AMD Platform Security Processor BIOS Architectural Design Guide

Publication # **54267** Revision: **1.00**
Issue Date: **January 2014**

Contents

List of Figures

List of Tables

Revision History

Date	Revision	Description
January 2014	1.00	Initial NDA release.

Definitions

Table 1. Definitions, Acronyms and Abbreviations

Term	Definition	Comments
AES	Advanced Encryption Standard	
AGESA	AMD Generic Encapsulated Software Architecture	AMD software stack to initialize Si
AP	Application Processor	Secondary core in a multi-core cluster
CCP	Cryptographic Co-Processor	
CRTM	Core Root of Trust for Measurement	
DMA	Direct Memory Access	
DRAM	Dynamic Random Access Memory	
DXE	Driver Execution Environment	Driver Execution Environment phase, that run after memory has been initialized
ECC	Elliptic Curve Cryptography	
EFI	Extensible Firmware Interface	
FFS	Firmware File System	A binary storage format that is well suited to firmware volumes. The abstracted model of the FFS is a flat file system
fTPM	Firmware TPM	same as iTPM
FV	Firmware Volume	A FV is a simple Flash File System that starts with a header and contains files that are named by a GUID. The file system is flat and does not support directories. Each file is made up of a series of sections that support encapsulation.
FW	Firmware	
HOB	Hand-Off Block	A structure used to pass information from one boot phase to another (i.e., from the PEI phase to the DXE phase)
HMAC	keyed-hash message authentication code	In cryptography, a keyed-hash message authentication code (HMAC) is a specific construction for calculating a message authentication code (MAC) involving a cryptographic hash function in combination with a secret cryptographic key

Table 1. Definitions, Acronyms and Abbreviations (Continued)

Term	Definition	Comments
HSM	Hardware Security Module	A hardware security module (HSM) is a physical computing device that safeguards and manages digital keys for strong authentication and provides cryptoprocessing without revealing keying material
IBV	Independent BIOS Vendor	
iTPM	Integrated TPM	Firmware TPM
ML	Mullins	Code name used for the AMD Family 16h Models 30h-3Fh Processor, otherwise known as the AMD FT3b processor or the AMD FP4 processor.
MTM	Mobile Trusted Module	A firmware version of a TPM
OEM	Original Equipment Manufacturer	
OS	Operating System	
PEI	Pre-EFI Initialization	Set of drivers usually designed to initialize memory and the CPU so that DXE phase can run.
PKCS	Public Key Cryptography Standards	
PSP	Platform Security Processor	
RNG	Random Number Generator	
ROM	Read Only Memory	
RoT	Root of Trust	
RSA	Rivest-Shamire-Adleman encrypton algorithm	
RTM	Root of trust for measurement	
SEC	Security Phase	Initial starting point for boot process, first code executed after hardware reset. Responsible for 1) Establishing root trust in the software space; 2) Initializing architecture specific configuration to establish memory space for the C code stack.
SHA	Secure Hash Algorithm	

Table 1. Definitions, Acronyms and Abbreviations (Continued)

Term	Definition	Comments
SMM	System Management Mode	An operating mode in which all normal execution (including the operating system) is suspended, and special separate software (usually firmware or a hardware-assisted debugger) is executed in high-privilege mode.
SPI	Serial Peripheral Interface Bus.	Also referred to as the Non-volatile ROM chip on this Bus
SRAM	Static Random Access Memory	
TCG	Trusted Computing Group	A standards organization
TEE	Trusted Execution Environment	TrustZone is one example of a technology that establishes a TEE
TPM	Trusted Platform Module	A hardware root of trust
	AMD Signing Key	A 2048 bit RSA key pair generated by AMD. The private key is used to sign the public portion of OEM signing key.
	OEM Signing Key	An asymmetric key pair generated by OEMs. The private key is used to sign the RTM volume of BIOS. The public portion of signed OEM key is stored in the SPI BIOS image
	PSP Directory	A simple directory at certain SPI location that lists various firmware images and respective location in the SPI space
	BIOS RTM Volume	BIOS firmware Volume that is root of trust of x86 BIOS execution. The code in this volume is executed at x86reset. Based on OEM implementation this can be SEC volume or combined SEC-PEI volume. PSP firmware authenticates BIOS RTM volume before releasing the x86 core.

Chapter 1 Introduction

1.1 Scope of This Document

This document's primary focus is to cover BIOS requirements and to suggest implementation guidelines. This document does not cover the details of the Platform Security Processor (PSP) firmware and PSP functionality. This document covers only the services and interfaces that BIOS provides to PSP firmware. The document also assumes that the platform BIOS will run on x86 CPU core under an ACPI-aware operating system and that BIOS uses SPI ROM storage.

1.1.1 PSP Overview

The PSP is an integrated hardware security subsystem that runs independently from the main cores of the platform. It provides various security features as listed in the following subsections. PSP executes its own firmware and shares the SPI flash storage that BIOS uses.

1.1.2 Key Features of the PSP

1.1.2.1 Hardware Validated Boot

- The PSP validates the signature of the initial BIOS Boot code prior to starting BIOS boot. PSP is the Core Root of Trust for Measurement (CRTM) and the x86 cores are only released from reset if the BIOS image is authentic.
- Only validated BIOS is allowed to boot.
- The initial block of BIOS code is responsible for subsequently validating the signatures of all other BIOS code blocks loaded from the system read only memory (ROM).

1.1.2.2 Integrated Trusted Platform Module (TPM) Functions

Implements the TPM 2.0 functions required (for some categories of systems) by Microsoft® Windows® 8.

1.1.2.3 Cryptographic Acceleration Support

Provides hardware acceleration of cryptographic algorithms for PSP FW and x86 software. Also provides true random number generator (RNG) support.

Chapter 2 Overview of Feature Implementation

2.1 Hardware Validated Boot

Hardware Validated Boot is an AMD specific form of secure boot, which roots the trust to hardware in an immutable PSP on-chip ROM firmware and validates the integrity of the system ROM firmware (BIOS).

Figure 1 shows the scope of hardware validated boot as it relates to the UEFI secure boot.

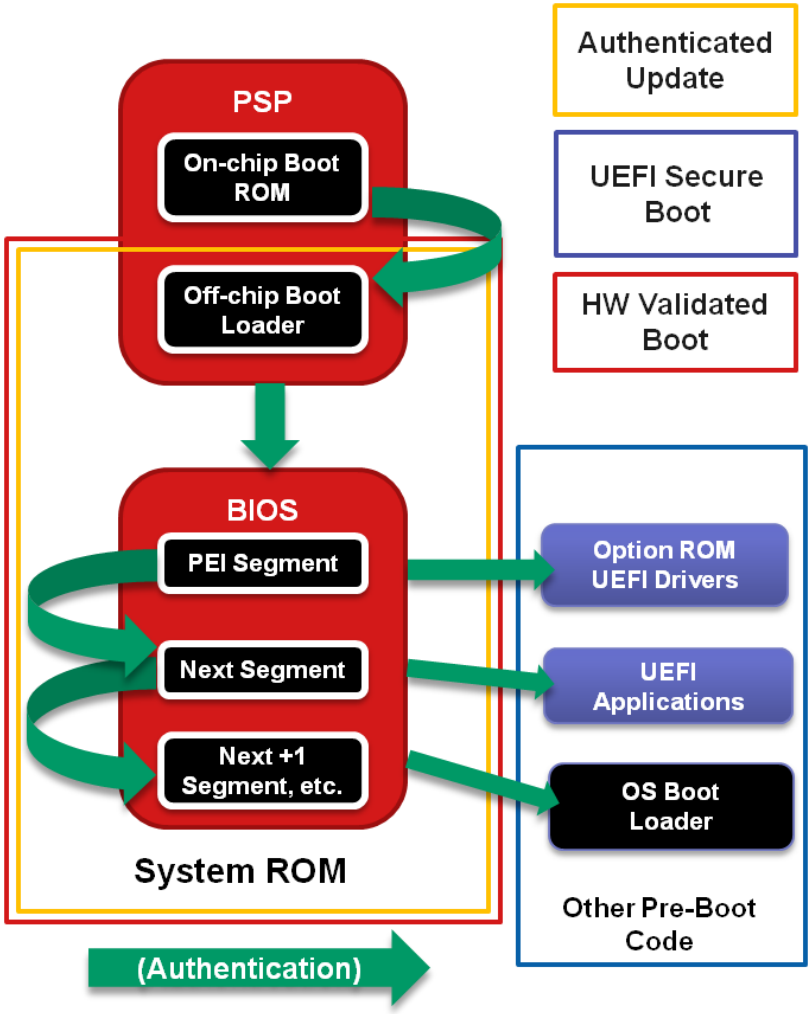

Figure 1. Hardware Validated Boot Overview

The idea behind AMD Hardware Validated Boot is to build a trusted boot environment even before starting the x86 cores. In the Hardware Validated Boot mode, the PSP subsystem is the core root of trust for measurement.

During cold boot and under Hardware Validated Boot, the PSP runs its own firmware; all of the x86 cores are held in the reset state while PSP firmware performs basic initialization and authenticates the x86 reset code (i.e., the first block of BIOS). PSP firmware searches for this fraction of the BIOS image in the SPI flash area and validates its signature. After validating the BIOS signature, the PSP firmware configures the necessary hardware registers to release the x86 cores. The x86 cores, upon reset, start execution of the BIOS code authenticated by PSP firmware. The BIOS maintains this trust chain by first authenticating all firmware components before passing control to these firmware components. UEFI Specification 2.3.1, Chapter 27, provides further guideline for BIOS expected behavior. This document does not discuss those details and the BIOS writers are expected to follow those guidelines in addition to the ones listed later in this document.

On resume from sleep, PSP firmware restores the memory controller, validates resume vector and releases the x86 cores. Once released, the x86 cores fetch code straight from Dynamic Random Access Memory (DRAM) based on boot time BIOS configuration. Chapter 5, BIOS S3-Resume Path Handling, on page 36 covers details regarding the BIOS-PSP information exchange to make this execution flow possible.

2.1.1 Integrated TPM Functions

The PSP software solution-stack offers firmware based TPM 2.0 services based on Microsoft whitepaper – “Trusted execution environment ACPI profile”. BIOS writers are expected to follow the guidelines and provide BIOS support as outlined in the Microsoft whitepaper. BIOS must wait for memory to be available before sending firmware trusted platform module (fTPM) command to PSP.

The PSP subsystem does not have its own storage space to save the TPM data. Instead, it relies on BIOS to provide the storage services to PSP firmware. The PSP firmware uses BIOS system management mode (SMM) mailbox services to save PSP data in SPI space. The PSP firmware encrypts the data block and uses BIOS runtime SMM handler services to store or update this data to SPI flash storage. The BIOS is expected to (a) reserve part of SPI flash region for PSP data storage, (b) provide services to PSP firmware to store and update PSP data to this SPI region, and (c) protect this region of SPI flash from writing by unauthorized code (using the chipset-provided flash locking mechanisms and Secure Flash Update). PSP firmware is expected to manage any updated TPM data within its own local memory until BIOS makes those storage services available to PSP firmware; in other words, the BIOS storage services are not expected to be available during early boot and resume path and PSP firmware is expected to not rely on BIOS storage services during that time. Separately, BIOS can use the firmware TPM services for BIOS measurements as outlined in TCG specifications. In this usage model, the BIOS replaces the discrete-TPM PEI/DXE driver with a firmware-based-TPM PEI/DXE driver; and BIOS exposes the TPM

protocol defined in “TCG EFI protocol” specification. Also BIOS is expected to use the TPM2.0 command-set to communicate with integrated Trusted Platform Module (iTPM).

2.1.2 Cryptographic Acceleration Support

The PSP solution-stack also offers hardware based cryptographic acceleration services such as support for SHA, AES, RSA, ECC, etc. BIOS makes use of PSP cryptographic acceleration for its own purposes such as validating the digital signature of other BIOS components, OS boot loader etc.

The CCP also provides a true RNG function, which may optionally be used to seed UEFI entropy. The mechanism for seeding UEFI entropy is described by Microsoft in the document entitled, “UEFI Entropy-gathering Protocol”.

Chapter 3 PSP Components

PSP components are described in the following subsections.

3.1 On-chip PSP Boot ROM

PSP Boot ROM is an immutable part of the SOC and it embeds in it a SHA-256 hash of the public part of the AMD signing key, which forms the hardware core root of trust for the Hardware Validated Boot process. PSP microcontroller (A5) starts executing the On-chip Boot ROM code, in secure mode. It loads the off-chip PSP firmware into PSP static random access memory (SRAM) and after authenticating the PSP off-chip firmware it passes control to it.

3.2 Off-chip PSP Boot Loader

When PSP on-chip Boot ROM transfers control to PSP off-chip Boot Loader, it communicates the pre-loaded PSP Directory table address in PSP SRAM, in mailbox area at a pre-defined address within PSP SRAM.

Once the PSP Boot Loader starts execution, it first determines if the system is booting from S5 or resuming from S3/S0i3 state by reading the sleep type and resume flag from FCH.

PSP Boot Loader performs the following sequence of operations as part of the S5 boot:

1. Locates and Loads the Off-chip SMU FW into PSP SRAM; and verifies its signature using the AMD Signing RSA Public Key. If the signature verification passes, Loads the verified Off-chip SMU FW image into SMU SRAM and notifies the SMU Boot ROM code to transfer control to the SRAM FW.
2. Locates and Loads the BIOS Signing RSA Public Key token into PSP SRAM; and validates its signature using the AMD Signing RSA Public Key.
   a. If the signature validation fails then it writes an error code to the TDR register and enters an infinite loop.
   b. If the signature validation passes, then proceeds to step 3.
3. Locates and Loads the BIOS RTM FW and signature block, into PSP SRAM; and validates its signature using the BIOS Signing RSA Public Key. The BIOS RTM FW signature is computed over the BIOS RTM FW code concatenated with PSP Directory Table.
   a. If the signature validation fails then it writes an error code to the TDR register and enters an infinite loop.
   b. If the signature validation passes, then proceeds to step 4.
4. Initializes the BIOS-PSP mailbox interface
5. Releases the CPU BSP core to begin execution. At this point the CPU microcode starts executing the BIOS RTM code from a fixed CPU reset vector 0xFFFF_FFF0.

PSP Boot Loader performs the following sequence of operations as part of the S3/S0i3 resume:

1. Locates and Loads the S3 Restore Buffer generated by the platform BIOS and saved to SPI-ROM by the PSP Secure OS during S5 boot sequence into PSP SRAM; and validates its HMAC using the HMAC key.
   a. If the HMAC validation fails then it writes an error code to the TDR register and enters an infinite loop.
   b. If the HMAC validation passes, then proceeds to step 2.
2. Processes the S3 restore buffer and restore pre self-refresh data to the memory interface, reprograms memory interface, executes the programming sequence to take the DRAM out of self-refresh and writes the S3 restore post self-refresh data to the memory interface.
3. Releases the CPU BSP core to resume execution. At this point the CPU BSP core starts executing the BIOS Resume code located in SMM memory and pointed to by CPU S3-resume vector.
4. Determines if the system is resuming from S3 or S0i3 state
5. Derives the PSP SRAM Content Encryption Key
6. Decrypts the PSP SRAM content (saved during S3/S0i3-suspend) from DRAM into PSP SRAM
7. Validates the HMAC of the loaded data using the HMAC key
   a. If the HMAC validation fails then it writes an error code to the TDR register and enters an infinite loop.
   b. If the HMAC validation passes, then it transfers control to the restored PSP Secure OS resume entry point.

3.3 Off-chip PSP Secure OS

When off-chip PSP Boot Loader transfers control to PSP Secure Operating System (OS), it communicates certain state information such as the Sleep (Sx) state, fTPM state and S3 Restore Buffer in a mailbox area at a pre-defined address with in PSP SRAM.

Once the PSP Secure OS starts execution, it first determines if the system is booting from S5 or resuming from S3/S0i3 state by reading the Sx state variable in the SRAM mailbox address.

PSP Secure OS performs the following sequence of operations as part of the S5 boot :

1. Performs the necessary initialization of OS internal structures and instantiates the TPM 2.0 compliant fTPM as a trusted application
2. Sets-up CPU-PSP interface registers' access control policy and interrupt mechanism
3. Re-initializes BIOS-PSP mailbox interface
4. Waits for BIOS SMM environment to be set up and BIOS to notify the SMM space reserved for PSP and parameters needed for PSP to generate SMI

5. Generates an HMAC over the S3 Restore Buffer content in SRAM using an HMAC key and stores S3 Restore Buffer contents appended with the HMAC to SPI-ROM by placing the payload in SMM space and generating an SMI interrupt to BIOS
6. Enters steady-state idling and waiting for commands from host interfaces

In steady state, when the system begins to enter S3/S0i3-state, as shown in Figure 3, on page 27 and Figure 4, on page 30, PSP Secure OS is notified. Upon receiving this notification, PSP Secure OS prepares to enter the S3/S0i3 state.

Chapter 4 Overview of BIOS Support for PSP

4.1 SPI Flash Region Layout

The SPI flash storage is shared by both x86 BIOS binary and PSP firmware; i.e., BIOS code and data as well as PSP firmware code and data coexist in the same storage space. PSP firmware (on-chip and off-chip) parses the SPI flash for various components in order to locate, authenticate and execute PSP binaries. A PSP directory table is needed to aid PSP firmware in finding various components in the SPI storage. This PSP directory is a simple table of various entries. Each of these entries provides information about various firmware components in the SPI flash area such as their type, size and location. The PSP directory can be anywhere in the SPI storage.

4.1.1 PSP Directory Table

Figure 2 on page 20 describes the layout of the PSP Directory Table in SPI-ROM and how it is used to locate various PSP related public key tokens, firmware images and corresponding signatures.

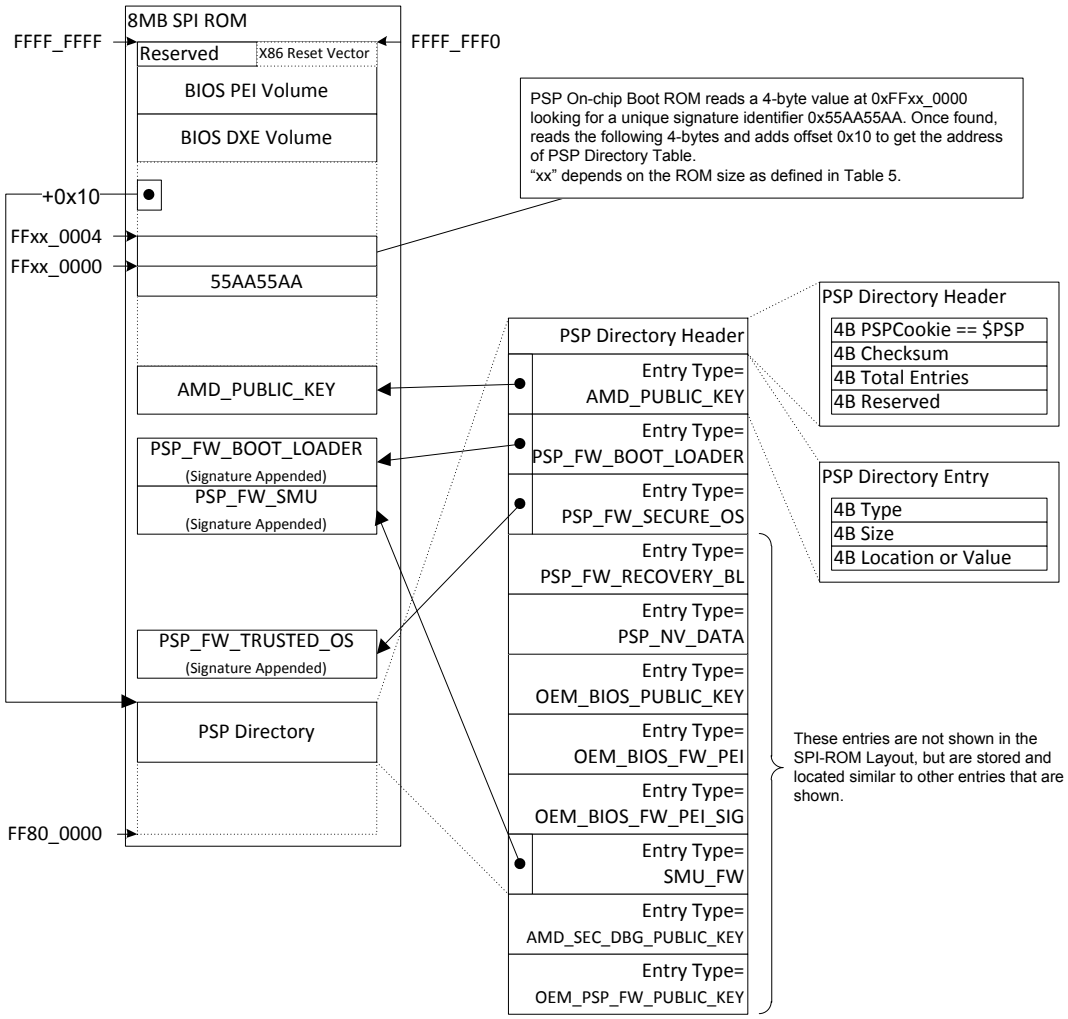

Figure 2. PSP Directory Table

Table 2 describes the Embedded Firmware Signature (0x55AA55AA) locations for different ROM sizes:

Table 2. Embedded Firmware Signature Target Locations

ROM Size	Address PSP Boot ROM checks (i.e. 0xFFxx_0000)
512 KB	0xFFFA_0000
1 MB	0xFFF2_0000
2 MB	0xFFE2_0000
4 MB	0xFFC2_0000
8 MB	0xFF82_0000
16 MB	0xFF02_0000

PSP Boot ROM does not have ROM size information and as such it sequentially reads all of the listed address from 0xFFFA_000 to 0xFF02_0000 until it finds the signature 0x55AA55AA.

Table 3 describes the fields of PSP Directory Table header structure:

Table 3. PSP Directory Table Header Structure

Field Name	Offset (Hex)	Size (in Bytes)	Description/Purpose
PSP Cookie	0x00	4	PSP cookie "$PSP" to recognize the header
Checksum	0x04	4	32 bit CRC value of the header items below this field and the including all entries. Fletcher's checksum algorithm is used for CRC calculation.
Total Entries	0x08	4	Number of PSP Directory Table Entries in the table
Reserved	0x12	4	Reserved – Set to zero

Table 4 describes the fields of PSP Directory Table Entry structure:

Table 4. PSP Directory Table Entry Fields

Field Name	Offset (Hex)	Size (in Bytes)	Description/Purpose
Type	0x00	4	Type of PSP entry
Size	0x04	4	Size of the PSP entry in bytes
Union of { Location or Value }	0x08	8	Depending on the Entry Type Location: Address/Offset of SPI-ROM location where the data for the corresponding PSP Entry is located OR Value: 64-bit value for the PSP Entry

Table 5. PSP Directory Entry Type Encodings

PSP Entry Type	Description/Purpose
0x00	AMD public Key
0x01	PSP Boot Loader firmware
0x02	PSP Secure OS firmware
0x03	PSP Recovery Boot Loader
0x04	PSP Non Volatile data
0x05	(OEM) BIOS public key signed with AMD key
0x06	BIOS RTM Volume

Table 5. PSP Directory Entry Type Encodings (Continued)

PSP Entry Type	Description/Purpose
0x07	BIOS RTM volume Signature using OEM private key
0x08	SMU offchip firmware
0x09	AMD Secure Debug Key
0x0A	(OEM) PSP Secure OS public key signed with AMD key
0x0B	PSP Soft Fuse Chain (VALUE = 0, Secure part can't be unlocked, Value = 1, Secure part can be unlocked)
0x0C	PSP boot-loaded trustlet binaries
0x0D	Trustlet public key signed with AMD key
0x0E-0x5E	Reserved for AMD use
0x5F	Software Configuration Settings Data Block
0x60-0x7F	Reserved for AMD use
0x80-0xFF	Reserved for OEM use

Note: *The First four entries of PspDirectory MUST be in the following order:*

First entry- AMD Public Key (Type 0x00)

Second entry- PSP Boot Loader firmware (Type 0x01)

Third entry- SMU Firmware (Type 0x08)

Fourth entry- PSP Recovery Firmware (Type 0x03)

When PSPType Enum is of type Soft Fuse Chain (i.e. type 0x0B) the Size tag is set to 8-bytes and the next 64bit field Location/Value is used to represent the soft fuse value itself.

The following table defines the PSP Soft Fuse Chain 1:

Table 6. PSP Soft Fuse Chain 1

Bit Number	Purpose
0	PSP Secure Debug Control Flag 1- Enabled , 0 – Disabled
1..63	Reserved for AMD Use

Table 7. PSP Soft Fuse Chain Bit Assignment

Bit Index	Description/Purpose
0	PSP Secure Debug Control Flag (0-Disabled, 1-Enabled)
1..63	Reserved

4.1.2 Crisis Recovery Path with PSP Enabled

BIOS already has an existing, mature recovery scheme in case of a flash update failure during a non-secure boot sequence (PSP disabled). The primary focus here is to handle the case where PSP is enabled. The off-chip PSP boot loader will run prior to releasing the X86 cores, and the PSP secure OS/Trustlet will run in parallel with X86 codes.

The recommendation is to separate the PSP SPI ROM into 2 categories, accompanied by BIOS and PSP FW changes.

Recommended SPI ROM Organization:

1. Fixed/Protected Region
   The contents in this region are unchanged and not updatable during flash update, it is protected by the IBV or OEM flash tool.
2. Updateable Region
   This area can be updated by new releases containing bug fixes, enhancements, and new feature enablement code.

If the following data areas in the SPI flash are corrupted, the result will be permanent boot failure, and the recovery mechanism will fail – ROMSIG (Embedded Firmware Signature), PSP Directory Table Header, PSP Directory Table Entries. Details for each of these items follow:

- ROMSIG should not be changed during flash update as the damage will result in the PSP on-chip FW not being able to locate the PSP Directory header, and the whole system will hang with the X86 cores held in reset.
- The PSP Directory Table Header should not be changed during flash update as the corruption will lead to a checksum failure and the PSP OROM will stall the system.
- The items pointed to by some of the PSP entries region should preserve a maximum size for further Update.
- Directory Table Entries may be updated, and others should remain fixed. Table 8 on page 24 shows the type for each entry. Target items marked as “Fixed” should be kept in the protected region that is not updated during flash update, as this is the base of the PSP recovery mechanism, and the corruption of those binaries will lead unexpected behavior

Table 8. PSP Entry SPI ROM Property Assignment

Type ID (Hex)	Name	SPI ROM property
00	AMD public Key	Fixed
01	PSP Boot Loader firmware	Updateable
02	PSP Secure OS firmware	Updateable
03	PSP Recovery Boot Loader	Fixed
04	PSP Non Volatile data	Fixed
05	(OEM) BIOS public key signed with AMD key	Fixed
06	BIOS RTM Volume	Fixed
07	BIOS RTM volume signature using OEM private key	Fixed
08	SMU offchip firmware	Updateable
09	AMD Secure Debug Key	Fixed
0A	(OEM) PSP Secure OS public key signed with AMD key	Fixed
0C	PSP boot-loaded trustlet binaries	Updateable
0D	Trustlet public key signed with AMD key	Fixed
5F	Software Configuration Settings Data Block	Updateable

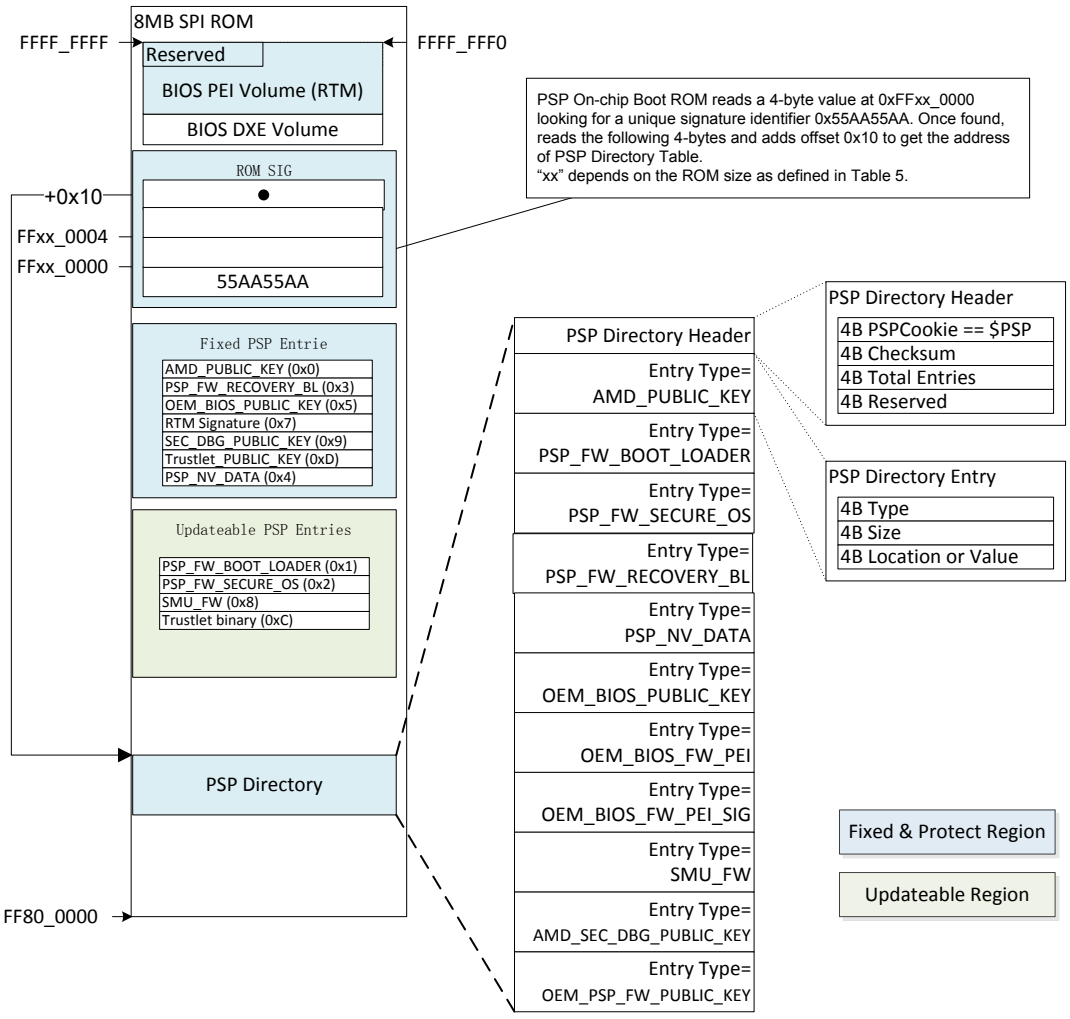

Figure 3 Overall SPI ROM layout (Recovery Supported Design)

The crisis-recovery flow for a non-secure system must be updated accordingly for a system with PSP enabled. (See also the "PSP Components" section, which covers details beyond crisis recovery.)

- When the system reset signal is de-asserted, the X86 cores will be held in reset, and the PSP on-chip boot ROM will start execution. After loading and authenticating the AMD Public key, the PSP on-chip boot ROM will load and validate the PSP off-chip boot loader.
- If verification fails, the recovery process is initiated and the PSP recovery off-chip boot loader will be loaded instead. The Recovery PSP Boot Loader verifies the BIOS PEI (not updatable), set a status bit to indicate recovery process required and releases x86.
- If verification succeeds, the PSP off-chip boot loader will authenticate the SMU FW. If SMU FW authentication fails, PSP off-chip Boot Loader, skips loading SMU FW, verifies the BIOS PEI (not updatable), set a status bit to indicate recovery process is required and releases x86. Given that the BIOS RTM and RTM signature are in the fixed region and not

updated during flash update, it is assumed that the authentication of BIOS RTM will always be successful in normal as well as recovery boot paths.

- The PSP off-chip Recovery boot loader does not load the SMU FW.
- Considering that the SMU firmware may not loaded in the recovery path, BIOS boot code must check whether the SMU firmware is loaded before issuing any messages to the SMU. In other words, the BIOS modules that are executed during the recovery path (including: AGESA PEI module, Platform PEI module, Recovery module, VBIOS) should not depend on the services provided by SMU FW.
- The PSP BIOS PEI module sends the DRAM ready message to the PSP FW once DRAM has been initialized. PSP FW will return the recovery status bit along with other status bits indications. The PSP PEI module publishes this information through PPI/HOB. The platform BIOS recovery path should locate and check this PPI/HOB to retrieve PSP FW integrity status along with DXE FV integrity. If the PPI/HOB shows problems with the status of either of these items, the BIOS should run in the recovery path.
- The secure flash mechanism should be followed in the recovery path as well as the normal BIOS update path. This means the new BIOS image must be authenticated before the update and the fixed SPI region as mentioned above must remain protected

Figure 3 on page 27 illustrates the details of the crisis recovery flow with PSP; zoom in to make it more legible.

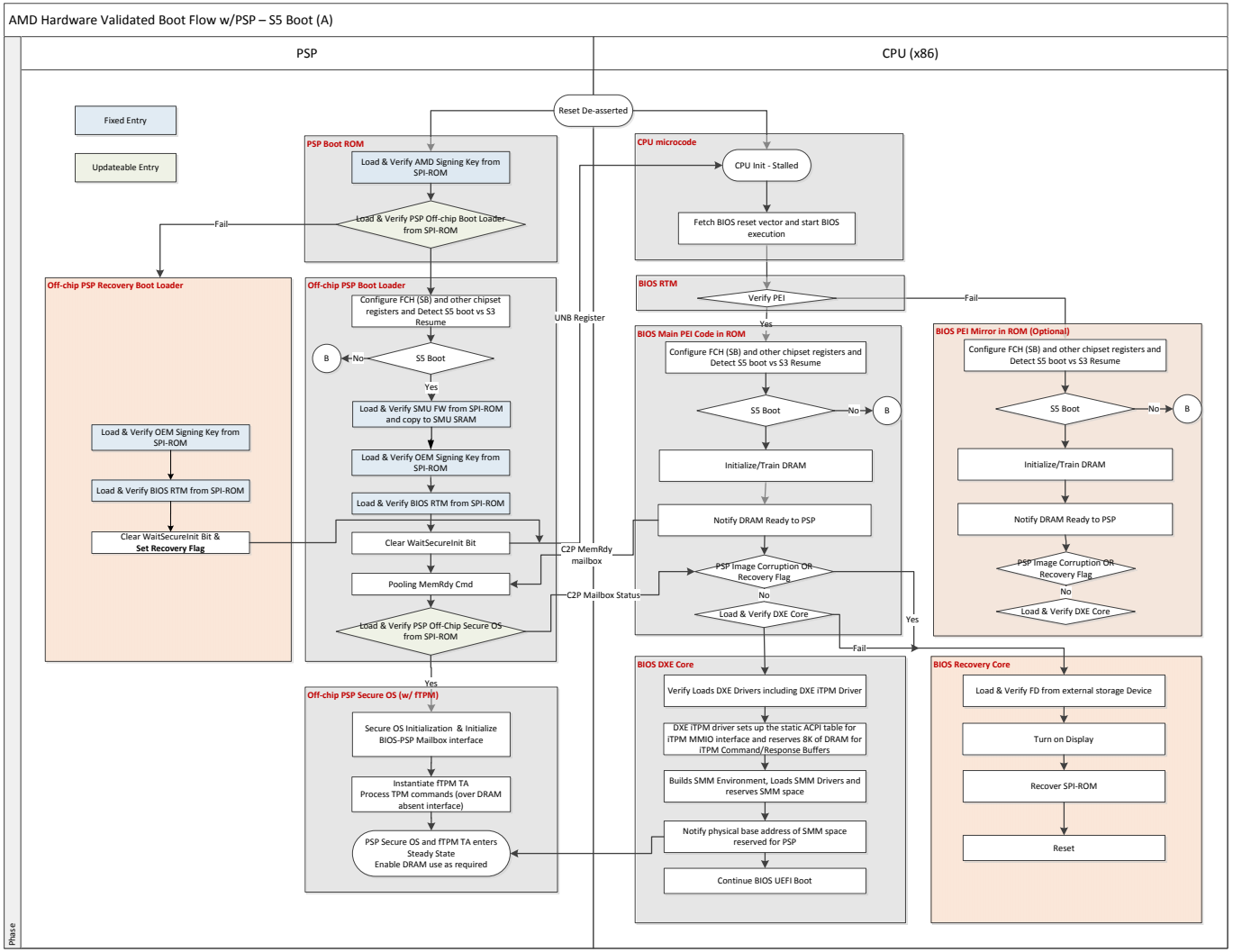

Figure 3. Crisis Recovery Flow With PSP

4.2 Signing of BIOS Component- OEM Signing Key, PEI Volume

The OEM must sign the BIOS RTM volume using the private portion of secure RSA key. This key that is used to sign the BIOS RTM volume is referred here as OEM signing key. OEM keeps the private portion of the OEM signing key in secure place (HSM, etc.) and submits the public portion of OEM key to AMD. AMD will perform one time signing of public portion of OEM signing key. This process enables PSP firmware to authenticate OEM public key.

On secure PSP parts, the PSP firmware authenticates the BIOS image in two steps before releasing x86 core. PSP firmware first parses PSP directory to locate the signed OEM public key and authenticates the OEM public key that was signed with AMD signing key. Next, after the public portion of OEM signing key is authenticated, the PSP firmware uses the OEM public key to further authenticate the BIOS RTM volume that was signed by OEM secure private key. PSP encrypts the hash with the OEM public key and then compares the resulting hash with the hash in the signature.

If the signature matches, the BIOS is considered trusted and x86 cores are released.

One note about the signed BIOS RTM volume- The signed BIOS blob is generated by first concatenating BIOS RTM volume with PSP directory blob and signing this combined blob using private portion of OEM key. Integrity of both PSP directory and RTM volume integrity can now be checked together when PSP firmware authenticates this blob. Note, this combined blob must be signed with the BIOS Signing RSA Private Key using the RSASSA-PSS signing scheme used as signature scheme with SHA-256 used as the hashing algorithm for both message and mask generation. The resulting signature data is stored in the PSP directory entry as the entry type 0x07. The size of the signature data will be 256-byte for 2048-bits key or 512-byte for 4K key.4096-bits key.

This two-step authentication removes unnecessary dependence on AMD signing server or build processes where BIOS is built on regular basis by IBV/OEM. AMD signing server will sign the public portion of OEM signing key once at the beginning of project and is separate from the BIOS build process; during the normal BIOS build process the private portion of OEM signing key will be used to sign BIOS RTM volume as part of OEM build process without any AMD signing server involvement. The BIOS image includes the AMD public key as well as signed public OEM signing key in the PSP image that was generated at the beginning of the project. This allows the OEM to use internal signing processes without external dependency.

After the PSP firmware releases x86 core for execution, the BIOS is expected to maintain the chain of trust to authenticate next set of bios code before executing it. It is left to OEM/IBV to choose appropriate BIOS implementation to insure the trust chain. At x86 core release the RTM volume has been authenticated by PSP firmware. BIOS RTM volume must authenticate the next volume before handling of the control. If only the SEC code is the BIOS RTM volume, then SEC code must authenticate PEI volume before handling off control to PEI core. If BIOS RTM volume

is entire PEI volume it must authenticate DXE volume. The DXE IPL code in BIOS PEI volume needs to further authenticate DXE volume before handing off control to DXE code.

There are various possible methods to authenticate each of BIOS drivers in DXE volume. In one possible method the DXE IPL driver in PEI volume validates the entire DXE volume before handing off control to DXE core. In such implementation the DXE volume can be signed and the static public key used for signing of DXE volume can be saved in PEI volume itself. This digital signature of DXE volume signature can be saved anywhere in SPI space with some provisions for DXE IPL driver to locate this key during cold boot. During cold boot the DXE IPL code in PEI volume will use this public key and validate DXE volume against the digital certificate of DXE volume saved in the SPI space and if the digital certificate is authentic the DXE IPL handoff control to DXE code.

Another option may be simply to save the SHA1/2 digest of DXE volume in PEI volume and during cold boot compare the DXE volume against the digest in PEI volume. If the digest matches the DXE volume is considered trusted. To implement such process the BIOS build process can compute the DXE digest and append the digest after PEI volume. The OEM signing key will sign the concatenated blob of PEI volume + DXE digest + PSP directory and this signature will be reflected in PSP directory for BIOS RTM entry. In this implementation PSP firmware will authenticate PEI volume as well as DXE digest. Later during cold boot DXE IPL driver can use this digest to authenticate DXE volume as well. Any tampering of DXE digest in SPI area can be detected by PSP since DXE digest is part of signed BIOS RTM.

AMD will provide the signed OEM public key and AMD public key. The format of AMD signing key and OEM signing key is shown in Appendix E on page 71.

4.3 BIOS Build Process

The SPI image includes BIOS components as well as PSP components. In the below process BIOS PEI volume is considered BIOS RTM volume.

To support hardware validated boot the BIOS PEI FV needs to be signed and PSP directory needs to be present to provide information regarding various signed entities. Figure 4 on page 30 summarizes the above discussion to illustrate how various entities listed above can be combined to build the final SPI image.

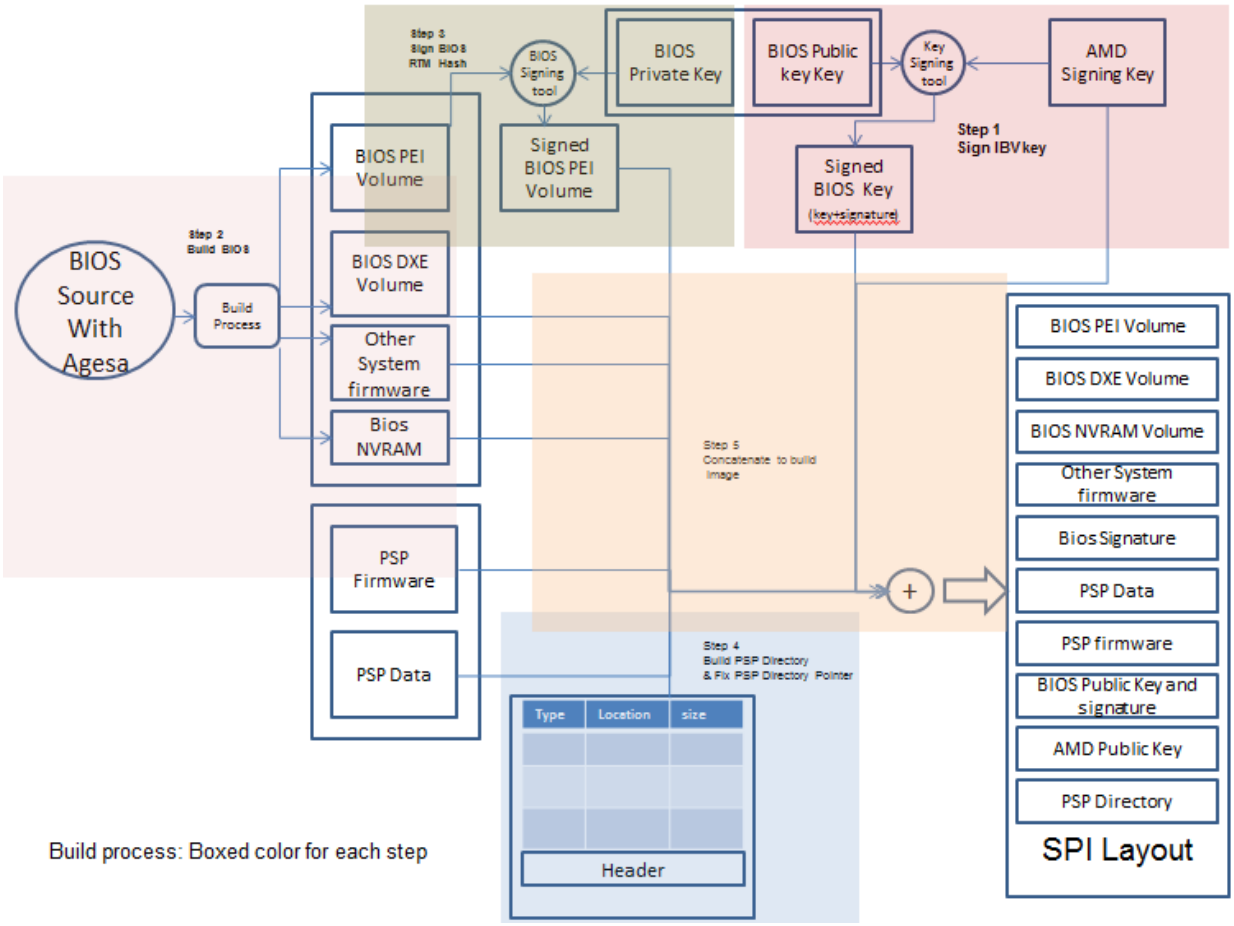

Figure 4. Final SPI Image

- As shown above, the first step is generation of the digital signature of public portion of OEM signing key. This may happen just once before the start of project. The OEM/IBV submits the public key portion of their OEM signing key to AMD. AMD signs this key using the AMD RSA key and passes back it to IBV/OEM. The AMD public key and the signed OEM public key should be part of final BIOS SPI image. Next, the BIOS source code is compiled and various BIOS components (PEI Volume, DXE volume, NVRAM storage, EC binary, etc.) are built as usual.
- As part of build process, DXE volume hash is generated and this hash is saved within BIOS PEI volume. This is required as part of hardware validated boot sequence for PEI kernel code to authenticate DXE volume before passing control to DXE core and extend the trust chain.
- The PSP directory is built next. PSP directory table points to location of various firmware entities. The PSP directory holds the location information of following entries-
  - AMD Public Key
  - PSP firmware i.e., Boot loader, Secure OS, PSP Recovery OS
  - AMD SMU firmware

  - PSP NVRAM data
  - OEM Signing Key signed with AMD signature
  - BIOS PEI Volume signature (signed Private portion of OEM signing key)
  - BIOS PEI Volume

- BIOS binaries, PSP directory and various firmware binaries are combined to build the SPI BIOS image.
- Finally, the OEM signing server builds the signed BIOS RTM signature based on blob of BIOS PEI volume concatenated with PSP Directory , and generates the digital signature of this using private portion of OEM signing key. The SPI location for signed BIOS RTM code is finally updated with this signature blob.

After the above steps the final SPI BIOS image will be ready.

4.3.1 Hardware validated Boot BIOS development bypass mechanism (Mullins Only)

Some IBVs have neither key management nor signing infrastructure. They have no way to securely generate and handle the OEM Signing Key. Thus, they have no way to sign their BIOS for the continuation of the secure boot mechanism. During bring-up and product development, the IBVs rev BIOS versions very quickly. Manually signing the BIOS block for them works initially but quickly becomes a drain on key AMD resources. Automating the signing of the BIOS block requires resources to create and support the signing portal. Automated checking of the submitted block is infeasible which means that AMD could inadvertently sign malicious BIOS blocks. Instead of either of these solutions, a specifically crafted and AMD private key signed data blob will be created to terminate the Hardware validated boot chain. This blob can only be used during development by IBVs and OEM. The blob mechanism will be removed in production release, AMD does not authorize the use of BLOB-based HVB for field use with production platforms..

The blob interception point is in the PSP secure off-chip bootloader. The PSP bootloader checks whether the entity pointed to by the BiosRtmFirmware entry (Entry Type6) of the PSP Directory is at the reset vector. If not, the PSP firmware considers it a blob and does not concatenate the PSP Directory table to the entity for signature validation. The blob is validated against the RTM.

This means that the PSP directory is not validated in this boot flow since it is not concatenated with the blob before the blob is signed. Since we are purposefully placing the token in a location that does not match the reset vector and thereby breaking the secure boot chain, this does not have additional negative security impact.

4.4 Runtime Execution Flow

The high level description of execution flow is listed below.

4.4.1 5.4.1 Pre x86 Initialization

At power on reset the following execution flow takes place. The Pre x86 code Init sequence defines the sequence of events by PSP firmware before the x86 core is released from reset. Details on this flow are defined in PSP Software Architecture design document.

1. Under Hardware Validated Boot mode, the x86 cores are held in reset during the system Power-On sequence while the PSP begins executing code.
2. The PSP runs its on-chip firmware at reset; this code is an immutable part of the Silicon.
3. The PSP on-chip firmware scans for the PSP off-chip code in the SPI space. PSP on-chip firmware first scans SPI space for Firmware Location signature (0x55AA55AA); this signature is scanned at specific SPI locations as defined in FCH porting guide. Offset 0x10 of this structure points to location of PSP directory.
4. The PSP on-chip firmware scans the PSP directory to find the AMD public key, the PSP firmware and the PSP data in the SPI ROM; PSP on-chip code next loads these binaries into PSP's secure memory. After authenticating these binaries the PSP on-chip firmware passes control to the PSP off-chip firmware.
5. The PSP off-chip firmware uses the PSP Directory to find the signed OEM key, the BIOS RTM signature, and the BIOS RTM code.
6. PSP off-chip firmware authenticates the BIOS in a three-step process

- PSP on-chip code authenticates the AMD public key by comparing this digest of the key value against the value saved in immutable ROM.
- PSP off-chip code authenticates the OEM public key by using AMD public key.
- After the signature of the OEM public key is validated, PSP firmware uses the OEM public key to authenticate BIOS PEI (RTM) Volume and PSP directory (signed RTM blob build with PEI volume concatenated with PSP Directory).

7. After authenticating the BIOS PEI volume, the PSP firmware writes to a hardware register to release the x86 cores.
8. PSP boot loader continues to load PSP secure OS from SPI to secure SRAM in parallel to x86 BIOS code execution. X86 BIOS must ensure the SPI region decode range remain valid. If the SPI decode range is changed by BIOS SEC or other code than it will also affect the PSP subsystem that is attempting to read from SPI space.

4.4.2 BIOS Boot x86 Initialization

After the x86 cores are released from reset, the BIOS boot phase starts and performs the following steps:

1. The CPU fetches the BIOS reset vector from SPI flash ROM at memory address 0xFFFFFFF0 and BIOS SEC execution starts.
2. The BIOS SEC code loads and executes rest of PEI driver in PEI (RTM)volume that has already been authenticated by PSP firmware.
3. If the platform wishes to use a TPM, the platform may use firmware TPM services offered by PSP or it may use discrete TPM, but not both; the correct BIOS TPM drivers need to be present to perform necessary TPM operation. PSP firmware offers TPM 2.0 support as outlined in Microsoft specification. BIOS must wait for memory to be available before sending any command to PSP.
4. Once memory is initialized the BIOS sends mailbox command “MboxBiosCmdDramInfo” to PSP to inform availability of memory. PSP firmware can now use memory for its own use.
5. At the end of PEI stage the PEI kernel must authenticate DXE volume before handing off control to DXE IPL. As part of the build process, the digest of DXE volume is saved within PEI volume. PEI core computes the digest of DXE volume and compares it against saved digest in PEI volume. If the digest matches, the DXE IPL is considered trusted and PEI kernel code continues to load the DXE driver.
6. At this point the BIOS boot process moves to the DXE phase.
7. The DXE core authenticates various DXE driver modules, such as Option ROMs, third party DXE drivers, etc. that are not part of the SPI flash (i.e. not part of DXE volume). The authentication steps follow secure boot flow as outlined in UEFI specification.
8. If the BIOS is using integrated TPM, then the iTPM drivers are loaded to perform measurements of other DXE components. The iTPM DXE driver allocates necessary memory space for TPM request/response buffer and builds the TPM2 Static ACPI table as outlined in the Microsoft whitepaper on “Trusted Execution Environment ACPI Profile”. In addition it extends necessary protocol outlined in TCG specification to aid BIOS to perform measurement and other TPM use. A platform should only include discrete or integrated TPM and not both. The only exception is AMD reference board where both kind of TPM are present on board for validation purposes only. In such validation boards BIOS need to support both kind of TPM driver and provide setup option (default to internal TPM) for test user to select discrete vs. internal TPM device for test.
9. Next the SMM environment is built and SMM drivers are loaded. The SMM drivers are needed to handle PSP firmware storage request. PspP2Cmbox SMM driver: This driver handles SMI requests coming from PSP firmware and also provides information to PSP FW of how to trigger specific SMI to BIOS. The Fake SMI has been selected as the SMI source. Currently all the P2C mailbox commands are only used for providing the SPI ROM access to PSP FW. Once the SMM driver handler receives the request from PSP FW, it will call

IBV/OEM customized storage library in the backend. Interface of the customized storage library is same as EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.

a. The customized storage library operates on SPI space for PSP NVRAM and enables write/erase operation on that space. It also ensures NVRAM region is appropriately locked against unauthorized update on this SPI space. Note, this library is required to work in SMM mode, and should not depend on any boot available services.
b. The PspP2Cmbox SMM driver will perform the following steps:
    1. The driver allocates transfer buffer in SMRAM for PSP to pass the parameter.
    2. The driver sends a mailbox command "MBoxBiosCmdSmmInfo" to the PSP firmware and informs the space reserved for PSP communicating as well as other relevant information for PSP to trigger SMI at runtime.
    3. When PSP firmware triggers SMI for BIOS services, the callback handler checks the integrity of data that PSP firmware writes into SMM buffer and uses EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL liked library to service PSP NVRAM access request.

10. Now PSP firmware can call the BIOS services to store TPM data.
11. A new PSP DXE/SMM driver is loaded to handle the resume path. This driver prepares the BIOS resume code (discussed in Chapter 5, BIOS S3-Resume Path Handling, on page 36) in memory and sends a mailbox command to inform the PSP firmware about the location of the BIOS S3 resume vector. In addition it registers to service the SlpxSx SMM trap. (See Chapter 5 on page 36 for more details on S3 resume path).
12. The IBV Storage code needs to lock the PSP and other critical regions of the SPI flash. The SPI region must be locked before running any Option ROMs or other non-System ROM code and only trusted SMM code should be able to unlock and update this region.
13. PSP DXE driver will register a callback on ReadyToBoot event to perform the tasks right before handling control to OS boot loader. The details steps of the callback are:
    a. Initialize the RDRAND instruction related register
    b. Save the SMM resume vector and Core context to specific MSR *(**Note**, should be only called once during boot*)
    c. Send MboxBiosCmdBootDone C2P mailbox command to PSP FW
14. The remaining DXE drivers are loaded and finally OS boot loader is loaded and after proper authentication BIOS hands off control to the OS boot loader.

4.4.3 BIOS Runtime Functionality

The high-level execution flow during runtime is listed below.

1. After BIOS hands off control to the OS boot loader, the OS boot loader loads the operating system (e.g. Windows 8).
2. The OS takes control and loads rest of the OS drivers.

3. The OS references the iTPM ACPI table and loads iTPM OS driver to offer TPM support.
4. The OS uses the TPM software stack and sends TPM commands that are handled by PSP firmware. The PSP firmware services any OS requests for TPM commands.
5. If the PSP firmware needs to perform a write to SPI storage, it performs the following additional steps:
   a. PSP firmware copies necessary data as well as command parameters into SMM space reserved for PSP-BIOS communication.
   b. PSP firmware triggers SMI for BIOS to service the request. BIOS SMM entry code invokes all registered SMM handlers. In this case, the P2CMbox SMM Handler gets control and checks if the PSP firmware generated the SMM.
   c. The P2CMbox SMM callback handler locates and invokes the EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL liked library to write the data into SPI.
   d. Finally, the P2CMbox SMM driver reports the appropriate status to the PSP firmware and resumes out of SMM mode.
6. The PSP firmware returns the appropriate status and data values to the operating system.

Chapter 5 BIOS S3-Resume Path Handling

5.1 BIOS S3 Transition Flow on ACPI Aware OS

BIOS implements the following support for S3 transitions:

1. During the cold boot path, the BIOS sets up an SMM trap on the Sleep transition.
2. When the OS transitions to the S3 path, it writes the SLP_TYP to the PmControl IO register to transition the platform to the S3 state.
3. This action generates an SMM trap and the BIOS SMM handler gets control.
4. The BIOS SMM routine sends mailbox command “MBoxBiosCmdSxInfo” to the PSP firmware and this mailbox command informs PSP about the final Sx sleep state the system will transition to.
5. BIOS then waits for the PSP to finish the pre-S3 transition items.
6. Once the PSP acknowledges mailbox command completion, the BIOS finally writes the SLP_TYPE value to the PmControl IO register to transition the system to the sleep state.

5.2 BIOS S3 Resume

Under secure mode the PSP firmware restores the memory during resume. Hence, the DRAM is already available when the x86 cores come out of reset on resume.

The availability of DRAM provides an opportunity to optimize the BIOS resume path. For example, there is no need to perform Cache as RAM initialization. Also, a new CPU MSR is added and any write to this MSR results in CPU context (MSR, microcode, etc.) being automatically saved in protected fenced memory. During resume from sleep transition, the CPU core automatically restores to this CPU context saved in fenced memory; and x86 cores immediately execute from the resume vector that was passed as part of write to MSR during cold boot. The availability of DRAM and properly restored CPU context at x86 reset time provide an opportunity to improve BIOS resume time.

The existing UEFI BIOS currently does not comprehend this kind of S3 resume. Conventionally, BIOS PEI driver is expected to run from ROM code and perform memory restore. Historically, the reset addresses for x86 cold boot and S3-resume were the same (i.e. 0xFFFFFFF0). Hence, during the resume path the BIOS code executed the same PEI driver in SPI space that it would execute during cold boot. In this case the BIOS resume vector can be the DRAM location and the resume path can be optimized to make use of this fact.

Various possible ways to provide the DRAM resume path are listed in the following sections.

5.2.1 Custom Resume Path

One of the quickest ways to resume is to build very specialized resume code in memory that is targeted to perform Si restore and initialization. The BIOS code during boot copies this custom resume code to memory and informs the PSP of the location of this code. The BIOS code marks this region reserved so that OS will leave this region intact. During the resume path the PSP firmware releases x86 core and control is transferred to this custom binary. This custom binary can replay the hardware restore sequence and perform additional initialization. After initialization, control is handed off to the ACPI OS.

Even though this option may provide the quickest resume path, maintainability may be an issue for some BIOS designs. The BIOS PEI volume holds the code to perform Si restore. In the existing UEFI BIOS design, the resume handling is distributed across various PEI drivers; consolidating all these resume components that are spread across various drivers into just one custom binary may be difficult to manage or port, considering PSP may not be enabled in some cases (requiring the traditional resume path).

5.2.2 Separate Firmware Volume for Resume Code

The second option is to have separate firmware volumes specific to the resume path. This firmware volume will contain the resume-specific drivers. The drivers in the volume are aware of DRAM availability to make better use of the environment. The BIOS during cold boot will load these volumes into the memory and perform any necessary address adjustment and data structure initialization. The BIOS will copy this resume firmware volume into memory and mark this region reserved. During the cold boot path, the MSR's are setup such that the S3 resume vector will run code from this volume.

During the resume path, the x86 cores reset to directly run code from this volume in memory; this firmware volume is resume specific and should provide a quick resume path.

5.2.3 SMM Resume

The most secure option is to resume directly to SMM space and perform restore operations from SMM. This path will have following advantages:

- The SMM code is protected from the OS space and this mechanism will keep the resume code away from prying eyes and protect it from alteration.
- Unlike PEI code that requires some data structure initialization (HOB, etc.), the SMM code is prepared during boot and can be immediately used on resume. This should save some time that would otherwise be spent on PEI kernel initialization.
- The SMM implementation based on "UEFI PI SMM Core interface" specification provides a richer UEFI environment (including 64-bit mode) compared to PEI drivers in terms of PI protocol and driver interaction.

To enable such resume path the BIOS SMM driver, during cold boot, will write to this new MSR while x86 is running in SMM mode; CPU context snapshot will be automatically saved in fenced memory. During resume from S3 the CPU will restore context from fenced memory and reset in SMM mode and run the resume vector in SMM memory that was passed as part of MSR write during cold boot. The SMM resume vector will perform basic initialization and call other registered SMM-driver callback handlers to perform respective Si restore. Once the restore operation is complete the SMM driver will update the SMM save state such that resume (rsm) will cause x86 core to run a piece of BIOS code that will be copied outside memory at a reserved location; this BIOS code will finally jump to OS resume vector per ACPI FACS WakeVector.

This implementation will also require substantial changes in existing BIOS code to move the resume specific code from PEI driver to SMM drivers.

5.2.4 Modified Conventional Resume

Another simpler option is to perform a conventional resume with some improvements. In this path, the x86 resume from sleep path will begin the x86 core executing code from DRAM and then jump to ROM code to continue conventional resume. During the cold boot the S3 reset vector code is copied, via write to new MSR, at OS visible memory or protected SMM space. In one such implementation the BIOS SMM function perform special MSR write to save CPU context and resume vector in SMM space as explained before. Just like above case the BIOS SMM code will write to the MSR so that on resume from S3 when PSP firmware restores DRAM and x86 microcode restores CPU context from memory the x86 core will jump to SMM code on X86 reset. This SMM function will update the SMM save area of each core and resume out of SMM to jump to alternate BIOS ROM entry point in PEI Core code. The SMM save state that will be updated by SMM resume driver includes information regarding location of PEI GDT table, RIP and RSP for the code outside SMM; e.g., the GDT location in SMM save area is patched to point to GDT table in ROM; in addition stack pointer (RSP) location in SMM save area will be set to use BIOS reserved memory locations as stack and finally the RIP in SMM state is patched to jump to alternate PEI entry point in PEI volume. The SMM code may further authenticate PEI volume, if OEM desires; after the basic setup the SMM code will resume (RSM) out of SMM mode and the control will be transferred to alternate entry point of PEI core. The alternate PEI code will then use DRAM and not cache (as RAM) for stack and continue with regular resume path and later handoff control to OS. Other enhancements can be added in the PEI kernel code on resume path.

On AMD reference boards; this approach will be used for initial validation.

Chapter 6 TPM Software Interface

The PSP software solution-stack offers firmware based TPM 2.0 services based on Microsoft whitepaper – “Trusted execution environment ACPI profile”.

The TPM software interface is being defined by the TCG’s PC client work group and will be ratified as part of TCG PC Client Specific Platform TPM Specification; the current draft TPM software interface section of this specification is referred to as the Command/Response Buffer Interface and is defined in the TPM Command/Response Buffer Interface w/Locality Support, Version 0.56, DRAFT, 01-09-2013. This specification is undergoing review and is expected to be ratified as a TCG standard defining the TPM 2.0 interface.

6.1 TPM 2.0 Command/Response Buffer Interface

This interface defines a control area structure as shown in Table 9. The physical address of this control area is specified in a TPM 2 ACPI table set up by the BIOS.

The command/response buffers used in this interface are allocated in DRAM by BIOS once the memory is available.

Table 9. Control Area Layout

Field	Byte Length	Offset	Description
Miscellaneous	4	00h	Used for Power Transition
Status	4	04h	SET by the TPM to indicate an error condition or that it is in idle state
Cancel	4	08h	SET by software to abort command processing
Start	4	0Ch	SET by software to indicate that a command is available for processing.
Interrupt Control	8	10h	Reserved. (Must be zero.)
Command Size	4	18h	Size of the Command buffer
Command Address	8	1Ch	This field contains the physical address of the command buffer.
Response Size	4	24h	Size of the Response Buffer
Response Address	8	28h	This field contains the physical address of the response buffer.

This interface is described under section 1.2 of TPM Command/Response Buffer Interface w/Locality Support, Version 0.56, DRAFT, 01-09-2013 in greater details including how these fields are controlled in submitting commands and receiving responses, canceling the commands

with comprehensive state and sequence diagrams. As such, those details are not repeated here and specification should be consulted for implementation.

6.2 AMD Implementation of TPM 2.0 Interface

TPM interface definition structure as defined in the TPM Command/Response Buffer Interface w/Locality Support, Version 0.56, DRAFT, 01-09-2013 that includes the interface identifier, control area and control area extension are mapped to a set of CPU-PSP MMIO message registers, as shown in Figure 5.

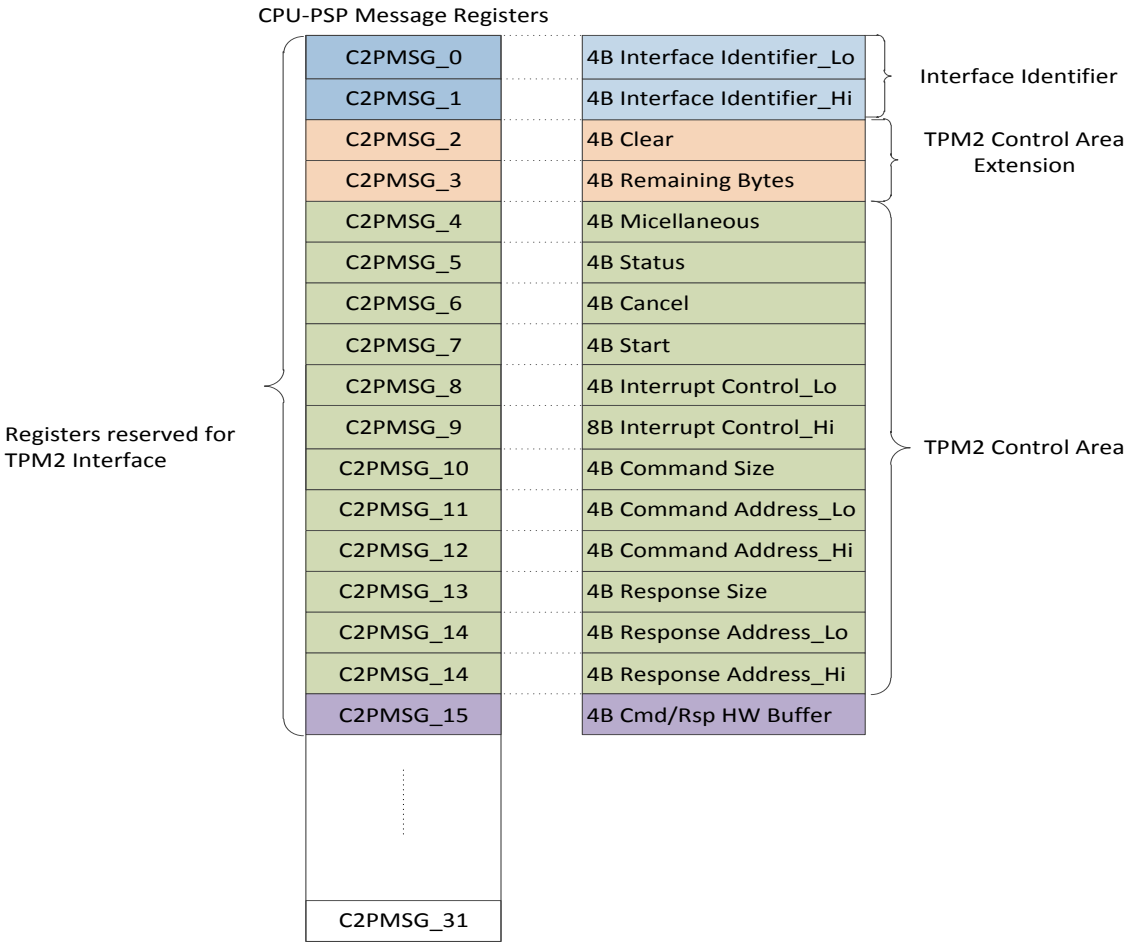

Figure 5. TPM2 Command/Response Buffer Interface

The TPM interface identification structure as defined in the TPM Command/Response Buffer Interface w/Locality Support, Version 0.56, DRAFT, 01-09-2013, which is common to both TPM 1.2 and TPM 2.0 is expected to be located at a well-known fixed physical address. However, in AMD's Mullins implementation of TPM 2.0, it is not possible for this structure to be placed at a fixed address as these fields are mapped to CPU-PSP mailbox registers as shown in Figure 5.

Therefore, the BIOS iTPM driver that initializes and uses this interface is required to be aware of the AMD specific implementation on where this structure is placed.

Once, BIOS has completed DRAM training it initializes the Command/Response Buffer interface by:

- Building the TPM2 static ACPI table, and set the start method to 2. (Uses the ACPI Start method.)
- Reserving TPM 2.0 command/response buffers in DRAM,
- Updating the ACPI control area with the physical address of the command/response buffers.
- Implement TPM ACPI Start Method reside in the _DSM.

Note: *TPM 2.0 interface and TPM 2.0 commands are only supported once the DRAM becomes available.*

Chapter 7 BIOS PSP Mailbox interaction

The BIOS-to-PSP and PSP-to-BIOS communication interfaces are implemented as mailboxes mapped to a set of MMIO CPU-PSP registers and to a portion of SMM memory area reserved for PSP by BIOS respectively, as shown in Figure 6:

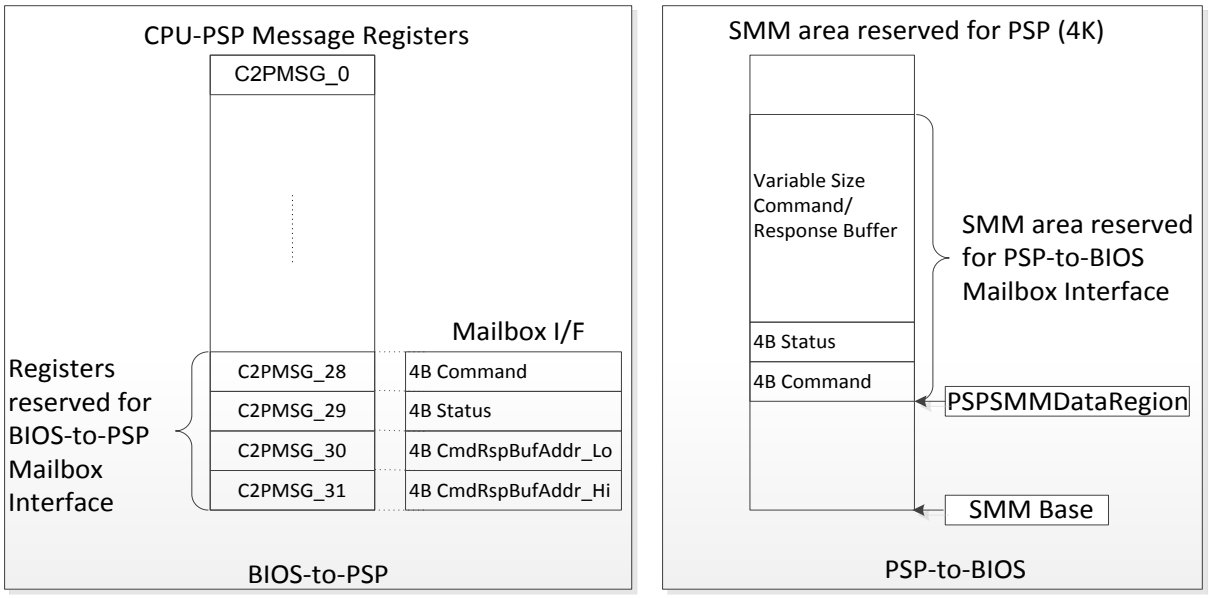

Figure 6. BIOS-PSP Mailbox Interface

The mailbox interface consists of 4-byte status and command fields; and a command/response buffer.

In the case of BIOS-to-PSP mailbox the CmdRspBufAddr_Lo and CmdRspBufAddr_Hi fields are set to the bits 0:31 and bits 32:63 of a 64-bit physical address of the command/response buffer in DRAM.

In the case of PSP-to-BIOS mailbox the command/response buffer starts at offset 0x08 from the PSPSMMDataRegion.

The command field is a 32-bit value initialized to zero by the target. The host submitting the command first writes the command data into the command buffer and then writes the 32-bit command identifier to this field to indicate a new command has been placed in the command buffer. It then waits for the command field to be cleared by the target.

The target processing the command clears this field when it has completed processing the command.

The status field consists of bit-fields indicating the status of the interface as well as the last-processed command. The target sets the status bits immediately before clearing the command field.

The host, after writing to the command field, waits for the target to clear the command field and when the command field is cleared reads the status bits and response data from the command/response buffer.

Figure 7 shows the command execution sequence over this mailbox interface.

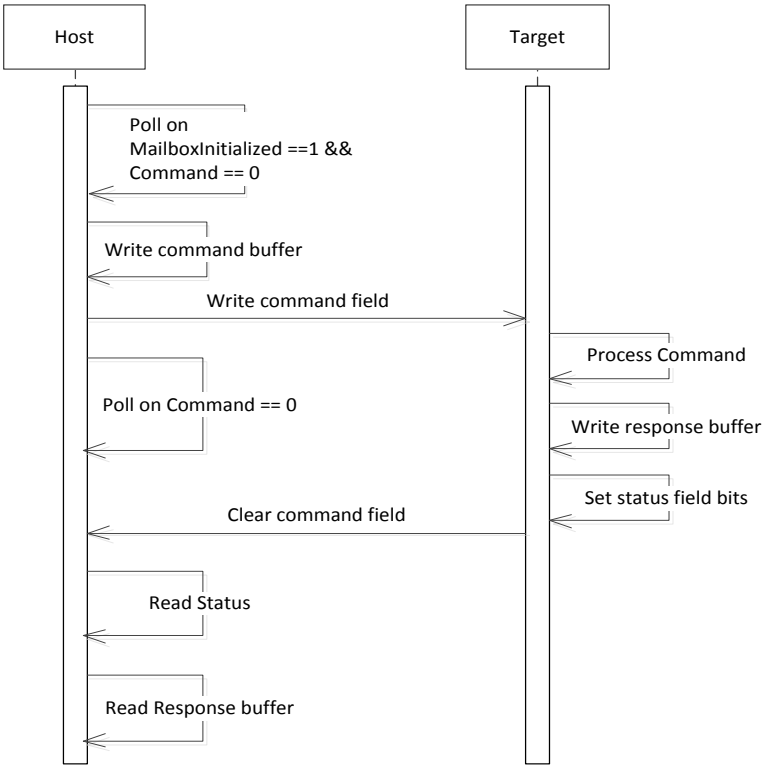

Figure 7. BIOS-PSP Mailbox Command Execution Sequence

PSP FW, in its role as a target on the BIOS-to-PSP communication interface, can enable interrupts to be generated on BIOS writes to the command register.

PSP FW, in its role as a host on the PSP-to-BIOS communication interface uses the System Management Interrupt (SMI) mechanism to generate interrupt to BIOS. PSP FW generates the SMI interrupt by writing a pre-defined value either to an IO or Memory address. The value to be written and the type of address are communicated to PSP FW by BIOS through BIOS-to-PSP communication interface.

Table 8 defines the bit-fields of the status field:

Table 10. BIOS-PSP Mailbox Status Register Bit Fields

Field Name	Bit Index	Description/Purpose
MboxInitialized	0	Set by the target to indicate the mailbox interface state. 0 – Interface is not initialized. 1 – Interface is initialized.
Error	1	Set by the target to indicate error condition of the last processed command. 0 – No Error
Terminated	2	Set by the target to indicate the last command is terminated.
Halt	3	Set by the target to indicate unrecoverable error at the interface.
Recovery required	4	Set by the target to indicate some PSP entry point by PSP directory has been corrupted.
Reserved	31:5	Reserved and set zero.

Table 11 defines the set of BIOS-to-PSP commands currently defined:

Table 11. BIOS-to-PSP Mailbox Commands

Command	Value	Description/Purpose
MboxBiosCmdDramInfo	0x01	Notification that DRAM is trained and ready for use.
MboxBiosCmdSmmInfo	0x02	Provides details on SMM memory area reserved for PSP. It includes the physical addresses of SMM Base and PSP SMM data region and the length of PSP SMM data region.
MboxBiosCmdSxInfo	0x03	Notification that the platform is entering S3-suspend state.
MboxBiosCmdRsmInfo	0x04	Information on BIOS Resume Module stored in SMM memory which includes the BIOS resume vector and size of the resume code.
MboxBiosCmdPspQuery	0x05	Command to get the list of capabilities supported by PSP FW. This is used to communicate if iTPM is supported or not in PSP FW.
MboxBiosCmdBootDone	0x06	Notification that BIOS has completed BIOS POST.
MboxBiosCmdClearS3Sts	0x07	Inform PSP clear S3ExitReset
MboxBiosS3DataInfo	0x08	Bios will send this command to inform PSP to save the blob, which needed to restore memory during resume from S3
MboxBiosCmdNop	0x09	Notification that BIOS has completed PSP-to-BIOS command submitted via PSP-to-BIOS mailbox interface.

Table 12 defines the set of PSP-to-BIOS commands currently defined:

Table 12. PSP-to-BIOS Mailbox Commands

Command	Value	Description/Purpose
MboxPspCmdSpiGetAttrib	0x81	Get SPI-ROM attributes such as the size and polarity
MboxPspCmdSpiSetAttrib	0x82	Set SPI-ROM attributes
MboxPspCmdGetBlockSize	*0x83*	Get SPI-ROM block size
MboxPspCmdReadFV	0x84	Read PSP NVRAM firmware volume
MboxPspCmdWriteFV	0x85	Write SP NVRAM firmware volume
MboxPspCmdEraseFV	0x86	Erase PSP NVRAM firmware volume

The PSP-to-BIOS mailbox commands are used by PSP FW in managing the SPI-ROM area reserved for PSP to be used as non-volatile RAM (NVRAM) storage. PSP FW uses the firmware volume block2 protocol defined by the UEFI Platform Initialization Specification [4] to manage the NVRAM storage as a firmware volume.

The BIOS-PSP mailbox related data structures can be found under section 12.2 as part of Appendix A on page 60.

7.1 BIOS to PSP Mailbox Commands

To send mailbox command to PSP, the BIOS builds the command/response buffer in system memory and updates the 64 bit MMIO CmdRspBufAddr pointer to point to this command/response buffer in the memory. BIOS then writes the mailbox command, as listed below, in the MMIO command register and waits for the command register to return to zero (as controlled by PSP firmware). Once the command is processed, BIOS reads the status of the mailbox operation from the mailbox status register as well as the status value in the command/response buffer.

The generic format of the command/response buffer is below

```
typedef struct {
  UINT32    TotalSize;
  UINT32    Status;
} MBOX_BUFFER_HEADER;

typedef struct {
  MBOX_BUFFER_HEADER          Header;
  COMMAND_SPECIFIC_BUFFER     Buffer;
} MBOX_COMMAND_RESPONSE_BUFFER;
```

Each command/response buffer starts with a standard mailbox buffer header. The field TotalSize is set to the size of command/response buffer. The field Status is updated by PSP firmware to

reflect the status of the command that is being handled by PSP firmware. The standard mailbox buffer header is followed by a command-specific buffer that is different per each mailbox command.

7.1.1 MboxBiosCmdDramInfo (MboxCmd = 0x01)

BIOS sends this command to PSP after memory is configured by BIOS. After this mailbox command, the PSP will use the protected fenced DRAM memory that is not accessible to X86 cores.

No additional parameters are needed for this command. CmdRspBufAddr points to a standard mailbox buffer.

7.1.2 MboxBiosCmdSmmInfo (MboxCmd = 0x02)

After the SMM environment is ready the PSP SMM driver sends this mailbox command to PSP firmware. The PSP firmware will use information supplied via this command to later trigger SMM, and request BIOS services for the PSP NVRAM region in SPI space. The command/response buffer is of structure type MBOX_SMM_BUFFER as defined below:

```
typedef struct {
  UINT64  Address;
  UINT32  AddressType;
  UINT32  ValueWidth;
  UINT32  ValueAndMask;
  UINT32  ValueOrMask;
} SMM_TRIGGER_INFO;

typedef struct {
  UINT64            SmmBase;
  UINT64            SmmLength;
  UINT64            PsPSmmDataRegion;
  UINT64            PspSmmDataLength;
  SMM_TRIGGER_INFO  SmmTrigInfo;
} SMM_REQ_BUFFER;
```

The command/response buffer structure starts with a standard mailbox header.

`SmmBase-` This field provides the location of SMM base.

`SmmLength-` This field provides the length of SMM space.

`PspSmmDataRegion-` The PSP SMM driver allocates SMM space for PSP use and this field provides the location of SMM space that is carved out for PSP use.

`PspSmmDataLength-` This field provides the length of the PSP region in SMM space. PSP firmware must not access SMM space beyond this allotted space.

SMM_TRIGGER_INFO provides information on how the PSP firmware can trigger an SMI. There are various ways to trigger SMI- write to MMIO, IO or PCI config space. This structure provides the following information:

`Address-` The Physical address where PSP needs to write to trigger SMI

`AddressType-` The type to access to trigger SMI; IO (0), MMIO (1) or PCI (2)

`ValueWidth-` Write length- Byte (0) , Word (1) , Dword (2), Qword (3) on Address

`ValueAndMask-` AndMask

`ValueOrMask-` OrMask

To trigger an SMI, the PSP firmware first reads from the Address location as defined by AddressType and performs an "AND" operation based on ValueAndMask followed by an "OR" operation based on ValueOrMask and writes it back to the Address location.

7.1.3 MboxBiosCmdSxInfo (MboxCmd = 0x03)

The BIOS SMM driver sends this command right before the system transitions to sleep state. PSP firmware performs any last minute save operations just prior to S3 transition when BIOS sends this command to PSP.

The command/response buffer has the following structure. It starts with the standard mailbox buffer header, followed by the SleepType field that informs the PSP of the sleep state the system is about to transition to.

```
typedef struct {
  MBOX_BUFFER_HEADER    Header;
  UINT8              SleepType;
 } MBOX_SX_BUFFER;
```

7.1.4 MboxBiosCmdRsmInfo (MboxCmd = 0x04)

BIOS send this command to PSP if the platform design requires PSP to authenticate the BIOS resume code before releasing the x86 core. The BIOS informs PSP of the location and length of BIOS resume code in memory. In response to this command, PSP firmware calculates the digest of this memory region. During resume, PSP firmware accesses memory and calculates the digest of resume code; it compares this digest against the boot time calculated digest to ensure the BIOS Resume code has not been tampered. PSP releases the x86 core if the calculated digest remains unchanged. If the digest is changed, PSP will not release x86 core.

The command/response buffer has the following structure. It starts with the standard mailbox buffer header followed by ResumeVectorAddress and ResumeVectorLength:

```
typedef struct {
  MBOX_BUFFER_HEADER    Header;
  UINT64             ResumeVectorAddress;
  UINT64             ResumeVectorLength;
  } MBOX_RSM_BUFFER;
```

ResumeVectorAddress- Location of BIOS resume vector in memory

ResumeVectorLength- Length of BIOS resume vector in memory

7.1.5 MboxBiosCmdPspQuery (MboxCmd = 0x05)

BIOS sends this command to PSP to find the capabilities offered by PSP. In response to this command, PSP firmware updates the Capability field in the command/response buffer as shown below:

```
typedef struct {
  UINT32 Capabilities;
 } CAPS_REQ_BUFFER;

// Bitmap defining capabilities
#define PSP_CAP_TPM (1 << 0)

typedef struct {
  MBOX_BUFFER_HEADER  Header;
  UINT32              Capability;
 } MBOX_CAPS_BUFFER
```

The command/response buffer starts with a standard mailbox buffer followed by a Capability field that is updated by PSP firmware. Each bit in the Capability field represents a feature supported by PSP. If PSP supports firmware TPM then Bit0 of Capability field will be set after PSP firmware process the command.

7.1.6 MboxBiosCmdBootDone (MboxCmd = 0x06)

BIOS sends this command to PSP before handing off control to the OS. This mailbox command is an indication to PSP firmware to no longer handle any more BIOS mailbox commands other than commands coming from SMM space. After this command is sent, PSP will only handle a mailbox command if the command buffer is within SMM region.

No additional parameters are needed for this command. CmdRspBufAddr points to the mailbox buffer.

7.1.7 MboxBiosCmdClearS3Sts (MboxCmd = 0x07)

BIOS sends this command to PSP after last AP resume from sleep. This command inform PSP that all the cores are successfully resumed out of S3 state

No additional parameters are needed for this command. `CmdRspBufAddr` points to the mailbox buffer.

7.1.8 MboxBiosS3DataInfo (MboxCmd = 0x08)

BIOS sends this command to PSP after building the necessary memory controller restore data block that is used by PSP firmware to restore the memory controller during resume path. BIOS sends this data block as part of this mailbox command parameter.. PSP FW needs save this blob to a temporary buffer, and save to SPI ROM after the PSP to BIOS mailbox interface has been established. The PSP to BIOS mailbox interface is built via MboxBiosCmdSmmInfo (MboxCmd = 0x02).

Additional parameters associate with this command are like below:

```
typedef struct {
  UINT64 S3RestoreBufferBase;               ///< Address of the blob
  UINT64 S3RestoreBufferSize;               ///< Size of the blob
} S3DATA_REQ_BUFFER;

typedef struct {
  MBOX_BUFFER_HEADER  Header;               ///< Header
  S3DATA_REQ_BUFFER   Req;                  ///< Req
} MBOX_S3DATA_BUFFER;
```

7.1.9 MBOX_S3DATA_BUFFER;MboxBiosCmdNop (MboxCmd = 0x09)

This is the NOP (No Operation) command. BIOS send this command to generate an interrupt to the PSP after servicing PSP to BIOS mailbox requests.

7.2 PSP to BIOS Mailbox Commands

To request BIOS services the PSP firmware builds mailbox commands in SMM space and generates an SMI. PSP firmware uses information per the BIOS mailbox command MboxBiosCmdSmmInfo and uses the PSP SMM area and SMM trigger mechanism to trigger BIOS services. The structure of the BIOS mailbox interface is defined below:

```
typedef struct {
  VOLATILE MBOX_COMMAND     MboxCmd;
  VOLATILE MBOX_STATUS      MboxSts;
  MBOX_BUFFER_HEADER        Buffer;
 } BIOS_MBOX;
```

PSP_MBOX location is offset 0 of SMM region- PSPSMMDataRegion that BIOS SMM driver has allocated for PSP communication. During cold boot path BIOS SMM driver allocates this region and informs PSP firmware about this region through MboxBiosCmdSmmInfo mailbox command. PSP firmware must never use SMM space beyond the allocated SMM space. When BIOS SMM code is ready to serve PSP SMM requests it sets the MBOX_STATUS_INITIALIZED bit in MboxSts register. The Buffer location is updated by PSP firmware with the command specific request buffer. PSP firmware next sets MboxCmd with a specific mailbox command that PSP needs BIOS to service. After the BIOS_MBOX structure is setup by PSP firmware it triggers an SMI to the x86 core. The BIOS SMM driver parses the structure and invokes the appropriate routine to service the PSP request. After the completion of the request the BIOS SMM driver updates the MboxSts register to reflect completion status and clears the MboxCmd location to indicate completion of the PSP command. BIOS finally sends the MboxBiosCmdNop to trigger an interrupt to PSP firmware and return out of SMM mode.

The structure of command/response buffer at offset 8 of BIOS_MAILBOX will change based on PSP command. The generic format of command/response buffer is similar to the BIOS to PSP command and is listed below:

```
typedef struct {
  UINT32    TotalSize;
  UINT32    Status;
} MBOX_BUFFER_HEADER;

typedef struct {
  MBOX_BUFFER_HEADER       Header;
 COMMAND_SPECIFIC_BUFFER   Buffer;
} MBOX_COMMAND_RESPONSE_BUFFER;
```

Each command/response buffer starts with a standard mailbox buffer header. The field TotalSize is set to size of command/response buffer. After the PSP command is serviced the field Status is updated by the BIOS SMM driver to reflect the status of the command requested by PSP firmware. The standard mailbox buffer header is followed by a command specific buffer that is different per each mailbox command.

PSP firmware should never attempt to write/erase SPI space beyond PSP NVRAM region. BIOS SMM code must reject any such access.

7.2.1 MboxPspCmdSpiGetAttrib (MboxCmd = 0x081)

PSP firmware sends this command to find the attributes of the SPI part. PSP firmware send this command to request EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetAttributes() service on PSP NVRAM region in SPI space. In response to this command BIOS updates the Capability field in the Attribute field in the command-specific buffer structure listed below:

```
typedef struct {
  UINT64 Attribute;
 } SPI_ATTRIB_REQUEST;
```

The possible attribute values are defined in the UEFI specification for enum type EFI_FVB_ATTRIBUTES_2.

7.2.2 MboxPspCmdSpiSetAttrib (MboxCmd = 0x082)

At this time this is an unsupported command. PSP firmware sends this command to request EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.SetAttributes() service on PSP NVRAM region in SPI space.

7.2.3 MboxPspCmdSpiGetBlockSize (MboxCmd = 0x083)

PSP firmware sends this command to request EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetBlockSize() service on PSP NVRAM region in SPI space. PSP firmware requires this information to detect the size of each block in PSP NVRAM region.

```
typedef struct {
  UINT64  Lba;
  UINT64  BlockSize;
  UINT64  NumberOfBlocks;
} SPI_INFO_REQ;
```

This structure maps to EFI_FVB_GET_BLOCK_SIZE structure in the UEFI specification.

Note: *The difference from UEFI definition point of view is SPI_INFO_REQ are absolute addresses instead of pointers defined in EFI_FVB_GET_BLOCK_SIZE structure.*

BIOS returns the Lba as (zero), BlockSize and total blocks of PSP NVRAM region.

7.2.4 MboxPspCmdSpiReadFV (MboxCmd = 0x084)

PSP firmware sends this command to request EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.Read() service on PSP NVRAM region in SPI space.

PSP firmware may use this command to read its data from PSP NVRAM space. Generally PSP firmware will directly perform the SPI read and not use this command. The command-specific structure for this command is below

```
typedef struct {
  UINT64  Lba;
  UINT64  Offset;
  UINT64  NumByte;
  UINT8   Buffer[1];
```

```
} SPI_RW_REQ;
```

Lba is the logical block address in NVRAM that PSP firmware requests to read. Offset is the specific location within the Lba block. NumByte field reflects the number of bytes PSP firmware needs to read. BIOS SMM code is expected to read the requested NVRAM offset and update the SMM space starting at offset Buffer.

7.2.5 MboxPspCmdSpiWriteFV (MboxCmd = 0x085)

PSP firmware sends this command to request EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.Write() service on PSP NVRAM region in SPI space. PSP firmware may use this command to write to PSP NVRAM space. To protect the PSP NVRAM region BIOS SMM code is expected to lock this region unless PSP made this write request. The structure for this command is the same as above SPI_RW_REQ. In this command, BIOS reads the SMM location starting at Buffer offset of the structure and writes this buffer content to the appropriate SPI location.

7.2.6 MboxPspCmdSpiEraseFV (MboxCmd = 0x086)

PSP firmware sends this command to request EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.Erase() service on PSP NVRAM region in SPI space.

BIOS SMM driver erases the SPI space in response to this command. The request buffer for this command specific structure is defined below:

```
typedef struct {
  UINT64  Lba
  UINT64  NumberOfBlocks;
} SPI_ERASE_REQ;
```

In response to this command, BIOS erases the SPI space per `Lba & NumberofBlocks` as defined in the structure.

Chapter 8 Platform BIOS Requirements for PSP Implementation

The below BIOS requirement applies when platform BIOS supports Hardware validated boot and PSP firmware TPM feature of Mullins PSP

- Platform BIOS MUST reserve 1MB of space in SPI storage for PSP components. This includes PSP firmware, PSP NVRAM data and various keys in the PSP Directory [IBV|OEM]
- Platform BIOS must include PSP Directory
  For more information about the tool used to build PSP Directory please refer to Appendix A, PSP Directory Structure, on page 58.
  - AGESA PI package must include all PSP and SMU binaries
  - Platform BIOS must include PSP directory. All PSP appropriate entries defined in Table 5,PSP Directory Entry Type Encodings, on page 21 must be included in PSP Directory.[IBV|OEM]
  - Public portion of OEM signing key MUST be signed by AMD and both OEM signed key and AMD public key MUST be present in SPI image and also locatable via PSP directory. [IBV|OEM]
- Platform BIOS MUST set up PSP code and data pages in SPI space such that the PSP code and data page can be independently erased/updated without affecting the rest of the BIOS components in SPI space.[IBV|OEM]
- BIOS MUST support single binary that supports both secure/non-secure boot and resume environment.[IBV|OEM]
- BIOS MUST include necessary support to ensure the trust chain is maintained from x86 BIOS reset to OS handoff. [IBV|OEM]
  - IBV|OEM MUST sign the BIOS RTM code (PEI Volume or SEC volume) using private key based on RSASSA-PSS signing scheme. The private portion of IBV|OEM signing key MUST be protected per IBV|OEM process choice. If IBV|OEM private keys are compromised the overall hardware validated boot feature can be completely compromised across all systems. [IBV|OEM]
  - Platform BIOS MUST build PSP Directory that provide location information of BIOS_RTM location, signed BIOS RTM location and signed BIOS public key in PSP directory.[IBV|OEM]
  - If BIOS RTM volume has only SEC code then it MUST authenticate the PEI volume before handing off control to PEI core.[IBV|OEM]
  - BIOS PEI volume must authenticate DXE and other firmware volume during cold boot and resume before executing BIOS drivers from these volumes to protect the integrity of trust chain. [IBV|OEM]

  - Platform BIOS must authenticate external software component such as external Option Rom and OS Boot loader as defined in secure boot section of UEFI specification.
- Platform BIOS MUST send PSP mailbox commands as outlined in the PSP-BIOS mailbox section. [AGESA/OEM|IBV]
  - AGESA PSP/iTPM PEI, DXE and SMM driver MUST send the all BIOS-PSP mailbox command to PSP at appropriate point during boot. Similarly AGESA PSP/iTPM driver should handle PSP-BIOS mailbox command at runtime[AGESA]
  - Platform BIOS code MUST ensure these AGESA drivers are appropriately loaded and protected.[IBV|OEM]
  - PSP MMIO region MUST be always accessible for BIOS-PSP mailbox communication. [AGESA/IBV|OEM]
- Platform BIOS MUST provide an SMM interface to allow PSP data to be saved in SPI space. This includes the support EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL liked library interface as defined in section 10.3.2 in SMM space that will be used to update PSP NVRAM data.[IBV|OEM]
  - Platform BIOS MUST ensures this interface is inaccessible outside SMM. Also the PSP region of SPI MUST remain write protected outside this interface use such that NVRAM cannot be tempered [IBV|OEM]
  - AGESA P2Cmbox SMM driver MUST use this library interface in SMM space to locate the storage protocol [AGESA]
- BIOS MUST reserve 4K SMM memory for BIOS-PSP runtime communication.[AGESA]
  - AGESA PspP2Cmbox SMM driver MUST reserve region for BIOS-PSP communication [AGESA]
  - Platform BIOS must provide SMM kernel services to allow SMM memory allocation with in SMM space.[IBV|OEM]
- Platform BIOS MUST ensure the SMM environment is protected
  - SMM region MUST be locked before exiting the BIOS boot environment. The SMM code must remain protected such that non-authenticated code cannot access SMM data or code area. Some additional note, the SMM lock need be done after PspDxe driver save CPU core context through the specific MSR, if SMM region need to access during the SMM resume path. [IBV|OEM]
- Platform BIOS MUST ensure the SMI source used by PSP is configured properly for SMM to be non-blocking during runtime.[AGESA]
  - AGESA PSP driver MUST install appropriate SMM handle to service SMI events (FakeSts0/Software SMI) from PSP firmware. AGESA PSP driver must also provide this information to PSP firmware via mailbox interface.
  - Platform BIOS MUST ensure the same SMI mechanism is not used by other SMM component. Platform BIOS must also ensure the chipset registers remain configured to trigger SMI at run time. [IBV|OEM]

- BIOS MUST adequately reserve system memory during boot to appropriately set up stack/heap for DRAM ready x86 SMM resume path on a secure PSP part.[AGESA/IBV|OEM]
  - AMD AGESA SMM driver MUST reserve the SMM location [AGESA]
  - Platform driver MUST produce AMD_PSP_PLATFORM_PROTOCOL instance that AMD AGESA SMM driver can use to reserve SMM space.[IBV|OEM]
  - If platform BIOS build a separate resume volume as outlined in section 5.2, on page 36 the platform BIOS must reserve this memory region to protect against any OS use. Also during resume platform BIOS MUST ensure this resume volume in DRAM has not been tempered.[IBV|OEM]
- BIOS MUST support Windows 8 Secure Boot requirements as specified in Microsoft's certification requirements.[IBV|OEM]
- BIOS MUST support Windows 8 firmware TPM interfaces as specified in Microsoft's whitepaper "Trusted execution environment ACPI profile". This includes reserving part of system memory that will be used by PSP firmware as a TPM Command and Response Buffer.[IBV|OEM]
  - BIOS MUST send fTPM command after memory is available.
  - BIOS MUST protect the PSP code and data area in the SPI space. This area must be write-protected via SMM or physical lock so that only BIOS SMM code can update this region.[IBV|OEM]
- BIOS flash utility MUST ensure PSP firmware is properly updated and also the SPI region for PSP firmware is properly protected during regular boot.[IBV|OEM]
  - During flash update platform BIOS must ensure the integrity of contents of PSP Directory as well as various binaries referenced by PSP directory. PSP firmware searches for PSP Directory in multiple locations as defined in Section 4.1.1, on page 19. BIOS vendor MAY use these other location as backup location. PSP firmware looks for PSP directory in the order as defined in Table 1
- BIOS must support UEFI TPM protocol for TPM devices. For ex. "TCG EFI Protocol Specification" for onboard discrete TPM1.2 device and EFI_TREE_PROTOCOL [9] for TPM2.0 devices.[IBV|OEM]
- When a platform BIOS (such as Larne reference board) supports both discrete and firmware TPM for development and validation purposes ONLY, the platform BIOS MUST support both kind of TPM stack (i.e., TPM1.2/TPM2.0 as well as dTPM/fTPM) in the BIOS ROM image. During cold boot the platform BIOS MUST load appropriate TPM driver stack based on user configuration (such as BIOS setup option) option. On AMD reference validation board, the default TPM selection MUST be internal firmware TPM use.[IBV|OEM]
- Platform BIOS MAY use PSP entry type 0x80-0xFF for OEM or IBV specific proprietary use. PSP Entry Type from 0x00 to 0x7F is reserved for AMD use.
- Platform BIOS MAY use PSP Hardware Cryptographic accelerator to improve the boot time associated with cryptography operation such as AES, SHA, RSA, ECC, RNG.[AGESA/IBV|OEM]

  - AMD AGESA MAY provide the library for the base CCP operation mentioned above [AGESA]
  - IBV|OEM CCP library MAY use the AMD AGESA CCP library to make use of PSP CCP.
- IBV BIOS MUST ensure SPI decode range in FCH controller remain properly programmed for PSP firmware to access other PSP component from SPI space.

Chapter 9 Standards

The PSP and software components must comply with many standards including the following:

9.1 UEFI 2.3.1c Chapter 27 Secure Boot

Affected component(s): System BIOS

http://www.uefi.org/specs/

9.2 Microsoft® Trusted Execution Environment UEFI Protocol

Affected component(s): System BIOS, Firmware TPM, PSP OS

http://msdn.microsoft.com/en-us/library/windows/hardware/jj923068.aspx

9.3 Microsoft®Trusted Execution Environment ACPI Profile

Affected component(s): System BIOS, Firmware TPM, PSP OS

http://msdn.microsoft.com/en-us/library/windows/hardware/jj923067.aspx

9.4 AMD PSP 1.0 Software Architecture Design Document

Affected component(s): System BIOS, Firmware TPM, PSP OS

Appendix A PSP Directory Structure

```
typedef struct {
    UINT32  PspCookie;      // "$PSP"
    UINT32  Checksum;       // 32 bit CRC of header items below and the entire
table
    UINT32  TotalEntries;   // Number of PSP Entries
} PSP_DIRECTORY_HEADER;

enum _PSP_DIRECTORY_ENTRY_TYPE {
    AMD_PUBLIC_KEY                  = 0,    // PSP entry pointer to AMD public
key
    PSP_FW_BOOT_LOADER              = 1,    // PSP Entry points to PSP boot
loader in SPI space
    PSP_FW_TRUSTED_OS               = 2,    // PSP Entry points to PSP Firmware
region in SPI space
    PSP_FW_RECOVERY_BOOT_LOADER     = 3,    // PSP recovery boot loader
    PSP_NV_DATA                     = 4,    // PSP entry points to PSP data
region in SPI space
    BIOS_PUBLIC_KEY                 = 5,    // PSP entry points to BIOS public
key stored in SPI space
    BIOS_RTM_FIRMWARE               = 6,    // PSP entry points to BIOS RTM
code (PEI volume) in SPI space
    BIOS_RTM_SIGNATURE              = 7,    // PSP entry points to signed BIOS
RTM hash stored  in SPI space
    SMU_OFFCHIP_FW                  = 8,    // PSP entry points to SMU off-chip
firmware
    PSP_AMD_SECURE_DEBUG_KEY        = 9,    // AMD Secure Debug key
    PSP_SECURE_OS_SIGNING_KEY       = 10,   // PSP Secure OS OEM signing key

    AMD_SOFT_FUSE_CHAIN_01          = 11,   // PSP entry pointer to 64bit PSP
Soft Fuse Chain
    PSP_BOOT_TIME_TRUSTLETS         = 12,   // PSP entry points to boot-loaded
trustlet binaries
    PSP_BOOT_TIME_TRUSTLETS_KEY     = 13,   // PSP entry points to key of the
boot-loaded trustlet binaries

};

typedef UINT32 PSP_DIRECTORY_ENTRY_TYPE;

typedef struct
{
    PSP_DIRECTORY_ENTRY_TYPE    Type;       // Type of PSP entry; 32 bit long
    PSP_UINT32                  Size;       // Size of PSP Entry in bytes
    union
    {
       PSP_UINT64       Location;   // Address of PSP Entry in SPI-ROM space
```

```
       PSP_UINT64       Value;      // Value of certain PSP Entry in SPI-ROM
space;
    }Content;// Location/Value union type
} PSP_DIRECTORY_ENTRY;


typedef struct {
    PSP_DIRECTORY_HEADER  Header;
    PSP_DIRECTORY_ENTRY   PspEntry[1];    // Array of PSP entries each pointing
to a binary in SPI flash
                                          // The actual size of this array
comes from the
                                          // header
(PSP_DIRECTORY.Header.TotalEntries)
} PSP_DIRECTORY;
```

Appendix B PSP –BIOS Mailbox

```
//===========================================================================
```

// Define Mailbox Command

```
//===========================================================================
```

// Mbox command list. Only one command can be send till target processes it; the only exception is Abort command that BIOS may send in case of timeout etc.

///

```
typedef enum {
  MboxCmdRsvd               = 0x00,   ///< Unused

  MboxBiosCmdDramInfo       = 0x01,   ///< Bios -> PSP: Memory DRAM information
(ie. PspBuffer address etc)
  MboxBiosCmdSmmInfo        = 0x02,   ///< Bios -> PSP: Bios will provide SMM
inf - SmmBase,
                                      ///
PspSmmDataRegion,PspSmmDataRegionLength, SoftSmiValue, SoftSmiPort
  MboxBiosCmdSxInfo         = 0x03,   ///< Bios -> PSP: Sx transition info (S3,
S5)
  MboxBiosCmdRsmInfo        = 0x04,   ///< Bios -> PSP: Resume transition info
(Vector, Size of resume code)
  MboxBiosCmdPspQuery       = 0x05,   ///< Bios -> PSP: Bios Find supported
feature
  MboxBiosCmdBootDone       = 0x06,   ///< Bios -> PSP: Bios is done with BIOS
POST
  MboxBiosCmdClearS3Sts     = 0x07,   ///< Bios -> PSP: Inform PSP clear
S3ExitReset
```

MboxBiosS3DataInfo = 0x08, ///< Bios -> PSP: Bios will send this command to inform PSP to save the data needed to restore memory during resume from S3

MboxBiosCmdNop = 0x09, ///< Bios -> PSP: Bios will send this NOP command to indicate to PSP that is is done servicing PSP SMM request

```
  MboxPspCmdSpiGetAttrib    = 0x81,   ///< PSP -> BIOS: Get location of PSP
NVRam region
  MboxPspCmdSpiSetAttrib    = 0x82,   ///< PSP -> BIOS: Get location of PSP
NVRam region
  MboxPspCmdSpiGetBlockSize = 0x83,   ///< PSP -> BIOS: Get Block size info
  MboxPspCmdSpiReadFV       = 0x84,   ///< PSP -> BIOS: Read PSP NVRAM firmware
volume
  MboxPspCmdSpiWriteFV      = 0x85,   ///< PSP -> BIOS: Write PSP NVRAM
firmware volume
```

```
  MboxPspCmdSpiEraseFV      = 0x86,   ///< PSP -> BIOS: Erase PSP NVRAM
firmware volume


  MboxCmdAbort              = 0xfe,   ///< Abort the last command (BIOS to PSP
in case of timeout etc)
} MBOX_COMMAND;


//=============================================================================
//
// Define Mailbox Status field
//
//=============================================================================
//
/// MBox Status MMIO space
///
typedef struct {
  UINT32 MboxInitialized:1;   ///< Target will set this to 1 to indicate it is
initialized  (for ex. PSP/TPM ready)
  UINT32 Error:1;             ///< Target in adddtion to Done bit will also set
this bit to indicate success/error on last command
  UINT32 Terminated:1;        ///< Target will set this bit if it aborted the
command due to abort request
  UINT32 Halt:1;              ///< Target will set this error if there is
critical error that require reset etc
} MBOX_STATUS;

//
// Above defined as bitmap
#define MBOX_STATUS_INITIALIZED       0x00000001ul    ///< Mailbox Status:
Initialized
#define MBOX_STATUS_ERROR             0x00000002ul    ///< Mailbox Status:
Error
#define MBOX_STATUS_ABORT             0x00000004ul    ///< Mailbox Status:
Abort
#define MBOX_STATUS_HALT              0x00000008ul    ///< Mailbox Status: Halt

///
/// Each MMIO Block will have Command, Status and Buffer pointer entries.
/// The 8 dword wide MMIO mailbox will be part of PSP-CPU MMIO space
///
typedef struct {
  VOLATILE MBOX_COMMAND     MboxCmd;        ///< Mbox Command 32 bit wide
  VOLATILE MBOX_STATUS      MboxSts;        ///< Mbox status  32 bit wide
  MBOX_BUFFER               *Buffer;        ///< 64 bit Ponter to memory with
additional parameter.
} PSP_MBOX;
```

```
///
/// structure in DRAM for additional parameter
///
typedef struct {
  UINT32    TotalSize;                    ///< Total Size of MBOX_BUFFER
(including this field)
  UINT32    Status;                       ///< Status value if any:e
  //UINT8   ReqBuffer[x];                 ///< X byte long Request buffer for
additional parameter.
} MBOX_BUFFER_HEADER;


//===========================================================================
//
// Below defines Request buffer for various commands. This structure is based
on Command
//
//===========================================================================
///
/// structure of ReqBuffer for MboxBiosS3DataInfo mailbox command
///
typedef struct {
  UINT64 S3RestoreBufferBase;           ///< PSP reserve memory near TOM
  UINT64 S3RestoreBufferSize;           ///< Size of PSP memory
} S3DATA_REQ_BUFFER;

/// MBOX buffer for S3Info data to bring memory out of self refresh info
typedef struct {
  MBOX_BUFFER_HEADER  Header;           ///< Header
  S3DATA_REQ_BUFFER   Req;              ///< Req
} MBOX_S3DATA_BUFFER;


/// Define structure of SMM_TRIGGER_INFO
typedef struct {
  UINT64  Address;                      ///< Memory or IO address (Memory
will be qword, IO will be word)
  UINT32  AddressType;                  ///< SMM trigger typr - Perform
write to IO/Memory
  UINT32  ValueWidth;                   ///< Width of value to write (byte
write, word write,..)
  UINT32  ValueAndMask;                 ///< AND mask of value after
reading from the address
  UINT32  ValueOrMask;                  ///< OR Mask of value to write to
this address.
} SMM_TRIGGER_INFO;

///
```

```
/// structure of ReqBuffer for MboxBiosCmdSmmInfo mailbox command
///
typedef struct {
  UINT64            SMMBase;               ///< SMM TSeg Base
  UINT64            SMMLength;             ///< Length of SMM area
  UINT64            PSPSmmDataRegion;      ///< PSP region base in Smm space
  UINT64            PspSmmDataLength;      ///< Psp region length in smm space
  SMM_TRIGGER_INFO  SmmTrigInfo;           ///< Information to generate SMM
} SMM_REQ_BUFFER;

/// MBOX buffer for SMM info
typedef struct {
  MBOX_BUFFER_HEADER  Header;              ///< Header
  SMM_REQ_BUFFER      Req;                 ///< Reques buffer
} MBOX_SMM_BUFFER;


///
/// structure of ReqBuffer for MboxBiosCmdSxInfo mailbox command
///
typedef struct {
  UINT8  SleepType;                        ///< Inform which sleep state the
system is going to
} SX_REQ_BUFFER;

/// MBOX buffer for Sx info
typedef struct {
  MBOX_BUFFER_HEADER  Header;              ///< Header
  SX_REQ_BUFFER       Req;                 ///< Request buffer
} MBOX_SX_BUFFER;


///
/// structure of ReqBuffer for MboxBiosCmdRsmInfo mailbox command
///
typedef struct {
  UINT64  ResumeVecorAddress;              ///< Address of BIOS resume vector
  UINT64  ResumeVecorLength;               ///< Length of BIOS resume vector
} RSM_REQ_BUFFER;

/// MBOX buffer for RSM info
typedef struct {
  MBOX_BUFFER_HEADER  Header;              ///< Header
  RSM_REQ_BUFFER      Req;                 ///< Req
} MBOX_RSM_BUFFER;

/// CAPS_REQ_BUFFER structure
```

```
typedef struct {
  UINT32 Capabilities;                    ///< PSP Writes capabilities into
this field when it returns.
} CAPS_REQ_BUFFER;

// Bitmap defining capabilities
#define PSP_CAP_TPM (1 << 0)

/// MBOX buffer for Capabilities Query
typedef struct {
  MBOX_BUFFER_HEADER  Header;             ///< Header
  CAPS_REQ_BUFFER     Req;                ///< Req
} MBOX_CAPS_BUFFER;

/// MBOX buffer for Exit BIOS info
typedef struct {
  MBOX_BUFFER_HEADER  Header;             ///< Header
} MBOX_DEFAULT_BUFFER;

//
// Define Malbox buffer comming from PSP->BIOS
//

///
/// structure of ReqBuffer for MboxPspCmdSpiGetAddress/MboxPspCmdSpiGetAddress
mailbox command
///
typedef struct {
  UINT64    Attribute;                    ///< Inform attribute of SPI part
} SPI_ATTRIB_REQ;

/// MBOX buffer for Spi Get/Set attribute info
typedef struct {
  MBOX_BUFFER_HEADER      Header;         ///< Header
  SPI_ATTRIB_REQ          Req;            ///< Req
} MBOX_SPI_ATTRIB_BUFFER;


///
/// structure of ReqBuffer for MboxPspCmdSpiGetBlockSize mailbox command
///
typedef struct {
  UINT64  Lba;                            ///< starting LBA
  UINT64  BlockSize;                      ///< Block size of each Lba
  UINT64  NumberOfBlocks;                 ///< Total number of blocks
} SPI_INFO_REQ;
```

```
/// MBOX buffer for Spi read block attribute
typedef struct {
  MBOX_BUFFER_HEADER    Header;             ///< Header
  SPI_INFO_REQ          Req;                ///< Req
} MBOX_SPI_INFO_BUFFER;


///
/// structure of ReqBuffer for MboxPspCmdSpiRead/Write mailbox command
///
typedef struct {
  UINT64  Lba;                              ///< starting LBA
  UINT64  Offset;                           ///< Offset in LBA
  UINT64  NumByte;                          ///< Total byte to read
  UINT8   Buffer[1];                        ///< Buffer to read the data
} SPI_RW_REQ;

/// MBOX buffer for Spi read block attribute
typedef struct {
  MBOX_BUFFER_HEADER    Header;             ///< Header
  SPI_RW_REQ            Req;                ///< Req
} MBOX_SPI_RW_BUFFER;


///
/// structure of ReqBuffer for MboxPspCmdSpiErase mailbox command
///
typedef struct {
  UINT64  Lba;                              ///< starting LBA
  UINT64  NumberOfBlocks;                   ///< Total number of blocks
} SPI_ERASE_REQ;

/// MBOX buffer for Spi read block attribute
typedef struct {
  MBOX_BUFFER_HEADER    Header;             ///< Header
  SPI_ERASE_REQ         Req;                ///< Req
} MBOX_SPI_ERASE_BUFFER;

/// Union of various structure
typedef union _MBOX_BUFFER {
  MBOX_DEFAULT_BUFFER       Dflt;           ///< Default

  MBOX_S3DATA_BUFFER        S3DataInfo;     ///< S3DataInfo
  MBOX_SMM_BUFFER           Smm;            ///< Smm
  MBOX_SX_BUFFER            Sx;             ///< Sx
  MBOX_RSM_BUFFER           Rsm;            ///< Rsm
```

```
  MBOX_CAPS_BUFFER          Cap;            ///< Cap

  MBOX_SPI_ATTRIB_BUFFER    SpiAttrib;      ///< SpiAttrib
  MBOX_SPI_INFO_BUFFER      SpiInfo;        ///< SpiInfo
  MBOX_SPI_RW_BUFFER        SpiRw;          ///< SpiRw
  MBOX_SPI_ERASE_BUFFER     SpiErase;       ///< SpiErase
} MBOX_BUFFER;
```

Appendix C PSP S5 Boot Flow

When SoC is powered-up and reset is de-asserted SMU On-chip Boot ROM code starts executing and early fuse loading takes place. At this point both PSP microcontroller (A5) and Host CPU (x86) are both in reset. Subsequently host x86 BSP core and PSP (A5) resets are both de-asserted.

Figure 8 on page 68 shows flow of operations that takes place starting from this stage during the S5 boot.

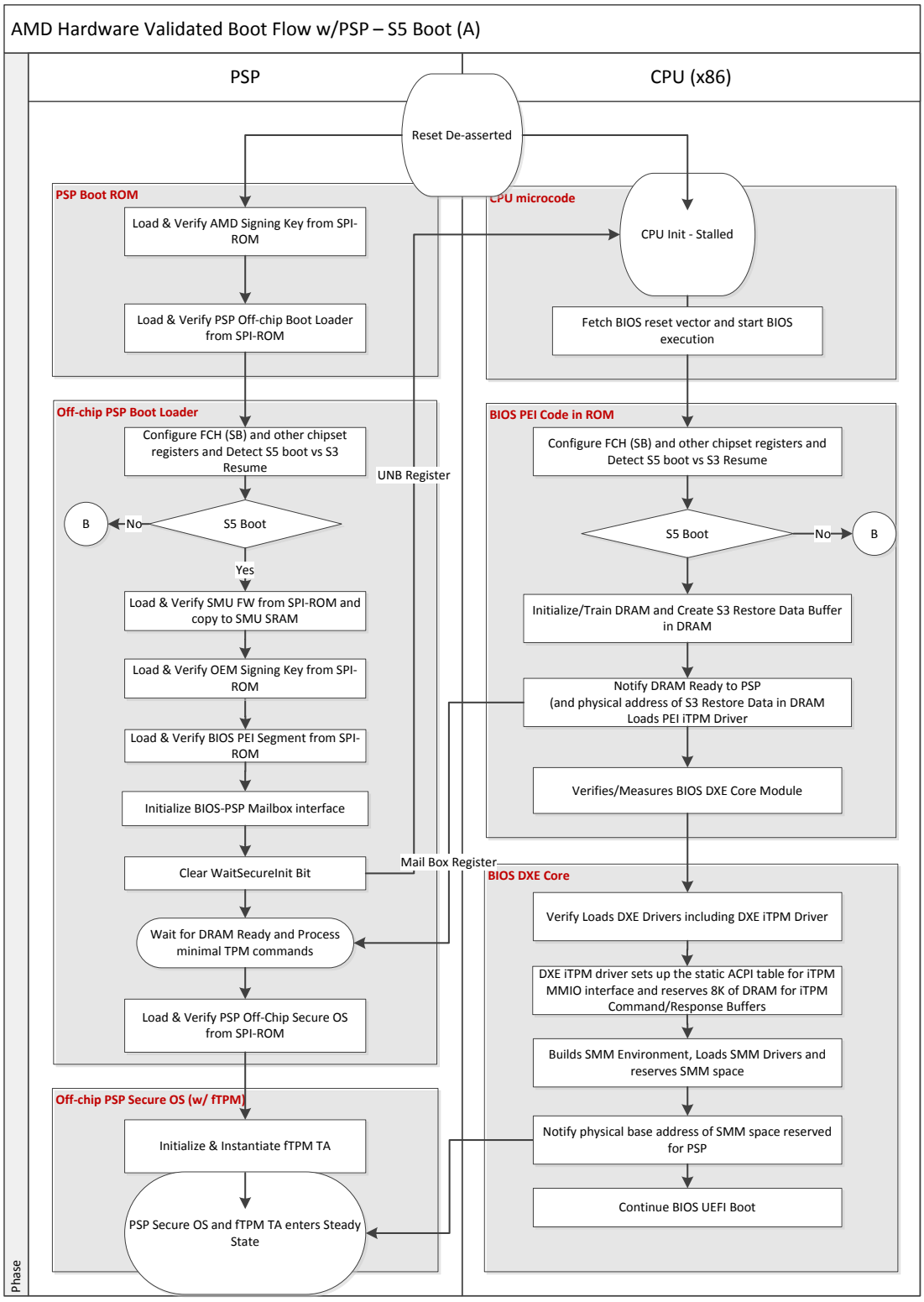

Figure 8. Hardware Validated Boot Flow – S5 Boot

Appendix D PSP S3/Resume

Figure 9 shows the S3 suspend flow:

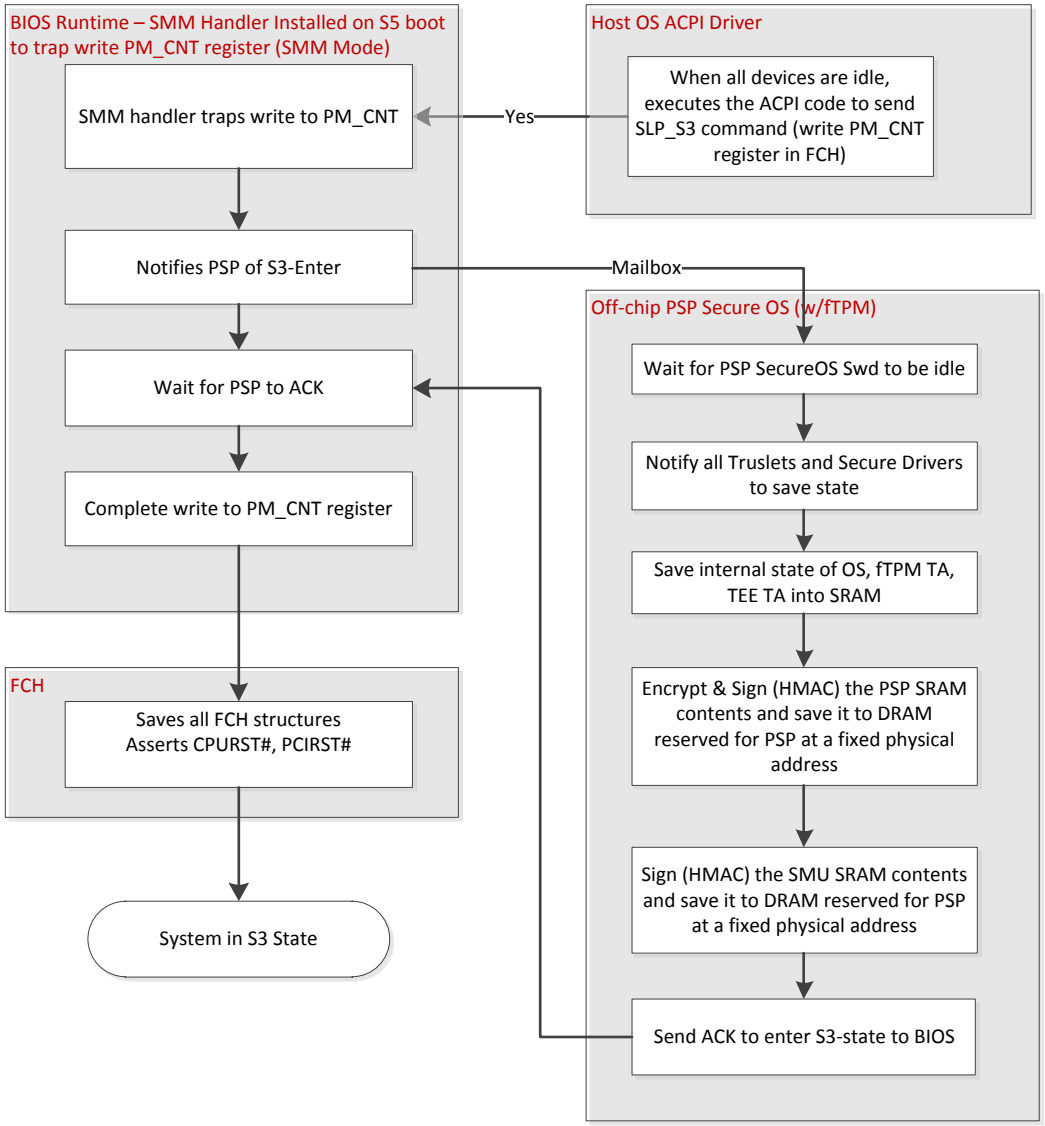

Figure 9. Hardware Validated Boot Flow – S3 Suspend

D.1 PSP S3 Resume Flow

Figure 10 shows the S3 resume flow:

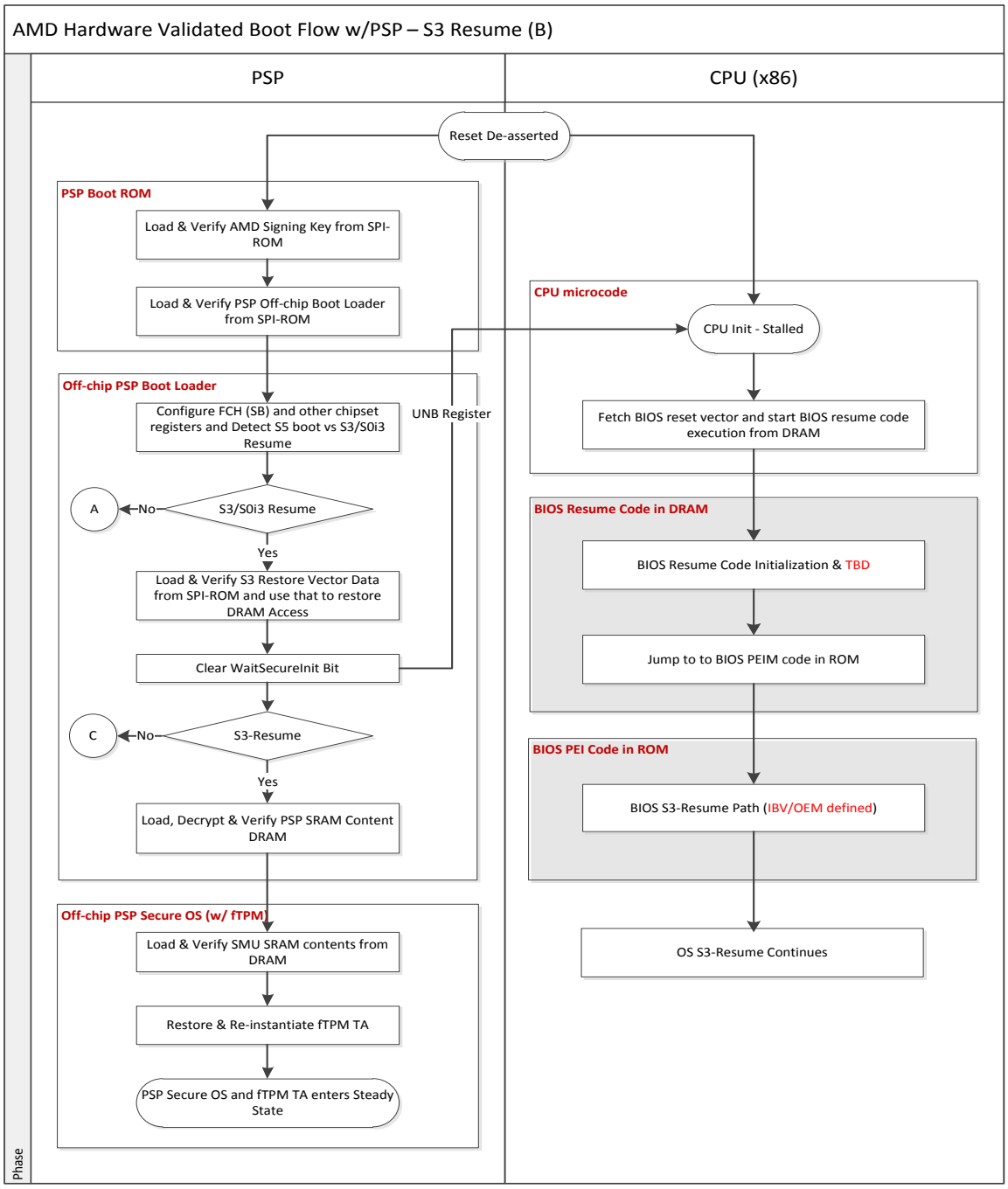

Figure 10. Hardware Validated Boot Flow – S3 Resume

Appendix E Key format

AMD will provide the following keys to OEM/IBV.

E.1 Public Part of the AMD Signing RSA-2048 bit Key

File: Agesa\Firmwares\ML\AmdPubKey.bin

Figure 11 describes the layout of the Root AMD Signing RSA-2048 bit Public Key (Root) stored in SPI-ROM.

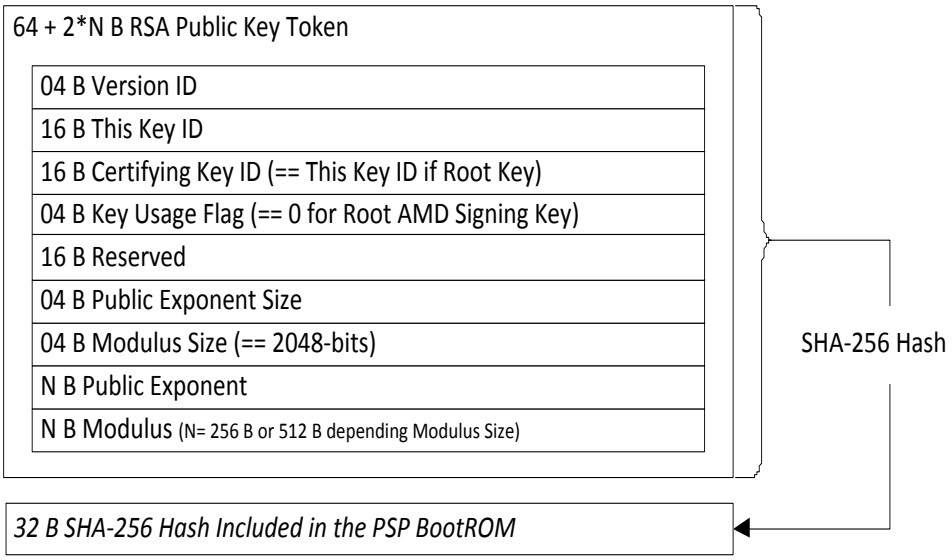

Figure 11. Root RSA Public Key Token Format

E.2 Certified Public Part of the Leaf/Intermediate RSA-2048 or RSA 4096-bit Key

File: Agesa\Firmwares\ML\RtmPubSigned.key (***Note:*** *Only used by AMD customer reference board*)

Figure 12 describes the layout of a leaf/intermediate RSA Public Key with sizes up to 4096-bit stored in SPI-ROM. The public part of the OEM BIOS signing RSA key will be stored in this format.

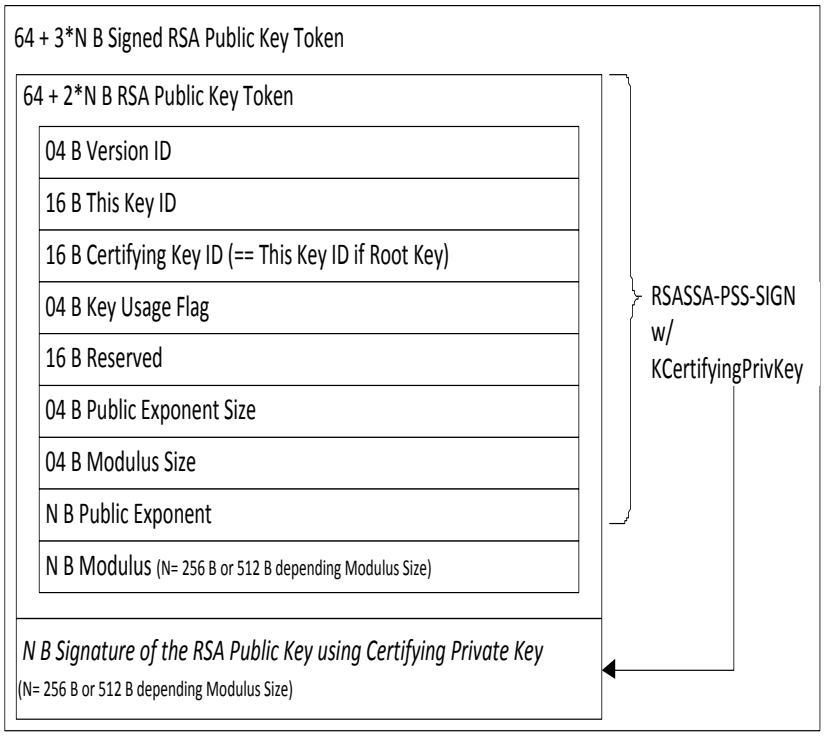

Figure 12. RSA Public Key Token Format

RSASSA-PSS signing scheme is used as signature scheme with SHA-256 used as the hashing algorithm for both message and mask generation.

Table 13 describes the fields of RSA Public Key Token structure:

Table 13. RSA Key Format Fields

Field Name	Offset (Hex)	Size (in Bytes)	Description/Purpose
Version ID	0x00	4	Version for key format structure
This Key ID	0x04	16	A universally unique key identifier
Certifying Key ID	0x14	16	A universally unique key identifier corresponds to the certifying key
Key Usage Flag	0x24	4	Signing Key Usage Flag 0 – Key authorized to sign AMD developed PSP Boot Loader and AMD developed PSP FW components and SMU FW. 1 – Key authorized to sign BIOS 2 – Key authorized to PSP FW (both AMD developed and OEM developed)
Reserved	0x24	16	Reserved – Set to zero
Public Exponent Size	0x38	4	Public Exponent Size (will be set to 256-bits)
Modulus Size	0x3C	4	Modulus size in bits
Public Exponent	0x40	N = 256 or 512 depending on public exponent size	Public Exponent For AMD Signing Public Key we may choose an exponent of size of 32-bytes and store that as sign-extended to 256-bit value and set the exponent size accordingly.
Modulus	0x80	N = 256 or 512 depending on modulus size	Modulus value zero extended to size of N*8 bits

Appendix F BuildPspDirectory Tool

F.1 PSP Directory Configure File Format

Line for declare BIOS related, start with BIOS_IMAGE

```
BIOS_IMAGE  INPUT_FILE = BIOS.fd,  PSP_FV_BASE=0x50000, PSP_FV_SIZE=0x130000, RTM_SIZE = 0xCB000
```

INPUT_FILE: Specify the input BIOS file's name

PSP_FV_BASE: FV base reserved for hold PSP images

PSP_FV_SIZE: FV size reserved for hold PSP images

RTM_SIZE: RTM (Root Trust Module) file's size

Line for declare PSP directory entry, start with PSP_ENTRY

```
PSP_ENTRY TYPE=0, FILE=AmdPubKey.bin
```
or

```
PSP_ENTRY TYPE=0xb, VALUE=0x1
```

TYPE: PSP directory entry type (Table 4 – PSP Directory Entry Type Encodings)

FILE: The PSP image file

VALUE: Value to be filled for some specific entry

F.2 Command Line Parameters

BuildPspDirectory tool support 3 functions: Build Directory table, Build PSP images, Dump Psp directory which specify as "bd", "bb", 'dp' in the command line. It only allowed executing one function at once; different function has its own defined parameters. Type "-h" following by can get the detail help for each specific function.

F.2.1 Build Directory Table (bd)

This command is used to build PSP Directory header which following the definition in 5.1.1 . Two positional parameters are required: 1) configuration file (Format defined in 12.6.1); 2) Output Psp Directory binary name.

"BuildPspDirectory.exe bd pspdirectory.cfg pspdir.bin" will build psp directory table binary named pspdir.bin, while the PSP directory entry information is get from the input configure file (pspdirectory.cfg).

Also to reduce the latency of detect specific PSP entries, and following the restriction of PSP entry's order, this function will optimize the order automatically regardless the order you written in the Psp Directory configure file as below:

Type 0 -> Type 1 -> Type 8 -> Type 3 -> Type 5 -> Type 6 -> Type 7 -> Type 2 -> Type 4 -> Others

F.2.2 Build PSP BIOS Image (bb)

This command is used to embed the images specified by input configuration file to the reserved PSP FV region. Two positional parameters are required: 1) configuration file (Format defined in 12.6.1); 2) Output BIOS image name after embedding.

"BuildPspDirectory.exe bb pspdirectory.cfg bios.fd" will embed images which specified in the configure file (pspdirectory.cfg) to the output BIOS image file bios.fd

F.2.3 Dump PSP Directory Information (dp)

This command is used to dump PSP directory information of a given BIOS image. Only one positional parameter is required: The input BIOS image, while below optional parameters can support dump the bios image in different manners: 1) –x Output the information in XML format; 2) –b output PSP binaries embedded; 3) –d output PSP configure file which file name is align with output of command '-b'

"BuildPspDirectory.exe dp bios.fd –b -d" will output the PSP binaries embedded in the PSP FV also the 'Pspdirectory.cfg' associated. The function can be used with 'bd' and 'bb' to achieve replaces PSP binary at binary level.

"BuildPspDirectory.exe dp bios.fd –x" will output the PSP information in XML format.

Appendix G PSP FW FW_STATUS

Below are the status codes definition, which PSP FW write to both IO Port 80h (Low 8bit), and FW_STATUS register (Full 32btis)

BootLoader Error Codes and Progress Codes in FW_STATUS is with prefix 0x100.

Table 14. PSP BootLoader Error Codes

Error Codes in FW_STATUS	Value	Description
BL_ERROR_INVALID_BOOTMODE	0x01	consult with AMD FCH/PSP team
BL_ERROR_INVALID_APERCONFIG	0x02	consult with AMD PSP FW team
BL_ERROR_SMUFW	0x03	Preclusion check if SMU FW content/signature is corrupted or correctly placed in BIOS
BL_ERROR_OEMSIGNING	0x04	Preclusion check if OEM signing key content/signature is corrupted or correctly placed in BIOS
BL_ERROR_BIOS_PEI	0x05	Preclusion check if BIOS PEI volume or PSP directory table content/signature is corrupted or correctly placed in BIOS
BL_ERROR_SECUREOS	0x06	Preclusion check if PSP SecureOS content/signature is corrupted or correctly placed in BIOS
BL_ERROR_LOAD_SMUFW	0x07	Can not locate SMU FW in PSP directory table, preclusion check the PSP directory table.
BL_ERROR_LOAD_OEMSIGNING	0x08	Can not locate OEM SINGING KEY in PSP directory table, preclusion check the PSP directory table.
BL_ERROR_LOAD_BIOS_PEI	0x09	Can not locate BIOS PEI in PSP directory table, preclusion check the PSP directory table.
BL_ERROR_LOAD_TRUSLETKEY	0x0a	Can not locate Truslet key in PSP directory table, preclusion check the PSP directory table.
BL_ERROR_LOAD_SECUREOS	0x0b	Can not locate SECURE OS in PSP directory table, preclusion check the PSP directory table.

Table 14. PSP BootLoader Error Codes (Continued)

Error Codes in FW_STATUS	Value	Description
BL_ERROR_INVALID_PSP_DIRENTRY	0x0c	invalid PSP directory entry, preclusion check the PSP directory table.
BL_ERROR_RELEASE_BSPCORE_FAIL	0x0d	Fail to release the BSP Core/x86 - consult with AMD PSP FW team
BL_ERROR_RETURNED_FROM_OS	0x0e	Control is not returned from secure OS, abnormal status - consult with AMD PSP FW team
BL_ERROR_LOAD_RESTOREVEC_FAIL	0x0f	Preclusion check if SPI-ROM S3 datablob content/signature is corrupted in SPI-ROM
BL_ERROR_RESTORE_SECUREBOOT_REG	0x10	Only available for A0, consult with AMD FCH/PSP team
BL_ERROR_S0I3_STEPS_FAIL	0x11	consult with AMD FCH/PSP team

Table 15. PSP BootLoader Progress Codes

Progress Codes in FW_STATUS	Value	Description
BL_SUCCESS_C_MAIN	0x40	Entered Boot Loader
BL_SUCCESS_CONFIG_FCHSB	0x41	configured FCH/SB
BL_SUCCESS_BOOTMODE_S4S5	0x42	entered S4/S5 bootmode
BL_SUCCESS_COLD_CRYPTO_POINTER	0x43	passed function pointers from Boot Rom
BL_SUCCESS_COLD_LOAD_SMUFW	0x44	loaded SMU FW
BL_SUCCESS_COLD_VERIFY_SMUFW	0x45	verified SMU FW
BL_SUCCESS_COLD_LOAD_OEM_KEY	0x46	loaded OEM Signing Key
BL_SUCCESS_COLD_VERIFY_OEM_KEY	0x47	verified OEM Signing Key
BL_SUCCESS_COLD_LOAD_BIOS_PEI	0x48	loaded BIOS PEI segment
BL_SUCCESS_COLD_VERIFY_BIOS_PSPDIR	0x49	verified BIOS PEI segment and PSP directory table
BL_SUCCESS_COLD_RELEASE_BSPCORE	0x4a	released BSPCORE
BL_SUCCESS_COLD_LOAD_TRUSTLETKEY	0x4b	loaded Truslet Key
BL_SUCCESS_COLD_VERIFY_TRUSTLETKEY	0x4c	verified Truslet Key and hash the Truslet Key
BL_SUCCESS_COLD_LOAD_SECURE_OS	0x4d	loaded secure OS
BL_SUCCESS_COLD_VERIFY_SECURE_OS	0x4e	verified secure OS

Table 15. PSP BootLoader Progress Codes (Continued)

Progress Codes in FW_STATUS	Value	Description
BL_SUCCESS_COLD_MB_BL_SECURE_OS	0x4f	set up Bootloader-to-SecureOS mailbox
BL_SUCCESS_COLD_TRANSFER_SECURE_OS	0x50	Bootloader transfer control to secure OS
BL_SUCCESS_BOOTMODE_S3S0I3	0x51	entered S3/S0i3 bootmode - Warm Boot
BL_SUCCESS_WARM_CRYPTO_POINTER	0x52	passed function pointers - Warm Boot
BL_SUCCESS_WARM_LOAD_RESTORE	0x53	loaded DRAM restore data - Warm Boot
BL_SUCCESS_WARM_VERIFY_RESTORE	0x54	verified DRAM restore data - Warm Boot
BL_SUCCESS_WARM_AGESA_RESTORE	0x55	restored DRAM access - Warm Boot
BL_SUCCESS_WARM_SETS3EXIT_BIT	0x56	set the S3 Exit Bit
BL_SUCCESS_WARM_RELEASE_BSPCORE	0x57	released BSPCORE - Warm Boot
BL_SUCCESS_WARM_S0I3_STEP_DONE	0x58	executed S0i3 steps on resume - Warm Boot
BL_SUCCESS_WARM_LOAD_SECURE_OS	0x59	unused
BL_SUCCESS_WARM_VERIFY_SECURE_OS	0x5a	verified secure OS - Warm Boot
BL_SUCCESS_WARM_MB_BL_SECURE_OS	0x5b	set up Bootloader-to-SecureOS mailbox - Warm Boot
BL_SUCCESS_WARM_MB_SRAMHMAC_PASS	0x5c	successfully verified the signature of SRAM - Warm Boot
BL_SUCCESS_WARM_MB_TRANSFER2OS	0x5d	successfully transferred control to Secure OS - Warm Boot
BL_ERROR_WARM_MB_SRAMHMAC_FAIL	0x5e	failed validation of SRAM signature - Warm Boot

After the PSP Boot Loader passes control to Secure OS FW, the format of the FW_STATUS register is following:

```
xx_xx_xxxx
|  |  |
|  |  |_ error code
|  |
|  |_ Secure OS progress code
|
|_ 0x80|<last executed BIOS command>
```

Example: 0x84070000 means that Secure OS successfully finished initialization sequence and the last BIOS command the OS received was 0x4 (MBOX_C2P_RSM_INFO) from BIOS.

Table 16. Progress Codes during Secure OS Initialization

Secure OS Progress Value	Description
0x01	Entered Secure OS
0x02	Successfully initialized access to CC6 buffers (if stuck at previous code, check CC6 configuration)
0x03	Next step in OS init process (contact PSP FW team)
0x04	Initialized internal data structures and mailbox buffers
0x05	Initialized Swd part of the OS and started notification handler thread
0x06	Loaded and initialized System Trustlets (if stuck at previous code, check Trustlet binary in SPI ROM)
0x07	Finished Secure OS initialization, entered steady state

Progress codes during S3 cycle. There could be other progress codes, for troubleshooting contact PSP FW team

Table 17. Progress Codes during S3 Cycle

Secure OS Progress Value	Description
0x20	Secure OS Entered S3 suspend
0x21	Secure OS entered idle state before suspend
0x22	Secure OS resumed to the Nwd part
0x23	Secure OS finished S3 resume