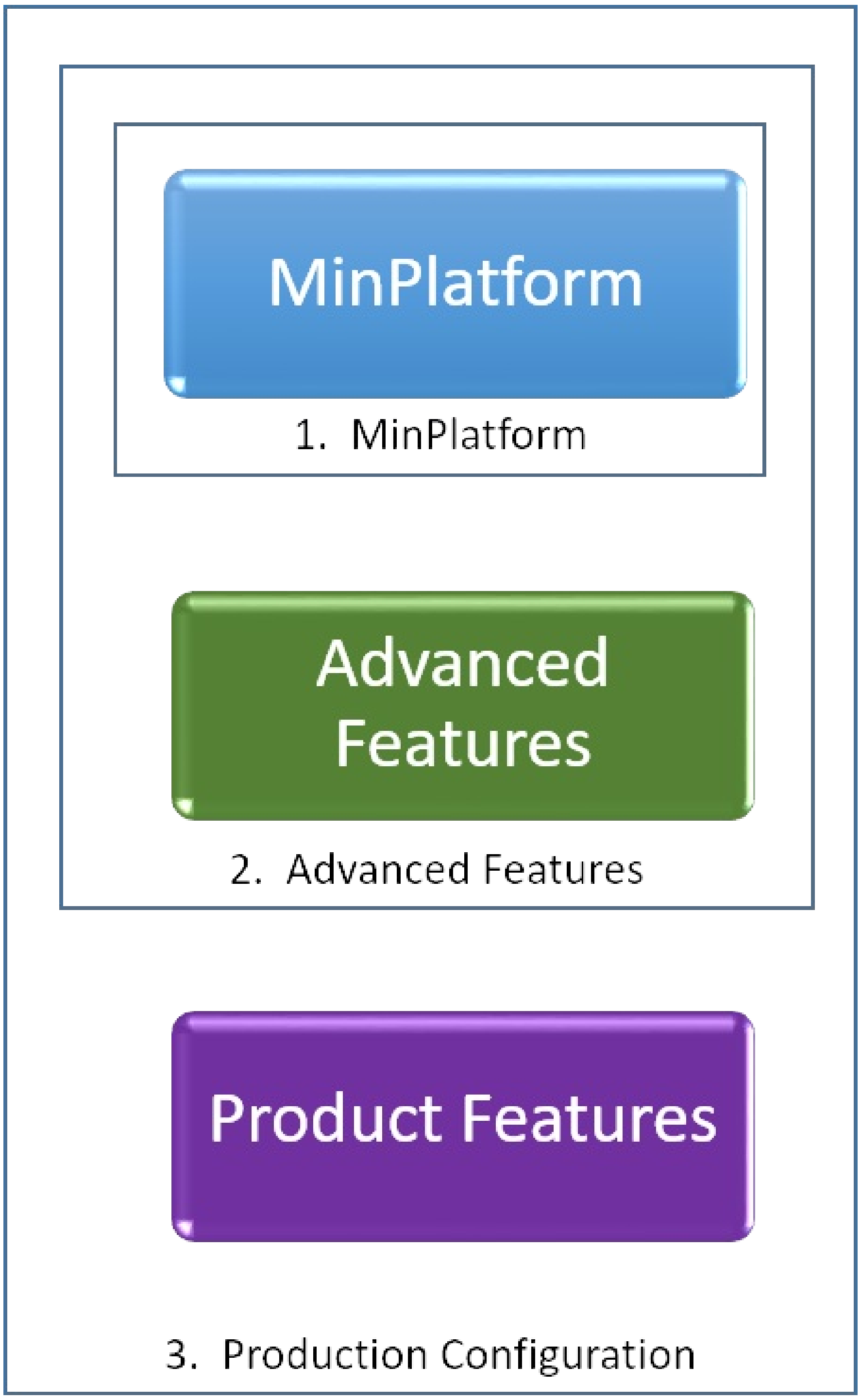


Figure 2 Minimum Platform Architecture High Level Sequence

Figure 2 shows the same basic idea can be represented as a Venn diagram with three fundamental steps.

This architecture is reflected in these areas:

- Source code in modules, packages, and resulting binary components
- Execution in control, data, error, and debug flows
- Functionality as solutions evolve from initial to complete
- Scaling from silicon development to products

2.1 Staged Architecture

The Minimum Platform Architecture defines a number of stages that are integral to the design and implementation of conformant firmware solutions. Stages define what code needs to be built, what functionality is required, what binary components are required at the FV and UEFI PI Architecture executables level, and what the system capabilities are available as a result of successfully executing through a firmware stage.

`Stage`	`Functional Objective`	`Example Capabilities`
`I`	Minimal Debug	Serial Port, Port 80, External debuggers Optional: Software debugger
`II`	Memory Functional	Basic hardware initialization including main memory
`III`	Boot to UEFI Shell	Generic DXE driver execution
`IV`	Boot to OS	Boot a general purpose operating system with the minimally required feature set. Publish a minimal set of ACPI tables.
`V`	Security Enabled	UEFI Secure Boot, TCG trusted boot, DMA protection, etc.
`VI`	Advanced Feature Selection	Firmware update, power management, networking support, manageability, testability, reliability, availability, serviceability, non-essential provisioning and resiliency mechanisms
`VII`	Tuning	Size and performance optimizations

Table 4 Architecture Stages

The stages correspond well to bootstrapping a system and to developing a production-worthy solution. The stages are defined in order to detail the minimum items required. It is expected that there will be more required and more present than what is defined in this specification for an end product. The stages capture what is minimally required to support the strategic objectives of transparency and quality as well as the more tactical objectives of structure, consistency, cohesion, coupling, and compatibility. Note that the stages represent enabling steps, not necessarily the order of execution. For example, ACPI initialization necessary in Stage IV may be performed before Stage III would be considered complete. Further, the stages may not necessarily be strictly additive once enabling is complete. For example, the UEFI shell may not be required in the production firmware image based on product requirements, but it must have been enabled and therefore capable of being loaded in the final firmware if chosen by a firmware engineer supporting the firmware in the final production image.

2.2 Modularity

Throughout the architecture you will find a mix of static and dynamic modularity. Static libraries are used to make the platform and board code simpler and more approachable. Dynamic linking in the form of UEFI PI Architecture components (PEIM and drivers) as well as large grain Firmware Volume containers are also used extensively in order to increase leverage, scalability, and large grain FV container updates.

An important concept is that the minimum platform is defined with a mix of solutions for modularity, but that said modularity will be flexible. It is intended that products will be able to scale from fully statically-linked embedded solutions to fully dynamically-linked leveraged validation solutions as they reach the end of Stage VII optimization activities. These advanced solutions can be considered derivatives of the platform architecture and should not undermine the strategic objectives of transparency and quality, nor the more tactical objectives of structure, consistency, cohesion, coupling, and compatibility.

2.3 Execution

To provide for a simpler progression through functionality, there are controls that enable building and executing the minimal code to achieve the goal for a particular stage. There is a PCD control that platform and board code can use to limit the scope of what must be functional at a given state of development. For example, if the board is configured for Stage I functionality, developers should not be burdened with errors because required porting to make memory functional has not been done yet. Similarly, initializing memory should not be burdened with the functionality for authenticating cryptographic hashes required for Stage V.

2.4 Organization

The architecture makes use of four primary classifications of code that are generally instantiated in different EDK II packages.

- **Common** (EDK II) is code that does not have any direct HW requirements other than the basics required to execute machine code on the processor (stack, memory, IA registers, etc).
  - **Producer(s): TianoCore.org**
- **Silicon**, also often called hardware code, has some tie to a specific class of physical hardware. Sometimes governed by industry standards, sometimes proprietary. Silicon or hardware code is usually not intended to have multiple implementations for the same hardware.
  - **Producer(s): Silicon vendor**
- **Platform** defines the actions needed to enable a specific platform's capabilities. In this architecture, capabilities are divided into mandatory and advanced features. Mandatory features are enabled in stages prior to Stage VI. Advanced features are enabled in Stage VI and later.
  - **Minimum Platform Producer(s): TianoCore.org**
  - **Advance Feature Producer(s): TianoCore.org, OEM, BIOS vendor**
- **Board** packages contains board specific code for one or more motherboards.
  - **Producer(s):** Device manufacturer, BIOS vendor, Board user

The architecture is designed to support a maintainer ownership model. For example, board developers should not directly modify (fork) the platform, silicon, or common code. More details on conventional usage of the package classifications can be found in supplemental literature from UEFI Forum, TianoCore.org, and others.

For the purposes of this document, the board package is considered an integration point of the firmware image in addition to providing board-specific functionality. The silicon package provides supported silicon initialization support for one or more silicon products. The minimum platform package represents common elements of this architecture that may depend upon board and silicon interfaces. In order to meet the security objectives of this specification, the minimum platform package must not depend upon deprecated EDK II packages. Other packages composed within the firmware solution described in this specification should consist of widely known elements of the EDKII ecosystem from TianoCore.org.

An example of a firmware stack compliant with this specification for three classes of computer systems is shown in the below figure.

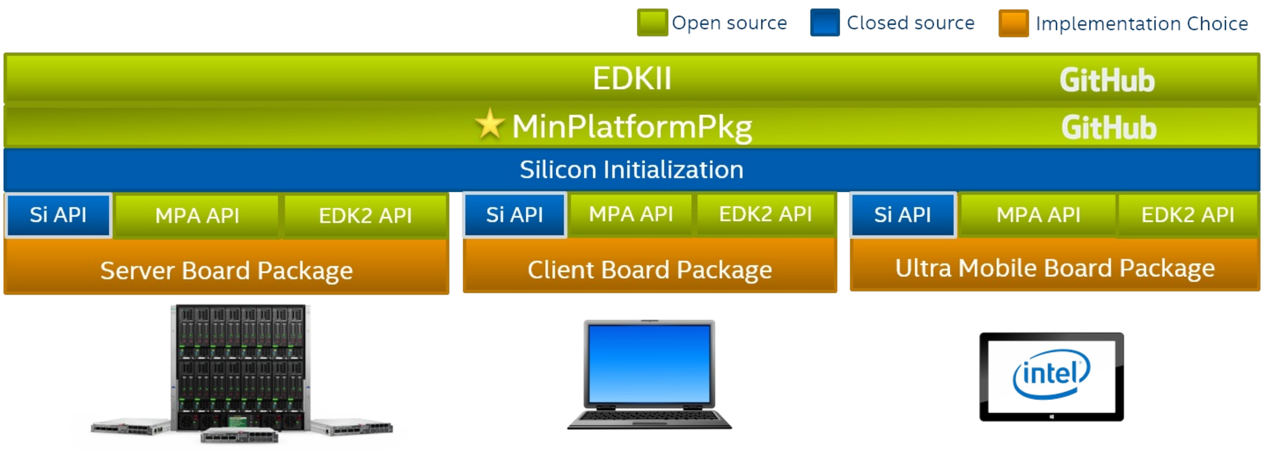


Figure 3 Example of a Minimal Platform Firmware Stack

2.5 Platform and Board Layer

The Minimum Platform Architecture is designed to provide consistency across boot flows with the control flows defined in this specification. These control flows are common to all platforms. Therefore, a single implementation is intended to serve all platforms. This implementation is maintained in the MinPlatformPkg on TianoCore.org. Throughout the standardized boot flow, implementation-specific details are required including board resource information for devices, buses, GPIO settings, etc. In addition, logic is necessary to integrate the silicon solution. For Intel® FSP, this involves invoking the appropriate FSP API with the proper parameters at the proper time in the boot flow. Such details are implemented in the board package behind the board API defined in this specification. This results in flexibility at the board layer for a custom software design that allows substitution of details like silicon initialization flow while maintaining a common control flow with other platforms. The board package is also typically responsible for other implementation-specific integration such as providing a custom build environment that prepares a firmware image processed by the tools required to produce an image compatible for a given board.

3.1 Overview

The objective of Stage I is to provide a foundation for more complex development in later stages. The board should implement a board-specific minimal code path capable of firmware debug output. Basic debug capability serves as a base for development activities in later stages. As Stage I is inherently foundational to product execution it may include more content and complexity than the functionality strictly required for debug output.

3.1.1 Major Execution Activities

- Establish temporary memory.
- Perform pre-memory board-specific initialization.
- Board detection
- GPIOs
- Serial Port initialization (Example: SIO, HSUART)

It is not necessary for the developer to fully configure GPIO at this time. The only required board configuration is that necessary to reach system debug activities.

3.1.2 Main Control Flow

Stage I is contained within SEC and PEI phases. Code must not be compressed and content must be capable of being mapped to memory by hardware or other firmware.

The high level control flow is described in the diagram below.

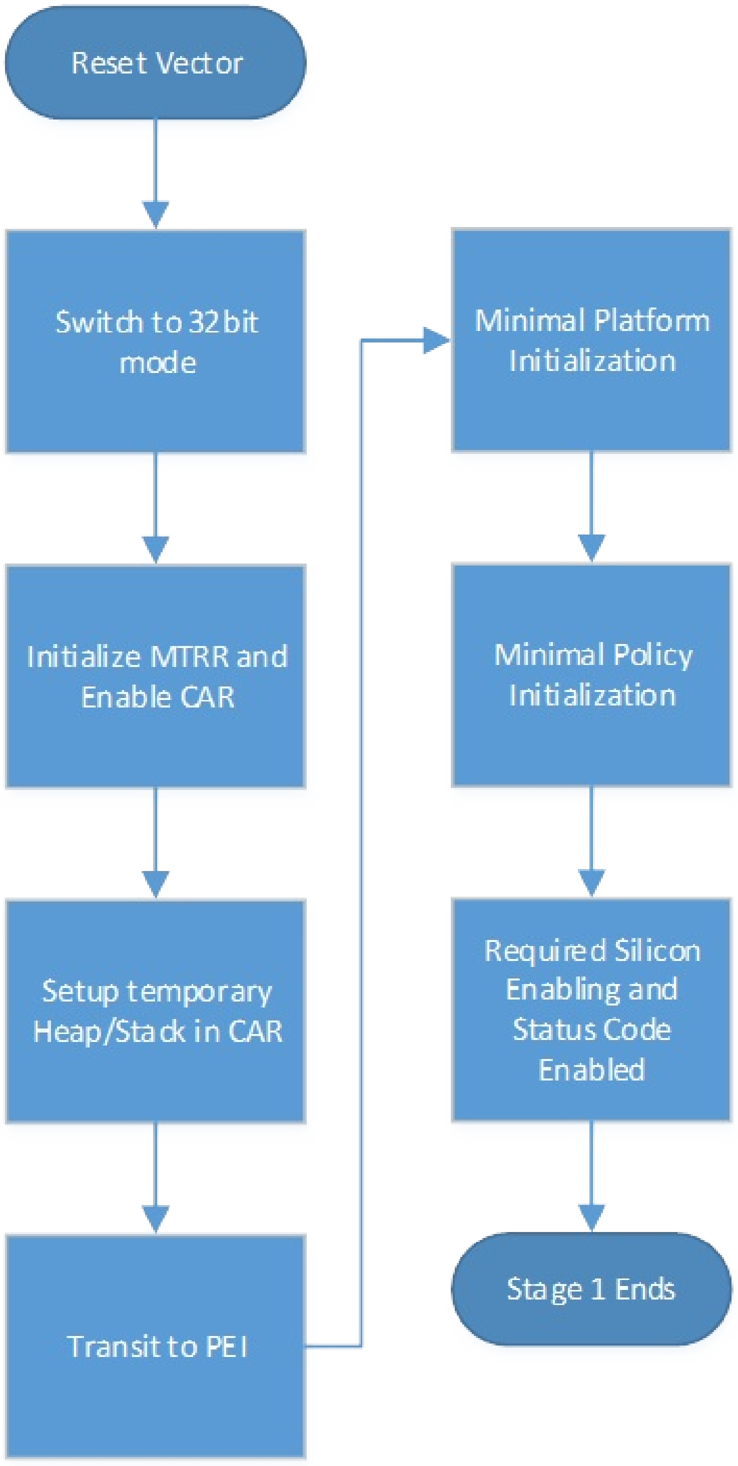

Reset Vector
Switch to 32bit mode
Initialize MTRR and Enable CAR
Setup temporary Heap/Stack in CAR
Transit to PEI
Minimal Platform Initialization
Minimal Policy Initialization
Required Silicon Enabling and Status Code Enabled
Stage 1 Ends

Figure 4 Stage I Main Control Flow

These activities do not map 1:1 to the required functions. Some of this flow is already well defined by UEFI or EDK II specifications. The following details are focused on the Platform, Silicon, and Board interactions, and minimal requirements for structure, consistency, and portability.

3.2 Firmware Volumes

The Stage I functionality is contained in these Firmware Volumes with these attributes.

Name	Content	Compressed	Parent FV
FvPreMemory	SEC + StatusCode	No	None
FvBspPreMemory	Pre-memory board initialization	No	None
FvFspT	SEC silicon initialization	No	None
FvFspM	Memory initialization	No	None
FvPreMemorySilicon	Pre-memory silicon initialization	No	FvFspM
FvFspS	Silicon initialization	No	None
FvPostMemorySilicon	Post-memory silicon initialization	Yes	FvFspS

Table 5 Stage I Firmware Volumes

As the foundational stage for further functionality, Stage I may include additional content beyond what is strictly required to meet the stage objective. Typically this will include silicon initialization code that may be packaged in a variety of mechanisms including varying size binary blobs. In the layout shown in Table 5, the Intel® FSP solution is shown as an example. In this case, the FSP binary can be rebased and integrated in one step rather than distributing the work for the FSP-M and FSP-S rebase unnecessarily across later stages. Note that a child FV is a FV embedded within the parent FV. The child FV is identified by a file type of EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE.

Combining the FVs with full set of silicon binary components yields this example flash map for MMIO storage:

Binary	FV	Components	Purpose
Stage I	FvPreMemory.fv	SecCore.efi	• Reset Vector • Passes PEI core the address of FvFspM • Passes PEI core the debug configuration
		ReportFvPei.efi	• Installs firmware volumes
		SiliconPolicyPeiPreMemory.efi	• Publishes silicon initialization configuration
		PlatformInitPreMemory.efi	• Performs pre memory initialization
		Additional Components	• Additional pre-memory components required for Stage I boot
	FvBspPreMemory.fv	Additional Components	• Advanced pre-memory board support components
	FvFspT.fv	PlatformSec.efi	• Initializes T-RAM silicon functionality
			• Tests T-RAM functionality

		Additional Components	
	FvFspM.fv	PeiCore.efi	• PEI services and dispatcher
		PcdPeim.efi	• PCD service
		FspPlatform.efi	• Converts UPD to Policy PPI
		FvPreMemorySilicon.fv	
		(child FV)	
		Additional Components	• Pre-memory silicon initialization components
		ReportStatusCodeRouterPei.efi	• Provide status code infrastructure
		StatusCodeHandlerPei.efi	• Provide status code listeners
		Additional Components	
	FvFspS.fv	FvPostMemorySilicon.fv	
		(child FV)	
		Additional Components	• Post-memory silicon initialization components
		Additional components	

Table 6 Stage I FV and Component Layout

Note that many of the components included above do not actually participate in producing Stage I functionality, and will not be executed when the boot stage target is set to Stage I. For systems that use non-volatile storage technology that does not provide memory map capabilities, this layout may be modified where necessary. However, the firmware execution path must remain scoped to only perform actions required to achieve the boot stage objective.

See Appendix: Full FV Map for a more complete example Firmware Volume layout.

3.3 Modules

The architecture requires the following modules. Only modules found in the BoardPkg should be modified for board porting. MinPlatformPkg and other common package contents must not be directly modified. BoardPkg and SiliconPkg modules will have multiple instances to support different boards and different silicon.

3.3.1 UEFI Components

These components are required. They enable orderly board porting and add the support for extensibility in later stages. The libraries consumed are the subset of libraries required by this specification. Some libraries are defined in this specification, and some are defined in EDK II documentation.

Item	Producing Package	Libraries Consumed
SecCore.efi	UefiCpuPkg	PlatformSecLib, SerialPortLib
PeiCore.efi	MdeModulePkg	
PcdPeim.efi	MdeModulePkg	
ReportStatusCodeRouterPei.efi	MdeModulePkg	
StatusCodeHandlerPei.efi	MdeModulePkg	SerialPortLib
ReportFvPei.efi	MinPlatformPkg	ReportFvLib, TestPointCheckLib
SiliconPolicyPeiPreMemory.efi	MinPlatformPkg	SiliconPolicyInitLib,
		SiliconPolicyUpdateLib
PlatformInitPreMemory.efi	MinPlatformPkg	BoardInitLib, TestPointCheckLib

Table 7 Stage I UEFI Components Platform Architecture Libraries

3.3.2 Platform Architecture Libraries

Board porting will require creation of libraries identified as produced by the BoardPkg. Depending on the board, there may be existing libraries that are sufficient for a board, so it is important to assess the utility of existing library instances when developing board support.

Item	API Definition Package	Producing Package	Description
BoardInitLib	MinPlatformPkg	BoardPkg	Board initialization library.
ReportFvLib	MinPlatformPkg	MinPlatformPkg	Installs platform firmware volumes.
SerialPortLib	MdeModulePkg	BoardPkg	SIO vendor specific initialization to enable serial port.
SiliconPolicyInitLib	IntelSiliconPkg	SiliconPkg	Provides default silicon configuration policy data.
SiliconPolicyUpdateLib	IntelSiliconPkg	BoardPkg	Provides board updates to silicon configuration policy data.
PlatformSecLib	UefiCpuPkg	MinPlatformPkg	Reset vector and SEC initialization code.
TestPointCheckLib	MinPlatformPkg	MinPlatformPkg	Test point check library. It is called by PlatformInit module to perform stage-specific checks.

TestPointLib	MinPlatformPkg	MinPlatformPkg	Test point library. It provides helper functionality for TestPointCheck lib.

Table 8 Stage I Libraries

3.4 Required Functions

The following functions are required to exist and to execute in the listed order. The component that provides the function is not specified because it is not required by the architecture.

* In the common EDK II open source code.

3.4.1 Required SEC functions

Name	Purpose
ResetHandler (*)	The reset vector invoked by silicon
TempRamInit	Silicon initializes temporary memory
TestPointTempMemoryFunction	Test temporary memory functionality
SecStartup (*)	First C code execution, constructs PEI input
TestPointEndOfSec	Verify state before switching to PEI

Table 9 Stage I SEC Functions

3.4.2 Required PEI functions

Name	Purpose
PeiCore (*)	PEI entry point
PeiDispatcher (*)	Calls the entry points of PEIM
ReportPreMemFv	Installs firmware volumes required in pre-memory
BoardDetect	Board detection of the motherboard type
BoardDebugInit	Board specific initialization for debug device
PlatformHookSerialPortInitialize	Board serial port initialization. Called from SEC or PEI
TestPointDebugInitDone	Verify debug functionality

Table 10 Stage I PEI Functions

3.5 Configuration

This section defines the configurable items that must be available to achieve Stage I functionality.

Stage I configuration is largely concerned with hard-coded address space data and serial port data. In some platform architectures, a firmware or ROM may be responsible for handing off NEM configuration data to Stage I.

3.5.1 Flash Map Configuration

PCD	Purpose
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvPreMemoryBase	Pre-Memory FV base address.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvPreMemorySize	Pre-Memory FV size.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvFspTBase	Fsp-T FV base address.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvFspTSize	Fsp-T FV size.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvFspMBase	Fsp-M FV base address.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvFspMSize	Fsp-M FV size.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvFspSBase	Fsp-S FV base address.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvFspSSize	Fsp-S FV size.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvMicrocodeBase	Microcode FV base address.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvMicrocodeSize	Microcode FV size.

Table 11 Stage I Flash Map Configuration PCDs

3.6 Data Flows

This section defines key data structures and the ways this data flows through the system over time.

3.6.1 Serial Port Configuration

Serial port configuration spans build and boot. Serial port parameters come from the board and are used for debug features, serial input/output devices supporting local or remote consoles, and OS level debuggers.

1. Serial port default configuration options are defined in the MdePkg.dec.
2. Serial port configuration options may be overwritten by the BoardPkg.dsc.
3. Serial port configuration options are consumed by the SerialPortLib library class implementation.
4. SerialPortLib library class is used by the StatusCodeHandlerPei.inf component to initialize and display messages to a serial port.
5. Serial port configuration options are published via a SERIAL_PORT_CONFIGURATION_HOB.
6. SERIAL_PORT_CONFIGURATION_HOB is consumed by MinPlatformSerialDxe.inf to produce EFI_SERIAL_IO_PROTOCOL.

3.6.2 Debug Configuration

PCD	Purpose
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialBaudRate	Baud rate for the 16550 serial port
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialUseMmio	Enable serial port MMIO addressing
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialUseHardwareFlowControl	Enable serial port HW flow control
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialDetectCable	Enable blocking Tx if no cable
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialRegisterBase	Register the serial port base address
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialLineControl	Serial port line control configuration
gEfiMdeModulePkgTokenSpaceGuid.PcdSerialFifoControl	Serial port FIFO control
gMinPlatformPkgTokenSpaceGuid.PcdSecSerialPortDebugEnable	Enable serial port debug in SEC phase
gEfiMdePkgTokenSpaceGuid.PcdFixedDebugPrintErrorLevel	Control build time optimization based on debug print level
gEfiMdePkgTokenSpaceGuid.PcdDebugPropertyMask	Control DebugLib behavior
gEfiMdePkgTokenSpaceGuid.PcdDebugPrintErrorLevel	Control run time debug print level
gEfiMdePkgTokenSpaceGuid.PcdReportStatusCodePropertyMask	Control display of status codes

Table 12 Stage I Debug Configuration

3.7 Additional Control Flows

None

3.8 Build Files

UEFI system firmware by nature is often constructed with a large number of modules and components. EDK II modules are typically written to be generic and reusable. As such, much of the build file content is the same for all platforms. The platform architecture provides further structure and consistency by defining dedicated build files that are exposed to all consumers of the platform package. The modular separation of the build files is based on the UEFI PI Architecture phases, the platform architecture stages, and optional features. The board package is ultimately able to leverage as much of this content as applicable for a given system. The build is coordinated by the board package which should include standard build files from the minimum platform package or other dependent packages such as a silicon package.

`Name`	**`Consumer`**	**`Standalone Buildable`**	**`FV Produced`**	**`Comments`**
MinPlatformPkg \Include\CoreCommonLib.dsc	Board	No	None	Stage I-V common libraries
MinPlatformPkg \Include\CorePeiInclude.dsc	Board	No	None	Combination of Stage I-V that will be processed by compilation building
MinPlatformPkg \Include\CorePeiLib.dsc	Board	No	None	Combination of Stage I-V that will be processed by compilation building
BoardPkg \BoardName\BoardPkg.dsc	Build	Yes	None	Primary build file.
`Name`	`Consumer`	`Standalone Buildable`	`FV Produced`	`Comments`
MinPlatformPkg \Include\CorePreMemoryInclude.fdf	Board	No	None	Combination of Stage I-II that will be processed by compilation building
BoardPkg \BoardName\BoardPkg.fdf	Build	Yes	Yes	Combination of Stage I-V that will be processed by compilation building

Table 13 Stage I Build Files

3.9 Test Point Results

Test Point Function	Test Subject	Test Overview	Reporting Mechanism
TestPointTempMemoryFunction ()	Temporary Memory	Reads/writes results on the stack and heap	Reported through PPI
TestPointDebugInitDone ()	Debug Capability	Dumps a struct of debug configuration parameters to the log	Serial Port shows debug log Port 80 shows number

Table 14 Stage I Test Point Results

3.10 Functional Exit Criteria

1. Temporary Memory is initialized.
2. The debug device is initialized and the platform has written a message to indicate Stage I termination.

3.11 Stage Enabling Checklist

The following steps should be followed to enable a board for Stage I.

1. Copy the EDK II packages locally to the workspace.
2. Select an appropriate silicon initialization solution such as Intel® FSP.
3. Review the requirements for the silicon initialization solution.
4. Gather the silicon initialization requirements for the given board.
5. Customize the silicon initialization solution to configure the system to the board-specific requirements.
   i. For Intel® FSP, this includes setting minimal policy configuration.
   ii. For other silicon solutions, similar parameter customization may be needed if the silicon solution is not written for a particular system design.
6. Determine other firmware and software components required for the system to function properly.
   i. For an Intel platform, this may include firmware images such as the appropriate microcode patch, EC firmware image, power management controller firmware, and others.
   ii. Additional third-party add-in components such as specific UEFI drivers may also be required.
7. Determine board-specific information required to fetch code and show debug output.
   i. This includes details such as the NVRAM layout and strap information.
8. Copy a reference GenerationOpenBoardPkg/BoardXXX and update the board interfaces in Required Functions.
   i. Add serial port initialization code in `PlatformHookSerialPortInitialize ()` at BoardPkg/Library/BasePlatformHookLib.
      i. This is SIO related code.
   ii. Add Board detection code in `BoardDetect ()`,

BoardPkg/BoardInitLib/PeiBoardXXXInitPreMemoryLib.c`

1. If the project only supports one board, this function can return directly.
2. Add Board pre-memory debug initialization code in `BoardDebugInit ()`, BoardPkg/BoardInitLib/PeiBoardXXXInitPreMemoryLib.c.
   i. This is for debug channel related initialization.
3. Audit BoardPkg/Stage1.dsc, ensure all PCDs in the Configuration section are correct for your board.
   i. Set `gMinPlatformPkgTokenSpaceGuid.PcdBootStage` = 1
   ii. Follow "Debug Lib Selection" to enable serial debug capability.
4. Audit all other PCD settings in the board DSC file to verify the values are correct for your board.
5. Verify the flash layout is compliant with this specification.
6. Verify the required binaries will be included in the image produced in the build.
7. Verify the code execution path for Stage I will only perform the actions required to achieve debug output.
8. Boot the system, collect the debug log, and verify the test point results defined in the Test Point section are correct.

4.1 Overview

The objective of Stage II is to enable a minimal boot path memory initialization code execution that successfully installs permanent memory.

4.1.1 Major Execution Activities

- Complete execution of the memory initialization module.
- Discover, train and install permanent memory.
- Migrate the temporary memory/stack to permanent memory.
- Migrate any code modules from temporary RAM to permanent memory.
- Perform cache configuration for a post-memory environment.
- Execute memory installed notification actions.

`Stage II Functionality`
Non-volatile storage read-only access
Pre-memory silicon policy initialization
Basic services like cache and CPU IO
Initialization of decompression capability
Memory initialization and basic memory test

4.1.2 Main Control Flow

Stage II extends the Stage I control flow by executing the platform and silicon initialization required for memory initialization. The stage is completed when permanent memory is installed. Since execution prior to memory initialization typically occurs in a resource-constrained environment, the code in this stage is not compressed. To simplify silicon enabling which may be opaque to the board engineer in the form of a binary blob, Stage II enabling does not strictly constrain the extent of silicon initialization. In particular, it is recommended to perform standard security lock functionality such as register locks, privilege level changes, and other actions that are in the system requirements to reduce conditional logic and therefore potential for error in enabling those settings. This only pertains to security settings within the chipset. This does not include larger industry standard security features such as UEFI Secure Boot and TCG measured boot. Those features are enabled in Stage V Security Enable.

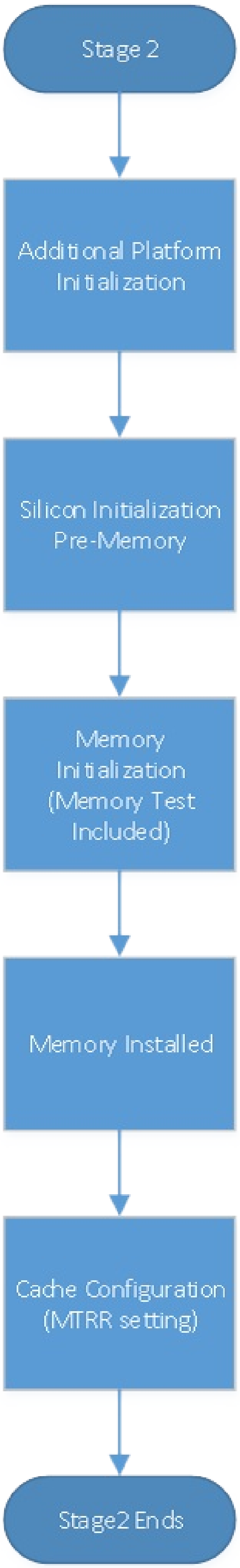

Stage 2
Additional Platform Initialization
Silicon Initialization Pre-Memory
Memory Initialization (Memory Test Included)
Memory Installed
Cache Configuration (MTRR setting)
Stage2 Ends

Figure 5 Stage II Main Control Flow

4.2 Firmware Volumes

Stage II leverages most of the Stage I content. Additional firmware volumes include:

Name	Content	Compressed	Parent FV
FvPostMemory	Post-memory modules	Yes	None
FvBsp.fv	Post-memory board support	Yes	None

Table 15 Stage II Firmware Volumes

Which yields this example extension of the flash map for MMIO storage (append to Stage I map):

Binary	FV	Components	Purpose
Stage II	FvPostMemory.fv	ReadOnlyVariable.efi	Common core variable services
		SiliconPolicyPeiPostMemory.efi	Publishes silicon initialization configuration
		PlatformInitPostMemory.efi	Performs post memory initialization
		DxeIpl.efi	Load and invoke DXE
		ResetSystemRuntimeDxe.efi	Provides reset service
		PciHostBridge.efi	PCI host bridge driver
		Additional Components	Additional post-memory components required for Stage II boot
	FvBsp.fv	Additional Components	Post-memory board support components

Table 16 Stage II FV and Component Layout

See Appendix: Full FV Map for a more complete example Firmware Volume layout.

4.3 Modules

Only modules in the board package should be modified in the process of board porting. The minimum platform package and other common package contents must not be directly modified. The board package and silicon package modules may have multiple instances to support different boards and different silicon. These components are required. They enable orderly board porting and add the support for extensibility in later stages. The libraries consumed are the subset of libraries required by this specification. Some libraries are defined in this specification, some are defined in EDK II documentation.

4.3.1 UEFI Components (DXE)

Item	Producing Package	Libraries Consumed
DxeIpl.efi	MdeModulePkg	
SiliconPolicyPeiPostMemory.efi	MinPlatformPkg	SiliconPolicyInitLib SiliconPolicyUpdateLib
PlatformInitPostMemory.efi	MinPlatformPkg	BoardInitLib TestPointCheckLib
ResetSystemRuntimeDxe.efi	MdeModulePkg	ResetSystemLib
PciHostBridge.efi	MdeModulePkg	PciHostBridgeLib

Table 17 Stage II DXE UEFI Components

4.3.2 Platform Architecture Libraries (PEI)

Item	API Definition Package	Producing Package	Description
BoardInitLib	MinPlatformPkg	BoardPkg	Board initialization library.
SiliconPolicyInitLib	IntelSiliconPkg	SiliconPkg	Provides default silicon configuration policy data.
SiliconPolicyUpdateLib	IntelSiliconPkg	BoardPkg	Provides board updates to silicon configuration policy data.
TestPointCheckLib	MinPlatformPkg	MinPlatformPkg	Test point check library. It is called by PlatformInit module to perform stage-specific checks.
TestPointLib	MinPlatformPkg	MinPlatformPkg	Test point library. It provides helper functionality for TestPointCheck lib.

Table 18 Stage II PEI Platform Architecture Libraries

4.3.3. Platform Architecture Libraries (DXE)

Stage II contains some DXE items needed to enable Stage III. No board porting of these libraries is required. Board integrators should ensure that their silicon package provides the necessary libraries. These libraries and the UEFI Components (DXE) are functionally irrelevant to Stage II functionality.

Item	API Definition Package	Producing Package	Description
ResetSystemLib	MdeModulePkg	SiliconPkg	For DXE reset architecture protocol
PciHostBridgeLib	MdeModulePkg	SiliconPkg	For DXE PCI host bridge driver

Table 19 Stage II DXE Platform Architecture Libraries

4.4 Required Functions

The following functions are required to exist and to execute in the listed order. The component that provides the function is not specified because it is not required by the architecture.

4.4.1 Required PEI functions

* In the common EDK II open source code.

Name	Purpose
BoardBootModeDetect	Board hook for EFI_BOOT_MODE detection
BoardInitBeforeMemoryInit	Board specific initialization prior to permanent memory initialization (e.g. GPIO configuration)
SiliconPolicyInitPreMemory	Pre-memory silicon policy default initialization
SiliconPolicyUpdatePreMemory	Pre-memory silicon policy update logic
SiliconPolicyDonePreMemory	Opportunity to implement a board-specific indicator that silicon policy initialization is done
MemoryInit	Permanent memory initialization
InstallEfiMemory	Install permanent memory to core
PeiCore (*)	PEI entry point (post-memory entry)
SecTemporaryRamDone (*)	Optional API defined in the PI specification to disabled temporary RAM
ReportPostMemFv	Firmware volume installation in post-memory
TestPointPostMemoryFvInfoFunctional	Test to verify correctness of the firmware volume map in post-memory
BoardInitAfterMemoryInit	Board initialization after memory is installed
SetCacheMtrr	Configuration of MTRR settings for post-memory
TestPointPostMemoryMtrrAfterMemoryDiscoveredFunctional	Test to verify the correctness of the MTRR settings in post-memory
TestPointPostMemoryResourceFunctional	Test to verify correctness of permanent memory

Table 20 Stage II PEI Functions

4.4.2 Interfaces

* In the common EDK II open source code.

Component	Name	Consumer	Purpose
BoardInitLib	BoardBootModeDetect	Platform	Board-specific boot mode

BoardInitLib	BoardBootModeDetect	Platform	detection
	BoardInitAfterMemoryInit	Platform	Board specific initialization after memory initialization
	BoardInitBeforeTempRamExit	Platform	Board specific hook before temporary RAM exit
	BoardInitAfterTempRamExit	Platform	Board specific hook after temporary RAM exit
SiliconPolicyInitLib	SiliconPolicyInitPreMemory	Platform	Initialize silicon policy default values
	SiliconPolicyDonePreMemory	Platform	Platform-specific behavior to indicate the policy update is done
SiliconPolicyUpdateLib	SiliconPolicyUpdatePreMemory	Platform	Board updates default policy

Table 21 Stage II Interfaces

4.5 Configuration

This section defines the configurable items that must be available to achieve Stage II functionality.

4.5.1 Intel® FSP Related Configuration

PCD	Purpose
gIntelFsp2WrapperTokenSpaceGuid.PcdFspmBaseAddress	FSP-M FV base address
gIntelFsp2WrapperTokenSpaceGuid.PcdPeiMinMemorySize	Indicates the PEI memory size reported by the platform
gIntelFsp2WrapperTokenSpaceGuid.PcdPeiRecoveryMinMemorySize	Indicates the PEI recovery memory size reported by the platform

Table 22 Stage II FSP Related Configuration

4.5.2 FV Related Configuration

PCD	Purpose
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvPostMemoryBase	Post-memory FV base address
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvPostMemorySize	Post-memory FV size

Table 23 Stage II FV Related Configuration

4.6 Data Flows

This section defines architectural data structures and the flows in which these structures move through the boot over time.

4.6.1 Silicon Policy Flow and Rules

4.6.1.1 Silicon Module Provides Default Silicon Policy Data

One silicon policy data structure is created per silicon module. The data structure should only contain data. Functions should not be used in silicon policy data.

When a silicon module installs this policy data, it should consider the most common usage as the default policy data. Therefore, a board module must only update non-default values instead of all fields.

This silicon code may expose the APIs below.

- An API to initialize all policy data to the default value, based upon the current silicon.
- An API to tell silicon code that all policy data have been updated, and they are ready to consume.

`Library`	`Interface`	`Definition Location`	`Producer`	`Consumer in FSP Boot Path`	`Consumer in EDK II Path`
SiliconPolicyInitLib	SiliconPolicy InitPreMemory	Silicon	Silicon	FspWrapper PlatformLib	Platform
	SiliconPolicy DonePreMemory	Silicon	Silicon	FspWrapper PlatformLib	Platform
	SiliconPolicy InitPostMemory	Silicon	Silicon	FspWrapper PlatformLib	Platform
	SiliconPolicy DonePostMemory	Silicon	Silicon	FspWrapper PlatformLib	Platform
	SiliconPolicy InitLate	Silicon	Silicon	FspWrapper PlatformLib	Platform
	SiliconPolicy DoneLate	Silicon	Silicon	FspWrapper PlatformLib	Platform
SiliconPolicyUpdateLib	SiliconPolicy UpdatePreMemory	Minimum Platform / Silicon	Board	FspWrapper PlatformLib	Platform
	SiliconPolicy Update PostMemory	Minimum Platform / Silicon	Board	FspWrapper PlatformLib	Platform
	SiliconPolicy UpdateLate	Minimum Platform / Silicon	Board	FspWrapper PlatformLib	Platform

Table 24 Silicon Policy Libraries

NOTE: A general guideline is that pointers should not be used in policies that persist across the CAR to permanent memory boundary as the pointer addresses will become invalid. Pre-memory only and post-memory only policies are not affected by the memory transition. A pre-memory policy installation PEIM should only be used for policies that must be updated in pre-memory for early use by silicon code. The post-memory policy installation PEIM can be compressed and does risk pointers becoming invalidated due to the memory transition.

The policy configuration HOBs are inherently passed to DXE, so a silicon DXE module can locate and install a policy protocol for a particular policy if it is used in DXE.

4.6.1.2 Board Module Silicon Policy Data

Using the SiliconPolicyUpdateLib, the board package may reference a variety of sources to obtain the board-specific policy values, including but limited to the following common sources.

1. PCD database
2. UEFI Variable
3. Binary Blob
4. Built-in C structure
5. Hardware information

4.6.1.3 Board Module Silicon Policy Update Completion

After policy configuration is completed, the board may indicate that the policy is configured with board-specific actions in the SiliconPolicyDonePreMemory ( ) API in SiliconPolicyInitLib.

4.6.1.4 Non-Intel® FSP Policy Data Flow

The SiliconPolicyPeiPreMemory module in the minimum platform package will invoke the following policy configuration functions in the given order.

- BoardPreMemoryInit ()
  - SiliconPolicyInitPreMemory ()
  - SiliconPolicyUpdatePreMemory ()
    - Minimum platform: Minor update of relevant options
    - Fully featured platform (Stage VI and greater): Full update for all platform features
  - SiliconPolicyDonePreMemory ()

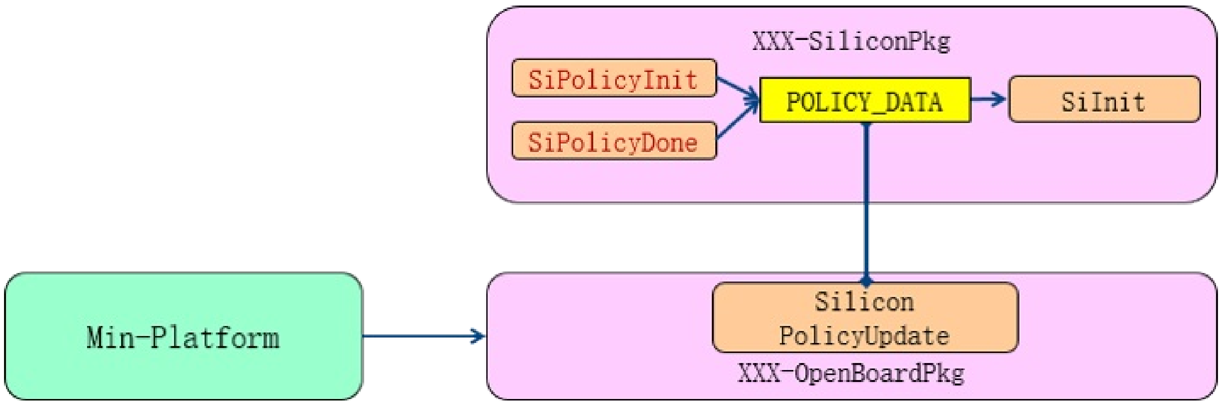


Figure 6 Non-FSP Policy Data Flow

4.6.1.5 Intel® FSP Policy Data Flow

- UpdateFspmUpdData (Upd)
- SiliconPolicyInitPreMemory ()
- SiliconPolicyUpdatePreMemory ()
- Minimum platform: Minor update of relevant options
- Fully featured platform (Stage VI and greater): Full update for all platform features
- SiliconPolicyDonePreMemory ()

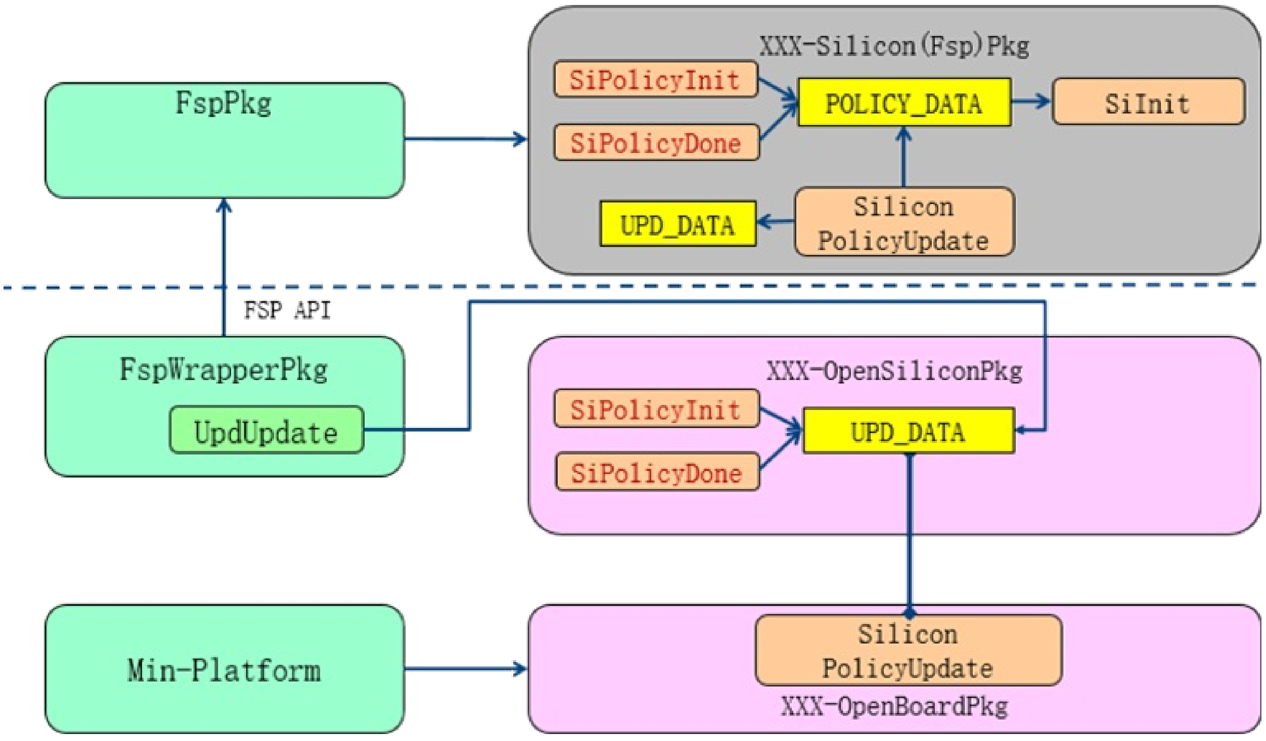


Figure 7 FSP Policy Data Flow

4.6.2 HOB Output

1. Intel® FSP HOB to PEI HOB transition

   In an Intel® FSP API mode boot with an EDK II wrapper, the system will have two HOB lists - one maintained in the FSP PEI environment and the other in the FSP wrapper PEI environment. The FSP wrapper environment is responsible for converting data from the FSP HOB to a PEI HOB. These HOBs are platform-specific; examples include the SmbiosHob and GraphicInfoHob.

2. PEI HOBs for Phase Handoff to DXE

4.6.3 MTRR Configuration Settings in Post-Memory

The system MTRR settings are typically configured in two locations after permanent memory initialization.

1. After permanent memory installation

   At this point, cache attributes are set for PEI memory usage. This specification does not require any particular MTRR configuration, as it is ultimately dependent upon platform goals such as functionality and performance given the device and storage technologies present on the platform. The most common ranges configured are the default memory setting as UC, the DRAM region as WB, and the SPI flash MMIO region as WP. These settings are usually applied in a memory installation notification function or a PEIM shadow. The architecture requires that the settings be applied in the board package.

2. Prior to DXE IPL

At this point PEI execution has completed and control is transitioning to the DXE phase. The MTRR settings are typically modified to prepare for the DXE environment. The most common ranges configured are the default memory as WB, the TSEG (SMRAM) region as UC, and MMIO as UC. These settings are usually applied in an end of PEI notification function. The architecture requires that the settings be applied in the board package.

4.7 Additional Control Flows

None

4.8 Build Files

This is appended to previous Build Files section.

Name	Consumer	Standalone Buildable	FV Produced	Comments
MinPlatformPkg \Include\CorePostMemoryInclude.fdf	Board	No	None	

Table 25 Stage II Build Files

4.9 Test Point Results

`Test Point`	`Test Subject`	`Test Overview`	`Reporting Mechanism`
TestPointMemory DiscoveredMtrr Functional ()	MTRR after memory discovered	Verifies MTRR settings. (No overlap, PEI memory WB, Flash region is WP, MMIO UC)	Dump result to serial log. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPointMemory DiscoveredMemory Resource Functional ()	Resource description HOB	No memory resource overlap.	Dump result to serial log. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPointMemory DiscoveredFvInfo Functional ()	FV HOB and FV info PPI	FV HOB and FV info PPI.	Dump result to serial log. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT

Table 26 Test Point Results

NOTE: ADAPTER_INFO_PLATFORM_TEST_POINT_STRUCT can be updated by TestPointCheckLib. The format is similar to the HSTI. See Appendix - Interface TestPoint.

4.10 Functional Exit Criteria

1. Permanent memory is initialized.
2. Temporary memory is disabled.
3. PEI phase MTRR configuration settings are applied.
4. Resource description HOB is built.

4.11 Stage Enabling Checklist

The following steps should be followed to enable a platform for Stage II.

1. Update GenerationOpenBoardPkg/BoardXXX
   i. Add Board boot mode detection code in `BoardBootModeDetect ()`, BoardXXX/BoardInitLib/PeiBoardXXXInitPreMemoryLib.c.
      i. The boot mode can be hardcoded. It should reflect actual functionality based upon the feature, such as S3 (silicon register), Capsule (variable), Recovery (GPIO).
   ii. Add Board pre-memory initialization code in `BoardInitBeforeMemoryInit ()` and `BoardInitAfterMemoryInit ()`, BoardXXX/BoardInitLib/PeiBoardXXXInitPreMemLib.c.
      i. It initializes board specific hardware devices, such as GPIO.
      ii. It also updates pre-memory policy configuration by using PCD
   iii. Add Board policy update code in `SiliconPolicyUpdatePreMemory ()`, BoardXXX/PeiSiliconPolicyUpdateLib/PeiBoardXXXInitPreMemoryLib.c.
      i. The PCD updated in `BoardInitBeforeMemoryInit ()` might be used here.
2. Ensure all PCDs in the configuration section (DSC files) are correct for your board.
   i. Set `gMinPlatformPkgTokenSpaceGuid.PcdBootStage` = 2
3. Ensure all required binaries in the flash file (FDF files) are correct for your board.
4. Boot, collect log, verify test point results defined in section 4.9 are correct.

5.1 Overview

The primary objective of Stage III is to enable a minimal boot path that successfully loads the UEFI Shell. A secondary objective for Stage III is to be silicon and board agnostic. All silicon and board specific details should be leveraged from Stages I and II. Demonstrating the capability to load the UEFI shell does not imply that the UEFI shell is a required component in the end product firmware. It does ensure that the platforms with a terminal boot stage target greater than Stage II can load the UEFI shell so the system can be analyzed and configured in the UEFI boot services environment with well-defined behavior in a consistent manner with other Minimum Platform specification-compliant systems.

The minimal UI capability that is required is serial console. UEFI variables must be supported with at least emulated variable behavior. UEFI variable storage to a non-volatile media such as SPI NOR flash is acceptable if the platform requirements mandate such support. Additional capabilities are optional and must not be assumed. These include USB input devices, graphics devices, and other storage devices.

5.1.1 Major Execution Activities

- DXE Initial Program Load (IPL)
- DXE Core initialization and dispatcher execution
- Initialize the generic infrastructure required for the DXE environment o Installation of DXE architectural protocols o Initialization of architecturally required hardware such as timers
- Post-memory silicon policy initialization
- Serial console input and output capabilities

`Stage III Functionality`
Universally usable infrastructure: DXE Core, Minimal BDS, console infrastructure
Silicon agnostic architectural protocol producing hardware modules
UEFI Variable support (emulation allowed)
UEFI Shell
Tests for Memory Map, Cache Map, architectural hardware

5.1.2 Main Control Flow

Stage III extends Stage II control flow by executing Driver Execution Environment (DXE), executing Boot Device Selection (BDS) and invoking the UEFI Shell.

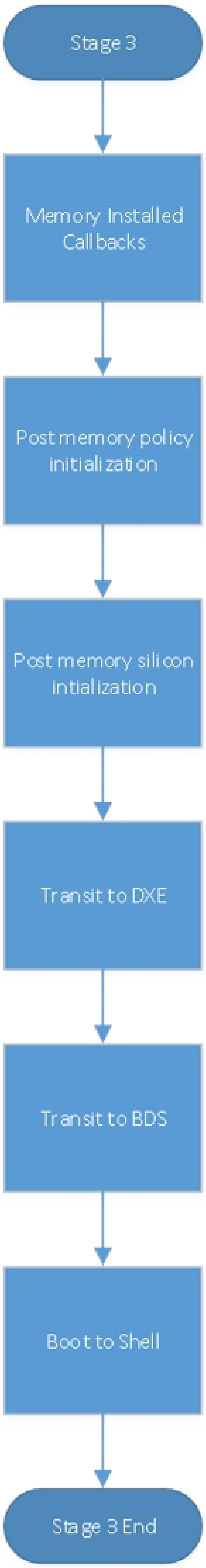

Stage 3
Memory Installed Callbacks
Post memory policy initialization
Post memory silicon intialization
Transit to DXE
Transit to BDS
Boot to Shell
Stage 3 End

Figure 8 Stage III Main Control Flow

After memory is installed during Stage II, the remaining silicon and platform initialization must take place in the PEI phase only. All silicon initialization tasks should have been completed in Stage II, and there should be no silicon-specific initialization required in the DXE phase. The default console information should be transferred via a HOB and initialized and used in Stage III.

5.2 Firmware Volumes

Stage III finalizes silicon and prepares DXE/BDS services. Additional firmware volumes include:

Name	Content	Compressed	Parent FV
FvUefiBoot	Common DXE/BDS Services	Yes	None

Table 27 Stage III Firmware Volumes

Which yields this example extension of the flash map for MMIO storage (add to Stage I + II map):

Binary	FV	Components	Purpose
Stage III	FvUefiBoot.fv	DxeCore.efi	DXE services and dispatcher
		PcdDxe.efi	Provides PCD services
		ReportStatusCodeRouterDxe.efi	Provides status code infrastructure
		StatusCodeHandlerRuntimeDxe.efi	Provides status code listeners
		BdsDxe.efi	Provides Boot Device Selection phase
		CpuDxe.efi	Provides processor services
		Metronome.efi	Provides metronome HW abstraction
		MonotonicCounterRuntimeDxe.efi	Provides monotonic counter service
		PcatRealTimeClockRuntimeDxe.efi	Provides RTC abstraction
		WatchdogTimer.efi	Provides watchdog timer service
		RuntimeDxe.efi	Provides UEFI runtime service functionality
		HpetTimerDxe.efi	Provide timer service
		EmuVariableRuntimeDxe.efi	Provides UEFI variable service
		CapsuleRuntimeDxe.efi	Provides capsule service
		PciBusDxe.efi	PCI bus driver
		GraphicsOutputDxe.efi	Provides graphics support
		TerminalDxe.efi	Provides terminal services
		GraphicsConsoleDxe.efi	Provides graphics console
		ConSplitterDxe.efi	Provides multi console support
		EnglishDxe.efi	Provides Unicode collation services
		GenericMemoryTestDxe.efi	Provide memory test
		DevicePathDxe.efi	Provides device path services
		DiskIo.efi	Provides disk IO services
		Partition.efi	Provides disk partition services
		Fat.efi	Provides FAT filesystem services

		Additional Components	Additional post-memory components required for Stage III boot

Table 28 Stage III FV and Component Layout

See Appendix: Full FV Map for a more complete example Firmware Volume layout.

5.3 Modules

Only modules in the board package should be modified in the process of board porting. The minimum platform package and other common package contents must not be directly modified. The board package and silicon package modules may have multiple instances to support different boards and different silicon. These components are required. They enable orderly board porting and add the support for extensibility in later stages. The libraries consumed are the subset of libraries required by this specification. Some libraries are defined in this specification, some are defined in EDK II documentation.

5.3.1 UEFI Components (DXE)

Item	Producing Package	Libraries Consumed	Comments
DxeCore.efi	MdeModulePkg		
PcdDxe.efi	MdeModulePkg		
BdsDxe.efi	MdeModulePkg	PlatformBootManagerLib	
CpuDxe.efi	UefiCpuPkg		Architecture Protocol
Metronome.efi	MdeModulePkg		Architecture Protocol
MonotonicCounterRuntimeDxe.efi	MdeModulePkg		Architecture Protocol
PcatRealTimeClockRuntimeDxe.efi	PcAtChipsetPkg		Architecture Protocol
WatchdogTimer.efi	MdeModulePkg		Architecture Protocol
RuntimeDxe.efi	MdeModulePkg		Architecture Protocol
SecurityStubDxe.efi	SecurityPkg		Architecture Protocol
HpetTimerDxe.efi (*)	PcAtChipsetPkg		Architecture Protocol
VariableRuntimeDxe.efi /	MdeModulePkg		Architecture Protocol
VariableSmmRuntimeDxe.efi			
CapsuleRuntimeDxe.efi	MdeModulePkg		Architecture Protocol
PciBusDxe.efi	MdeModulePkg		PCI
TerminalDxe.efi	MdeModulePkg		Terminal
ConSplitterDxe.efi	MdeModulePkg		Console
EnglishDxe.efi	MdeModulePkg		Localization
DevicePathDxe.efi	MdeModulePkg		Other
`Optional drivers`			
GraphicsOutputDxe.efi	MdeModulePkg		Graphics
GraphicsConsoleDxe.efi	MdeModulePkg		Console

MemoryTest.efi	MdeModulePkg		Other
ReportStatusCodeRouterDxe.efi	MdeModulePkg		Status code
StatusCodeHandlerRuntimeDxe.efi	MdeModulePkg	SerialPortLib	Status code

Table 29 Stage III DXE UEFI Components

* An alternative timer module may be used to produce an instance of gEfiTimerArchProtocolGuid.

5.3.2 Platform Architecture Libraries

No board porting of these libraries is required.

`Item`	`API Definition Package`	`Producing Package`	`Description`
SerialPortLib	MdeModulePkg	MinPlatformPkg	Serial port leveraging PEI and HOB initialization.
PlatformBoot ManagerLib	MdeModulePkg	MinPlatformPkg	Basic platform boot manager port.

Table 30 Stage III Platform Architecture Libraries

5.4 Required Functions

The following functions are required to exist and to execute in the listed order. The component that provides the function is not specified because it is not required by the architecture.

5.4.1 Required PEI functions

Name	Purpose
BoardInitBeforeSiliconInit	Board initialization hook
SiliconPolicyInitPostMemory	Silicon post memory policy initialization
SiliconPolicyUpdatePostMemory	Board updates silicon policies
SiliconPolicyDonePostMemory	Complete post memory silicon policy data collection
BoardInitAfterSiliconInit	Board specific initialization after silicon is initialized
DxeLoadCore (*)	DXE IPL locate and call DXE Core
SetCacheMtrrAfterEndOfPei	Sets cache map in preparation for DXE
TestPointEndOfPei	Verify expected state as we exit PEI phase
TestPointPostMemoryMtrrEndOfPeiFunctional	Basic test for cache configuration before entering DXE

Table 31 Stage III Required PEI Functions

* In the common EDK II open source code.

5.4.2 PEI Interfaces

Component	Name	Consumer	Purpose
BoardInitLib	BoardInitBeforeSiliconInit	Platform	Board specific initialization before silicon initialization
	BoardInitAfterSiliconInit	Platform	Board specific initialization after silicon initialization
SiliconPolicyInitLib	SiliconPolicyInitPostMemory	Platform	Silicon provides default policy
	SiliconPolicyDonePostMemory	Platform	Platform to indicate the policy update is done
	SiliconGetPolicySubData	Board	Return policy data for update.
SiliconPolicyUpdateLib	SiliconPolicyUpdatePostMemory	Platform	Board updates default policy

Table 32 Stage III PEI Functions

5.4.3 Required DXE functions

Name	Purpose
DxeMain (*)	DXE entry point

CoreStartImage (*)	DXE driver entry point
SiliconPolicyInitLate	Silicon policy late (DXE) initialization
SiliconPolicyUpdateLate	Silicon policy late (DXE) update from the board package
SiliconPolicyDoneLate	Silicon policy late (DXE) indication policy initialization is done
CoreAllEfiServicesAvailable (*)	Verify all architectural protocols are installed
BdsEntry (*)	BDS entry point
PlatformBootManagerBeforeConsole (*)	Platform-specific BDS functionality before console
BoardInitAfterPciEnumeration	Board-specific hook after PCI enumeration completion
TestPointPciEnumerationDone	Test to verify PCI enumeration assignment
ExitPmAuth	Signal key security events EndOfDxe and SmmReadyToLock
TestPointEndOfDxe	Test to verify expected state after EndOfDxe
TestPointDxeSmmReadyToLock	Test to verify expected state after SmmReadyToLock
EfiBootManagerDispatchDeferredImages (*)	Dispatch deferred third party UEFI driver OPROMs
PlatformBootManagerAfterConsole (*)	Platform specific BDS functionality after console
BootBootOptions (*)	Attempt each boot option
EfiSignalEventReadyToBoot (*)	Signals the ReadyToBoot event group
BoardInitReadyToBoot	Board hook on ReadyToBoot event
TestPointReadyToBoot	Test to verify expected state after ReadyToBoot event signal
UefiMain (*)	UEFI Shell entry point

Table 33 Stage III DXE Functions

5.4.4 DXE Interfaces

Component	Name	Consumer	Purpose
BoardInitLib	BoardNotificationInit	Platform	Board specific initialization hook at DXE phase

Table 34 Stage III DXE Interfaces

5.5 Configuration

This section defines the configurable items that must be available to achieve Stage III functionality.

These definitions may be both source and binary in nature.

5.5.1 FV Related Configuration

PCD	Purpose
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvUefiBootBase	UefiBoot FV base address.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvUefiBootSize	UefiBoot FV size.

Table 35 Stage III Flash Map Configuration PCDs

5.5.2 Driver Related Configuration

PCD	Purpose
gEfiMdeModulePkgTokenSpaceGuid.PcdEmuVariableNvModeEnable	Enables UEFI variable emulation mode.

Table 36 Stage III Driver Configuration

5.6 Data Flows

This section defines the architecturally defined data structures and the ways this data flows through the system over time. In addition to the definition and lifecycle for important pieces of data.

5.6.1 Memory Map Flow

The detailed description on memory map can be found in the whitepaper A Tour Beyond BIOS Memory Map and Practices in UEFI BIOS.

5.7 Additional Control Flows

None

5.8 Build Files

This is appended to previous Build files section.

Name	Consumer	Standalone Buildable	FV Produced	Comments
MinPlatformPkg \Include\CoreUefiBootInclude.fdf	Board	No	None	

Table 37 Stage III Build Files

5.9 Test Point Results

`Test Point`	`Test Subject`	`Test Overview`	`Reporting Mechanism`
TestPoint EndOfPeiMtrr Functional ()	MTRR after EndOfPei	Confirm MTRR settings. Example: • No overlap • DXE memory is WB • MMIO is UC • Flash region is UC	Dump result to serial log. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint EndOfPei SystemResource Functional ()	Resource HOB SMRAM HOB	SMRAM No system resource overlap	Dump result to serial log. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint EndOfPei PciBusMaster Disabled ()	PCI device BME	Check if BME is cleared	Dump result to serial log. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint PciEnumerationDone PciResource Allocated ()	PCI device resource	Check if all PCI devices have been assigned proper resources.	Dump PCI resource assignment. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint PciEnumerationDone PciBusMaster Disabled ()	PCI device BME	Check if BME is cleared	Dump result to serial log. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint EndOfDxe NoThirdParty PciOptionRom ()	3rd party PCI option ROMs	Check if any 3rd party PCI option ROMs have been dispatched before EndOfDxe.	Dump LoadedImage. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot UefiMemoryAttribute TableFunctional ()	UEFI memory attribute table	Table is reported. Image code and data is consistent with the table.	Dump UEFI Table and UEFI Image Info. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot MemoryTypeInformation Functional ()	Memory type information	Inspect and verify memory type information is correct. Confirm no fragmentation exists in the ACPI/Reserved/Runtime memory	Dump the memory type information settings to the debug log. Set

		regions.	ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot UefiConsoleVariable Functional ()	Console	Inspect and verify console variable information is correct.	Dump the variable information to the serial log Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot UefiBootVariable Functional ()	Boot Option	Inspect and verify boot option information is correct.	Dump the variable information to the serial log Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT

Table 38 Stage III Test Point Results

5.10 Functional Exit Criteria

1. UEFI Shell can be loaded and invoked by the platform firmware.
2. The DXE MTRRs are set correctly and verified in the test point results.

5.11 Stage Enabling Checklist

The following steps should be followed to enable a platform for Stage III.

1. Add board post-memory initialization code in `BoardInitBeforeSiliconInit ()` and `BoardInitAfterSiliconInit ()`, BoardPkg/BoardInitLib/PeiBoardXXXInitPostMemoryLib.c.
   i. Initialize board-specific hardware device, such as GPIO.
   ii. Update post-memory policy configuration by using PCD.
2. Add board policy update code in `SiliconPolicyUpdatePostMemory ()`, BoardPkg\PeiSiliconPolicyUpdateLib \PeiBoardXXXInitLib.c.
   i. The PCD updated in `BoardInitBeforeSiliconInit ()` might be used here.
3. Add board initialization DXE code in `BoardInitAfterPciEnumeration ()`, `BoardInitReadyToBoot ()`, `BoardInitEndOfFirmware ()`.
   i. NOTE: The functions may be empty if nothing needs to be updated.
4. Ensure all PCDs in the configuration section (DSC files) are correct for your board.
   i. Set `gMinPlatformPkgTokenSpaceGuid.PcdBootStage` = 2
5. Ensure all required binaries in the flash file (FDF files) are correct for your board.
6. Boot, collect debug log, and verify the test point results defined in section 5.9 are correct.

6.1 Overview

The objective of Stage IV is to enable a minimal boot path that successfully boots a commercial operating system such as Linux or Windows, with UEFI interfaces exposed to the OS implemented in compliance with the UEFI specification. The minimal boot path only involves functionality necessary to load the OS to a state where a user may begin performing more complex interactions. This involves successfully reaching an environment that allows the user to launch applications. The stage does not include support for all applications that, for example, may require certain CPU or GPU features enabled. Nor does it require any further support, including but not limited to device and system power management, full hardware performance support enabled, system reset support, etc.

Any additional functionality is classified as an advanced feature. Those features are collectively enabled in Stage VI.

6.1.1 Major Execution Activities

`Stage IV Modules`
Minimum ACPI table initialization
Additional input, output, and storage support based on platform and operating system requirements
SMM
Perform ACPI enable/disable
Kernel debug support
UEFI variable support

6.1.2 Main Control Flow

Stage IV introduces additional functionality to meet the minimal requirements for a UEFI-compliant operating system. Much of the support required will be performed during the DXE phase interleaving Stage IV control flows with pre-existing control flows from Stage III. A minimum set of ACPI tables, namely RSDT, FACP, FACS, FADT, MADT, HPET and DSDT, need to be initialized and published. If there are alternative and/or additional operating system expectations such as full DeviceTree support, that should be enabled to allow the operating system to be loaded.

It is recommended that only the mandatory boot option devices are connected in BDS to minimize complexity and boot time in the minimal execution path to the operating system. In the flow diagram below, the left half is identical to the functionality enabled by Stage III prior to entering the BDS phase. It is expected that the Stage III components are reused to complete Stage IV tasks.

The green blocks in Figure 9 Stage IV Control Flow reuse the existing blocks from Stage III.

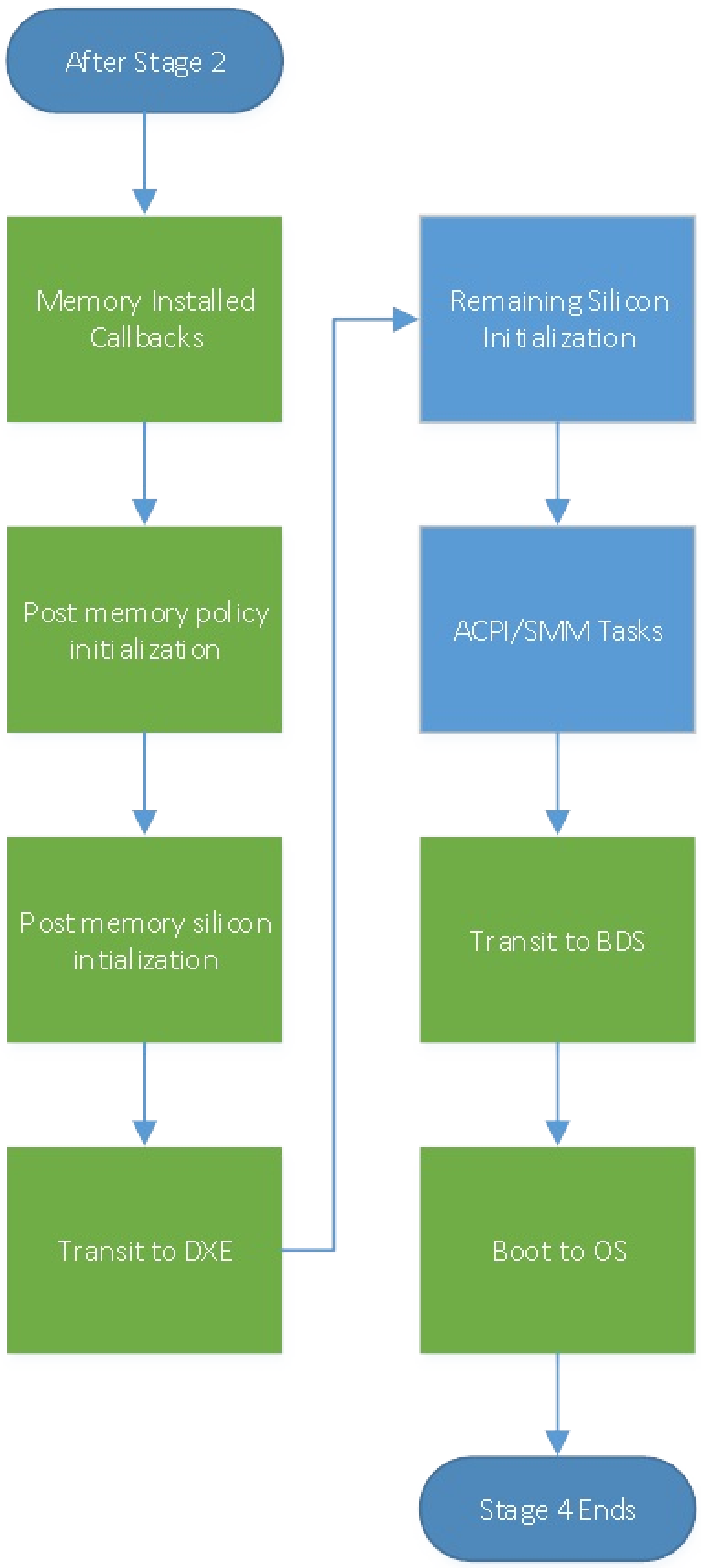

After Stage 2
Memory Installed Callbacks
Post memory policy initialization
Post memory silicon intialization
Transit to DXE
Remaining Silicon Initialization
ACPI/SMM Tasks
Transit to BDS
Boot to OS
Stage 4 Ends

Figure 9 Stage IV Main Control Flow

6.2 Firmware Volumes

Stage IV finalizes silicon initialization, adds basic operating system required interfaces, and supports minimally featured operating system boot. The new components are support in a dedicated firmware volume.

Name	Content	Compressed	Parent FV
FvOsBoot	DXE/BDS Services	Yes	None
FvLateSilicon	ACPI and SMM silicon support	No	FvOsBoot

Table 39 Stage IV Firmware Volumes

Which yields this example extension of the flash map for MMIO storage (add to Stage I + II + III map):

Binary	FV	Components	Purpose
Stage IV	FvOsBoot.fv	FvLateSilicon.fv (child FV)	
		Additional Components	Additional silicon initialization support that is performed late in the boot
		AcpiTable.efi	Provides common ACPI services
		PlatformAcpi.efi	Provides MinPlatform ACPI content
		BoardAcpi.efi	Provides board ACPI content
		PiSmmIpl.efi	SMM initial loader
		PiSmmCore.efi	SMM core services
		ReportStatusCodeRouterSmm.efi	SMM status code infrastructure
		StatusCodeHandlerSmm.efi	SMM status code handlers
		PiSmmCpu.efi	SMM CPU services
		CpuIo2Smm.efi	SMM CPU IO services
		FaultTolerantWriteSmm.efi	SMM fault tolerant write services
		SpiFvbServiceSmm.efi	SMM SPI FLASH services
		Additional Components	Additional post-memory components required for Stage IV boot

Table 40 Stage IV FV and Component Layout

See Appendix: Full FV Map for a more complete example Firmware Volume layout.

6.3 Modules

Only modules in the board package should be modified in the process of board porting. The minimum platform package and other common package contents must not be directly modified. The board package and silicon package modules may have multiple instances to support different boards and different silicon. These components are required. They enable orderly board porting and add the support for extensibility in later stages. The libraries consumed are the subset of libraries required by this specification. Some libraries are defined in this specification, some are defined in EDK II documentation.

6.3.1 UEFI Components (DXE)

These components are required. They enable orderly board porting and orderly extensibility to add functionality over time.

The libraries consumed are the subset of libraries required by this specification. Some libraries are defined in this specification, some are defined in EDK II documentation.

`Item`	`Producing Package`	`Libraries Consumed`
AcpiTable.efi	MdeModulePkg	
PlatformAcpi.efi	MinPlatformPkg	BoardAcpiLib

Table 41 Stage IV ACPI DXE UEFI Components

6.3.2 UEFI Components (DXE)

`Item`	`Producing Package`	`Libraries Consumed`
SataControllerDxe.efi	MdeModulePkg	
AtaAtapiPassThru.efi	MdeModulePkg	
AtaBusDxe.efi	MdeModulePkg	
UhciDxe.efi	MdeModulePkg	
EhciDxe.efi	MdeModulePkg	
XhciDxe.efi	MdeModulePkg	
UsbBusDxe.efi	MdeModulePkg	
UsbMassStorageDxe.efi	MdeModulePkg	
UsbKbDxe.efi	MdeModulePkg	

Table 42 Stage IV DXE UEFI Components

6.3.3 UEFI Components (SMM)

`Item`	`Producing Package`	`Libraries Consumed`
PiSmmIpl.efi	MdeModulePkg	
PiSmmCore.efi	MdeModulePkg	
ReportStatusCodeRouterSmm.e fi	MdeModulePkg	
StatusCodeHandlerSmm.efi	MdeModulePkg	SerialPortLib

PiSmmCpu.efi	UefiCpuPkg	
CpuIo2Smm.efi	UefiCpuPkg	
FaultTolerantWriteSmm.efi	MdeModulePkg	
SpiFvbServiceSmm.efi	MinPlatformPkg	

Table 43 Stage IV SMM UEFI Components

6.3.4 Platform Architecture Libraries

Board porting will require creation of libraries identified as produced by the BoardPkg. Depending on the board, there may be existing libraries that are sufficient for a board, so it is important to assess the utility of existing library instances when developing board support.

`Item`	`API Definition Package`	`Producing Package`	`Description`
BoardAcpiLib	MinPlatformPkg	BoardPkg	Services for ACPI table creation

Table 44 Stage IV Platform Architecture Libraries

6.4 Required Functions

The following functions are required to exist and to execute in the given order. The component that provides the function is not specified because it is not required by the architecture.

The required functions for Stage IV are organized by phase and subsystem (e.g. ACPI, SMM, etc). See Appendix: Full Functions Map for a complete ordering for all stages.

6.4.1 Required DXE Functions

Name	Purpose
PlatformCreateAcpiTable	Create the minimum set of platform-specific ACPI tables
PlatformUpdateAcpiTable	Update data in platform-specific in ACPI tables
PlatformInstallAcpiTable	Install platform-specific ACPI tables
CoreExitBootServices (*)	Dismantles UEFI boot services and enter UEFI run time
BoardInitEndOfFirmware	Board hook for the ExitBootServices event
TestPointExitBootServices	Test to verify state after ExitBootServices has been invoked
RuntimeDriverSetVirtualAddressMap (*)	Sets virtual address mode

Table 45 Stage IV DXE Functions

* In the common EDK II open source code.

6.4.2 DXE Interfaces

Component	Name	Consumer	Purpose
BoardInitLib	BoardNotificationInit	Platform	Board specific initialization hook at DXE phase

Table 46 Stage IV DXE Interfaces

6.4.3 Required SMM Functions

Name	Purpose
SmmIplEntry (*)	SMM IPL
SmmMain (*)	SMM Core entry point
PiCpuSmmEntry (*)	SMM CPU driver
SmmRelocateBases (*)	Relocation
_SmiEntryPoint (*)	SMI entry point
SmmEntryPoint (*)	Dispatch SMI handlers
PchSmmCoreDispatcher	Dispatch PCH child SMI handlers
TestPointSmmEndOfDxe	Verify state after SmmEndOfDxe
TestPointSmmEndOfDxe	Verify state after SmmEndOfDxe
TestPointSmmReadyToLock	Verify state after SmmReadyToLock

PlatformEnableAcpiCallback	Switch the system to ACPI mode
BoardEnableAcpiCallback	Board hook for ACPI mode switch

Table 47 Stage IV SMM Functions

* In the common EDK II open source code.

6.4.4 SMM Interfaces

`Component`	`Name`	`Consumer`	`Purpose`
BoardAcpiLib	BoardEnableEcAcpiMode ()	Platform	Board specific ENABLE_ACPI_MODE action
	BoardDisableEcAcpiMode ()	Platform	Board specific DISABLE_ACPI_MODE action

Table 48 Stage IV SMM Interfaces

6.5 Configuration

This section defines the configurable items that must be available to achieve Stage IV functionality.

These definitions may be both source and binary in nature.

6.5.1 Memory Type Information Related Configuration

PCD	Purpose
gMinPlatformPkgTokenSpaceGuid.	Memory size reserved for ACPI reclaim memory
PcdPlatformEfiAcpiReclaimMemorySize	
gMinPlatformPkgTokenSpaceGuid. PcdPlatformEfiAcpiNvsMemorySize	Memory size reserved for ACPI NVS memory
gMinPlatformPkgTokenSpaceGuid.	Memory size reserved for EFI reserved memory
PcdPlatformEfiReservedMemorySize	
gMinPlatformPkgTokenSpaceGuid. PcdPlatformEfiRtDataMemorySize	Memory size reserved for EFI runtime data memory
gMinPlatformPkgTokenSpaceGuid. PcdPlatformEfiRtCodeMemorySize	Memory size reserved for EFI runtime code memory

Table 49 Memory Type Information Configuration

6.5.2 FV Related Configuration

PCD	Purpose
`gMinPlatformPkgTokenSpaceGuid.PcdFlashFvOsBootBase`	OsBoot FV base address
`gMinPlatformPkgTokenSpaceGuid.PcdFlashFvOsBootSize`	OsBoot FV size

Table 50 Flash Map Configuration PCDs

6.6 Data Flows

This section defines the architecturally defined data structures and the ways this data flows through the system over time. In addition to the definition and lifecycle for important pieces of data.

ACPI tables in this stage are located using AcpiSdtProtocol supplied by MdeModulePkg. These tables are updated with the help existing HOBs, policies for different silicon components. APIs to update these tables are located in the AcpiPlatfrom DXE driver.

6.7 Additional Control Flows

None

6.8 Build Files

This is appended to previous Build files section.

Name	Consumer	Standalone Buildable	FV Produced	Comments
MinPlatformPkg \Include\CoreOsBootInclude.fdf	Board	No	None	Stage IV required components

Table 51 Stage IV Build Files

6.9 Test Point Results

Test Point	Test Subject	Test Overview	Reporting Mechanism
TestPoint ReadyToBoot AcpiTable Functional ()	ACPI table(s)	• Table is reported. • MADT is consistent with MP services.	Dump ACPI tables. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint SmmReadyToLock SecureSmmCommunication Buffer ()	SMM communication buffer	Only CommBuffer(s) and MMIO are mapped in the page table.	Dump memory map and GCD map at SmmReadyToLock and check at SmmReadyToBoot.
TestPoint SmmReadyToLock SmmMemoryAttributeTable Functional ()	SMM memory page attribute table	Table is reported. Image code/data mapping is accurate. • GDT, IDT, and page table are RO • Data is NX • Code is RO	Dump SMM table and SMM Image Info. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint SmmEndOfDxe Smrr Functional ()	SMRR	• SMRR is aligned. • SMRR matches SMRAM_INFO	Dump SMRR and SMRAM_INFO. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint SmmReadyToBoot SmmPageProtection ()	SMM page table	SMM page table matches SmmMemoryAttribute table.	Report error based upon check. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint DxeSmmReadyToLock SmramAligned ()	SMRAM info	SMRAM is aligned.	Dump SMRAM region table. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint DxeSmmReadyToLock WsmtTable Functional ()	WSMT table	WSMT is reported.	Dump WSMT table. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint DxeSmmReadyToBoot SmiHandlerInstrument ()	SmiHandler profile	SmiHandler profile.	Dump SMI Handler profile. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT

Table 52 Stage IV Test Point Results

6.10 Functional Exit Criteria

1. Successfully load a UEFI compatible operating system such that all firmware interfaces required for the OS to load are satisfied (optional interfaces not necessary for loading are not required).
2. On an x86 compatible system which supports SMM, SMM is initialized.
    - Core/CPU
        - CPU code relocates SMM base.
        - CPU code sets SMRR correctly.
        - CPU sets SMM_CODE_CHECK.
3. Silicon
    - Silicon provides software SMI registration capability.
    - Silicon provides capability Sx SMI registration capability.
4. The minimum ACPI tables described in this section are installed.

6.11 Stage Enabling Checklist

The following steps should be followed to enable a platform for Stage IV.

1. Install the minimal DSDT
   i. In rare cases: Install board-specific SSDT
2. Ensure all PCDs in the configuration section (DSC files) are correct for your board.
   i. Set `gMinPlatformPkgTokenSpaceGuid.PcdBootStage` = 4
3. Ensure all required binaries in the flash file (FDF files) are correct for your board.
4. Boot, collect log, verify test point results defined in section 6.9 Test Point Results are correct

7.1 Overview

The objective of Stage V is to establish the basic system security foundation for a production environment. Given the importance of security for all connected systems, the platform architecture considers the following basic security features as minimum requirements for any product and thus an important part of the effort to produce a minimal platform. This stage is concerned with enabling security technologies described in industry specifications. Lower-level chipset-specific security technologies such as register locks may exist and those should be enabled during standard silicon initialization flows in earlier stages.

7.1.1 Major Execution Activities

`Stage V Modules`
Full UEFI variable services support (i.e. non-volatile, volatile, and authenticated)
Authenticated boot (HW and UEFI)
TCG trusted boot (if TPM HW is present)
DMA protection

7.1.2 Main Control Flow

Stage V introduces new modules and requirements to the boot incrementally over Stage IV. The key requirement is to satisfy industry standard security specifications applicable to the platform. The security technologies enabled in this stage are not strictly bound to the definition in this specification and may consist of a subset or superset of the content described in this section. However, the only case in which a modern production system should not implement a form of any of these technologies is if the necessary hardware is not available. In all other cases, the system must at least implement a form of the following:

- Hardware rooted authenticated boot that can establish a Static Root of Trust for Verification (S-RTV) and continue an authenticated chain of verification throughout the boot process.
- System measurement capability that allows the firmware to serve as a Static Root of Trust for Measurement (S-RTM).
- Protection from Direct Memory Access (DMA) attacks.

The TCG measured boot chain of trust is should be enabled in this stage. At this point, Authenticated UEFI Variable support must be completely functional. This is a basic requirement for secure authentication and management of the UEFI Secure Boot keys.

7.2 Firmware Volumes

Stage V supports key security features. Additional FV are:

Name	Content	Compressed	Parent FV
FvSecurity	Security related modules	No	None
NvStorage	Real NV storage on flash	No	None

Table 53 Stage V Firmware Volumes

Which yields this example extension of the flash map for MMIO storage (add to Stage I - IV map):

Binary	FV	Components	Purpose
Stage V	FvSecurity.fv	Tcg2Dxe.efi	TPM2 services
		Tcg2ConfigDxe.efi	TPM2 configuration UI.
		Tcg2PlatformDxe.efi	TPM2 platform module.
		Tcg2Smm.efi	TPM2 ACPI services.
		TcgMor.efi	TCG Memory Override support
		IntelVTdPmrPei.efi	IOMMU PEI services.
		IntelVTdDxe.efi	IOMMU DXE services.
		SecurityStubDxe.efi	Provide security architecture protocol.
		FaultTolerantWriteSmm.efi	Fault-tolerant services in SMM.
		VariableSmm.efi	Provide Variable service in SMM.
		VariableSmmRuntimeDxe.efi	Provide Variable service in UEFI.
		SecureBootConfigDxe.efi	SecureBoot configuration UI.
		Additional Components	Additional post-memory components required for Stage V boot

Table 54 Stage V FV and Components Layout

See Appendix: Full FV Map for a more complete example Firmware Volume layout.

7.3 Modules

Only modules in the board package should be modified in the process of board porting. The minimum platform package and other common package contents must not be directly modified. The board package and silicon package modules may have multiple instances to support different boards and different silicon. These components are required. They enable orderly board porting and add the support for extensibility in later stages. The libraries consumed are the subset of libraries required by this specification. Some libraries are defined in this specification, some are defined in EDK II documentation.

7.3.1 UEFI Components (PEI)

These components are required. They enable orderly board porting and orderly extensibility to add functionality over time.

The libraries consumed are the subset of libraries required by this specification. Some libraries are defined in this specification, some are defined in EDK II documentation.

`Item`	`Producing Package`	`Libraries Consumed`
Tcg2Pei.efi	SecurityPkg	
Tcg2ConfigPei.efi	SecurityPkg	
Tcg2PlatformPei.efi	MinPlatformPkg	
IntelVTdPmrPei.efi	IntelSiliconPkg	

Table 55 Stage V PEI UEFI Components

7.3.2 UEFI Components (DXE)

These components are required. They enable orderly board porting and orderly extensibility to add functionality over time.

The libraries consumed are the subset of libraries required by this specification. Some libraries are defined in this specification, some are defined in EDK II documentation.

`Item`	`Producing Package`	`Libraries Consumed`
TcgMor.efi	SecurityPkg	
Tcg2Dxe.efi	SecurityPkg	
Tcg2ConfigDxe.efi	SecurityPkg	
Tcg2PlatformDxe.efi	MinPlatformPkg	
VariableSmmRuntimeDxe.efi	MdeModulePkg	
SecureBootConfigDxe.efi	SecurityPkg	
SecurityStubDxe.efi	MdeModulePkg	
IntelVTdDxe.efi		

Table 56 Stage V DXE UEFI Components

7.3.3 UEFI Components (SMM)

These components are required. They enable orderly board porting and orderly extensibility to add functionality over time.

The libraries consumed are the subset of libraries required by this specification. Some libraries are defined in this specification, some are defined in EDK II documentation.

Item	Producing Package	Libraries Consumed
Tcg2Smm.efi	SecurityPkg	
FaultTolerantWriteSmm.efi	MdeModulePkg	
VariableSmm.efi	MdeModulePkg	

Table 57 Stage V SMM UEFI Components

7.3.4 Platform Architecture Libraries

Board porting will require creation of libraries identified as produced by the BoardPkg. Depending on the board, there may be existing libraries that are sufficient for a board, so it is important to assess the utility of existing library instances when developing board support.

Item	API Definition Package	Producing Package	Description

Table 58 Stage V Platform Architecture Libraries

7.4 Required Functions

The following functions are required to exist and to execute in the given order. The component that provides the function is not specified because it is not required by the architecture.

* In the common EDK II open source code.

The required functions for Stage IV are presented organized by phase and subsystem (e.g. ACPI, SMM, etc). See Appendix: Full Functions Map for a complete ordering for all stages.

7.4.1 Required PEI functions

Name	Purpose
PeimEntryMA (*)	Entry point for the TPM2 PEIM
IntelVTdPmrInitialize (*)	Entry point for the VT-d PEIM

Table 59 Stage V PEI Functions

* In the common EDK II open source code.

7.4.2 Required DXE functions

Name	Purpose
DriverEntry (*)	Entry point for the TPM2 DXE module
IntelVTdInitialize(*)	Entry point for the VT-d DXE module
UserPhysicalPresent (*)	Indicates whether a physical user is present for UEFI secure boot
ProcessTcgPp	Process the TPM physical presence (PP) request
ProcessTcgMor	Process the TPM memory overwrite request (MOR)

Table 60 Stage V DXE Functions

* In the common EDK II open source code.

7.4.3 Required SMM functions

Name	Purpose
InitializeTcgSmm (*)	Entry point for the TPM2 SMM module
MemoryClearCallback (*)	Callback function for setting the MOR variable

Table 61 Stage V SMM Functions

* In the common EDK II open source code.

7.5 Configuration

This section defines the configurable items that must be available to achieve Stage IV functionality.

These definitions may be both source and binary in nature.

7.5.1 Security Related Configuration

Component	Name	Producer	Consumer	Purpose	Porting Category
Post Build	PK	Board	Core	PK variable	Platform Policy: UEFI Secure Boot
	KEK	Board	Core	KEK variable	Platform Policy: UEFI Secure Boot
	db	Board	Core	db variable	Platform Policy: UEFI Secure Boot
	dbx	Board	Core	dbx variable	Platform Policy: UEFI Secure Boot
PcdTpmInstance Guid	GUID	Board	Core	Select TPM instance	Platform Policy: TCG trusted boot
PcdTpm2 InitializationPolicy	UINT8	Board	Core	Choose if TPM driver need send Tpm2Init.	Platform Policy: TCG trusted boot
PcdTpm2Self TestPolicy	UINT8	Board	Core	Choose if TPM driver need send Tpm2SelfTest	Platform Policy: TCG trusted boot
PRE_MEM_SILICON_POLICY	MOR data	Board	Silicon	The board code consumes the MOR variable and pass it to MemoryInit module as policy	Platform Policy: TCG MOR
L"MemoryOverwrite RequestControl"	MOR Variable	OS	Board	OS indicates to UEFI FW the MOR request.	Platform Policy: TCG MOR
PcdVTdPolicy PropertyMask	VTd policy mask	Platform	Core	VTd policy	Platform Policy: DMA

Table 62 Stage V Security Configuration

7.5.2 FV Related Configuration

PCD	Purpose
gEfiMdeModulePkgTokenSpaceGuid. PcdFlashNvStorageVariableBase	Base address of the NV variable range in flash device.
gEfiMdeModulePkgTokenSpaceGuid. PcdFlashNvStorageVariableSize	Size of the non-volatile variable range. Note that this value should less than or equal to PcdFlashNvStorageFtwSpareSize.
gEfiMdeModulePkgTokenSpaceGuid. PcdFlashNvStorageFtwWorkingBase	Base address of the FTW working block range in flash device.
gEfiMdeModulePkgTokenSpaceGuid. PcdFlashNvStorageFtwWorkingSize	Size of the FTW working block range.
gEfiMdeModulePkgTokenSpaceGuid. PcdFlashNvStorageFtwSpareBase	Base address of the FTW spare block range in flash device. Note that this value should be block size aligned.
gEfiMdeModulePkgTokenSpaceGuid. PcdFlashNvStorageFtwSpareSize	Size of the FTW spare block range. Note that this value should larger than PcdFlashNvStorageVariableSize and block size aligned.
gMinPlatformPkgTokenSpaceGuid. PcdFlashFvSecurityBase	Security FV base address.
gMinPlatformPkgTokenSpaceGuid. PcdFlashFvSecuritySize	Security FV size.

Table 63 Stage V Flash Map Configuration PCDs

7.5.3 Feature Related Configuration

PCD	Purpose
gMinPlatformModuleTokenSpaceGuid.PcdSmiHandlerProfileEnable	Enable SMI handler profile.
gMinPlatformModuleTokenSpaceGuid.PcdTpm2Enable	Enable TPM2.
gMinPlatformModuleTokenSpaceGuid.PcdUefiSecureBootEnable	Enable UEFI Secure Boot.

Table 64 Stage V Feature Configuration

7.6 Data Flows

This section defines the architecturally defined data structures and the ways this data flows through the system over time. In addition to the definition and lifecycle for important pieces of data.

7.7 Additional Control Flows

This section describes how the security features are embedded in the control flows. PSCS/ChipSec required features should be enabled in this stage in addition to other general security flow. This section will also elaborate on each security feature and the platform code implementation required to enable the feature.

Note: Some of these features can be treated as an advanced feature and can be turned on or off based on system-specific usage. However, this section serves as a guideline to develop platform code for security features.

7.7.1 UEFI Secure Boot

Refer to the UEFI specification and the whitepaper A Tour Beyond BIOS - Implementing UEFI Authenticated Variables in SMM with EDK II

7.7.2 Hardware Authenticated Boot

UEFI Secure boot provides verification of 3rd party drivers, such as the OS loader or PCI option ROMs.

A platform may provide additional authentication for firmware volume.

For example: Intel Boot Guard, or PI signed FV.

- Intel® Boot Guard provides a hardware way to verify the initial boot block (IBB) code. After power on, the CPU Microcode finds a Boot Guard ACM and executes the Boot Guard ACM, which is signed by Intel. Then the Boot Guard ACM takes the Boot Guard manifest and verifies the IBB code.
- The PI specification also provides the verification for the system firmware code on the board. Refer to PI specification, EFI Signed Firmware Volumes and EFI Signed Sections.

The whole hardware based secure boot flow on an Intel Boot Guard platform is:

1. Startup ACM or some equivalent module verifies the initial boot block of the system firmware.
   - Intel® Boot Guard Technology is one possible implementation
2. The initial boot block verifies the rest of the system firmware.
   - PI signed FV is one possible implementation. An implementation may choose PKCS7 or RSA2048_SHA256 based signing verification.
   - The other option is just to use the HASH for the rest of the system firmware. In PEI phase, the code who installs the addition FV for the post memory phase need verify the HASH of the system firmware.
3. The system firmware verifies 3rd party code.
   - UEFI secure boot is the implementation.

7.7.3 TCG Trusted Boot and Memory Overwrite Request (MOR)

Refer to TCG platform specification and the white paper A Tour Beyond BIOS - Implementing TPM Support in EDK II

7.7.4 DMA (VT-d) Protection

Refer to Intel® VT-d specification and the white paper Using IOMMU for DMA Protection in UEFI

7.8 Build Files

This is appended to the previous Build files section.

`Name`	`Consumer`	`Standalone Buildable`	`FV Produced`
MinPlatformPkg\Include\CoreSecurityPreMemoryInclude.fdf	Board	No	None
MinPlatformPkg\Include\CoreSecurityPostMemoryInclude.fdf	Board	No	None
MinPlatformPkg\Include\CoreSecurityLateInclude.fdf	Board	No	None

Table 65 Stage V Build Files

7.9 Test Point Results

Test Point	Test Subject	Test Overview	Reporting Mechanism
TestPoint EndOfDxe DmarTable Funtional ()	DMAR table	DMAR table is reported.	Dump DMAR table. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot AcpiTable Functional ()	ACPI table	ACPI tables are valid.	Dump installed ACPI tables. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot GcdResource Functional ()	GCD resource	Memory resources are described consistently in ACPI tables and GDT.	Dump installed ACPI tables and GDT. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot HstiTable Functional ()	HSTI table	HSTI table is reported.	Dump HSTI table. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot EsrtTable Functional ()	ESRT table	ESRT table is reported.	Dump ESRT table. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot PiSignedFvBoot Enabled ()	PI signed FV boot	Verify PI signed FV boot is enabled.	Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot UefiSecureBoot Enabled ()	UEFI Secure Boot	SecureBoot variable is set.	Dump the SecureBoot variable. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot TcgTrustedBoot Enabled ()	TCG trusted boot	TCG protocol is installed.	Dump TCG protocol capability. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint ReadyToBoot TcgMor Enabled ()	TCG MOR	MOR variable is set.	Dump the MOR UEFI variable. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT
TestPoint DiscoveredDma Protection Enabled ()	DMA protection	DMA protection in PEI.	Dump DMA ACPI table. Set ADAPTER_INFO_ PLATFORM_TEST_ POINT_STRUCT

TestPointEndOfDxeDmaAcpiTableFunctional ()	DMA protection	DMA ACPI table is reported.	Dump DMA ACPI table. Set ADAPTER_INFO_PLATFORM_TEST_POINT_STRUCT
TestPointEndOfDxeDmaProtectionEnabled()	DMA protection	DMA protection in DXE.	Dump DMA ACPI table. Set ADAPTER_INFO_PLATFORM_TEST_POINT_STRUCT

Table 66 Stage V Test Point Results

7.10 Functional Exit Criteria

1. UEFI secure boot is enabled.
2. TCG trusted boot is enabled.
3. TCG MOR is enabled.

7.11 Stage Enabling Checklist

The following steps should be followed to enable a platform for Stage V.

1. Update BoardPkg/Board.
   i. Deploy the UEFI secure boot variables (PK/KEK/db/dbx)
   ii. Configure `PcdTpmInstanceGuid` to select TPM hardware. Default of `gEfiTpmDeviceInstanceTpm20DtpmGuid` value is usually correct.
2. UEFI secure boot
   i. Update `PlatformSecureLib` : `UserPhysicalPresent ()` , to check if a user is physically present to authorize change of authenticated variables
3. For TCG trusted boot
   i. May select TPM2 instance `PcdTpmInstanceGuid` .
   ii. May set `PcdFirmwareDebuggerInitialized` based on whether or not a Firmware Debugger is attached to the platform
4. For DMA Protection
   i. May include IOMMU driver to do DMA protection, if the silicon supports IOMMU.
5. Ensure all PCDs in the configuration section (DSC files) are correct for your board.
   i. Set `gMinPlatformPkgTokenSpaceGuid.PcdBootStage` = 5
6. Ensure all required binaries in the flash file (FDF files) are correct for your board.
7. Boot, collect log, verify test point results defined in section 7.9 Test Point Results are correct

8.1 Overview

Advanced features are non-essential features. Essential features are defined as being support required to meet earlier stage boot objectives. An advanced feature must be implemented as highly cohesive and stand-alone software to only support a specific feature. Modularizing such features, reducing dependencies on other advanced features, and eliminating dependencies on specific implementations of other advanced features is critical and results in a variety of benefits:

- The minimum platform serves as a basic enabling vehicle ready to support various roles for a given hardware platform. This yields a minimum platform solution that is open to extension but closed for modification.
- System power-on is simplified because unnecessary code paths and silicon paths can be avoided or deferred.
- Platforms can be composed in a more modular and portable manner allowing generic advanced features to be readily shared among participants.
- Feature adoption benefits from modular design that is simple to maintain.

Organizing advanced features in the platform architecture enables better realization of the benefits in UEFI specification compliant firmware with highly cohesive and lowly coupled component interactions.

This chapter provides guidance on how to design and integrate advanced features. The source code layout and other maintenance details are outside the scope of this specification.

The core advanced feature requirements that must be met:

- Cohesive, the feature should not contain any functionality unrelated to the feature.
- Complete, the feature must have a complete design that minimizes dependencies. A feature package cannot directly depend on another feature package.
- Easy to Integrate, the feature should expose well-defined software interfaces to use and configure the feature.
  - It should also present a set of simple and well-documented standard EDK II configuration options such as PCDs to configure the feature.
  - In general, features should be self-contained and started by the dispatcher. The board firmware should be required to perform as few steps as possible to enable the feature.
  - All features are required to have a feature enable PCD (`PcdFeatureEnable`). Any effort to enable the feature besides this PCD should be carefully considered. Default configuration values should apply to the common case.
- Portable, the feature is not allowed to depend on other advanced feature or board source code packages. For example, if Feature A depends on output Feature B, a board integration module should use a generic interface in Feature A to get the output and pass it to a generic interface in Feature B. Structures should not be shared between feature packages. Most structures should be defined in a common package such as MdePkg if the structure is industry standard, IntelSiliconPkg if the structure is specific to Intel silicon initialization, etc. Feature-specific structures are of course allowed to be defined within a feature package and used by libraries and modules in that package.
- Self Documenting, the feature should follow software best practices to allow others to debug the code and contribute changes. In addition to source code, advanced features must have a Readme.md with sufficient information for a newcomer to understand the feature.
- Single Instance, the feature should not have more than one instance of a source solution. If an existing feature package does not solve a specific instance of a problem for the feature, the feature package should be re-worked to consider new requirements instead of duplicating feature code.

8.1.1 Major Execution Activities

`Stage VI Modules`
Execute the Enabled Advanced Features

The number of embedded features must be minimized in order to support the broadest compatibility of the minimal platform. Features should be designed to define an API that can be used to integrate the feature into generic platform configurations. The feature source code should never be modified to absorb details of a specific platform or board.

8.2 Firmware Volumes

Stage VI enables advanced features. There is a container FV for adding advanced features:

Name	Content	Compressed	Parent FV
FvAdvancedPreMemory	Advanced feature drivers that should be dispatched prior to memory initialization	No	None
FvAdvanced	Advanced feature drivers that should be dispatched after memory initialization	Yes	None

Table 67 Stage VI Firmware Volumes

Which yields this example extension of the flash map for MMIO storage (append to Stage I-Stage V map):

Binary	FV	Components	Purpose
Stage VI	FvAdvancedPreMemory.fv	FeatureStack1.fv	Feature 1
		Additional Feature Stacks	Additional pre-memory advanced features
	FvAdvanced.fv	FeatureStack1.fv	Feature 1
		FeatureStack2.fv	Feature 2
		FeatureStack3.fv	Feature 3
		Additional Feature Stacks	Additional advanced features

The modules that constitute a particular feature are not required to be contained within a single firmware volume and this might especially be the case in systems with limited flash storage capacity which could be impacted by firmware volume alignment requirements.

Table 68 Stage VI FV and Component Layout

The PEI core will create a FV HOB for each child firmware volume such that each DXE firmware volume is exposed to the DXE dispatcher.

8.3 Configuration

8.3.1 FV Related Configuration

PCD	Purpose
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvAdvancedPreMemoryBase	Pre-memory advanced features FV base address.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvAdvancedPreMemorySize	Pre-memory advanced features FV size.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvAdvancedBase	Advanced Features FV base address.
gMinPlatformPkgTokenSpaceGuid.PcdFlashFvAdvancedSize	Advanced Features FV size.

Table 69 Stage VI Flash Map Configuration PCDs

8.4 Advanced Feature Design

Advanced features should be designed such that they are easily portable between minimum platform compliant implementations. In consideration of portability, it is recommended to encapsulate each feature within a dedicated package. Such encapsulation enables rapid integration of the feature and a focused area for feature-related changes. For example, feature declarations for build elements such as GUIDs, PCDs, PPIs, and protocols are scoped within the feature package DEC file. Including the feature and consequently the package imports the feature tokens within the available namespace and changes affecting the feature are localized to the package which in turn exposes the change to all feature consumers.

The Advanced Feature template should be used to describe relevant configuration for integrating the feature into a minimum platform compliant system. Any board or silicon-specific details should be abstracted such that the information is provided to the feature via "feature APIs". Such dependencies are recommended to be exposed via binary interfaces such as PPIs and protocols and can be considered similar in purpose to the architectural PPIs and Protocols defined in the PI specification. Such requirements must be included in the "Required Functions" section of the advanced feature template. Though not required, to increase portability, advanced features should not depend upon deprecated EDK II packages and attempt to reduce exposure to packages other than MdePkg and UefiCpuPkg. In turn, this decreases risk of depending upon deprecated packages in the future.

9.1 Overview

Note: This is a proposed stage in the architecture and this section is reserved for future completion and definition of the stage. Any implementation may ignore this stage until this section is completed and this notice is removed.

It is anticipated that future versions of this architecture specification will provide details for embedded performance tuning, common component tuning, and more invasive customization. The objective for this section is to provide a spectrum of options that keep as many board designs as consistent as possible. Some potential topics follow, but should not be reviewed at this

In Stage VII, the fully featured is tuned for production.

First, it can be worthwhile to look at the embedded features and performance oriented options that have been designed into the core or minimum platform. For example, if you do not support network boot, the PciBus driver provides a PCD to disable dispatching the network option ROM. By default, network option ROM dispatch is enabled. This is a known tunable setting.

Second, it can be worthwhile to strip unused components from the defined FV. For simplicity and consistency of progressing through Stage VI, it is better to use the provided code consistent with the architecture. Once a stable and fully functional system is completed, it is intended that platform architecture compatible systems can still remove unneeded components in order to finely tune the product. The core provides a tool named FMMT that can be used to process the build output and remove unnecessary components. Alternatively, a board can copy and modify the provided Build DSC and FDF files in the MinPlatformPkg and SiliconPkg. The former increases build time. The latter increases integration effort for new core, MinPlatformPkg, and SiliconPkg releases.

Third, it is often necessary to enable and use tools to perform detailed analysis of performance and size to identify hotspots that need to be improved.

Appendix A Full Maps Overview

This appendix section provides full reference maps of concepts covered in the specification.

These maps are maintained in this section as they incorporate information across several sections.

A.1 Firmware Volume Layout

This is a logical firmware volume layout by stage.

`Binary`	`FV`	`Components`	`Purpose`
`Stage I`	`FvPreMemory.fv`	SecCore.efi	• Reset Vector • Passes PEI core the address of FvFspmM • Passes PEI core the debug configuration
		ReportFvPei.efi	• Installs firmware volumes
		SiliconPolicyPeiPreMemory.efi	• Publishes silicon initialization configuration
		PlatformInitPreMemory.efi	• Performs pre memory initialization
		FvSecurityPreMemory.fv</br>(child FV)	
		Tcg2Pei.efi	• TPM2 initialization
		Tcg2ConfigPei.efi	• TPM2 selection
		Tcg2PlatformPei.efi	• TPM2 platform module
		Additional Components	• Additional pre-memory components required for Stage V boot
		Additional Components	• Additional pre-memory components required for Stage I boot
	`FvBspPreMemory.fv`	**FvAdvancedPreMemory.fv**</br>(child FV)	
		Additional Components	• Advanced feature pre-memory stacks
		Additional Components	• Additional pre-memory board support components
			• Initializes T-RAM silicon

	`FvFspT.fv`	PlatformSec.efi	• Tests T-RAM functionality
		Additional Components	
	`FvFspM.fv`	PeiCore.efi	• PEI services and dispatcher
		PcdPeim.efi	• PCD service
		FspPlatform.efi	• Converts UPD to Policy PPI
		FvPreMemorySilicon.fv</br>(child FV)	
		Additional Components	• Pre-memory silicon initialization components
		ReportStatusCodeRouterPei.efi	• Provide status code infrastructure
		StatusCodeHandlerPei.efi	• Provide status code listeners
		Additional Components	
	`FvFspS.fv`	**FvPostMemorySilicon.fv**</br>(child FV)	
		Additional Components	• Post-memory silicon initialization components
		Additional components	
`Binary`	`FV`	`Components`	`Purpose`
`Stage II`	`FvPostMemory.fv`	ReadOnlyVariable.efi	• Common core variable services
		SiliconPolicyPeiPostMemory.efi	• Publishes silicon initialization configuration
		PlatformInitPostMemory.efi	• Performs post memory initialization
		DxeIpl.efi	• Load and invoke DXE
		ResetSystemRuntimeDxe.efi	• Provides reset service
		PciHostBridge.efi	• PCI host bridge driver
			• Additional post-

		Additional Components	components required for Stage II boot
	`FvBsp.fv`	Additional Components	• Post-memory board support components
`Binary`	`FV`	`Components`	`Purpose`
`Stage III`	`FvUefiBoot.fv`	DxeCore.efi	• DXE services and dispatcher
		PcdDxe.efi	• Provides PCD services
		ReportStatusCodeRouterDxe.efi	• Provides status code infrastructure
		StatusCodeHandlerRuntimeDxe.efi	• Provides status code listeners
		BdsDxe.efi	• Provides Boot Device Selection phase
		CpuDxe.efi	• Provides processor services
		Metronome.efi	• Provides metronome HW abstraction
		MonotonicCounterRuntimeDxe.efi	• Provides monotonic counter service
		PcatRealTimeClockRuntimeDxe.efi	• Provides RTC abstraction
		WatchdogTimer.efi	• Provides watchdog timer service
		RuntimeDxe.efi	• Provides UEFI runtime service functionality
		Security.efi	• Provides security services to core
		HpetTimerDxe.efi	• Provide timer service
		EmuVariableRuntimeDxe.efi	• Provides UEFI variable service
		CapsuleRuntimeDxe.efi	• Provides capsule service

		PciBusDxe.efi	• PCI bus driver
		GraphicsOutputDxe.efi	• Provides graphics support
		TerminalDxe.efi	• Provides terminal services
		GraphicsConsoleDxe.efi	• Provides graphics console
		ConSplitterDxe.efi	• Provides multi console support
		EnglishDxe.efi	• Provides Unicode collation services
		MemoryTest.efi	• Provide memory test
		DevicePathDxe.efi	• Provides device path services
		DiskIo.efi	• Provides disk IO services
		Partition.efi	• Provides disk partition services
		Fat.efi	• Provides FAT filesystem services
		Additional Components	• Additional post-memory components required for Stage III boot
`Binary`	`FV`	`Components`	`Purpose`
`Stage IV`	`FvOsBoot.fv`	**FvLateSilicon.fv** (child FV)	
		Additional Components	• Additional silicon initialization support that is performed late in the boot
		AcpiTable.efi	• Provides common ACPI services
		PlatformAcpi.efi	• Provides MinPlatform ACPI content
		BoardAcpi.efi	• Provides board ACPI content
		PiSmmIpl.efi	• SMM initial loader

		PiSmmCore.efi	• SMM core services
		ReportStatusCodeRouterSmm.efi	• SMM status code infrastructure
		StatusCodeHandlerSmm.efi	• SMM status code handlers
		PiSmmCpu.efi	• SMM CPU services
		CpuIo2Smm.efi	• SMM CPU IO services
		FaultTolerantWriteSmm.efi	• SMM fault tolerant write services
		SpiFvbServiceSmm.efi	• SMM SPI FLASH services
		Additional Components	• Additional post-memory components required for Stage IV boot

`Binary`	`FV`	`Components`	`Purpose`
`Stage V`	`FvSecurity.fv`	Tcg2Dxe.efi	• TPM2 services
		Tcg2ConfigDxe.efi	• TPM2 configuration UI
		Tcg2PlatformDxe.efi	• TPM2 platform module
		Tcg2Smm.efi	• TPM2 ACPI services
		TcgMor.efi	• TCG Memory Override support
		IntelVTdPmrPei.efi	• IOMMU PEI services
		IntelVTdDxe.efi	• IOMMU DXE services
		SecurityStubDxe.efi	• Provide security architecture protocol.
		FaultTolerantWriteSmm.efi	• Fault-tolerant services in SMM.
		VariableSmm.efi	• Provide Variable service in SMM.
		VariableSmmRuntimeDxe.efi	• Provide Variable

		VariableSmmRuntimeDxe.efi	service in UEFI.
		SecureBootConfigDxe.efi	• SecureBoot configuration UI.
		Additional Components	• Additional post-memory components required for Stage V boot
`Binary`	`FV`	`Components`	`Purpose`
Stage VI	`FvAdvancedPreMemory.fv`	**FeatureStack1.fv** (child FV)	• Feature 1
		FeatureStack2.fv (child FV)	• Feature 2
	`FvAdvanced.fv`	**FeatureStack1.fv** (child FV)	• Feature 1
		FeatureStack2.fv (child FV)	• Feature 2
		FeatureStack3.fv (child FV)	• Feature 3
		Additional Feature Stacks	• Features

Table 71 Full Firmware Volume Layout

A.2 Key Function Invocation

Name	Purpose
ResetHandler (*)	The reset vector invoked by silicon
TempRamInit	Silicon initializes temporary memory
TestPointTempMemoryFunction	Test temporary memory functionality
SecStartup (*)	First C code execution, constructs PEI input
TestPointEndOfSec	Verify state before switching to PEI
PeiCore (*)	PEI entry point
PeiDispatcher (*)	Calls the entry points of PEIM
ReportPreMemFv	Installs firmware volumes required in pre-memory
BoardDetect	Board detection of the motherboard type
BoardDebugInit	Board specific initialization for debug device
PlatformHookSerialPortInitialize	Board serial port initialization. Called from SEC or PEI
TestPointDebugInitDone	Verify debug functionality
BoardBootModeDetect	Board hook for EFI_BOOT_MODE detection
BoardInitBeforeMemoryInit	Board specific initialization, e.g. GPIO
SiliconPolicyInitPreMemory	Silicon pre memory policy initialization
SiliconPolicyUpdatePreMemory	Board updates silicon policies
SiliconPolicyDonePreMemory	Complete pre memory silicon policy data collection
MemoryInit	Silicon initializes permanent memory
InstallEfiMemory	Install permantent memory to core
PeiCore (*)	PEI entry point (post memory entry)
SecTemporaryRamDone (*)	Call SEC to tear down temporary memory
ReportPostMemFv	Installs firmware volumes required in post-memory
TestPointPostMemoryFvInfoFunctional	Test for Firmware Volume map
BoardInitAfterMemoryInit	Board initialization after memory is installed
SetCacheMtrr	Configure cache map for permanent memory
TestPointPostMemoryMtrrAfterMemoryDiscoveredFunctional	Test post-memory cache map
TestPointPostMemoryResourceFunctional	Test resources
TestPointPostMemoryFvInfoFunctional	Test for Firmware Volume map

BoardInitBeforeSiliconInit	Board initialization hook
SiliconPolicyInitPostMemory	Silicon post memory policy initialization
SiliconPolicyUpdatePostMemory	Board updates silicon policies
SiliconPolicyDonePostMemory	Complete post memory silicon policy data collection
BoardInitAfterSiliconInit	Board specific initialization after silicon is initialized
DxeLoadCore (*)	DXE IPL locate and call DXE Core
SetCacheMtrrAfterEndOfPei	Sets cache map in preparation for DXE
TestPointEndOfPei	Verify expected state as we exit PEI phase
TestPointPostMemoryMtrrEndOfPeiFunctional	Basic test for cache configuration before entering DXE
PeimEntryMA (*)	Entrypoint for TPM2 PEIM
IntelVTdPmrInitialize (*)	Entrypoint for VT-d PEIM
DxeMain (*)	DXE entry point
CoreStartImage (*)	Calls the entry points of DXE drivers
SiliconPolicyInitLate	Silicon late policy initialization
SiliconPolicyUpdateLate	Board updates silicon policies
SiliconPolicyDoneLate	Complete late silicon policy data collection
CoreAllEfiServicesAvailable (*)	Check if required architectural protocols are installed
SmmIplEntry (*)	SMM IPL
SmmMain (*)	SMM Core entrypoint
PiCpuSmmEntry (*)	SMM CPU driver
SmmRelocateBases (*)	Relocation
_SmiEntryPoint (*)	SMI entry point
SmmEntryPoint (*)	Dispatch SMI handlers
PchSmmCoreDispatcher	Dispatch PCH child SMI handlers
InitializeTcgSmm (*)	Entrypoint for TPM2 SMM
MemoryClearCallback (*)	Callback function for MOR setting
PlatformCreateAcpiTable	Create the minimum set of platform specific tables
PlatformUpdateAcpiTable	Update platform specific data in ACPI tables - FADT.
PlatformInstallAcpiTable	Install platform specific ACPI tables
DriverEntry (*)	Entrypoint for TPM2 DXE
IntelVTdInitialize(*)	Entrypoint for VT-d DXE
	Return if physical user is present for

	UEFI secure boot
ProcessTcgPp	Process TPM PP request
ProcessTcgMor	Process TPM MOR request
BdsEntry (*)	BDS entry point
PlatformBootManagerBeforeConsole (*)	Platform specific BDS functionality before console
BoardInitAfterPciEnumeration	Board-specific hook on PCI enumeration completion
TestPointPciEnumerationDone	Verify PCI
ExitPmAuth	Signal key security events EndOfDxe and SmmReadyToLock
TestPointEndOfDxe	Verify expected state after EndOfDxe
TestPointDxeSmmReadyToLock	Verify expected state after SmmReadyToLock
TestPointSmmEndOfDxe	Verify state after SmmEndOfDxe
TestPointSmmEndOfDxe	Verify state after SmmEndOfDxe
TestPointSmmReadyToLock	Verify state after SmmReadyToLock
EfiBootManagerDispatchDeferredImages (*)	Dispatch deferred third party UEFI driver OPROMs
PlatformBootManagerAfterConsole (*)	Platform specific BDS functionality after console
BootBootOptions (*)	Attempt each boot option
EfiSignalEventReadyToBoot (*)	Signals the ReadyToBoot event group
BoardInitReadyToBoot	Board hook on ReadyToBoot event
TestPointReadyToBoot	Verify state after ReadyToBoot event signal
UefiMain (*)	UEFI Shell entry point
CoreExitBootServices (*)	Dismantles UEFI boot services and enters runtime
BoardInitEndOfFirmware	Board hook for ExitBootServices event
TestPointExitBootServices	Verify state after ExitBootServices has been called
RuntimeDriverSetVirtualAddressMap (*)	Set virtual address mode
PlatformEnableAcpiCallback	Switch the system to ACPI mode
BoardEnableAcpiCallback	Board hook for ACPI mode switch

Table 72 Key Function Invocation

* In the common EDK II open source code.

A.3 BDS Hook Points

BDS Hook Point Summary

Four new event signal groups are defined that will be signaled at the point shown in Table 16 Event groups as described in the UEFI specification are collections of events identified by a shared EFI_GUID that when one member event group is signaled, all other event groups are signed and their individual notification actions are taken. These event groups should be used in combination with the pre-existing notification mechanisms: signal of gEfiEndOfDxeEventGroupGuid and installation of the gEfiPciEnumerationCompleteProtocolGuid or gEfiDxeSmmReadyToLockProtocolGuid. Preference should always be given to the notification points defined outside this specification to make code dependent upon the notification as portable as possible.

PlatformBootManagerBeforeConsole () [1] Event: PCI enumeration complete - Install gEfiPciEnumerationCompleteProtocolGuid

```
* Minimum Platform action(s) performed:
    * Trusted consoles added
```

[2] Event: SignalBeforeConsoleAfterTrustedConsole

```
* Minimum Platform action(s) performed:
    * Enumerate USB keyboard
    * Connect controller for trusted graphics console
    * Register default boot option (UEFI shell)
    * Register static hot keys (F2/F7)
    * Process TCG Physical Presence
    * Process TCG MOR
    * Perform memory test
```

[3] Event: SignalBeforeConsoleBeforeEndOfDxe

```
* Minimum Platform action(s) performed:
    * None
```

[4] Event: End of DXE - Signal gEfiEndOfDxeEventGroupGuid

```
* Minimum Platform action(s) performed:
    * None
```

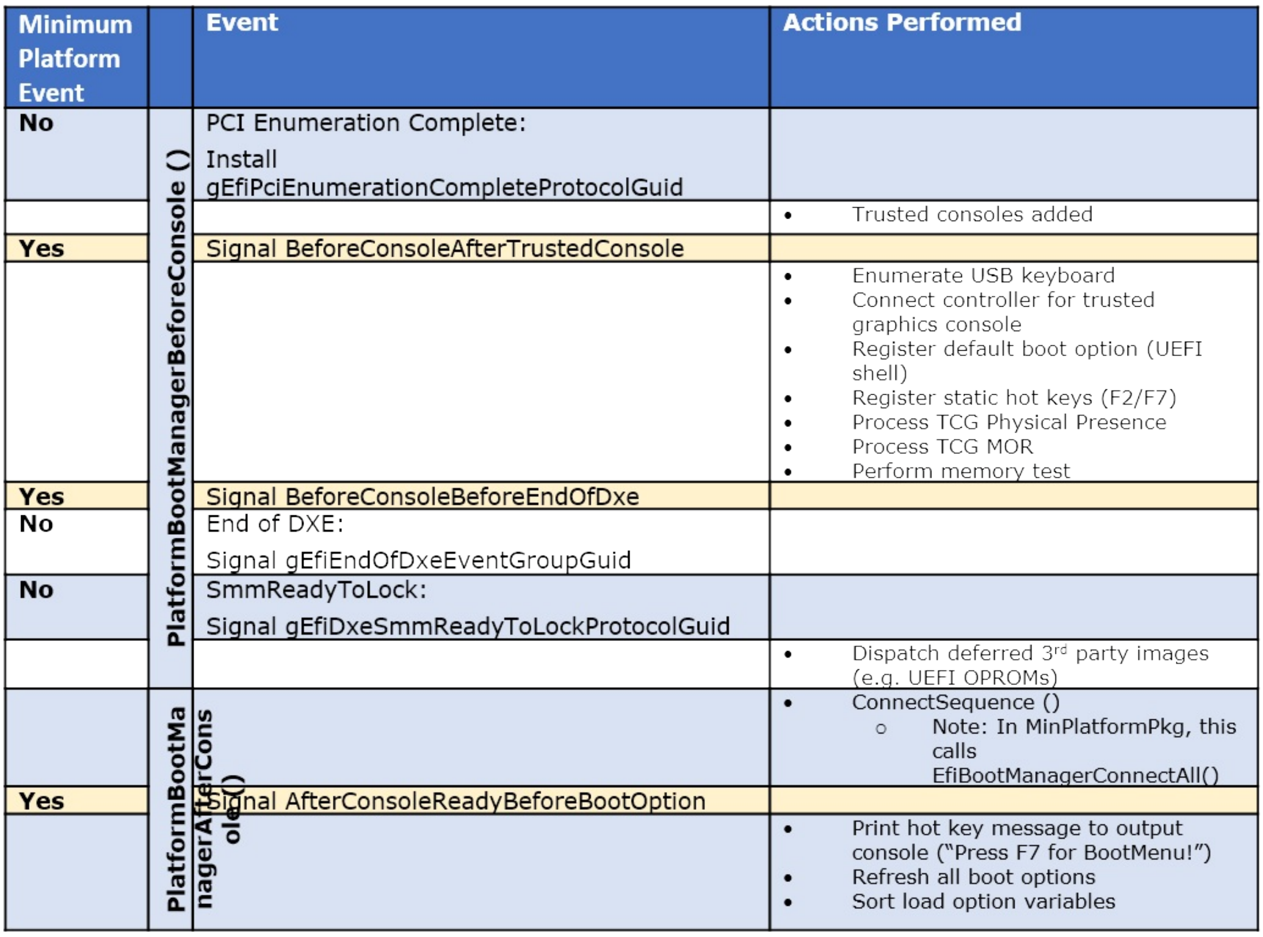
[5] Event: SmmReadyToLock: Signal gEfiDxeSmmReadyToLockProtocolGuid

```
* Minimum Platform action(s) performed:
    * Dispatch deferred 3rd party images (e.g. UEFI OPROMs)
```

PlatformBootManagerAfterConsole () [1] Invoke ConnectSequence ()

[2] Event: Signal AfterConsoleReadyBeforeBootOption

```
* Minimum Platform action(s) performed:
    * Print hot key message to output console ("Press F7 for BootMenu!")
    * Refresh all boot options
    * Sort load option variables
```



Minimum Platform Event		Event	Actions Performed
No	PlatformBootManagerBeforeConsole ()	PCI Enumeration Complete: Install gEfiPciEnumerationCompleteProtocolGuid	
			• Trusted consoles added
Yes		Signal BeforeConsoleAfterTrustedConsole	
			• Enumerate USB keyboard • Connect controller for trusted graphics console • Register default boot option (UEFI shell) • Register static hot keys (F2/F7) • Process TCG Physical Presence • Process TCG MOR • Perform memory test
Yes		Signal BeforeConsoleBeforeEndOfDxe	
No		End of DXE: Signal gEfiEndOfDxeEventGroupGuid	
No		SmmReadyToLock: Signal gEfiDxeSmmReadyToLockProtocolGuid	
			• Dispatch deferred 3rd party images (e.g. UEFI OPROMs)
	PlatformBootManagerAfterConsole()		• ConnectSequence () ○ Note: In MinPlatformPkg, this calls EfiBootManagerConnectAll()
Yes		Signal AfterConsoleReadyBeforeBootOption	
			• Print hot key message to output console (“Press F7 for BootMenu!”) • Refresh all boot options • Sort load option variables

Figure 10 Full BDS Hook Point Map

Appendix B Global Configuration Overview

This appendix section provides configuration mechanisms that are global and therefore are not constrained to any particular stage section.

B.1 Stage Configuration

```
[PcdsFeatureFlag]
  # Stage I - Boot to Debug Enabled
  # Stage II - Boot to Memory Initialization
  # Stage III - Boot to UEFI Shell
  # Stage IV - Boot to Operating System
  # Stage V - Boot to Operating System with Security Enabled
  gMinPlatformPkgTokenSpaceGuid.PcdBootStage|5|UINT8|0xF00000A0
```

The default value of `PcdBootStage` should be Stage V Boot to Operating System with Security Enabled. The stage selection PCD might influence code paths within shared modules between stages or add and remove stages from the build.

B.2 Test Point Check Infrastructure

Today's platforms are tested against several test suites such as Chipsec, Windows Hardware Security Test Infrastructure (HSTI), Windows Hardware Logo Kit (HLK), Linux UEFI Validation (LUV), and others. However, platforms may have platform-specific requirements not covered by test suites enforcing specification or general hardware compliance. The Test Point Check infrastructure is intended to test that actions such as MTRRs are configured correctly, FV HOBs are reported properly, no 3rd party options ROMs are executed before allowed, MemoryTypeInformation is reported correctly, and any other custom logic that platform implementer considers appropriate based on the platform requirements.

The Test Point infrastructure is supported by two primary libraries, TestPointLib and TestPointCheckLib. TestPointLib reports test results via the `ADAPTER_INFO_PLATFORM_TEST_POINT` structure defined below. The test result is validated in the TestPointCheckLib.

```
typedef struct {
  UINT32 Version;
  UINT32 Role;
  CHAR16 ImplementationID[256];
  UINT32 FeaturesSize;

  //UINT8 FeaturesImplemented[]; <- PCD set to define features
  //UINT8 FeaturesVerified[]; <- PCD read and set to determine features verified
  //CHAR16 ErrorString[];
} ADAPTER_INFO_PLATFORM_TEST_POINT;
```

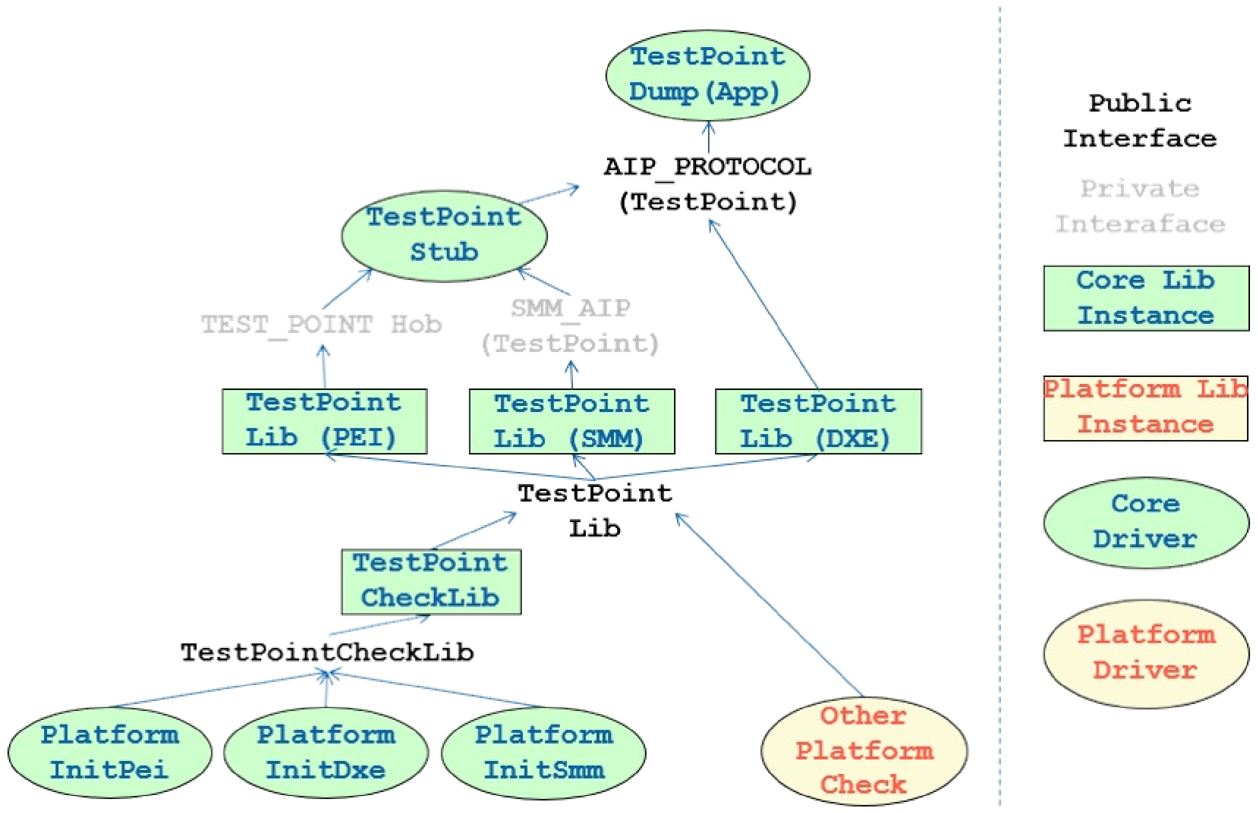


Figure 11 Test Point Check Infrastructure

Appendix C Advanced Configuration and Power Interface (ACPI) Overview

This section documents the layout and implementation guidelines of the ACPI tables generated for the platform. The implementation guidelines for ACPI code will focus on coding practices for ASL irrespective of platform code details.

C.1 Layout

ACPI tables will be organized into a set of mandatory tables defined in this section and optional tables provided in the form of an SSDT.

C.1.1 Mandatory Tables

The mandatory tables are composed of the minimum set of tables required to boot an ACPI compliant OS. These tables are intended to be present in Stage IV and later stages. The contents of these tables might differ based on build stage, it is described in the Table Contents sub-section.

The following tables fall under the mandatory tables list:

1. RSDP
2. RSDT/XSDT
3. FADT/FACS
4. DSDT
5. MADT
6. MCFG
7. HPET

C.2 ACPI Table Contents

There are three types of tables supported. Standard Static Tables, Differentiated System Description Table (DSDT), Secondary System Description Table (SSDT). The standard static tables have a defined structure in the ACPI specification. The contents of the DSDT and SSDT are described in this specification.

C.2.1 DSDT Contents

DSDT is a mandatory fixed table that is pointed to by the FADT (Fixed ACPI Description Table).

C.2.1.1 Stage IV Build

Stage IV is intended to have the minimum configuration to boot a platform with basic features and minimal set of devices enabled. Similarly ACPI implementation should have a minimal framework implemented for ACPI compliant OS.

The DSDT in this case should have a root and system bus defined. In addition to that, the DSDT will have device scopes for all the devices present in the minimum platform required packages (Section 8.1.1).

C.2.1.2 Stage VI Build

In this case, DSDT will include the following Device scopes and objects:

1. Device scopes for all PCI devices that need an ACPI component
2. Global NVS area region defined
3. Interrupt routing (_PRT method)

C.3 ACPI Device Categorization

ACPI device description tables such as DSDT and SSDT are comprised of device scopes and methods that define the capabilities and resources of an ACPI device. For scalability purposes, the ACPI devices are categorized in a manner that they can be easily plugged in and out of UEFI FW.

C.3.1 Silicon Specific Devices

These devices are silicon specific and are assumed to not change with different SKUs and stepping of the silicon. These devices will become part of DSDT as it is a mandatory table containing the fixed devices for the systems.

The number of silicon devices present in the DSDT will be decided by the scope of minimum and full build.

C.3.2 SKU Specific Devices

These devices are SKU specific and are assumed to change based on various SKUs. They are considered to be dynamic as they can be enabled/disabled/modified based on setup knobs or softstraps etc.

Because of their dynamic nature, these devices are added in the Platform SSDT. Every SKU will have a unique Platform SSDT installed. It will only contain devices present on that platform.

C.3.3 Board Specific Devices

These devices are board specific and are assumed to change based on the various board SKUs. These devices will also become part of SSDTs. These devices can have an SSDT of their own or get added to the platform SSDT depending on their availability on multiple SKUs. A fairly common board device will be added to the platform SSDT and the other devices can have a SSDT of their own.

C.3.4 Feature Specific Devices/Methods

These devices or methods are optional as they are exposed to handle certain advanced features. They will be added to DSDT or SSDT depending on the device they are being added for. For example a special DSM (device specific method) is to be added for an Audio codec, then it will fall under the Platform/Board SSDT.

C.4 Flow Diagrams

This section describes the flow for table integration and installing during boot. Device nodes will be generated and added to DSDT or SSDT by the device modules. Once the DSDT and SSDT are complete, they will be installed and published by the ACPI DXE driver.

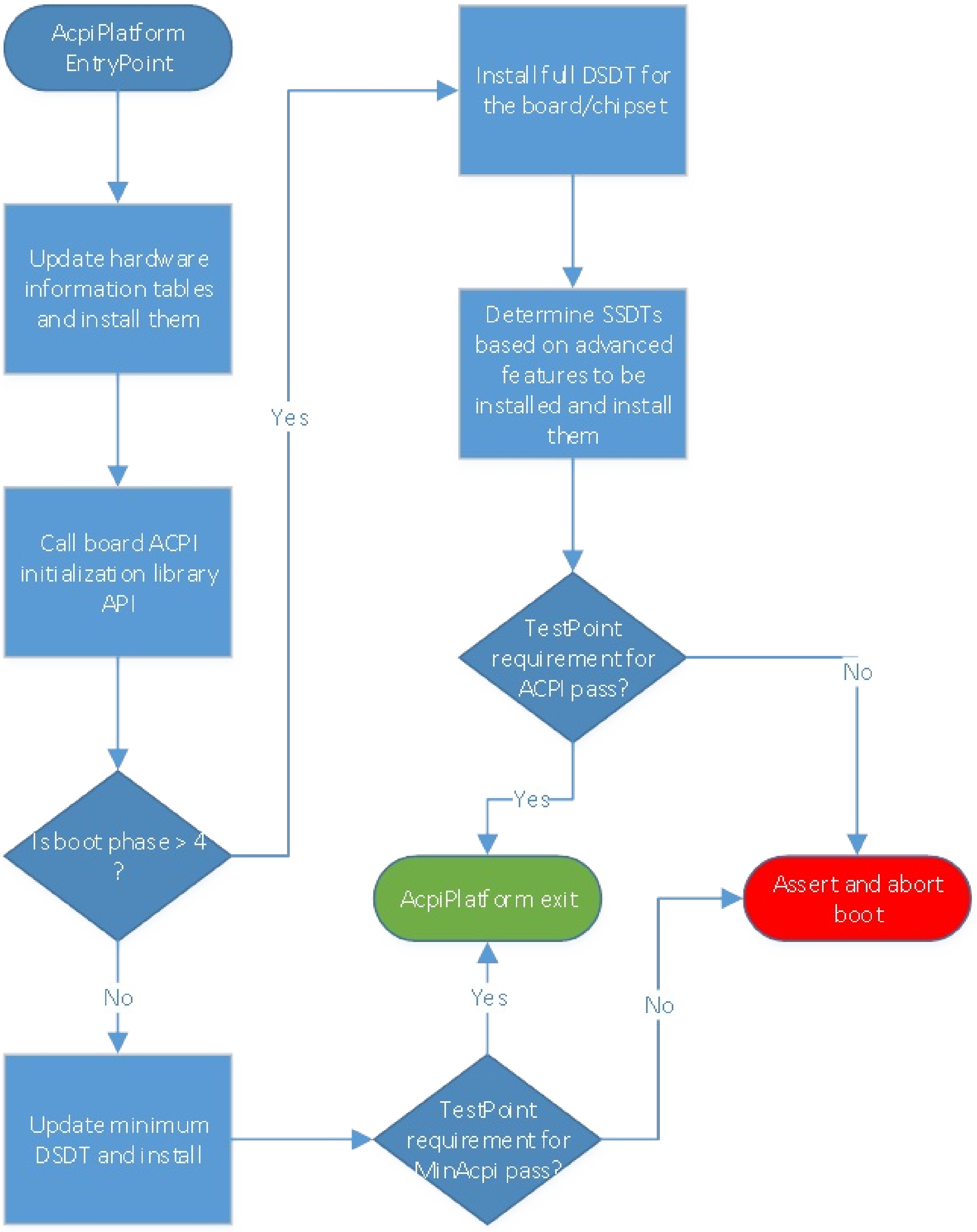

AcpiPlatform EntryPoint
Update hardware information tables and install them
Call board ACPI initialization library API
Is boot phase > 4 ?
Yes
No
Update minimum DSDT and install
TestPoint requirement for MinAcpi pass?
Yes
No
Install full DSDT for the board/chipset
Determine SSDTs based on advanced features to be installed and install them
TestPoint requirement for ACPI pass?
Yes
No
AcpiPlatform exit
Assert and abort boot

Figure 12 ACPI Platform Flow

Appendix D Interface Definitions Overview

This section contains interface definitions defined in the Minimum Platform architecture.

D.1 Required Functions

D.1.1 BoardPorting.SEC

D.1.1.1 ResetHandler (*)

```
; For IA32, the reset vector must be at 0xFFFFFFF0, i.e., 4G-16 byte
; Execution starts here upon power-on/platform-reset.
;
ResetHandler:
    nop
    nop
ApStartup:
    ;
    ; Jmp Rel16 instruction
    ; Use machine code directly in case of the assembler optimization
    ; SEC entry point relative address will be fixed up by some build tool.
    ;
    ; Typically, SEC entry point is the function _ModuleEntryPoint() defined in
    ; SecEntry.asm
    ;
    DB      0e9h
    DW      -3
```

D.1.2 BoardPorting.PEI

D.1.2.1 ReportPreMemFv

```
VOID
ReportPreMemFv (
  VOID
  );
```

D.1.2.2 BoardDetect

```
EFI_STATUS
EFIAPI
BoardDetect (
  VOID
  );
```

D.1.2.3 BoardDebugInit

```
EFI_STATUS
EFIAPI
BoardDebugInit (
  VOID
  );
```

D.1.2.4 PlatformHookSerialPortInitialize

```
/**
  Performs platform specific initialization required for the CPU to access
  the hardware associated with a SerialPortLib instance.  This function does
  not initialize the serial port hardware itself.  Instead, it initializes
  hardware devices that are required for the CPU to access the serial port
  hardware.  This function may be called more than once.
  @retval RETURN_SUCCESS       The platform specific initialization succeeded.
  @retval RETURN_DEVICE_ERROR  The platform specific initialization could not be completed.

**/
RETURN_STATUS
EFIAPI
PlatformHookSerialPortInitialize (
  VOID
  );
```

D.1.2.5 BoardBootModeDetect

```
EFI_BOOT_MODE
EFIAPI
BoardBootModeDetect (
  VOID
  );
```

D.1.2.6 BoardInitBeforeMemoryInit

```
EFI_STATUS
EFIAPI
BoardInitBeforeMemoryInit (
  VOID
  );
```

D.1.2.7 SiliconPolicyUpdatePreMemory

```
/**
  Performs silicon pre-memory policy update.

  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a PPI, etc.

  The input Policy must be returned by SiliconPolicyDonePreMemory().

  1) In FSP path, the input Policy should be FspmUpd.
  A platform may use this API to update the FSPM UPD policy initialized
  by the silicon module or the default UPD data.
  The output of FSPM UPD data from this API is the final UPD data.

  2) In non-FSP path, the board may use additional way to get
  the silicon policy data field based upon the input Policy.

  @param[in, out] Policy       Pointer to policy.

  @return the updated policy.
**/
VOID *
EFIAPI
SiliconPolicyUpdatePreMemory (
  IN OUT VOID *Policy
  );
```

D.1.2.8 ReportPostMemFv

```
VOID
ReportPostMemFv (
  VOID
  );
```

D.1.2.9 BoardInitAfterMemoryInit

```
EFI_STATUS
EFIAPI
BoardInitAfterMemoryInit (
  VOID
  );
```

D.1.2.10 SetCacheMtrrAfterMemoryDiscovered

TODO: Add prototype

D.1.2.11 BoardInitBeforeSiliconInit

```
EFI_STATUS
EFIAPI
BoardInitBeforeSiliconInit (
  VOID
  );
```

D.1.2.12 SiliconPolicyUpdatePostMemory

```
/**
  Performs silicon post-memory policy update.

  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a PPI, etc.

  The input Policy must be returned by SiliconPolicyDonePostMemory().

  1) In FSP path, the input Policy should be FspsUpd.
  A platform may use this API to update the FSPS UPD policy initialized
  by the silicon module or the default UPD data.
  The output of FSPS UPD data from this API is the final UPD data.

  2) In non-FSP path, the board may use additional way to get
  the silicon policy data field based upon the input Policy.

  @param[in, out] Policy       Pointer to policy.

  @return the updated policy.
**/
VOID *
EFIAPI
SiliconPolicyUpdatePostMemory (
  IN OUT VOID *Policy
  );
```

D.1.2.13 BoardInitAfterSiliconInit

```
EFI_STATUS
EFIAPI
BoardInitAfterSiliconInit (
  VOID
  );
```

D.1.2.14 SetCacheMtrrAfterEndOfPei

```
/**
  Update MTRR setting and set write back as default memory attribute.
  @retval  EFI_SUCCESS  The function completes successfully.
  @retval  Others       Some error occurs.
**/
EFI_STATUS
EFIAPI
SetCacheMtrrAfterEndOfPei (
  VOID
  )
```

D.1.3 BoardPorting.DXE

D.1.3.1 SiliconPolicyUpdateLate

```
/**
  Performs silicon late policy update.

  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a Protocol, etc.

  The input Policy must be returned by SiliconPolicyDoneLate().

  In FSP or non-FSP path, the board may use additional way to get
  the silicon policy data field based upon the input Policy.

  @param[in, out] Policy       Pointer to policy.

  @return the updated policy.
**/
VOID *
EFIAPI
SiliconPolicyUpdateLate (
  IN OUT VOID *Policy
  );
```

D.1.3.2 PlatformBootManagerBeforeConsole (*)

```
/**
  Do the platform specific action before the console is connected.

  Such as:
    Update console variable;
    Register new Driver#### or Boot####;
    Signal ReadyToLock event.
**/
VOID
EFIAPI
PlatformBootManagerBeforeConsole (
  VOID
  );
```

D.1.3.3 BoardInitAfterPciEnumeration

```
EFI_STATUS
EFIAPI
BoardInitAfterPciEnumeration (
  VOID
  );
```

D.1.3.4 PlatformBootManagerAfterConsole(*)

```
/**
```

```
  Do the platform specific action after the console is connected.

  Such as:
    Dynamically switch output mode;
    Signal console ready platform customized event;
    Run diagnostics like memory testing;
    Connect certain devices;
    Dispatch additional option roms.
**/
VOID
EFIAPI
PlatformBootManagerAfterConsole (
  VOID
  );
```

D.1.3.5 BoardInitReadyToBoot

```
EFI_STATUS
EFIAPI
BoardInitReadyToBoot (
  VOID
  );
```

D.1.3.6 PlatformCreateAcpiTable

TODO: Add prototype

D.1.3.7 PlatformUpdateAcpiTable

TODO: Add prototype

D.1.3.8 PlatformInstallAcpiTable

TODO: Add prototype

D.1.3.9 BoardInitEndOfFirmware

```
EFI_STATUS
EFIAPI
BoardInitEndOfFirmware (
  VOID
  );
```

D.1.4 BoardPorting.SMM

D.1.4.1 BoardEnableAcpiCallback

TODO: Add prototype

D.1.5 SiliconPorting.SEC

D.1.5.1 TempRamInit

TODO: Add prototype

D.1.6 SiliconPorting.PEI

D.1.6.1 SiliconPolicyInitPreMemory

```
/**
  Performs silicon pre-memory policy initialization.
  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a PPI, etc.

  The returned data must be used as input data for SiliconPolicyDonePreMemory(),
  and SiliconPolicyUpdateLib.SiliconPolicyUpdatePreMemory().

  1) In FSP path, the input Policy should be FspmUpd.
  Value of FspmUpd has been initialized by FSP binary default value.
  Only a subset of FspmUpd needs to be updated for different silicon sku.
  The return data is same FspmUpd.

  2) In non-FSP path, the input policy could be NULL.
  The return data is the initialized policy.

  @param[in, out] Policy       Pointer to policy.

  @return the initialized policy.
**/
VOID *
EFIAPI
SiliconPolicyInitPreMemory (
  IN OUT VOID *Policy OPTIONAL
  );
```

D.1.6.2 SiliconPolicyDonePreMemory

```
/*
  The silicon pre-memory policy is finalized.
  Silicon code can do initialization based upon the policy data.

  The input Policy must be returned by SiliconPolicyInitPreMemory().

  @param[in] Policy       Pointer to policy.

  @retval RETURN_SUCCESS The policy is handled consumed by silicon code.
*/
RETURN_STATUS
EFIAPI
SiliconPolicyDonePreMemory (
  IN VOID *Policy
  );
```

D.1.6.3 MemoryInit

```
/**
  This function
    1. Calling MRC to initialize memory.
    2. Install EFI Memory.
    3. Capsule coalesce if capsule boot mode.
    4. Create HOB of system memory.

  @param  PeiServices Pointer to the PEI Service Table

  @retval EFI_SUCCESS If it completes successfully.

**/
EFI_STATUS
MemoryInit (
  IN EFI_PEI_SERVICES          **PeiServices
  );
```

D.1.6.4 SiliconPolicyInitPostMemory

```
/**
```

```
  Performs silicon post-memory policy initialization.

  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a PPI, etc.

  The returned data must be used as input data for SiliconPolicyDonePostMemory(),
  and SiliconPolicyUpdateLib.SiliconPolicyUpdatePostMemory().

  1) In FSP path, the input Policy should be FspsUpd.
  Value of FspsUpd has been initialized by FSP binary default value.
  Only a subset of FspsUpd needs to be updated for different silicon sku.
  The return data is same FspsUpd.

  2) In non-FSP path, the input policy could be NULL.
  The return data is the initialized policy.

  @param[in, out] Policy       Pointer to policy.

  @return the initialized policy.
**/
VOID *
EFIAPI
SiliconPolicyInitPostMemory (
  IN OUT VOID *Policy OPTIONAL
  );
```

D.1.6.5 SiliconPolicyDonePostMemory

```
/*
  The silicon post-mem policy is finalized.
  Silicon code can do initialization based upon the policy data.

  The input Policy must be returned by SiliconPolicyInitPostMemory().

  @param[in] Policy       Pointer to policy.

  @retval RETURN_SUCCESS The policy is handled consumed by silicon code.
*/
RETURN_STATUS
EFIAPI
SiliconPolicyDonePostMemory (
  IN VOID *Policy
  );
```

D.1.6.6 SiliconInit

TODO: Add prototype

D.1.7 SiliconPorting.DXE

D.1.7.1 SiliconPolicyInitLate

```
/**
  Performs silicon late policy initialization.

  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a protocol, etc.

  The returned data must be used as input data for SiliconPolicyDoneLate(),
  and SiliconPolicyUpdateLib.SiliconPolicyUpdateLate().

  In FSP or non-FSP path, the input policy could be NULL.
  The return data is the initialized policy.

  @param[in, out] Policy       Pointer to policy.
  @return the initialized policy.
**/
VOID *
EFIAPI
SiliconPolicyInitLate (
  IN OUT VOID *Policy
  );
```

D.1.7.2 SiliconPolicyDoneLate

```
/*
  The silicon late policy is finalized.
  Silicon code can do initialization based upon the policy data.

  The input Policy must be returned by SiliconPolicyInitLate().

  @param[in] Policy       Pointer to policy.

  @retval RETURN_SUCCESS The policy is handled consumed by silicon code.
*/
RETURN_STATUS
EFIAPI
SiliconPolicyDoneLate (
  IN VOID *Policy
  );
```

D.1.7.3 SiliconInitAfterPciEnumeration

TODO: Add prototype

D.1.8 SiliconPorting.SMM

D.1.8.1 PchSmmCoreDispatcher

```
/**
  The callback function to handle subsequent SMIs.  This callback will be called by SmmCoreDispatcher.

  @param[in] SmmImageHandle             Not used
  @param[in] PchSmmCore                 Not used
  @param[in, out] CommunicationBuffer   Not used
  @param[in, out] SourceSize            Not used

  @retval EFI_SUCCESS                   Function successfully completed
**/
EFI_STATUS
EFIAPI
PchSmmCoreDispatcher (
  IN       EFI_HANDLE         SmmImageHandle,
  IN CONST VOID               *PchSmmCore,
  IN OUT   VOID               *CommunicationBuffer,
  IN OUT   UINTN              *SourceSize
  );
```

D.1.9 Test.DXE

D.1.9.1 ExitPmAuth

```
VOID
ExitPmAuth (
  VOID
  );
```

D.1.10 Debug.SEC

D.1.10.1 SecStartup (*)

```
/**
  Entry point to the C language phase of SEC. After the SEC assembly
  code has initialized some temporary memory and set up the stack,
  the control is transferred to this function.

  @param SizeOfRam           Size of the temporary memory available for use.
  @param TempRamBase         Base address of temporary ram
  @param BootFirmwareVolume  Base address of the Boot Firmware Volume.
**/
VOID
EFIAPI
SecStartup (
  IN UINT32                   SizeOfRam,
  IN UINT32                   TempRamBase,
  IN VOID                     *BootFirmwareVolume
  );
```

D.1.10.2 SecStartupPhase2 (*)

```
/**
  Caller provided function to be invoked at the end of InitializeDebugAgent().

  Entry point to the C language phase of SEC. After the SEC assembly
  code has initialized some temporary memory and set up the stack,
  the control is transferred to this function.

  @param[in] Context    The first input parameter of InitializeDebugAgent().

**/
VOID
NORETURN
EFIAPI
SecStartupPhase2 (
  IN VOID                     *Context
  );
```

D.1.11 Debug.PEI

D.1.11.1 PeiCore (*)

```
/**
  This routine is invoked by main entry of PeiMain module during transition
  from SEC to PEI. After switching stack in the PEI core, it will restart
  with the old core data.

  @param SecCoreDataPtr  Points to a data structure containing information about the PEI core's operating
                         environment, such as the size and location of temporary RAM, the stack location and
                         the BFV location.
  @param PpiList         Points to a list of one or more PPI descriptors to be installed initially by the PEI core.
                         An empty PPI list consists of a single descriptor with the end-tag
                         EFI_PEI_PPI_DESCRIPTOR_TERMINATE_LIST. As part of its initialization
                         phase, the PEI Foundation will add these SEC-hosted PPIs to its PPI database such
                         that both the PEI Foundation and any modules can leverage the associated service
                         calls and/or code in these early PPIs
  @param Data            Pointer to old core data that is used to initialize the
                         core's data areas.
                         If NULL, it is first PeiCore entering.

**/
VOID
```

```
EFIAPI
PeiCore (
  IN CONST EFI_SEC_PEI_HAND_OFF        *SecCoreDataPtr,
  IN CONST EFI_PEI_PPI_DESCRIPTOR      *PpiList,
  IN VOID                              *Data
  );
```

D.1.11.2 PeiDispatcher (*)

```
/**
  Conduct PEIM dispatch.

  @param SecCoreData     Pointer to the data structure containing SEC to PEI handoff data
  @param PrivateData     Pointer to the private data passed in from caller

**/
VOID
PeiDispatcher (
  IN CONST EFI_SEC_PEI_HAND_OFF  *SecCoreData,
  IN PEI_CORE_INSTANCE           *PrivateData
  );
```

D.1.11.3 SecTemporaryRamDone(*)

```
/**
  TemporaryRamDone() disables the use of Temporary RAM. If present, this service is invoked
  by the PEI Foundation after the EFI_PEI_PERMANANT_MEMORY_INSTALLED_PPI is installed.

  @retval EFI_SUCCESS           Use of Temporary RAM was disabled.
  @retval EFI_INVALID_PARAMETER Temporary RAM could not be disabled.

**/
EFI_STATUS
EFIAPI
SecTemporaryRamDone (
  VOID
  );
```

D.1.11.4 DxeLoadCore (*)

```
/**
   Main entry point to last PEIM

   @param This          Entry point for DXE IPL PPI
   @param PeiServices   General purpose services available to every PEIM.
   @param HobList       Address to the Pei HOB list

   @return EFI_SUCCESS              DXE core was successfully loaded.
   @return EFI_OUT_OF_RESOURCES     There are not enough resources to load DXE core.

**/
EFI_STATUS
EFIAPI
DxeLoadCore (
  IN CONST EFI_DXE_IPL_PPI *This,
  IN EFI_PEI_SERVICES      **PeiServices,
  IN EFI_PEI_HOB_POINTERS  HobList
  );
```

D.1.12 Debug.DXE

```
#### D.1.12.1 DxeMain (*)
```

~~~~
```c
/**
  Main entry point to DXE Core.

  @param  HobStart               Pointer to the beginning of the HOB List from PEI.

  @return This function should never return.

**/
VOID
EFIAPI
DxeMain (
  IN  VOID *HobStart
  );
~~~~

D.1.12.2 CoreStartImage (*)

```
/**
  Transfer control to a loaded image's entry point.

  @param  ImageHandle             Handle of image to be started.
  @param  ExitDataSize            Pointer of the size to ExitData
  @param  ExitData                Pointer to a pointer to a data buffer that
                                  includes a Null-terminated string,
                                  optionally followed by additional binary data.
                                  The string is a description that the caller may
                                  use to further indicate the reason for the
                                  image's exit.

  @retval EFI_INVALID_PARAMETER   Invalid parameter
  @retval EFI_OUT_OF_RESOURCES    No enough buffer to allocate
  @retval EFI_SECURITY_VIOLATION  The current platform policy specifies that the image should not be started.
  @retval EFI_SUCCESS             Successfully transfer control to the image's
                                  entry point.

**/
EFI_STATUS
EFIAPI
CoreStartImage (
  IN EFI_HANDLE  ImageHandle,
  OUT UINTN      *ExitDataSize,
  OUT CHAR16     **ExitData  OPTIONAL
  );
```

D.1.12.3 CoreAllEfiServicesAvailable (*)

```
/**
  Return TRUE if all AP services are available.

  @retval EFI_SUCCESS    All AP services are available
  @retval EFI_NOT_FOUND  At least one AP service is not available

**/
EFI_STATUS
CoreAllEfiServicesAvailable (
  VOID
  );
```

D.1.12.4 BdsEntry (*)

```
/**

  Service routine for BdsInstance->Entry(). Devices are connected, the
  consoles are initialized, and the boot options are tried.
```

```
  @param This           Protocol Instance structure.

**/
VOID
EFIAPI
BdsEntry (
  IN  EFI_BDS_ARCH_PROTOCOL *This
  );
```

IN EFI_BDS_ARCH_PROTOCOL ``*This

`);`

D.1.12.5 EfiBootManagerDispatchDeferredImages (*)

```
/**
  Dispatch the deferred images that are returned from all DeferredImageLoad instances.

  @retval EFI_SUCCESS       At least one deferred image is loaded successfully and started.
  @retval EFI_NOT_FOUND     There is no deferred image.
  @retval EFI_ACCESS_DENIED There are deferred images but all of them are failed to load.
**/
EFI_STATUS
EFIAPI
EfiBootManagerDispatchDeferredImages (
  VOID
  );
```

D.1.12.6 BootBootOptions(*)

```
/**
  Attempt to boot each boot option in the BootOptions array.

  @param BootOptions       Input boot option array.
  @param BootOptionCount   Input boot option count.
  @param BootManagerMenu   Input boot manager menu.

  @retval TRUE  Successfully boot one of the boot options.
  @retval FALSE Failed boot any of the boot options.
**/
BOOLEAN
BootBootOptions (
  IN EFI_BOOT_MANAGER_LOAD_OPTION    *BootOptions,
  IN UINTN                           BootOptionCount,
  IN EFI_BOOT_MANAGER_LOAD_OPTION    *BootManagerMenu OPTIONAL
  );
```

D.1.12.7 EfiSignalEventReadyToBoot (*)

```
/**
  Create, Signal, and Close the Ready to Boot event using EfiSignalEventReadyToBoot().

  This function abstracts the signaling of the Ready to Boot Event. The Framework moved
  from a proprietary to UEFI 2.0 based mechanism. This library abstracts the caller
  from how this event is created to prevent to code form having to change with the
  version of the specification supported.

**/
VOID
EFIAPI
EfiSignalEventReadyToBoot (
  VOID
  );
```

D.1.12.8 UefiMain (*)

```
/**
  The entry point for the application.

  @param[in] ImageHandle    The firmware allocated handle for the EFI image.
  @param[in] SystemTable    A pointer to the EFI System Table.

  @retval EFI_SUCCESS       The entry point is executed successfully.
  @retval other             Some error occurs when executing this entry point.

**/
EFI_STATUS
EFIAPI
UefiMain (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  );
```

D.1.12.9 CoreExitBootServices (*)

```
/**
  Terminates all boot services.

  @param  ImageHandle           Handle that identifies the exiting image.
  @param  MapKey                Key to the latest memory map.

  @retval EFI_SUCCESS           Boot Services terminated
  @retval EFI_INVALID_PARAMETER  MapKey is incorrect.

**/
EFI_STATUS
EFIAPI
CoreExitBootServices (
  IN EFI_HANDLE   ImageHandle,
  IN UINTN        MapKey
  );
```

D.1.12.10 RuntimeDriverSetVirtualAddressMap (*)

```
/**
  Changes the runtime addressing mode of EFI firmware from physical to virtual.

  @param  MemoryMapSize   The size in bytes of VirtualMap.
  @param  DescriptorSize  The size in bytes of an entry in the VirtualMap.
  @param  DescriptorVersion The version of the structure entries in VirtualMap.
  @param  VirtualMap      An array of memory descriptors which contain new virtual
                         address mapping information for all runtime ranges.

  @retval  EFI_SUCCESS            The virtual address map has been applied.
  @retval  EFI_UNSUPPORTED        EFI firmware is not at runtime, or the EFI firmware is already in
                                  virtual address mapped mode.
  @retval  EFI_INVALID_PARAMETER  DescriptorSize or DescriptorVersion is invalid.
  @retval  EFI_NO_MAPPING         A virtual address was not supplied for a range in the memory
                                  map that requires a mapping.
  @retval  EFI_NOT_FOUND          A virtual address was supplied for an address that is not found
                                  in the memory map.

**/
EFI_STATUS
EFIAPI
RuntimeDriverSetVirtualAddressMap (
  IN UINTN                  MemoryMapSize,
  IN UINTN                  DescriptorSize,
  IN UINT32                 DescriptorVersion,
  IN EFI_MEMORY_DESCRIPTOR  *VirtualMap
  );
```

D.1.13 Debug.SMM

D.1.13.1 SmmIplEntry (*)

```
/**
  The Entry Point for SMM IPL

  Load SMM Core into SMRAM, register SMM Core entry point for SMIs, install
  SMM Base 2 Protocol and SMM Communication Protocol, and register for the
  critical events required to coordinate between DXE and SMM environments.

  @param  ImageHandle    The firmware allocated handle for the EFI image.
  @param  SystemTable    A pointer to the EFI System Table.

  @retval EFI_SUCCESS    The entry point is executed successfully.
  @retval Other          Some error occurred when executing this entry point.

**/
EFI_STATUS
EFIAPI
SmmIplEntry (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  );
```

D.1.D.1 SmmMain (*)

```
/**
  The Entry Point for SMM Core

  Install DXE Protocols and reload SMM Core into SMRAM and register SMM Core
  EntryPoint on the SMI vector.

  Note: This function is called for both DXE invocation and SMRAM invocation.

  @param  ImageHandle    The firmware allocated handle for the EFI image.
  @param  SystemTable    A pointer to the EFI System Table.

  @retval EFI_SUCCESS    The entry point is executed successfully.
  @retval Other          Some error occurred when executing this entry point.

**/
EFI_STATUS
EFIAPI
SmmMain (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  );
```

D.1.13.3 PiCpuSmmEntry (*)

```
/**
  The module Entry Point of the CPU SMM driver.

  @param  ImageHandle    The firmware allocated handle for the EFI image.
  @param  SystemTable    A pointer to the EFI System Table.

  @retval EFI_SUCCESS    The entry point is executed successfully.
  @retval Other          Some error occurs when executing this entry point.

**/
EFI_STATUS
```

```
EFIAPI
PiCpuSmmEntry (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  );
```

D.1.13.4 SmmRelocateBases (*)

```
/**
  Relocate SmmBases for each processor.

  Execute on first boot and all S3 resumes

**/
VOID
EFIAPI
SmmRelocateBases (
  VOID
  );
```

D.1.13.5 _SmiEntryPoint (*)

TODO: Add prototype

D.1.13.6 SmmEntryPoint (*)

```
/**
  The main entry point to SMM Foundation.

  Note: This function is only used by SMRAM invocation.  It is never used by DXE invocation.

  @param  SmmEntryContext           Processor information and functionality
                                    needed by SMM Foundation.

**/
VOID
EFIAPI
SmmEntryPoint (
  IN CONST EFI_SMM_ENTRY_CONTEXT  *SmmEntryContext
  );
```

D.1.13.7 PlatformEnableAcpiCallback

TODO: Add prototype

D.2 BoardInit

D.2.1 BoardInitSupportLib

```
/** @file

Copyright (c) 2017, Intel Corporation. All rights reserved.<BR>
This program and the accompanying materials are licensed and made available under
the terms and conditions of the BSD License that accompanies this distribution.
The full text of the license may be found at
http://opensource.org/licenses/bsd-license.php.

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED.

**/

#ifndef _BOARD_INIT_LIB_H_
#define _BOARD_INIT_LIB_H_

#include <PiPei.h>
#include <Uefi.h>

EFI_STATUS
EFIAPI
BoardDetect (
  VOID
  );

EFI_STATUS
EFIAPI
BoardDebugInit (
  VOID
  );

EFI_BOOT_MODE
EFIAPI
BoardBootModeDetect (
  VOID
  );

EFI_STATUS
EFIAPI
BoardInitBeforeMemoryInit (
  VOID
  );

EFI_STATUS
EFIAPI
BoardInitAfterMemoryInit (
  VOID
  );

EFI_STATUS
EFIAPI
BoardInitBeforeTempRamExit (
  VOID
  );

EFI_STATUS
EFIAPI
BoardInitAfterTempRamExit (
  VOID
  );

EFI_STATUS
EFIAPI
BoardInitBeforeSiliconInit (
  VOID
  );

EFI_STATUS
EFIAPI
BoardInitAfterSiliconInit (
  VOID
  );

EFI_STATUS
EFIAPI
BoardInitAfterPciEnumeration (
  VOID
  );

EFI_STATUS
EFIAPI
BoardInitReadyToBoot (
  VOID
  );

EFI_STATUS
EFIAPI
BoardInitEndOfFirmware (
  VOID
  );

#endif
```

D.2.2 MultiBoardInitSupportLib

```
/** @file

Copyright (c) 2017, Intel Corporation. All rights reserved.<BR>
This program and the accompanying materials are licensed and made available under
the terms and conditions of the BSD License that accompanies this distribution.
The full text of the license may be found at
http://opensource.org/licenses/bsd-license.php.

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED.

**/

#ifndef _MULTI_BOARD_INIT_SUPPORT_LIB_H_
#define _MULTI_BOARD_INIT_SUPPORT_LIB_H_

#include <Library/BoardInitLib.h>

typedef
EFI_STATUS
(EFIAPI *BOARD_DETECT) (
  VOID
  );

typedef
EFI_STATUS
(EFIAPI *BOARD_INIT) (
  VOID
  );

typedef
EFI_BOOT_MODE
(EFIAPI *BOARD_BOOT_MODE_DETECT) (
  VOID
  );

typedef struct {
  BOARD_DETECT  BoardDetect;
} BOARD_DETECT_FUNC;
```

```
typedef struct {
  BOARD_INIT              BoardDebugInit;
  BOARD_BOOT_MODE_DETECT  BoardBootModeDetect;
  BOARD_INIT              BoardInitBeforeMemoryInit;
  BOARD_INIT              BoardInitAfterMemoryInit;
  BOARD_INIT              BoardInitBeforeTempRamExit;
  BOARD_INIT              BoardInitAfterTempRamExit;
} BOARD_PRE_MEM_INIT_FUNC;

typedef struct {
  BOARD_INIT              BoardInitBeforeSiliconInit;
  BOARD_INIT              BoardInitAfterSiliconInit;
} BOARD_POST_MEM_INIT_FUNC;

typedef struct {
  BOARD_INIT              BoardInitAfterPciEnumeration;
  BOARD_INIT              BoardInitReadyToBoot;
  BOARD_INIT              BoardInitEndOfFirmware;
} BOARD_NOTIFICATION_INIT_FUNC;

EFI_STATUS
EFIAPI
RegisterBoardDetect (
  IN BOARD_DETECT_FUNC  *BoardDetect
  );

EFI_STATUS
EFIAPI
RegisterBoardPreMemoryInit (
  IN BOARD_PRE_MEM_INIT_FUNC  *BoardPreMemoryInit
  );

EFI_STATUS
EFIAPI
RegisterBoardPostMemoryInit (
  IN BOARD_POST_MEM_INIT_FUNC  *BoardPostMemoryInit
  );

EFI_STATUS
EFIAPI
RegisterBoardNotificationInit (
  IN BOARD_NOTIFICATION_INIT_FUNC  *BoardNotificationInit
  );

#endif
```

D.3 SiliconPolicyInit

D.3.1 SiliconPolicyInitLib

The SiliconPolicyInitLib provides functions that silicon code initializes the default policy.

```
/** @file

Copyright (c) 2017, Intel Corporation. All rights reserved.<BR>
This program and the accompanying materials are licensed and made available under
the terms and conditions of the BSD License that accompanies this distribution.
The full text of the license may be found at
http://opensource.org/licenses/bsd-license.php.

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED.

**/

#ifndef _SILICON_POLICY_INIT_LIB_H_
#define _SILICON_POLICY_INIT_LIB_H_

/**
  Performs silicon pre-memory policy initialization.

  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a PPI, etc.

  The returned data must be used as input data for SiliconPolicyDonePreMemory(),
  and SiliconPolicyUpdateLib.SiliconPolicyUpdatePreMemory().

  1) In FSP path, the input Policy should be FspmUpd.
  Value of FspmUpd has been initialized by FSP binary default value.
  Only a subset of FspmUpd needs to be updated for different silicon sku.
  The return data is same FspmUpd.

  2) In non-FSP path, the input policy could be NULL.
  The return data is the initialized policy.

  @param[in, out] Policy       Pointer to policy.

  @return the initialized policy.
**/
VOID *
EFIAPI
SiliconPolicyInitPreMemory (
  IN OUT VOID *Policy OPTIONAL
  );

/*
  The silicon pre-mem policy is finalized.
  Silicon code can do initialization based upon the policy data.

  The input Policy must be returned by SiliconPolicyInitPreMemory().

  @param[in] Policy       Pointer to policy.

  @retval RETURN_SUCCESS The policy is handled consumed by silicon code.
*/
RETURN_STATUS
EFIAPI
SiliconPolicyDonePreMemory (
  IN VOID *Policy
  );

/**
  Performs silicon post-memory policy initialization.
  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a PPI, etc.

  The returned data must be used as input data for SiliconPolicyDonePostMemory(),
  and SiliconPolicyUpdateLib.SiliconPolicyUpdatePostMemory().

  1) In FSP path, the input Policy should be FspsUpd.
  Value of FspsUpd has been initialized by FSP binary default value.
  Only a subset of FspsUpd needs to be updated for different silicon sku.
  The return data is same FspsUpd.

  2) In non-FSP path, the input policy could be NULL.
  The return data is the initialized policy.

  @param[in, out] Policy       Pointer to policy.

  @return the initialized policy.
**/
VOID *
EFIAPI
SiliconPolicyInitPostMemory (
  IN OUT VOID *Policy OPTIONAL
  );

/*
  The silicon post-memory policy is finalized.
  Silicon code can do initialization based upon the policy data.

  The input Policy must be returned by SiliconPolicyInitPostMemory().

  @param[in] Policy       Pointer to policy.

  @retval RETURN_SUCCESS The policy is handled consumed by silicon code.
*/
RETURN_STATUS
EFIAPI
SiliconPolicyDonePostMemory (
  IN VOID *Policy
  );

/**
  Performs silicon late policy initialization.

  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a protocol, etc.

  The returned data must be used as input data for SiliconPolicyDoneLate(),
  and SiliconPolicyUpdateLib.SiliconPolicyUpdateLate().

  In FSP or non-FSP path, the input policy could be NULL.
  The return data is the initialized policy.

  @param[in, out] Policy       Pointer to policy.

  @return the initialized policy.
**/
VOID *
EFIAPI
SiliconPolicyInitLate (
  IN OUT VOID *Policy OPTIONAL
  );

/*
  The silicon late policy is finalized.
  Silicon code can do initialization based upon the policy data.

  The input Policy must be returned by SiliconPolicyInitLate().

  @param[in] Policy       Pointer to policy.

  @retval RETURN_SUCCESS The policy is handled consumed by silicon code.
*/
```

```
RETURN_STATUS
EFIAPI
SiliconPolicyDoneLate (
  IN VOID *Policy
  );

#endif
```

D.3.2 SiliconPolicyUpdateLib

The SiliconPolicyUpdateLib provides functions that board code overrides the default policy.

```
/** @file

Copyright (c) 2017, Intel Corporation. All rights reserved.<BR>
This program and the accompanying materials are licensed and made available under
the terms and conditions of the BSD License that accompanies this distribution.
The full text of the license may be found at
http://opensource.org/licenses/bsd-license.php.

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED.

**/

#ifndef _SILICON_POLICY_UPDATE_LIB_H_
#define _SILICON_POLICY_UPDATE_LIB_H_

/**
  Performs silicon pre-memory policy update.

  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a PPI, etc.

  The input Policy must be returned by SiliconPolicyDonePreMemory().

  1) In FSP path, the input Policy should be FspmUpd.
  A platform may use this API to update the FSPM UPD policy initialized
  by the silicon module or the default UPD data.
  The output of FSPM UPD data from this API is the final UPD data.

  2) In non-FSP path, the board may use additional way to get
  the silicon policy data field based upon the input Policy.

  @param[in, out] Policy       Pointer to policy.

  @return the updated policy.
**/
VOID *
EFIAPI
SiliconPolicyUpdatePreMemory (
  IN OUT VOID *Policy
  );

/**
  Performs silicon post-memory policy update.

  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a PPI, etc.

  The input Policy must be returned by SiliconPolicyDonePostMemory().

  1) In FSP path, the input Policy should be FspsUpd.
  A platform may use this API to update the FSPS UPD policy initialized
  by the silicon module or the default UPD data.
  The output of FSPS UPD data from this API is the final UPD data.

  2) In non-FSP path, the board may use additional way to get
  the silicon policy data field based upon the input Policy.
  @param[in, out] Policy       Pointer to policy.

  @return the updated policy.
**/
VOID *
EFIAPI
SiliconPolicyUpdatePostMemory (
  IN OUT VOID *Policy
  );

/**
  Performs silicon late policy update.

  The meaning of Policy is defined by silicon code.
  It could be the raw data, a handle, a Protocol, etc.

  The input Policy must be returned by SiliconPolicyDoneLate().

  In FSP or non-FSP path, the board may use additional way to get
  the silicon policy data field based upon the input Policy.

  @param[in, out] Policy       Pointer to policy.

  @return the updated policy.
**/
VOID *
EFIAPI
SiliconPolicyUpdateLate (
  IN OUT VOID *Policy
  );

#endif
```

D.4 TestPoint

D.4.1 TestPointLib

The TestPointLib provides helper functions for implementing test points. This library is optional.

```
/** @file

Copyright (c) 2017, Intel Corporation. All rights reserved.<BR>
This program and the accompanying materials are licensed and made available under
the terms and conditions of the BSD License that accompanies this distribution.
The full text of the license may be found at
http://opensource.org/licenses/bsd-license.php.

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED.

**/

#ifndef _TEST_POINT_LIB_H_
#define _TEST_POINT_LIB_H_

#include <PiPei.h>
#include <Uefi.h>

//
// Below is Test Point report definition.
//

//
// We reuse HSTI stype definition.
// ADAPTER_INFO_PLATFORM_TEST_POINT is similar to ADAPTER_INFO_PLATFORM_SECURITY.
//

#define PLATFORM_TEST_POINT_VERSION               0x00000001

#define PLATFORM_TEST_POINT_ROLE_PLATFORM_REFERENCE 0x00000001
#define PLATFORM_TEST_POINT_ROLE_PLATFORM_IBV       0x00000002
#define PLATFORM_TEST_POINT_ROLE_IMPLEMENTOR_OEM    0x00000003
#define PLATFORM_TEST_POINT_ROLE_IMPLEMENTOR_ODM    0x00000004

#define TEST_POINT_FEATURES_ITEM_NUMBER 2
```

D.4.1.1 ADAPTER_INFO_PLATFORM_TEST_POINT

```
typedef struct {
  UINT32  Version;
  UINT32  Role;
  CHAR16  ImplementationID[256];
  UINT32  FeaturesSize;
//UINT8   FeaturesImplemented[];
//UINT8   FeaturesVerified[];
//CHAR16  ErrorString[];
} ADAPTER_INFO_PLATFORM_TEST_POINT;

//
// Below is test point report library
//
```

D.4.1.2 TestPointLibSetTable

```
/**
  Publish TestPoint table in AIP protocol.
  One system should have only one PLATFORM_TEST_POINT_ROLE_PLATFORM_REFERENCE.

  @param TestPoint      TestPoint data
  @param TestPointSize  TestPoint size

  @retval EFI_SUCCESS          The TestPoint data is published in AIP protocol.
  @retval EFI_ALREADY_STARTED  There is already TestPoint table with Role and ImplementationID published in system.
  @retval EFI_VOLUME_CORRUPTED The input TestPoint data is invalid.
  @retval EFI_OUT_OF_RESOURCES There is not enough system resource to publish TestPoint data in AIP protocol.
**/
EFI_STATUS
EFIAPI
TestPointLibSetTable (
  IN VOID                     *TestPoint,
  IN UINTN                    TestPointSize
  );
```

D.4.1.3 TestPointLibGetTable

```
/**
  Search TestPoint table in AIP protocol, and return the data.
  This API will return the TestPoint table with indicated Role and ImplementationID,
  NULL ImplementationID means to find the first TestPoint table with indicated Role.

  @param Role             Role of TestPoint data.
  @param ImplementationID ImplementationID of TestPoint data.
                          NULL means find the first one match Role.
  @param TestPoint        TestPoint data. This buffer is allocated by callee, and it
                          is the responsibility of the caller to free it after
                          using it.
  @param TestPointSize    TestPoint size

  @retval EFI_SUCCESS          The TestPoint data in AIP protocol is returned.
  @retval EFI_NOT_FOUND        There is not TestPoint table with the Role and ImplementationID published in system.
**/
EFI_STATUS
EFIAPI
TestPointLibGetTable (
  IN UINT32                   Role,
  IN CHAR16                   *ImplementationID OPTIONAL,
  OUT VOID                    **TestPoint,
  OUT UINTN                   *TestPointSize
  );
```

D.4.1.4 TestPointLibSetFeaturesVerified

```
/**
  Set FeaturesVerified in published TestPoint table.
  This API will update the TestPoint table with indicated Role and ImplementationID,
  NULL ImplementationID means to find the first TestPoint table with indicated Role.

  @param Role             Role of TestPoint data.
  @param ImplementationID ImplementationID of TestPoint data.
                          NULL means find the first one match Role.
  @param ByteIndex        Byte index of FeaturesVerified of TestPoint data.
  @param BitMask          Bit mask of FeaturesVerified of TestPoint data.

  @retval EFI_SUCCESS          The FeaturesVerified of TestPoint data updated in AIP protocol.
  @retval EFI_NOT_STARTED      There is not TestPoint table with the Role and ImplementationID published in system.
  @retval EFI_UNSUPPORTED      The ByteIndex is invalid.
**/
EFI_STATUS
EFIAPI
TestPointLibSetFeaturesVerified (
  IN UINT32                   Role,
  IN CHAR16                   *ImplementationID, OPTIONAL
  IN UINT32                   ByteIndex,
  IN UINT8                    BitMask
  );
```

D.4.1.5 TestPointLibClearFeaturesVerified

```
/**
  Clear FeaturesVerified in published TestPoint table.
  This API will update the TestPoint table with indicated Role and ImplementationID,
  NULL ImplementationID means to find the first TestPoint table with indicated Role.

  @param Role             Role of TestPoint data.
  @param ImplementationID ImplementationID of TestPoint data.
                          NULL means find the first one match Role.
  @param ByteIndex        Byte index of FeaturesVerified of TestPoint data.
  @param BitMask          Bit mask of FeaturesVerified of TestPoint data.

  @retval EFI_SUCCESS          The FeaturesVerified of TestPoint data updated in AIP protocol.
  @retval EFI_NOT_STARTED      There is not TestPoint table with the Role and ImplementationID published in system.
  @retval EFI_UNSUPPORTED      The ByteIndex is invalid.
**/
EFI_STATUS
EFIAPI
TestPointLibClearFeaturesVerified (
  IN UINT32                   Role,
  IN CHAR16                   *ImplementationID, OPTIONAL
  IN UINT32                   ByteIndex,
  IN UINT8                    BitMask
  );
```

D.4.1.6 TestPointLibAppendErrorString

```
/**
  Append ErrorString in published TestPoint table.
  This API will update the TestPoint table with indicated Role and ImplementationID,
  NULL ImplementationID means to find the first TestPoint table with indicated Role.

  @param Role             Role of TestPoint data.
  @param ImplementationID ImplementationID of TestPoint data.
                          NULL means find the first one match Role.
  @param ErrorString      ErrorString of TestPoint data.

  @retval EFI_SUCCESS          The ErrorString of TestPoint data is updated in AIP protocol.
  @retval EFI_NOT_STARTED      There is not TestPoint table with the Role and ImplementationID published in system.
  @retval EFI_OUT_OF_RESOURCES There is not enough system resource to update ErrorString.
**/
EFI_STATUS
EFIAPI
TestPointLibAppendErrorString (
  IN UINT32                   Role,
  IN CHAR16                   *ImplementationID, OPTIONAL
  IN CHAR16                   *ErrorString
  );
```

D.4.1.7 TestPointLibSetErrorString

```
/**
  Set a new ErrorString in published TestPoint table.
  This API will update the TestPoint table with indicated Role and ImplementationID,
  NULL ImplementationID means to find the first TestPoint table with indicated Role.

  @param Role             Role of TestPoint data.
  @param ImplementationID ImplementationID of TestPoint data.
                          NULL means find the first one match Role.
  @param ErrorString      ErrorString of TestPoint data.
  @retval EFI_SUCCESS          The ErrorString of TestPoint data is updated in AIP protocol.
  @retval EFI_NOT_STARTED      There is not TestPoint table with the Role and ImplementationID published in system.
  @retval EFI_OUT_OF_RESOURCES There is not enough system resource to update ErrorString.
**/
EFI_STATUS
EFIAPI
TestPointLibSetErrorString (
  IN UINT32                  Role,
  IN CHAR16                  *ImplementationID, OPTIONAL
  IN CHAR16                  *ErrorString
  );

//
// TEST POINT SMM Communication command
//
#define SMI_HANDLER_TEST_POINT_COMMAND_GET_INFO           0x1
#define SMI_HANDLER_TEST_POINT_COMMAND_GET_DATA_BY_OFFSET 0x2

typedef struct {
  UINT32                             Command;
  UINT32                             DataLength;
  UINT64                             ReturnStatus;
} SMI_HANDLER_TEST_POINT_PARAMETER_HEADER;

typedef struct {
  SMI_HANDLER_TEST_POINT_PARAMETER_HEADER    Header;
  UINT64                                     DataSize;
} SMI_HANDLER_TEST_POINT_PARAMETER_GET_INFO;

typedef struct {
  SMI_HANDLER_TEST_POINT_PARAMETER_HEADER    Header;
  //
  // On input, data buffer size.
  // On output, actual data buffer size copied.
  //
  UINT64                                     DataSize;
  PHYSICAL_ADDRESS                           DataBuffer;
  //
  // On input, data buffer offset to copy.
  // On output, next time data buffer offset to copy.
  //
  UINT64                                     DataOffset;
} SMI_HANDLER_TEST_POINT_PARAMETER_GET_DATA_BY_OFFSET;

extern EFI_GUID gAdapterInfoPlatformTestPointGuid;

#endif
```

D.4.2 TestPointCheckLib

```
/** @file

Copyright (c) 2017, Intel Corporation. All rights reserved.<BR>
This program and the accompanying materials are licensed and made available under
the terms and conditions of the BSD License that accompanies this distribution.
The full text of the license may be found at
http://opensource.org/licenses/bsd-license.php.

THE PROGRAM IS DISTRIBUTED UNDER THE BSD LICENSE ON AN "AS IS" BASIS,
WITHOUT WARRANTIES OR REPRESENTATIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED.

**/

#ifndef _TEST_POINT_CHECK_LIB_H_
#define _TEST_POINT_CHECK_LIB_H_

#include <PiPei.h>
#include <Uefi.h>
```

D.4.2.1 Test Point Hook Points

```
//
// Below is Test Point Hook Point.
//
// Naming: TestPoint<Phase/Event><Function>
//
// Phase/Event(PEI) = MemoryDiscovered|EndOfPei
// Phase/Event(DXE) = PciEnumerationDone|EndOfDxe|DxeSmmReadyToLock|ReadyToBoot
// Phase/Event(SMM) = SmmEndOfDxe|SmmReadyToLock|SmmReadyToBoot
//
EFI_STATUS
EFIAPI
TestPointTempMemoryFunction (
  IN VOID   *TempRamStart,
  IN VOID   *TempRamEnd
  );

EFI_STATUS
EFIAPI
TestPointDebugInitDone (
  VOID
  );


EFI_STATUS
EFIAPI
TestPointMemoryDiscoveredMtrrFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointMemoryDiscoveredMemoryResourceFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointMemoryDiscoveredFvInfoFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointMemoryDiscoveredDmaProtectionEnabled (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointEndOfPeiSystemResourceFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointEndOfPeiMtrrFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointEndOfPeiPciBusMasterDisabled (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointPciEnumerationDonePciBusMasterDisabled (
  VOID
  );
```

```
EFI_STATUS
EFIAPI
TestPointPciEnumerationDonePciResourceAllocated (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointEndOfDxeNoThirdPartyPciOptionRom (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointEndOfDxeDmaAcpiTableFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointEndOfDxeDmaProtectionEnabled (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointDxeSmmReadyToLockSmramAligned (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointDxeSmmReadyToLockWsmtTableFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointSmmReadyToBootSmmPageProtection (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointReadyToBootAcpiTableFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointReadyToBootGcdResourceFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointReadyToBootMemoryTypeInformationFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointReadyToBootUefiMemoryAttributeTableFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointReadyToBootUefiBootVariableFunctional (
  VOID
  );
```

```
EFI_STATUS
EFIAPI
TestPointReadyToBootUefiConsoleVariableFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointReadyToBootHstiTableFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointReadyToBootEsrtTableFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointReadyToBootUefiSecureBootEnabled (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointReadyToBootPiSignedFvBootEnabled (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointReadyToBootTcgTrustedBootEnabled (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointReadyToBootTcgMorEnabled (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointDxeSmmReadyToBootSmiHandlerInstrument (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointExitBootServices (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointSmmEndOfDxeSmrrFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointSmmReadyToLockSmmMemoryAttributeTableFunctional (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointSmmReadyToLockSecureSmmCommunicationBuffer (
  VOID
  );
```

```
EFI_STATUS
EFIAPI
TestPointSmmReadyToBootSmmPageProtection (
  VOID
  );

EFI_STATUS
EFIAPI
TestPointSmmExitBootServices (
  VOID
  );
```

D.4.2.2 MinPlatformPkg Macro Definitions

```
//
// Below is detail definition for MinPlatform implementation
//
#define TEST_POINT_IMPLEMENTATION_ID_PLATFORM         L"Intel MinPlatform TestPoint"
#define TEST_POINT_IMPLEMENTATION_ID_PLATFORM_PEI     TEST_POINT_IMPLEMENTATION_ID_PLATFORM L" (PEI)"
#define TEST_POINT_IMPLEMENTATION_ID_PLATFORM_DXE     TEST_POINT_IMPLEMENTATION_ID_PLATFORM L" (DXE)"
#define TEST_POINT_IMPLEMENTATION_ID_PLATFORM_SMM     TEST_POINT_IMPLEMENTATION_ID_PLATFORM L" (SMM)"

#define TEST_POINT_FEATURE_SIZE           0x10

#define TEST_POINT_ERROR                  L"Error "
#define TEST_POINT_PLATFORM_TEST_POINT    L" Platform TestPoint"

// Byte 0 - SEC/PEI
#define TEST_POINT_TEMP_MEMORY_INIT_DONE  L" - Temp Memory Init Done - "
#define TEST_POINT_DEBUG_INIT_DONE        L" - Debug Init Done - "

#define TEST_POINT_BYTE0_TEMP_INIT_DONE   BIT0
#define TEST_POINT_BYTE0_DEBUG_INIT_DONE  BIT1

// Byte 1/2 - PEI
#define TEST_POINT_MEMORY_DISCOVERED      L" - Memory Discovered - "
#define TEST_POINT_END_OF_PEI             L" - End Of PEI - "

#define TEST_POINT_BYTE1_MEMORY_DISCOVERED_MTRR_FUNCTIONAL                       BIT0
#define TEST_POINT_BYTE1_MEMORY_DISCOVERED_MEMORY_RESOURCE_FUNCTIONAL            BIT1
#define TEST_POINT_BYTE1_MEMORY_DISCOVERED_FV_INFO_FUNCTIONAL                    BIT2
#define TEST_POINT_BYTE1_MEMORY_DISCOVERED_DMA_PROTECTION_ENABLED                BIT3
#define   TEST_POINT_BYTE1_MEMORY_DISCOVERED_MTRR_FUNCTIONAL_ERROR_CODE          L"0x01000000"
#define   TEST_POINT_BYTE1_MEMORY_DISCOVERED_MTRR_FUNCTIONAL_ERROR_STRING        L"Invalid MTRR Setting\r\n"
#define   TEST_POINT_BYTE1_MEMORY_DISCOVERED_MEMORY_RESOURCE_FUNCTIONAL_ERROR_CODE    L"0x01010000"
#define   TEST_POINT_BYTE1_MEMORY_DISCOVERED_MEMORY_RESOURCE_FUNCTIONAL_ERROR_STRING  L"Invalid Memory Resource\r\n"
#define   TEST_POINT_BYTE1_MEMORY_DISCOVERED_FV_INFO_FUNCTIONAL_ERROR_CODE       L"0x01020000"
#define   TEST_POINT_BYTE1_MEMORY_DISCOVERED_FV_INFO_FUNCTIONAL_ERROR_STRING     L"Invalid FV Information\r\n"
#define   TEST_POINT_BYTE1_MEMORY_DISCOVERED_DMA_PROTECTION_ENABLED_ERROR_CODE   L"0x01030000"
#define   TEST_POINT_BYTE1_MEMORY_DISCOVERED_DMA_PROTECTION_ENABLED_ERROR_STRING L"DMA protection disabled\r\n"

#define TEST_POINT_BYTE2_END_OF_PEI_SYSTEM_RESOURCE_FUNCTIONAL                             BIT0
#define TEST_POINT_BYTE2_END_OF_PEI_MTRR_FUNCTIONAL                                        BIT1
#define TEST_POINT_BYTE2_END_OF_PEI_PCI_BUS_MASTER_DISABLED                                BIT2
#define   TEST_POINT_BYTE2_END_OF_PEI_SYSTEM_RESOURCE_FUNCTIONAL_ERROR_CODE                L"0x02000000"
#define   TEST_POINT_BYTE2_END_OF_PEI_SYSTEM_RESOURCE_FUNCTIONAL_ERROR_STRING              L"Invalid System Resource\r\n"

#define   TEST_POINT_BYTE2_END_OF_PEI_MTRR_FUNCTIONAL_ERROR_CODE                           L"0x02010000"
#define   TEST_POINT_BYTE2_END_OF_PEI_MTRR_FUNCTIONAL_ERROR_STRING                         L"Invalid MTRR Setting\r\n"
#define   TEST_POINT_BYTE2_END_OF_PEI_PCI_BUS_MASTER_DISABLED_ERROR_CODE                   L"0x02020000"
#define   TEST_POINT_BYTE2_END_OF_PEI_PCI_BUS_MASTER_DISABLED_ERROR_STRING                 L"PCI Bus Master Enabled\r\n"

// Byte 3/4/5 - DXE
#define TEST_POINT_PCI_ENUMERATION_DONE                                                    L" - PCI Enumeration Done - "
#define TEST_POINT_END_OF_DXE                                                              L" - End Of DXE - "
#define TEST_POINT_DXE_SMM_READY_TO_LOCK                                                   L" - DXE SMM Ready To Lock - "

#define TEST_POINT_READY_TO_BOOT                                                           L" - Ready To Boot - "
#define TEST_POINT_EXIT_BOOT_SERVICES                                                      L" - Exit Boot Services - "
```

```
#define TEST_POINT_BYTE3_PCI_ENUMERATION_DONE_RESOURCE_ALLOCATED                              BIT0
#define TEST_POINT_BYTE3_PCI_ENUMERATION_DONE_BUS_MASTER_DISABLED                             BIT1
#define TEST_POINT_BYTE3_END_OF_DXE_NO_THIRD_PARTY_PCI_OPTION_ROM                             BIT2
#define TEST_POINT_BYTE3_END_OF_DXE_DMA_ACPI_TABLE_FUNCTIONAL                                 BIT3
#define TEST_POINT_BYTE3_END_OF_DXE_DMA_PROTECTION_ENABLED                                    BIT4
#define   TEST_POINT_BYTE3_PCI_ENUMERATION_DONE_RESOURCE_ALLOCATED_ERROR_CODE                 L"0x03000000"
#define   TEST_POINT_BYTE3_PCI_ENUMERATION_DONE_RESOURCE_ALLOCATED_ERROR_STRING               L"Invalid PCI Resource\r\n"
#define   TEST_POINT_BYTE3_PCI_ENUMERATION_DONE_BUS_MASTER_DISABLED_ERROR_CODE                L"0x03010000"
#define   TEST_POINT_BYTE3_PCI_ENUMERATION_DONE_BUS_MASTER_DISABLED_ERROR_STRING              L"PCI Bus Master Enabled\r\n"
#define   TEST_POINT_BYTE3_END_OF_DXE_NO_THIRD_PARTY_PCI_OPTION_ROM_ERROR_CODE                L"0x03020000"
#define   TEST_POINT_BYTE3_END_OF_DXE_NO_THIRD_PARTY_PCI_OPTION_ROM_ERROR_STRING              L"Third Party Option ROM disp
atched\r\n"
#define   TEST_POINT_BYTE3_END_OF_DXE_DMA_ACPI_TABLE_FUNCTIONAL_ERROR_CODE                    L"0x03030000"
#define   TEST_POINT_BYTE3_END_OF_DXE_DMA_ACPI_TABLE_FUNCTIONAL_ERROR_STRING                  L"No DMA ACPI table\r\n"
#define   TEST_POINT_BYTE3_END_OF_DXE_DMA_PROTECTION_ENABLED_ERROR_CODE                       L"0x03040000"
#define   TEST_POINT_BYTE3_END_OF_DXE_DXE_DMA_PROTECTION_ENABLED_ERROR_STRING                 L"DMA protection disabled\r\n"


#define TEST_POINT_BYTE4_READY_TO_BOOT_MEMORY_TYPE_INFORMATION_FUNCTIONAL                     BIT0
#define TEST_POINT_BYTE4_READY_TO_BOOT_UEFI_MEMORY_ATTRIBUTE_TABLE_FUNCTIONAL                 BIT1
#define TEST_POINT_BYTE4_READY_TO_BOOT_UEFI_BOOT_VARIABLE_FUNCTIONAL                          BIT2
#define TEST_POINT_BYTE4_READY_TO_BOOT_UEFI_CONSOLE_VARIABLE_FUNCTIONAL                       BIT3
#define TEST_POINT_BYTE4_READY_TO_BOOT_ACPI_TABLE_FUNCTIONAL                                  BIT4
#define TEST_POINT_BYTE4_READY_TO_BOOT_GCD_RESOURCE_FUNCTIONAL                                BIT5
#define   TEST_POINT_BYTE4_READY_TO_BOOT_MEMORY_TYPE_INFORMATION_FUNCTIONAL_ERROR_CODE        L"0x04000000"
#define   TEST_POINT_BYTE4_READY_TO_BOOT_MEMORY_TYPE_INFORMATION_FUNCTIONAL_ERROR_STRING      L"Invalid Memory Type Informa
tion\r\n"
#define   TEST_POINT_BYTE4_READY_TO_BOOT_UEFI_MEMORY_ATTRIBUTE_TABLE_FUNCTIONAL_ERROR_CODE    L"0x04010000"
#define   TEST_POINT_BYTE4_READY_TO_BOOT_UEFI_MEMORY_ATTRIBUTE_TABLE_FUNCTIONAL_ERROR_STRING  L"Invalid Memory Attribute Ta
ble\r\n"
#define   TEST_POINT_BYTE4_READY_TO_BOOT_UEFI_BOOT_VARIABLE_FUNCTIONAL_ERROR_CODE             L"0x04020000"
#define   TEST_POINT_BYTE4_READY_TO_BOOT_UEFI_BOOT_VARIABLE_FUNCTIONAL_ERROR_STRING           L"Invalid Boot Variable\r\n"
#define   TEST_POINT_BYTE4_READY_TO_BOOT_UEFI_CONSOLE_VARIABLE_FUNCTIONAL_ERROR_CODE          L"0x04030000"
#define   TEST_POINT_BYTE4_READY_TO_BOOT_UEFI_CONSOLE_VARIABLE_FUNCTIONAL_ERROR_STRING        L"Invalid Console Variable\r\
n"
#define   TEST_POINT_BYTE4_READY_TO_BOOT_ACPI_TABLE_FUNCTIONAL_ERROR_CODE                     L"0x04040000"
#define   TEST_POINT_BYTE4_READY_TO_BOOT_ACPI_TABLE_FUNCTIONAL_ERROR_STRING                   L"Invalid ACPI Table\r\n"
#define   TEST_POINT_BYTE4_READY_TO_BOOT_GCD_RESOURCE_FUNCTIONAL_ERROR_CODE                   L"0x04050000"
#define   TEST_POINT_BYTE4_READY_TO_BOOT_GCD_RESOURCE_FUNCTIONAL_ERROR_STRING                 L"Invalid GCD Resource\r\n"

#define TEST_POINT_BYTE5_READY_TO_BOOT_UEFI_SECURE_BOOT_ENABLED                               BIT0
#define TEST_POINT_BYTE5_READY_TO_BOOT_PI_SIGNED_FV_BOOT_ENABLED                              BIT1
#define TEST_POINT_BYTE5_READY_TO_BOOT_TCG_TRUSTED_BOOT_ENABLED                               BIT2
#define TEST_POINT_BYTE5_READY_TO_BOOT_TCG_MOR_ENABLED                                        BIT3
#define   TEST_POINT_BYTE5_READY_TO_BOOT_UEFI_SECURE_BOOT_ENABLED_ERROR_CODE                  L"0x05000000"
#define   TEST_POINT_BYTE5_READY_TO_BOOT_UEFI_SECURE_BOOT_ENABLED_ERROR_STRING                L"UEFI Secure Boot Disable\r\
n"
#define   TEST_POINT_BYTE5_READY_TO_BOOT_PI_SIGNED_FV_BOOT_ENABLED_ERROR_CODE                 L"0x05010000"
#define   TEST_POINT_BYTE5_READY_TO_BOOT_PI_SIGNED_FV_BOOT_ENABLED_ERROR_STRING               L"PI Signed FV Boot Disable\r
\n"
#define   TEST_POINT_BYTE5_READY_TO_BOOT_TCG_TRUSTED_BOOT_ENABLED_ERROR_CODE                  L"0x05020000"
#define   TEST_POINT_BYTE5_READY_TO_BOOT_TCG_TRUSTED_BOOT_ENABLED_ERROR_STRING                L"TCG Trusted Boot Disable\r\
n"
#define   TEST_POINT_BYTE5_READY_TO_BOOT_TCG_MOR_ENABLED_ERROR_CODE                           L"0x05030000"
#define   TEST_POINT_BYTE5_READY_TO_BOOT_TCG_MOR_ENABLED_ERROR_STRING                         L"TCG MOR not enabled\r\n"

// Byte 6/7 - SMM
#define TEST_POINT_SMM_END_OF_DXE                                                             L" - SMM End Of DXE - "
#define TEST_POINT_SMM_READY_TO_LOCK                                                          L" - SMM Ready To Lock - "
#define TEST_POINT_SMM_READY_TO_BOOT                                                          L" - SMM Ready To Boot - "
#define TEST_POINT_SMM_EXIT_BOOT_SERVICES                                                     L" - SMM Exit Boot Services -
 "

#define TEST_POINT_BYTE6_SMM_END_OF_DXE_SMRR_FUNCTIONAL                                       BIT0
#define TEST_POINT_BYTE6_SMM_READY_TO_LOCK_SMM_MEMORY_ATTRIBUTE_TABLE_FUNCTIONAL              BIT1
#define TEST_POINT_BYTE6_SMM_READY_TO_LOCK_SECURE_SMM_COMMUNICATION_BUFFER                    BIT2
#define TEST_POINT_BYTE6_SMM_READY_TO_BOOT_SMM_PAGE_LEVEL_PROTECTION                          BIT3
#define   TEST_POINT_BYTE6_SMM_END_OF_DXE_SMRR_FUNCTIONAL_ERROR_CODE                          L"0x06000000"
#define   TEST_POINT_BYTE6_SMM_END_OF_DXE_SMRR_FUNCTIONAL_ERROR_STRING                        L"Invalid SMRR\r\n"
#define   TEST_POINT_BYTE6_SMM_READY_TO_LOCK_SMM_MEMORY_ATTRIBUTE_TABLE_FUNCTIONAL_ERROR_CODE   L"0x06010000"
#define   TEST_POINT_BYTE6_SMM_READY_TO_LOCK_SMM_MEMORY_ATTRIBUTE_TABLE_FUNCTIONAL_ERROR_STRING L"Invalid SMM Memory Attribut
e Table\r\n"
```

```
#define   TEST_POINT_BYTE6_SMM_READY_TO_LOCK_SECURE_SMM_COMMUNICATION_BUFFER_ERROR_CODE         L"0x06020000"
#define   TEST_POINT_BYTE6_SMM_READY_TO_LOCK_SECURE_SMM_COMMUNICATION_BUFFER_ERROR_STRING       L"Unsecure SMM communication
buffer\r\n"
#define   TEST_POINT_BYTE6_SMM_READY_TO_BOOT_SMM_PAGE_LEVEL_PROTECTION_ERROR_CODE               L"0x06030000"
#define   TEST_POINT_BYTE6_SMM_READY_TO_BOOT_SMM_PAGE_LEVEL_PROTECTION_ERROR_STRING             L"SMM page level protection d
isabled\r\n"

#define TEST_POINT_BYTE7_DXE_SMM_READY_TO_LOCK_SMRAM_ALIGNED                                    BIT0
#define TEST_POINT_BYTE7_DXE_SMM_READY_TO_LOCK_WSMT_TABLE_FUNCTIONAL                            BIT1
#define TEST_POINT_BYTE7_DXE_SMM_READY_TO_BOOT_SMI_HANDLER_INSTRUMENT                           BIT2
#define   TEST_POINT_BYTE7_DXE_SMM_READY_TO_LOCK_SMRAM_ALIGNED_ERROR_CODE                       L"0x07000000"
#define   TEST_POINT_BYTE7_DXE_SMM_READY_TO_LOCK_SMRAM_ALIGNED_ERROR_STRING                     L"Invalid SMRAM Information\r
\n"
#define   TEST_POINT_BYTE7_DXE_SMM_READY_TO_LOCK_WSMT_TABLE_FUNCTIONAL_ERROR_CODE               L"0x07010000"
#define   TEST_POINT_BYTE7_DXE_SMM_READY_TO_LOCK_WSMT_TABLE_FUNCTIONAL_ERROR_STRING             L"No WSMT table\r\n"
#define   TEST_POINT_BYTE7_DXE_SMM_READY_TO_BOOT_SMI_HANDLER_INSTRUMENT_ERROR_CODE              L"0x07020000"
#define   TEST_POINT_BYTE7_DXE_SMM_READY_TO_BOOT_SMI_HANDLER_INSTRUMENT_ERROR_STRING            L"No SMI Instrument\r\n"

// Byte 8 - Advanced
#define TEST_POINT_BYTE8_READY_TO_BOOT_ESRT_TABLE_FUNCTIONAL                                    BIT0
#define TEST_POINT_BYTE8_READY_TO_BOOT_HSTI_TABLE_FUNCTIONAL                                    BIT1
#define   TEST_POINT_BYTE8_READY_TO_BOOT_ESRT_TABLE_FUNCTIONAL_ERROR_CODE                       L"0x08000000"
#define   TEST_POINT_BYTE8_READY_TO_BOOT_ESRT_TABLE_FUNCTIONAL_ERROR_STRING                     L"No ESRT\r\n"
#define   TEST_POINT_BYTE8_READY_TO_BOOT_HSTI_TABLE_FUNCTIONAL_ERROR_CODE                       L"0x08010000"
#define   TEST_POINT_BYTE8_READY_TO_BOOT_HSTI_TABLE_FUNCTIONAL_ERROR_STRING                     L"No HSTI\r\n"
```

D.4.2.3 ADAPTER_INFO_PLATFORM_TEST_POINT_STRUCT

```
#pragma pack (1)

typedef struct {
  UINT32  Version;
  UINT32  Role;
  CHAR16  ImplementationID[256];
  UINT32  FeaturesSize;
  UINT8   FeaturesImplemented[TEST_POINT_FEATURE_SIZE];
  UINT8   FeaturesVerified[TEST_POINT_FEATURE_SIZE];
  CHAR16  End;
} ADAPTER_INFO_PLATFORM_TEST_POINT_STRUCT;

#pragma pack ()

#endif
```