SOFTWARE SPECIFICATION

NVIDIA Mariko SoC Boot ROM

Revision History

Version	Date	Description
1.0	NOV 09, 2016	Initial Release

Table of Contents

1.0 INTRODUCTION

This document describes the enhancements, changes, and security features Boot ROM uses in the Mariko SoC. This guide should be used in conjunction with the Mariko Technical Reference Manual for details of referenced fuses, straps, and registers.

1.1 References

Refer to the documents or models listed in Table 1 for more information. Use the latest revision of all documents at all times.

Table 1: List of Related Documents

Document
Mariko Technical Reference Manual
Tegra X1 Technical Reference Manual

1.2 Abbreviations and Definitions

Table 2 lists abbreviations that may be used throughout this document and their definitions.

Table 2: Abbreviations and Definitions

Abbreviation	Definition
BL	Boot Loader. Note, the use of the term "Boot Loader" loosely applies to any piece of SW that the BR hands off control to. This could be the NV proprietary BL (Android / LDK), uBoot, UEFI, microboot, or nvtboot. The use of BL1 in this document also refers to the first BL loaded by the BR.
BEK	Boot Encryption Key
BSS	Uninitialized Data segment. This segment stores variables that do not have explicit initialization in source code (global and static variables). See https://en.wikipedia.org/wiki/Data_segment.
IROM	Internal ROM of the chip that stores BR code as well as secured assets such as crypto keys.
MGF	Mask Generation Function of the EMSA-PSS-ENCODE step of RSASSA-PSS signature generation.
PKC	Public Key Cryptography
RCM	USB Recovery Mode
SBK	Secure Boot Key
Unique ID / UID, ECID	Every NVIDIA chip has a unique value stored in fuses. This UID is 100-bits in length. BR stores the 100-bits UID in a structure of four 32-bit words. See References section for link to NVIDIA ECID format.
TZRAM	TZRAM is on-die SRAM used by TZ OS to store sensitive code and data.
TZDRAM	A region in DRAM where accesses are TZ-secured.
TZSRAM	Same as TZRAM above, but used to disambiguate TZDRAM and TZSRAM.
R	Read enabled
NR	Read protection enabled
W	Write enabled
NW	Write protection enabled

1.3 Chip States

- Preproduction - The chip state in which FUSE_PRODUCTION_MODE = 0, and FUSE_SECURITY_MODE = 0.
- NV Production mode - The chip state in which FUSE_PRODUCTION_MODE = 1, and FUSE_SECURITY_MODE = 0. This is the fused chip state in which customers receive their chips.
- FA - Failure Analysis Mode, when FUSE_FA_0 is burned.

2.0 DELTA CHANGES TEGRA X1 -> MARIKO

This section highlights the new Boot ROM changes made from Tegra X1 to Mariko.

2.1 Updated Fuses in Mariko

In Mariko, the following fuses were updated to add to the Boot ROM security:

- FUSE_BOOT_SECURITY_INFO
- FUSE_SW_RESERVED[2]

Refer to the Fuses chapter of the Mariko Technical Reference Manual for details.

Table 3: Summary of Authentication and Encryption options

FUSE_BOOT_SECURITY_INFO[0]* (Authentication Scheme)	FUSE_BOOT_SECURITY_INFO[2] (BL encryption using BEK)	Secure Boot Path
0	0	AES-CMAC authentication using SBK, no encryption (BCT, BL, SC7, RCM)
0	1	AES-CMAC authentication using SBK, BL encryption using BEK BCT encrypted section is encrypted using BEK SC7 firmware encrypted using BEK RCM messages encrypted using BEK
1	0	RSA-PSS, no encryption
1	1	RSA-PSS, BL encryption using BEK BCT encrypted section is encrypted using BEK SC7 firmware encrypted using BEK RCM messages encrypted using BEK

*BOOT_SECURITY_INFO[1] reserved and unused for Mariko.

2.2 IROM and FUSE Encryption Keys

FUSE/IROM encryption keys (FEK) is added to the MISC block for Mariko. These keys are accessible only by Boot ROM and are used to decrypt assets stored in IROM and fuses. For details, refer to the Security Engine (SE) chapter of the Mariko Technical Reference Manual.

2.2.1 Fuse Requirements

1. Add FUSE_BOOT_SECURITY_INFO fuse register (protected by SECURITY_MODE).
   a. 1 bit for enabling OEM fuse encryption (described in subsequent section).
   b. 3 bits for FEK key selection.
      See the key definitions in the Boot ROM Logical Layout of FEKs and the FUSE_BOOT_SECURITY_INFO bit field in the Fuses chapter of the Mariko Technical Reference Manual.
2. Add FEK bank select fuse register (protected by PRODUCTION_MODE). FUSE_RESERVED_PRODUCTION[2] is reserved as the FEK bank select.

   There are a total of 16 FEKs subdivided into two banks. Two FEKs are reserved for NVIDIA and test usage for each bank. Each bank also has 6 FEKs, all of which are assignable to OEMs. OEM FEK selection is done through FUSE_SECURITY_INFO[FEK Select].

a. FUSE_RESERVED_PRODUCTION[2] = 0 is Bank0. 1 is Bank1.

2.2.2 Boot ROM Logical Layout of FEKs

FEK key Number	Key name	Purpose	Bank
0	Test key	Development/test key	Bank 0
1	NVIDIA key	Encrypts the FSKP keys; NVIDIA owned	Bank 0
2	FEK 0	Encrypts OEM fuse assets	Bank 0
3	FEK 1	Encrypts OEM fuse assets	Bank 0
4	FEK 2	Encrypts OEM fuse assets	Bank 0
5	FEK 3	Encrypts OEM fuse assets	Bank 0
6	FEK 4	Encrypts OEM fuse assets	Bank 0
7	FEK 5	Encrypts OEM fuse assets	Bank 0
8	Test key	Development/test key	Bank 1
9	NVIDIA key	Encrypts the FSKP keys; NVIDIA owned	Bank 1
10	FEK 0	Encrypts OEM fuse assets	Bank 1
11	FEK 1	Encrypts OEM fuse assets	Bank 1
12	FEK 2	Encrypts OEM fuse assets	Bank 1
13	FEK 3	Encrypts OEM fuse assets	Bank 1
14	FEK 4	Encrypts OEM fuse assets	Bank 1
15	FEK 5	Encrypts OEM fuse assets	Bank 1

2.3 Secure Assets in IROM

The 4 KB of IROM space allocated for secure assets are encrypted at rest for Mariko. There are two types of keys stored in the Mariko IROM:

1. Factory secure provisioning keys (64 x 256-bit AES keys). Encrypted by the NVIDIA FEK. Note, the FEK bank select must always be heeded, such that the correct NV FEK is used.
2. Default SE keys, new for Mariko (12 x 128-bit AES keys). Encrypted by NV FEK.

2.3.1 Default SE keys

For Mariko, all SE key slots unused by the BR have AES keys pre-provisioned into them at cold boot. No specific use case is identified for this feature, but this gives OEMs flexibility should they require the use of some pre-provisioned keys.

12 x 128-bits of randomly generated data are added to the encrypted IROM key blob, which serve as the initialization vectors to the eventual derived key that is loaded into the twelve unused SE key slots.

Note: SE and SE2 are identically provisioned with the same keys at cold boot.

2.3.1.1 Boot ROM Default SE Key Derivation Procedure

1. The BR reads the FEK bank select fuse to assess if Bank0 or Bank1 is to be used. Depending on the bank selected, the BR uses a different AES key in the MISC registers to decrypt the assets in the IROM blob. If Bank0 is selected, the BR uses FEK key 1. If Bank1 is selected, the BR uses FEK key 9.
2. The BR loads the appropriate key from the MISC registers (as explained in 1. above) into the SE key slot 0.
3. BR copies the 12 x 128-bit=192byte blob from IROM to IRAM.
4. BR uses the SE to decrypt each 128-bit section of the 192-byte key blob into SE key slots 1 through 11 (skipping slot 0 until the last step).

5. BR decrypts the first 128-bits of the 192-byte key blob into SE key slot 0, which also overwrites the FEK in the process.

> **Note:** The default SE keys should be derived and loaded into the SE key slot before access to the FEK is locked down.

2.4 OEM Fuse Encryption

Mariko adds fuse encryption, to protect against microscope and/or decap attacks of certain high value OEM assets. The OEM can elect to encrypt their KEK, BEK, and SBK keys with an NVIDIA assigned Fuse Encryption Key (FEK). The FEK assigned can be one of the FEKs from Section 2.2.2 Boot ROM Logical Layout of FEKs. The OEM must also burn the OEM fuse encryption enable bit and FEK selection bits as defined in the Fuses chapter of the Mariko Technical Reference Manual. The OEM burned fuses not mentioned here are used by the BR as plaintext.

Fuse/Key	Burned by	Fuse encryption	Notes
FUSE_KEK00	OEM	ODM/BR	
FUSE_KEK01	OEM	ODM/BR	
FUSE_KEK02	OEM	ODM/BR	
FUSE_KEK03	OEM	ODM/BR	
FUSE_BEK0	OEM	ODM/BR	
FUSE_BEK1	OEM	ODM/BR	
FUSE_BEK2	OEM	ODM/BR	
FUSE_BEK3	OEM	ODM/BR	
FUSE_PRIVATE_KEY0	OEM	ODM/BR	private_key0-3 is the SBK.
FUSE_PRIVATE_KEY1	OEM	ODM/BR	
FUSE_PRIVATE_KEY2	OEM	ODM/BR	
FUSE_PRIVATE_KEY3	OEM	ODM/BR	
OEM/BR = Owned by OEM. Handled by Boot ROM (i.e., decrypted, loaded into appropriate place)			
Fuse reference: Refer to the Fuses chapter of the Mariko Technical Reference Manual.			

2.5 Encrypted Boot Loader

Mariko adds native first stage boot loader encryption support to the BR. One bit in FUSE_BOOT_SECURITY_INFO is reserved to enable this feature, burnable by the OEM.

A new 128-bit AES key is added to the secure boot flow to support the encrypted boot loader feature, named the Boot Encryption Key (BEK). This key is stored in OEM burnable fuses, is locked by FUSE_SECURITY_MODE, and can be encrypted by a Fuse Encryption Key (FEK) assigned by the OEM to the customer. The BEK is left in the SE key slot after BR exits, and the SE key slot is read protected.

The BEK encrypts the first stage BL, the BCT, as well as RCM messages, and LP0 recovery firmware.

The SBK carries over from Tegra X1 to Mariko. It is used as the key if AES-CMAC is used as the authentication scheme.

The BEK can be optionally encrypted by the selected OEM FEK.

2.6 Key Encryption Key (KEK)

A new per device encryption key called the Key Encryption Key (KEK) is added for OEM usage. The KEK is a 128-bit AES key, and is specified by the OEM and burned into FUSE_KEK0 to FUSE_KEK3. The BR loads the KEK into an SE key slot that is read locked. The KEK can be optionally encrypted by the selected OEM FEK.

2.7 TZRAM (TZSRAM) Size Increase from 64 KB to 256 KB

BR clears the whole TZSRAM to zero at cold boot. It does NOT clear TZSRAM at LP0 exit, traditionally to minimize resume time. Please refer to Section 1.2 Abbreviations and Definitions for the disambiguation of TZRAM/TZSRAM/TZDRAM.

2.8 SE Context Restoration

The SE context location has been moved from a non-secure scratch register to secure scratch register. Refer to Section 6.2.4.6 SE Context Restore for more details.

2.9 Separate Boot Loader Header from BCT

For Mariko, the BL header has been decoupled from the BCT. Therefore, it is no longer required to re-sign the BCT if the boot loader is updated, re-signed, and placed in the same location as before.

2.9.1 BCT Binding to BL

To preserve the ability to bind a BCT to a particular boot loader version the BR performs the following steps:

1. After authenticating the BCT, the BR tries to authenticate and load a BL based on the information in the `NvBootLoaderInfo` structure in the BCT. The NvBootLoaderInfo structure can store location information for up to 4 boot loaders.
2. The BR checks for Boot loaders from 0 to `Bct.BootLoadersUsed`.
3. If the `Bct.BootLoader[bootloader number].Version` field is non-zero, then BCT to BL binding is enabled for this copy of the boot loader. The `Bct.BootLoader[bootloader number].Version` field must match the `NvBootOemBootBinaryHeader.Version` field. If the Version numbers mismatch, the BR errors out and try to read the next copy of the Boot Loader.

2.10 AON Shadow TZRAM Handling by Boot ROM

2.10.1 At Cold Boot

The BR shall:

- Read the AON Shadow TZRAM power gating state from BCT. Bit 0 of `BootConfig2` field in the BCT is reserved for this feature.
- Program the PG controls for AON Shadow TZRAM:
  The value of `Bct.BootConfig2[0]]` is directly copied by the Boot ROM into `APBDEV_PMC_TZRAM_PWR_CNTRL_0_TZRAM_SD`. “0” in this bit means that the power gating feature is disabled. “1” means the power gating feature is enabled.
- If production mode is enabled, write lock the PG controls for AON Shadow TZRAM.

2.10.2 At SC7 Exit

Assuming that the PLLs into SE and TZRAM are running and stable and SE_SECURITY and TZRAM Security are in non-secure mode. The BR shall:

1. Read the AON Shadow TZRAM PG control at `APBDEV_PMC_TZRAM_PWR_CNTRL_0_TZRAM_SD`.
2. If `APBDEV_PMC_TZRAM_PWR_CNTRL_0_TZRAM_SD == 0` (i.e. not power gated) trigger the TZRAM restore sequence and wait for the restore to complete before loading the SC7 resumes firmware. The Boot ROM waits 1927 microseconds before timing out and returns an error, which in turn, triggers a full chip reset. The programming sequence to trigger the TZRAM restore from the AON domain is as follows:
   a. SE_TZRAM_OPERATION ← 0x3 (REQ = INITIATE, MODE= RESTORE).
   b. BR polls SE_TZRAM_OPERATION_0_BUSY field until it becomes IDLE (from BUSY).
   c. If 1927 microseconds has elapsed BR times out and returns an error, which in turn, triggers a full system reset.

Note: AON Shadow TZRAM restore must be triggered before SE context restore, as the SE context can be saved in TZRAM. It should also be triggered before SC7 firmware authentication and restore, since SC7 firmware can also be stored in TZRAM.

3.0 BOOT ROM SECURITY ENHANCEMENTS

This section aims to address the specific customer requests and requirements for the Mariko program.

3.1 Boot ROM Encryption

3.1.1 Requirement

1. Full BROM encryption only useful if execute-in-place; needs on-the-fly decryption support for memory controller
2. If encrypted BROM cannot be executed in place, prefer code in cleartext with critical data encrypted in BROM with FEK.

3.1.2 Response

- BROM code is in clear text
- Confidential data (FSKP, default SE keys) encrypted with FEK

3.1.3 Conclusion

NVIDIA shall support #2 as stated in the Requirement sub-section. IRAM Handling on Entry / Exit of Boot ROM

3.2 IRAM Clearing

3.2.1 Requirement

IRAM should be cleared when entering and exiting the Boot ROM.

3.2.2 Conclusion

It is not necessary to clear upon entry (due to latency) but it is required to clear on exit. NVIDIA shall list what is not cleared on exit. Code and Data from Non-Secure Regions

3.2.3 Requirement

- Code/data from non-secure regions must be copied once to IRAM
- Verification of code/data must fit and happen within IRAM
- Execution of the code must happen within IRAM and only with verified code

3.2.4 Response

- For cold boot, the destination of the first stage boot loader is flexible between SDRAM and IRAM, with the choice made by the customer. The destination, size, and entry point are specified in the boot loader header and are validated before any copy or execution takes place.
- For SC7 resume, the SC7 firmware location is fixed to IRAM location 0x4001_0000.

3.3 Boot ROM Removal of Unused Code

3.3.1 Requirement

Boot ROM should be as light as possible. Legacy code should be removed. AES-CMAC authentication scheme should be removed.

3.3.2 Response

NVIDIA agrees with the general principle of removing unneeded code and adheres to the following guidelines:

- If the code has purpose, then NVIDIA leaves it in.
- If there is no real use, then NVIDIA removes it.

3.4 Common Exit Path Security

3.4.1 Requirement

1. Exception vectors redirected after enabling Boot ROM secure access
2. Boot ROM should wait for security sync point before jumping to boot loader

3.4.2 Response

1. BR to remove redirection of exception vectors to IRAM. Exception vectors always points to BR during BR execution, and to BR after BR exit. Only TrustZone software can change the exception vector after BR exit.
2. Aligned on requirement #2 above. BR adds a read of SB_CSR_0 after programming it to make sure SECURE_BOOT_FLAG clearing has taken effect (flush register write pipeline in case of any timing issues; ensures secure PIROM region is locked).

3.4.3 Conclusion

1. BR action is the same as the response in #1.
2. BR action is the same as the response in #2.

3.5 Factory Secure Key Provisioning

3.5.1 Requirement

1. FSKP keys must be updated for Mariko.

3.5.2 Response

1. NVIDIA already updates FSKP keys for architecture of each new chip. No change in behavior.

4.0 BOOT ROM ARCHITECTURE

4.1 Core and Peripherals Segregation

Boot ROM architecture follows segregation of code into Core and Peripherals (IO).The Boot ROM has segregated the peripheral (IO) code from the core code to create separate drivers for each supported interface (eMMC and QSpi). This allows the drivers to be unit tested more easily and make them more modular to port to future projects.

Peripherals are invoked during the cold boot path of the BR flow. It is based on strap, fuses, and BCT data and in the recovery path to retrieve recovery firmware.

Peripheral boot interface is exposed via device manager API for initializing the hardware controller (clock/reset, pinmux/padring, setting up device controller context data structure in SYSRAM) for the core flow to enable and retrieve data (bct/bootloader/recovery image) from the media.

4.2 Dispatcher Architecture

This generation of Boot ROM introduces the concept of a task dispatcher which executes functions serially in deterministic order. If an error occurs in any function executed by a dispatcher, the BR branches to USB recovery mode (RCM).

The first dispatcher is the non-secure dispatcher. The term non-secure refers to the fact that this dispatcher, and the code which is executed, is stored in the non-secure section of the IROM; they do not get locked-down or hidden after BR execution.

The BR implements two dispatchers, one for tasks and code stored in the non-secure section of IROM, and one for tasks and code stored in the secure section of IROM.

The new dispatcher architecture has several advantages:

- Ease of maintainability of the flow of execution of the BR
- Ability to group a set of tasks together into its own task table (for example, SC7 tasks are grouped its own task table).
- Enhanced flow of execution logging ability
- Enhanced readability of the code and flow of execution

4.2.1 Dispatcher Implementation

The implementation of the task dispatcher is simple with little overhead. The dispatcher takes the task table the caller wishes to execute as the input. The dispatcher then executes each task in the task table. At first entry into the dispatcher, the task table ID is logged into the BIT. Furthermore, immediately before and after a task is called, the timestamp is logged for each individual task.

4.2.2 Example Task Table

The following code is the task table for cold boot. It lists the functions to be called in the event of a cold boot and a unique numerical ID is defined to identify the task.

```
// Tasks related to Coldboot flow
static const NvBootTask ColdBootTasks[] = {
    { &NvBootColdBootInit,        0x101 },
    { &NvBootColdBootReadBct,     0x102 },
    { &NvBootColdBootReInit,      0x103 },
    { &NvBootColdBootLoadBl,      0x104 },
    { &NvBootColdBootCopyBct,     0x105 },
};
```

4.3 Device Manager Architecture

4.3.1 Description

The BR device manager is an API that exposes commonly required functionality from boot devices, while keeping the underlying details of a boot device abstracted. The common set of device functions required is defined and grouped together as “device manager callback” functions in `nvboot_devmgr_int.h (NvBootDevMgrCallbacks)`.

When a new device is added to the supported list of secondary boot devices, a low level device driver implements all of the functions in `NvBootDevMgrCallbacks`. The device manager is then responsible to maintain a mapping of the device specific functions to the common device manager API functions.

Boot device initialization and re-initialization is done by the device manager. Upon detecting the boot device, the first initialization is done using known good values hard coded into the BR. After reading the BCT, the BR optionally reinitialize the device using parameters stored in the BCT. This can be done at the expense of some boot latency.

> **Note:** Note that the initialization time varies with the boot device.

4.3.2 Example Implementation

This section illustrates the relationship between the BR EMMC driver and the device manager. The common device functions that must be implemented by devices using the device manager are:

```
typedef struct NvBootDevMgrCallbacksRec
{
    NvBootDeviceGetParams      GetParams;
    NvBootDeviceValidateParams ValidateParams;
    NvBootDeviceGetBlockSizes  GetBlockSizes;
    NvBootDeviceInit           Init;
    NvBootDeviceRead           Read;
    NvBootDeviceQueryStatus    QueryStatus;
    NvBootDeviceShutdown       Shutdown;
    NvBootDeviceGetReaderBuffersBase GetReaderBuffersBase;
    NvBootDevicePinMuxInit     PadCtrlPinMux;
} NvBootDevMgrCallbacks;
```

The device manager will maintain a mapping of the above standardized functions in `nvboot_devmgr.c, in s_DeviceCallbacks[]`:

```
        {
            /* Callbacks for the SDMMC device */
            (NvBootDeviceGetParams)NvBootSdmmcGetParams,
            (NvBootDeviceValidateParams)NvBootSdmmcValidateParams,
            (NvBootDeviceGetBlockSizes)NvBootSdmmcGetBlockSizes,
            (NvBootDeviceInit)NvBootSdmmcInit,
            (NvBootDeviceRead)NvBootSdmmcReadPage,
            (NvBootDeviceQueryStatus)NvBootSdmmcQueryStatus,
            (NvBootDeviceShutdown)NvBootSdmmcShutdown,
            (NvBootDeviceGetReaderBuffersBase)NvBootSdmmcGetReaderBuffersBase,
          (NvBootDevicePinMuxInit)NvBootSdmmcPinMuxInit

    },
```

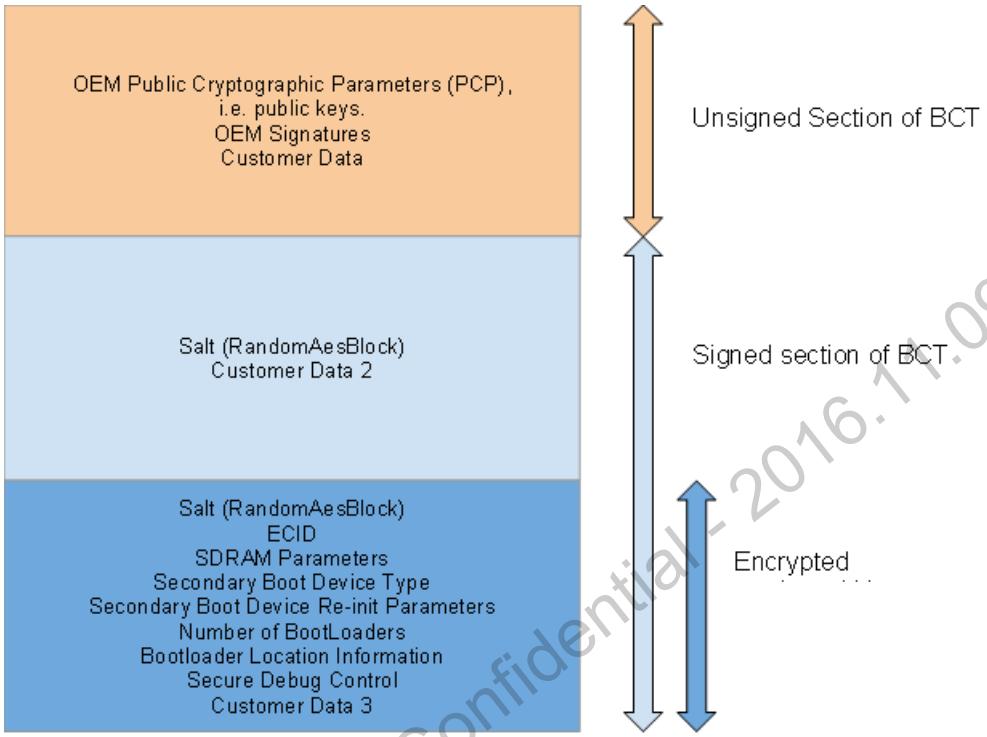
Upon cold boot, the cold boot driver detects the boot device selected via the fuses and the straps. Then it conveys this information to the device manager to select the proper grouping of device callbacks.

5.0 INPUTS TO BOOT ROM

In general, the inputs that BR relies on for state information and configuration are limited to straps, fuses, the BCT, and scratch registers.

5.1 Boot Configuration Table (BCT)

The BCT is stored on a secondary boot device, such as the eMMC. It stores configuration parameters used to initialize the secondary boot device, initialize and configure SDRAM, and locate the boot loader. It is signed by the OEM only.



5.1.1 Deltas from Tegra X1:

The following are a list of changes in BCT for Mariko:

- NvBootLoaderInfo has been updated. It now specifies the StartBlock and StartPage of the BL1 binary and header.
- A new struct named NvBootOemBootBinaryHeader has been created. This is a header appended to the BL1 binary, which shall store cryptographic signatures, size, destination and entry point of the BL1.
- Add some more reserved space in signed section of BCT that does not need to conform to reserved pattern.
- Remove 0x80...0x00 padding requirement in the reserved field of BCT.
- The signed section of the BCT is subdivided into two sections: one section which is unencrypted, and one which is. This allows for the placement of parameters that are not confidential and which can be used immediately after authentication (and without having to wait for decryption).
- The `SecureDebugControl` field is separated into two fields: one subject to ECID check, and one not subjected to ECID check.
- Add a 32-bit field for customer configurable features, named `BootConfig2`. Bit 0 of `BootConfig2` is reserved for AON TZRAM powergating control (0: Power gating disabled. 1: Power gating enabled).
- NvBootLoaderInfo simplified to remove parameters moved to NvBootOemBlInfo:
  `/**`

```
 * Stores information needed to locate and verify a boot loader.
 *
 * There is one \c NvBootLoaderInfo structure for each copy of a BL stored on
 * the device.
 */
typedef struct NvBootLoaderInfoRec
{
    /// Specifies a version number for the BL. The assignment of numbers is
    /// arbitrary; the numbers are only used to identify redundant copies
    /// (which have the same version number) and to distinguish between
    /// different versions of the BL (which have different numbers).
    NvU32      Version;

    /// Specifies the first physical block on the secondary boot device
    /// that contains the start of the BL. The first block can never be
    /// a known bad block.
    NvU32      StartBlock;

    /// Specifies the page within the first block that contains the start
    /// of the BL.
    NvU32      StartPage;

    /// Pad this struct to AES block size.
    NvU32      Unused;
} NvBootLoaderInfo;
```

5.2 Straps

Table 4: Boot ROM Straps

# Bits	Location	Function	Values	Notes
1	BOOT_FAST_UART (For internal reference only)	Support fast uart boot configuration in boot rom code for ATE test time savings (For internal reference only)	STRAPPING_OPT_A[9] (For internal reference only)	0 = Disable Fast UART (For internal reference only)
				1 = Enable Fast UART. Note: Input frequency REF need to be 12 MHz in order to support desired baud rate, 12 Mbps. (For internal reference only)
1	RCM_STRAPS	Forces USB Recovery Mode or RCM Debug mode	STRAPPING_OPT_A[12:10] + FUSE_ENABLE_2_BUTTON _RCM	
3	BOOT_SELECT[2:0]	Device Selection	STRAPPING_OPT_A[28:26]	-
			FUSE_PRODUCTION_MODE = X (don't care)	
			0x0 = eMMC_Config_0	x8 Boot ModeOff, DDR 51 MHz
			0x1 = SPI_Config_0	x1, 20.4 MHz, PLLP_OUT0
			0x2 = eMMC_Config_3	x8 Boot ModeOff, SDR 51 MHz
			0x4 = SPI_Config_1	x1, 19.2MHz, CLKM
			0x5 = Reserved	
			0x6 = Reserved	
			0x7 = Reserved	

# Bits	Location	Function	Values	Notes
			FUSE_PRODUCTION_MODE = 0	
			0x3 = UART Boot	UART boot via UART1
			FUSE_PRODUCTION_MODE = 1	
			0x3 = Use Fuses	Note: Default fuse config when no BOOT_DEVICE_INFO fuses are burned is eMMC configuration 0.
			Note: The above BOOT_SELECT options are only in effect in "regular boot" conditions i.e., cold boot. If "Forced RCM" or "Debug RCM" modes are detected, those modes take precedence over these boot device choices.	
1	NVPROD_UART	NvProduction UART boot mode	STRAPPING_OPT_A[13]	
			FUSE_PRODUCTION_MODE = 1; FUSE_SECURITY_MODE = 0	
			0x0 = Disabled	
			0x1 = Enabled	

5.3 Fuses

Mariko BR removes detailed user specifiable boot device settings, and BR supports a number of “enumerated configurations” that cover the most common use cases. A copy of the “IO configs” tab is pasted here for illustrative purposes:

SDMMC4 Configs	Description	In-use PLLs	Boot Strap
0x00	SDR, 51.0MHz, READ_MULTIPLE	PLLP_OUT0	Yes
0x01	SDR, 25.5MHz, READ_MULTIPLE	PLLP_OUT0	No
0x02	SDR, 25.5MHz, READ_SINGLE	PLLP_OUT0	No
0x03	DDR, 51.0MHz, READ_MULTIPLE	PLLP_OUT0	No
0x04	DDR, 25.5MHz, READ_MULTIPLE	PLLP_OUT0	No
QSPI0 Configs	**Description**	**In-use PLLs**	**Boot Strap**
0x00	PIO x1, 20.4MHz, NORMAL_READ	PLLP_OUT0	Yes
0x01	PIO x1, 19.2MHz, NORMAL_READ	CLK_M	No
0x02	PIO x4, 51MHz, QUAD_READ	PLLP_OUT0	No

5.4 SCRATCH Registers

5.4.1 Deltas from Tegra X1 to Mariko

Tegra X1 used PMC_SCRATCH1 to specify the SC7 resume firmware address. This is a non-secure scratch register. For Mariko, the scratch register used to store the SC7 resume firmware address shall be a secure scratch register: PMC_SECURE_SCRATCH119. Refer to the PMC chapter of the Mariko Technical Reference Manual.

5.4.2 Detecting the Chip State

The BR reads the following sources to detect the current chip state. Based on the values of these inputs, the BR decides what particular branch to take, as well as what actions may or may not be needed (such as, which authentication scheme, or debug policy to program).

FUSE_PRODUCTION_MODE	Specifies if the chip is in “NV production mode”. Chips sent to customers must always have this fused to 1.
BOOT_SECURITY_INFO	This fuse specifies the authentication scheme, whether encryption is enabled or not, and what FEK is selected.
FUSE_FA	Failure analysis mode fuse. If burned, the only boot path in BR is to UART.
BOOT_SELECT Strap of APB_MISC_PP_STRAPPING_OPT_A	Can be used to specify the boot device, as long as the FUSE_RESERVED_SW[3] is not burned.
PMC_SCRATCH0	If bit 0 is set, the BR will take the SC7 resume path.
FUSE_SECURE_PROVISION_INDEX	Only used in Secure Provisioning mode. Specifies the anti-cloning fuse.
FUSE_SECURE_PROVISION_INFO	Only used in Secure Provisioning mode. Specifies debug features.

6.0 MARIKO BOOT ROM FLOW

6.1 Expected Power on Reset State

It is very important for security reasons that the BPMP exception vectors at reset point to read-only memory.

In the Mariko implementation, the BPMP exception vectors (arevp.h) point to the beginning of IROM space, where the BR exception vector table resides:

Boot ROM exception vector table:

```
100000:     ea000019      b       10006c <reset>
100004:     ea000005      b       100020 <arm_data_abort>
100008:     ea000004      b       100020 <arm_data_abort>
10000c:     ea000003      b       100020 <arm_data_abort>
100010:     ea000002      b       100020 <arm_data_abort>
100014:     ea000001      b       100020 <arm_data_abort>
100018:     ea000000      b       100020 <arm_data_abort>
10001c:     eaffffff      b       100020 <arm_data_abort>
```

From arevp.h:

```
#define EVP_RESET_VECTOR_0_RESET_VECTOR_DEFAULT
_MK_MASK_CONST(0x100000)
#define EVP_UNDEF_VECTOR_0_UNDEF_VECTOR_DEFAULT
_MK_MASK_CONST(0x100004)
#define EVP_SWI_VECTOR_0_SWI_VECTOR_DEFAULT
_MK_MASK_CONST(0x100008)
#define EVP_PREFETCH_ABORT_VECTOR_0_PRE_ABORT_VECTOR_DEFAULT
_MK_MASK_CONST(0x10000c)
#define EVP_DATA_ABORT_VECTOR_0_DATA_ABORT_VECTOR_DEFAULT
_MK_MASK_CONST(0x100010)
#define EVP_RSVD_VECTOR_0_RSVD_VECTOR_DEFAULT
_MK_MASK_CONST(0x100014)
#define EVP_IRQ_VECTOR_0_IRQ_VECTOR_DEFAULT
_MK_MASK_CONST(0x100018)
#define EVP_FIQ_VECTOR_0_FIQ_VECTOR_DEFAULT
_MK_MASK_CONST(0x10001c)
```

All exceptions map to a single exception handler. The pseudo code for the exception handler is:

```
if(PRODUCTION_MODE_FUSE == 1) {
       if(FA_MODE_FUSE == 0) {
              Loop Forever
       }
} else {
       Loop for a long time (the time it takes the AVP at default POR clocks to loop
0xFFFF_FFFE times) for possible debugger connection, then reset.
}
```

6.1.1 First Instruction and Startup Code

The first instructions of the BR are contained in nvboot/startup/start.S. These instructions are written in assembly code. The startup code sets up the stack pointers, the protected IROM region start address, and the IROM patches that are present in the fuse block are applied at this time.

After the startup code, a series of “dispatchers” are moved, which executes functions one by one listed in table format (see Section 4.2 Dispatcher Architecture).

6.1.2 Startup Code Implementation Details

The general detail of the startup code is as follows:

1. Stack pointer setup for IRQ, FIQ, ABORT, UNDEFINED, SYSTEM/USER, and SUPERVISOR.
2. Zero out r3-r11.
3. Set protected IROM start address to SB_PIROM_START_0_PROTECTED_ROM_START_DEFAULT (0x1000), which partitions the IROM address space into non-secure and secure regions. The secure region of the IROM is read disabled at BR exit.
4. Apply any IROM patches at this point, so as to maximize the patchable area of the BR.
5. Jump to the first dispatcher, the non-secure dispatcher. Note that the UART boot path is branched to from the non-secure dispatcher, so the BR will never reach main().
6. Jump to secure IROM code entry (assembly code). Copy initialized data (.data segment), clear BSS (.bss segment). Jump to main().

6.1.3 Non-Secure Dispatcher

The function calls executed in the non-secure dispatcher is defined in nvboot/dispatcher/nvboot_tasks_ns.c:

```
static const NvBootTask VT_NONSECURE TasksNS[] = {
   //We do this cast because these routines do not return
   //This is OK for nonsecure dispatcher, but we should have made a diff type
   { (NvBootError(*)())&NvBootClocksEnablePmcClock, 0x01 },
   { (NvBootError(*)())&NvBootBpmpSetupOscClk, 0x2 },
   { (NvBootError(*)())&NvBootPadsExitDeepPowerDown, 0x3},
   // Note: NvBootBpmpNonsecureRomEnter may branch to UART boot here
   // if the UART boot conditions are satisfied.
   { (NvBootError(*)())&NvBootBpmpNonsecureRomEnter, 0x4 },
   // All code before this point must be in the non-secure
   // section (everything to get to UART boot for FA mode).
   // Move Pllp start and clock source switch of AVP to PLLP here,
   // which will speed up simulation time. This diverges from the T210
   // slightly, which had this function just before main() entry.
   { (NvBootError(*)())&NvBootBpmpEnablePllp, 0x05 },
   { (NvBootError(*)())&NvBootClocksSetAvpClockBeforeScatterLoad, 0x09 },
};
```

> It is possible to branch to the UART and Failure Analysis (FA) mode path in NvBootBpmpNonsecureRomEnter . If there is no branch to UART boot, the BR executes the rest of the non-secure dispatcher and then branches to the secure IROM entry code. The secure IROM entry code does some initialization (copy data segment, clear BSS to zero) before branching to "main()".

6.1.3.1 NvBootBpmpSetupOscClk

This function measures the oscillator frequency of the board. If the oscillator detection failed, the BR defaults to 38.4 MHz. If the oscillator detect was successful, the BR will program the oscillator frequency to `CLK_RST_CONTROLLER_OSC_CTRL_0`. The oscillator frequency is used by the clocking hardware to auto setup the parameters to PLLP and its dividers (see `arclk_rst.h`).

After programming the oscillator frequency in CAR, the microsecond timer is configured by programming `TIMERUS_USEC_CFG_0`. The values of the DIVIDEND and DIVISOR being programmed are defined in `s_UsecCfgTable` in nvboot_clocks.c.

6.2 Main Entry

After the execution of the non-secure dispatcher, the BR branches to the code in the secure section and to the main() entry point via `NvBootAsmSecureEnter`. `NvbootAsmSecureEnter` is the last portion of the startup code written in assembly, and copies initialized data (.data segment), clear BSS (.bss segment) and finally jump to main().

The main() entry is where the BR evaluates what state the chip is in, and what path to take next. After some common initialization in the security initialization dispatcher (SE init, SW crypto context init, key decryption), BR will branch to one of the major boot paths – Coldboot, SC7, RCM.

The main entry is in `nvboot/startup/boot0c.c`.

At main entry the BR:

- Detects the OEM authentication scheme and Encryption enable/disable to be used and store into the BIT.
- Updates flow status.
- Calls the secure BR code dispatcher
- Checks if this is SC7 boot. If so, go to SC7/Warm boot path.
- If this is not SC7, then either go to the RCM path or the Coldboot path.

If the BR is proceeding to the cold boot or RCM path, the crypto initialization dispatcher is run.

6.2.1 Secure Section Initialization Dispatcher

The BR secure section initialization dispatcher is called first before branching into the other major code paths. Tasks that are required to be done for all boot paths (and placed into the secure section of IROM) can be added here.

Task table:

```
/**
 * NvBootTask master task list for in secure mode of operation.
 *
 * IMPORTANT NOTE: Secure dispatcher expects functions to return
 *                 NvBootError.
*/
static const NvBootTask Tasks[] = {
    { &NvBootBpmpSubSystemInit, 0x50 },
    { &NvBootBpmpEnableApb2jtag, 0x51},
    { &NvBootBondOutRegUpdate, 0x52},
    { &NvBootBpmpSetupSrkFirewallForColdBoot, 0x53},
    { &NvBootMainSecureInit, 0x54},
};
```

6.2.2 Cold Boot

Cold boot is a boot sequence where everything is done without any prior context. The cold boot path is taken when an external main reset assertion to the Mariko chip occurs. The major phases of cold boot are:

- Initialize secondary boot device to known good safe values.
- Read BCT from secondary storage. Authenticate and decrypt BCT.
- Optional: Re-initialize secondary boot device to new values
- Read BL1 from secondary storage. Authenticate and decrypt using OEM keys.

6.2.2.1 Coldboot Init

Function: NvBootColdBootInit()

- Record boot type as cold boot in the BIT
- Set up boot device to known good safe values.
- Log the execution status as `NvBootFlowStatus_CBSetupBootDevice`, as well as the tick count in `NvBootTimeLog.NvBootSetupTickCnt`.
- Set `BootInfoTable.DevInitialized = TRUE`.

6.2.2.2 Reading BCT and Redundancy

Function: NvBootColdBootReadBct()

The BR uses fuses and straps to determine the secondary boot device to read from. The BCT is read from secondary storage into a fixed location in IRAM (determined at compile time by the linker script; but it is expected to be somewhere in the BR work area of IRAM A). Once the read phase is complete, the BR will attempt to authenticate the BCT using one of the supported authentication schemes[1]. If this step fails, the BR will attempt to read the next copy of the BCT (a maximum of 64 copies of the BCT are allowed) and authenticate it. If the authentication of the BCT is successful, the encrypted section of the BCT is optionally decrypted in place. Once decrypted, the BR does additional validation of BCT parameters to make sure they are of reasonable values – for example checking if the version is correct, or the block and page sizes are within the legal range, or if the correct padding format is present (this not a complete list, please refer to the function `ValidateBct` in `nvboot_bct.c`).

6.2.2.3 SDRAM Initialization

Function: `SetupSdram`()

If the BCT.NumSdramSets parameter in the bct is 0, skip this step. Otherwise, the BR initializes SDRAM based on the SDRAM parameters embedded in the BCT.

Pseudo code for the function is as follows:

```
if(BCT.NumSdramSets == 0)
       Return Success;

Read strap to select which SDRAM parameter set to use.

Return error if selected strap parameter set is higher than the number of sets in the
BCT.


if(EmcClockSource parameter is
   CLK_RST_CONTROLLER_CLK_SOURCE_EMC_0_EMC_2X_CLK_SRC_PLLM_UD
   Or
   CLK_RST_CONTROLLER_CLK_SOURCE_EMC_0_EMC_2X_CLK_SRC_PLLM_OUT0)
{
       Start PLLM.
}

Call SDRAM initialization function provided by MC team.

Set BootInfoTable.SdramInitialized = TRUE;

Return;
```

[1] See Secure Boot section for further details.

6.2.2.4 Process BCT Settings

If the BCT is authenticated, the BR can safely trust the data stored in the BCT. It is at this point where the function `NvBootColdBootEnableBctFlags` is called. This function typically deals with MSS key generation and distribution settings, but can always be expanded to add new functionality.

6.2.2.5 Secondary Boot Device Re-Initialization (Optional, Decision by OEM)

Function: NvBootColdBootReInit()

It is possible to store secondary boot device parameters in the BCT, which can override the default safe values used by the BR for the reading of the BCT (or, the limited configuration options stored in fuses). Device initialization comes at a cost of increased boot latency from the controller or device's re-initialization protocol.

6.2.2.6 Read and Authenticate BL1

Function: NvBootColdBootLoadBl()

The location and signature of the BL1 shall be separated from the BCT for Mariko. The BootLoaderInfo struct in the BCT points to the location where the BL1 copies reside in secondary storage. Sizing, signature, and other information are stored in the NvBootOemMb1Info header.

The read and authentication process is as follows:

1. Get the StartBlock, StartPage from the BCT's BootLoaderInfo struct.
2. Read in the BL1 header at StartBlock+StartPage.
3. Authenticate the header. Full authentication scheme details are in Section7.3.3.1 OEM BL Signing Flow.
4. If the header is authenticated, read in the BL1 binary into IRAM or SDRAM, depending on the LoadAddress specified in the NvBootOemMb1Info header. Note, BR needs to sanitize the LoadAddress and Length, to make sure the copy isn't into unauthorized memory regions (like BR stack) and take care of possible address overflow.
5. Authenticate the BL1 binary.
6. If authentication fails, check each copy of NvBootLoaderInfo in the BCT and repeat steps 1 through 5. The number of copies of the BL1 is indicated in the BootLoadersUsed field in the BCT.
7. If step 5 is successful, the BR proceeds to the secure exit dispatcher, which passes control to BL1.

6.2.3 USB Recovery Mode (RCM)

In general (but with some exceptions), if any errors in the boot process occur, the BR will trigger RCM mode. The main purpose of RCM is to allow BR to transfer control to a downloaded binary pushed through USB, what is normally referred to as the "USB applet". The USB applet can do anything that can be executed from the BPMP, like facilitate the re-flashing of the secondary storage device, do silicon testing, or initialize SDRAM.

Once BR recognizes RCM mode, it sits in a spin loop waiting for messages to come through USB. The BR accepts only signed messages in the proper format. RCM messages and payloads must be signed by the OEM.

This is the pseudo code of the loop that processes RCM messages:

```
while(DownloadAndExecuteNvBinary message has not be received)
{
Poll and wait for RCM message over USB.
Receive RCM message.
       if(Message is authentic)
              Issue appropriate response.
       Else
              Return error*
       Execute the message action (if any) if authentication was successful
```

```
}
DownloadAndExecuteBinary message is received. Authenticate message and payload and if
successful, proceed to Secure Exit.

*An Error return results in BR proceeding to the secure exit path. BR will exit to a
“branch to self” bootloader and spin.
```

6.2.3.1 Deltas from Tegra X1

The SecureDebugControl flag in the RCM header is split into two fields.

6.2.3.2 RCM Message Format

An “RCM message” in NVIDIA parlance, is made up of a RCM header plus optional binary payload. The RCM header is the struct `NvBootRcmMsg`, defined in `nvboot_rcm.h`. The only RCM message that requires a binary payload is the `DownloadAndExecuteNvBinary` message. The other messages can be sent with just the RCM header.

The RCM “opcode” specified in the header via the Opcode field determines the underlying command that the USB host wishes the BR to process. For example, the USB host may wish to update the debug policy of the chip before BR exit, and therefore should send an RCM message with the opcode “`NvBootRcmOpcode_SetDebugFeatures`”. For a full list of opcodes, please refer to the struct `NvBootRcmOpcode` in `nvboot_rcm.h`.

6.2.3.3 RCM Flow

The RCM flow of execution is dictated by the RCMTasks dispatcher:

```
static const NvBootTask RCMTasks[] = {
    { &NvBootRCMInit, 0x201 },
    { &NvBootRCMSendUniqueId, 0x202 },
    { &NvBootRCMProcessMsgs, 0x203 }
};
```

6.2.3.4 RCM Init and Connect

Function: NvBootRCMInit()

This is the initialization function for RCM mode. The BR:

- Reloads the WDT if enabled, the timeout value is specified in the define `WDT_TIMEOUT_VAL_RCM`.
- Updates BIT info that we have reached an RCM boot type.
- Explicitly re-initializes the secure debug control bits in the boot context struct to zero (meaning disabled) for safety.
- Explicitly initializes the default hand-off / execution address to jump to self code in NvBootMainNonSecureBootLoader.
- Exits at this point if the Debug RCM mode is detected.
- Sets up function pointers to USB hardware functions based on which USB port to use for RCM.
- Initializes USB hardware.
- Polls wait, Try to enumerate and or connect.

6.2.3.5 Send Unique ID

Function: NvBootRCMSendUniqueId()

Once a valid USB connection is detected, the BR will first send over 16-bytes of data. This data packet contains a 100-bit unique ID (“ECID”), as well as some extra chip state encoded into the unused upper bytes of the 16-byte data packet. The extra info helps the USB host on the other side of the connection know which cryptographic authentication scheme the BR accepts to authenticate RCM messages.

The proposed ECID encoding by BR is as follows:

ECID Bitfield	Usage	Additional Notes
[99:0]	ECID	
[103:100]	Mariko: APB_MISC_GP_HIDREV_0_MAJORREV	Tegra X1: 0x1 Mariko: 0x2, MAJORREV allows us to distinguish between Tegra X1 and Mariko.
[111:104]	Mariko: APB_MISC_GP_HIDREV_0_CHIPID	Chip ID i.e., 0x21
[115:112]	Mariko: APB_MISC_GP_MINORREV	
[123:116]	Reserved	
[126:124]	BOOT_SECURITY_INFO[2:0]	[1:0] = Authentication Scheme [2] = Secure Boot Encryption enable using BEK
[127]	FUSE_PRODUCTION_MODE	

6.2.3.6 Process Received Messages

Function: NvBootRCMProcessMsgs()

After the initial handshake, the BR waits in a loop for RCM messages from the USB host. Any number of messages can be received until an NvBootRcmOpcode_DownloadExecuteNvBinary message is received with a binary payload. Once this message is received and authenticated, the BR jumps to the binary payload (usually in this case the mb1_recovery binary) after completing the common secure exit path of the BR.

6.2.4 Warm Boot Flow

Warm boot also referred to as SC7-exit is exit from lower power state where most of SOC except PMC, AON units and DRAM is powered off. (This state is also called Deep Power Down or DPD). In this flow, Boot ROM is not expected to read boot images from an external medium but instead loads and validates boot images/resume-firmware from DRAM which was placed in self-refresh mode prior to SC7 entry.

Main responsibilities of Boot ROM in this flow are:

1. Initialize Memory Controller MC/EMC and bring DRAM out of self-refresh. SDRAM BCT params used for the initialization process are stored in PMC Scratch registers.
2. Load and validate resume firmware from DRAM into IRAM.
3. Decrypt and restore encrypted SE context from DRAM.

6.2.4.1 Warm Boot Dispatcher

Warm boot dispatcher table is part of the dispatcher library at dispatcher/nvboot_tasks_s.c:

> **Note:** This dispatcher table is subject to change during development.

```
{ &NvBootHaltAtWarmboot, 0x601},
{ &NvBootWarmBootUnPackSdramStartPllm, 0x602},
{ &NvBootCryptoMgrHwEngineInit, 0x604},
{ &NvBootCryptoMgrInit, 0x605},
{ &NvBootCryptoMgrDecKeys, 0x606},
{ &NvBootWarmBootSdramInit, 0x608},
{ &NvBootWarmBootOemProcessRecoveryCode, 0x60B},
{ &NvBootLP0ContextRestore, 0x60D},
```

6.2.4.2 Halt at Warm Boot

Function: NvBootHaltAtWarmboot

Debug feature that allows breaking into Boot ROM during warm boot flow on pre-production devices. This is controlled by bit 4 in Scratch register PMC_SCRATCH0_0.

6.2.4.3 Unpack SDRAM Parameters and Start PLLs

Function: NvBootWarmBootUnPackSdramStartPllm

Before SDRAM is brought out of self-refresh, PLLM is started using parameters stored in PMC SCRATCH2. For a breakdown, see include/t210/nvboot_pmc_scratch_map.h.

Next, SDRAM parameters are unpacked from PMC scratch. SDRAM parameters are stored in a tightly packed format in EMC scratch registers before sc7-entry. These are unpacked (using code drop from memory team) into IRAM. Since there is no BCT in SC7, the IRAM area normally reserved for the BCT can be reused.

6.2.4.4 Setup MC/EMC and Initialize SDRAM

Function: NvBootWarmBootSdramInit

Call MC code drop with SDRAM BCT constructed from unpacked scratch registers. MC Code drop sets up MC/EMC registers and brings DRAM out of self-refresh.

6.2.4.5 Recovery firmware Validation

Function: NvBootWarmBootOemProcessRecoveryCode

This purpose of this function is to validate and copy the SC7 recovery firmware from DRAM. The SC7 recovery firmware is signed by the OEM, and optionally encrypted with the BEK. The location of the SC7 firmware in DRAM is specified by PMC_SECURE_SCRATCH119. For full details on the authentication and validation steps, see Section 8.0 Secure SC7 / Warm Boot Resume.

6.2.4.6 SE Context Restore

SE Context prior to SC7-entry is encrypted with the Secure Restore Key (SRK) and stored in DRAM. For Mariko, there are two SRKs, one for each independent SE context blob. The first context blob stores the SE0 context. The second one stores the SE2 and PKA context.

The table below summarizes the scratch register usage for SE context restore.

> **Note:** The scratch registers have all been moved to secure scratch registers for Mariko.

SE0 encrypted context location	PMC_SECURE_SCRATCH117
SE2 and PKA encrypted context location	PMC_SECURE_SCRATCH116
SE0 SRK	PMC_SECURE_SCRATCH4 to 7
SE2 & PKA SRK	PMC_SECURE_SCRATCH120 to 123

6.2.4.7 Additional guidelines for SE context restore sequence

Prior to SE context restore, TZRAM must have been restored from the AO storage area first, since TZRAM is a valid storage location for the saved SE context.

7.0 SECURE BOOT

This section delves into the specifics of the Boot ROM portion of the Tegra Secure Boot process. The Boot ROM forms the root of trust of the whole system. When the Tegra chip is powered on, the BPMP executes BR code from read-only IROM. The BR is the first link in the chain that ensures that each software component of the system is trusted. From initial power on to rich-OS execution, each step is vetted before hand-off to the next link in the chain.

The Boot ROM supports the following secure boot / security related features:

- Fuse encryption of OEM assets (SBK, KEK, BEK)
- AES encryption; CBC mode of operation; OEM configurable
- Authentication using RSASSA-PSS[2], and AES-CMAC; OEM configurable
- Secure configurability of chip debug features
- Secure key storage in IROM (Encrypted with FEK)
- Factory secure provisioning of assets/keys
- Default SE AES keys, for OEM usage. See Section 2.3.1 for more details. These keys are encrypted by the NV FEK.

7.1 General Security Requirements

- Memory copy (memcpy) functionality must validate the destination, source and length parameters. No copies are allowed unless:
  - The destination is a valid memory address (IRAM, SDRAM, and TZSRAM).
    - Furthermore, the destination cannot be BR's own "work area" for stack, and other data structures (IRAM A region).
  - The data to be copied will fully fit into the destination area without overlapping into invalid memory space (Destination must be IRAM, SDRAM, or TZSRAM.)
  - The source is a valid memory address for BR (IRAM, SDRAM, and TZSRAM).
- If IRAM is the destination, BL1 is copied to a fixed location at the start of IRAM-B (0x40010000). The size of BL1 is capped at the size of 0x30000 KB (IRAM B, C, and D).
- If SDRAM is the destination, the BL1 is copied to the address specified in LoadAddress of the NvBootOemMb1Info header. The BR should check that the Length specified in the NvBootOemMb1Info header is not larger than the (size of DRAM - LoadAddress).
- SC7 firmware is copied to a fixed location at the start of IRAM-B (0x40010000). The size of the SC7 firmware is capped to 0x30000 KB (IRAM B, C, and D) minus the size of the SC7 header `NvBootWb0RecoveryHeader`.
- RCM header + applet is copied to a fixed location at the start of IRAM-B (0x40010000). The size of the header and applet is capped to 0x30000 bytes (IRAM B, C, and D).
- The copying of any payload must happen once into IRAM. The verification of the payload must occur in place in IRAM.
- The entry point of the code must be in the IRAM B, C, and D region for SC7 firmware.
- The entry point of the BL1 must be in the IRAM B, C, D region, or in SDRAM.
- Any parameters used by BR shall be from post-authenticated payloads/headers.

[2] Key sizes supported: 2048-bits (Legacy SE RSA engine)

7.2 Secure Boot Initialization

All of the common tasks that BR shall complete prior to branching to the major secure boot paths (secure cold boot, secure SC7, secure RCM) occurs during secure boot initialization.

These tasks are completed in the function `NvBootMainSecureInit`, as well as the tasks in the `CryptoInitTasks` dispatcher table.

The NvBootMainSecureInit function reinitializes BR context explicitly to default values for security reasons. The CryptoInitTasks dispatcher table mainly serves to initialize hardware engines (SE), the crypto manager, and prepare cryptographic keys for usage by BR.

7.2.1 Hardware Crypto Accelerator Engines Initialization

Dispatcher table: `CryptoInitTasks`

Function: `NvBootCryptoMgrHwEngineInit`()The crypto accelerators are initialized by the BR crypto manager at this step.

7.2.2 BR Crypto Manager Initialization

Dispatcher table: `CryptoInitTasks`

Function: `NvBootCryptoMgrInit()`

The crypto manager initializes its internal context at this step. As well, it calls `NvBootCryptoMgrSenseChipState()` which reads the appropriate OEM fuses corresponding to security configuration. Once this function is complete, the crypto manager is aware of which authentication schemes and encryption levels to use for the next stages of secure boot.

The main fuse to be read in this function is the FUSE_BOOT_SECURITY_INFO fuse. This is an OEM burnable fuse to set the authentication and confidentiality policy of the final shipped product.

7.2.2.1 Decrypt and Load OEM AES Keys (OEM fuse encryption optional)

Dispatcher table: `CryptoInitTasks`

Function: `NvBootCryptoMgrLoadOemAesKeys()`

Once the OEM FEK is loaded, the BR can now load the OEM AES keys from fuses. OEM fuses can be optionally encrypted with an FEK (see previous section) assigned to the OEM.

Upon function entry, the BR checks FUSE_BOOT_SECURITY_INFO[3], which controls the OEM fuse encryption feature.

Pseudo code for this function:

```
if( OEM fuse encryption == TRUE)
{
       for (each OEM key)
       {
Read the key from fuses into IRAM
Decrypt the key in IRAM directly into an SE key slot.
}
}
else
{
       for (each OEM key)
{
Read the key out into IRAM
Load the key into an SE key slot
}
}
```

OEM encryption key	Register location
Key Encryption Key	TBD
Boot Encryption Key (BEK)	TBD
Secure Boot Key (SBK)	FUSE_PRIVATE_KEY0 to FUSE_PRIVATE_KEY3

7.3 Secure Cold Boot

This section focuses on the secure boot aspect of the cold boot path. See the Section 6.2.2 Cold Boot for the overall cold boot flow.

The general secure cold boot chain of trust is: Boot ROM → Authenticate BCT (using OEM public key) → Decrypt BCT (using BEK) → Read BL1 Header → Authenticate BL1 Header → Read BL1 → Decrypt BL1 (using BEK) → Hand off control to BL1. Items in red are optional and decided by the OEM.

7.3.1 Authenticate BCT

7.3.1.1 Public Key Cryptography (PKC) Based Authentication

The following steps are taken to authenticate the BCT:

1. The BCT is read from secondary boot device storage wholly into IRAM at a fixed location. The location of the BCT is dynamically calculated at linking time. The BR linker script specifies the BCT to be placed immediately after the BIT.
2. The BR calculates the SHA256 hash of the set of public crypto parameters stored in the BCT, referred to as the *Pcp*. The Pcp is designed to house the public keys of all of the PKC authentication schemes supported by the BR. This is done to allow the BCT to be signed simultaneously by one or more of the PKC authentication schemes supported. This gives flexibility to customers (allows change of authentication scheme without resigning) phase as well as to NVIDIA (flexibility for verification). For Mariko, only RSASSA-PSS is supported.
3. The BR reads the SHA256 hash of the Pcp stored in fuses by the OEM at the chip provisioning stage. The fuses used are FUSE_PUBLIC_KEY0 to FUSE_PUBLIC_KEY7.
4. If the hash comparison fails, the BCT authentication step fails, and the BR will retry by reading any redundant copies of the BCT from secondary storage. See Section 6.2.2.2 Reading BCT and Redundancy for more information.
5. If the hash comparison is successful, the BR will load the authenticated public key into the appropriate crypto accelerator. For Mariko, the BR will use the SE with a 2048-bit key size.
6. The signature of the BCT is then calculated and verified with the signature stored in the BCT. The algorithm used is RSASSA-PSS from PKCS #1 v2.2, and the input message is in the signed section of the BCT from the `RandomAesBlock` field to the end of the BCT. If the RSASSA-PSS_VERIFY operation fails, BR will retry according to the BCT redundancy policy. If the signature comparison passes then proceed to BL1 read and verification steps.

7.3.1.2 Symmetric Key Based AES-CMAC Authentication

The following steps are taken to authenticate the BCT:

1. The BCT is read from secondary boot device storage wholly into IRAM at a fixed location. The location of the BCT is dynamically calculated at linking time. The BR linker script specifies the BCT to be placed immediately after the BIT.

   The MAC of the BCT is calculated using AES-CMAC (RFC 4493). The key used is the SBK. The input message is the signed section of the BCT from the `RandomAesBlock` field to the end of the BCT. If the MAC comparison

between the generated MAC and the MAC stored in the BCT fails, BR retries according to the BCT redundancy policy. If the signature comparison passes then proceed to the BL1 read and verification steps

7.3.2 BCT Decryption (Optional; OEM Controlled)

If BCT encryption is enabled, it must be decrypted before usage. It is at this stage of cold boot where this should occur. Note that the decryption should begin in the signed section of the BCT, but after the unencrypted but signed section (new for Mariko). See Section 5.1 for more details.

7.3.3 BL1 Signing and Verification

In the past, updating the boot loader binary required the BCT to be re-signed as well, since the `NvBootLoaderInfo` header is contained in the BCT. For Mariko, the dependency between BCT and `NvBootLoaderInfo` shall be removed. This allows the updating, re-signing, and flashing of the BL without having to re-flash and re-sign the BCT. The following changes are needed to support this feature:

- A new boot loader header is created, named NvBootOemBlHeader, containing the signature of the BL1 as well as the length, load address and entry point.

The NvBootLoaderInfo struct in the BCT is simplified. It still holds the start location of the new BL header in secondary storage (i.e. the `StartBlock` and `StartPage` fields from `NvBootLoaderInfo`).The new BL header uses the following structure:

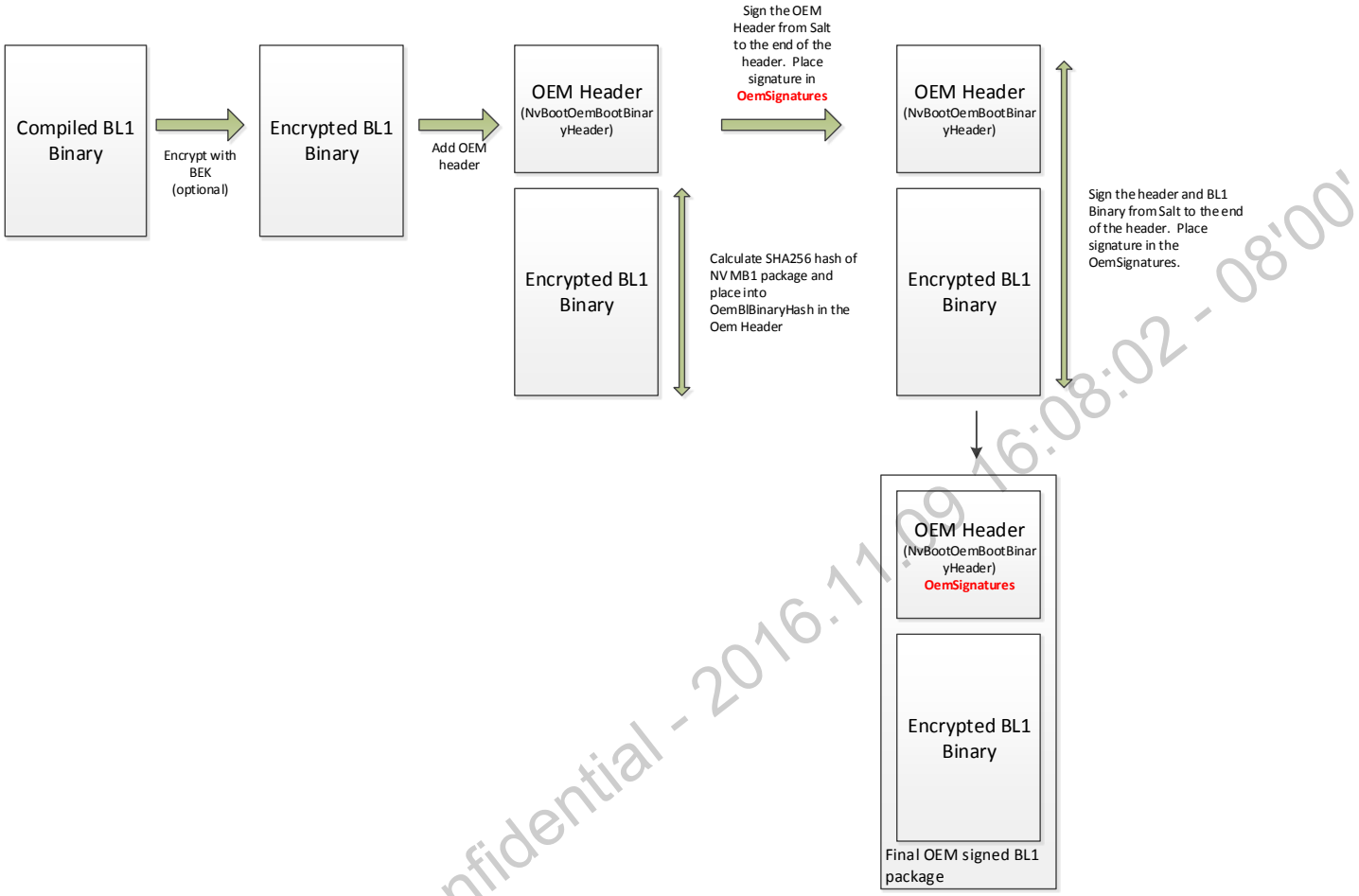
```
/**
* Stores information needed to locate and verify a boot loader.
    *
 * There is one \c NvBootLoaderInfo structure for each copy of a BL stored on
 * the device.
 */
typedef struct NvBootOemBlHeaderRec
{
    /// All cryptographic signatures supported will be stored here. The BL can be
    /// simultaneously signed by all cryptographic signature types.
    NvBootCryptoSignatures OemSignatures;

    /// Signed/hashed section starts here.
    /// Salt, i.e. Random bits of data.
    NvU32[8] Salt;

    /// The SHA256 hash of the BL binary.
    NvBootSha256Hash OemBootBinaryHash;

    /// Specifies a version number for the BL. The assignment of numbers is
    /// arbitrary; the numbers are only used to identify redundant copies
    /// (which have the same version number) and to distinguish between
    /// different versions of the BL (which have different numbers).
    NvU32      Version;

    /// Specifies the length of the BL in bytes. BLs must be padded
    /// to an integral number of 16 bytes with the padding pattern.
    /// @note The end of the BL cannot fall within the last 16 bytes of
    /// a page. Add another 16 bytes to work around this restriction if
    /// needed.
    /// This length is the length of NV header + MB1
    NvU32      Length;

    /// Specifies the starting address of the memory region into which the
    /// BL will be loaded.
    NvU32      LoadAddress;
```

```
        /// Specifies the entry point address in the loaded BL image.
        NvU32      EntryPoint;

        /// Specifies an attribute available for use by other code.
        /// Not interpreted by the Boot ROM.
        NvU32      Attribute; ///TODO: Do we need this?

        /// Reserved field. Pad by 12 bytes to make the signed section aligned to
        /// AES block size.
        NvU8 Reserved[12];
    } NvBootOemBlHeader;
```

7.3.3.1 OEM BL Signing Flow

1. The OEM compiles the BL1 binary from source. At this stage, the BL1 binary is in unencrypted / plain text format. The BL1 binary is padded to align at a 16-byte boundary (AES block size).
2. The OEM may optionally encrypt the BL1 binary using the BEK with AES-CBC, and an Initialization Vector (IV) of all zeroes. (Note, enable this in FUSE_BOOT_SECURITY_INFO).
3. The OEM creates a new instance of the BL1 Header, NvBootOemBootBinaryHeader, and fill in all the appropriate fields. The OEM may use the Salt member to mitigate pre-computation/rainbow table type attacks.
4. The OEM calculates the SHA256 hash of the BL binary. Place the resulting hash into NvBootOemBlHeader.OemBlBinaryHash.
5. The OEM signs the NvBootOemBootBinaryHeader using their chosen authentication/signing scheme, starting from Salt to the end of the struct. Place the resulting signature into the OemSignatures field.
6. At this stage, the BL1 is fully signed by the OEM, and ready for flashing onto the device. Note, the NvBootOemBootBinaryHeader and BL1 binary together are referred to as the “BL1 package”. The NvBootOemBootBinaryHeader header is placed at the beginning of the BL1 package, and the BL1 binary is placed contiguously with the header.

OEM Boot Loader Signing Flow



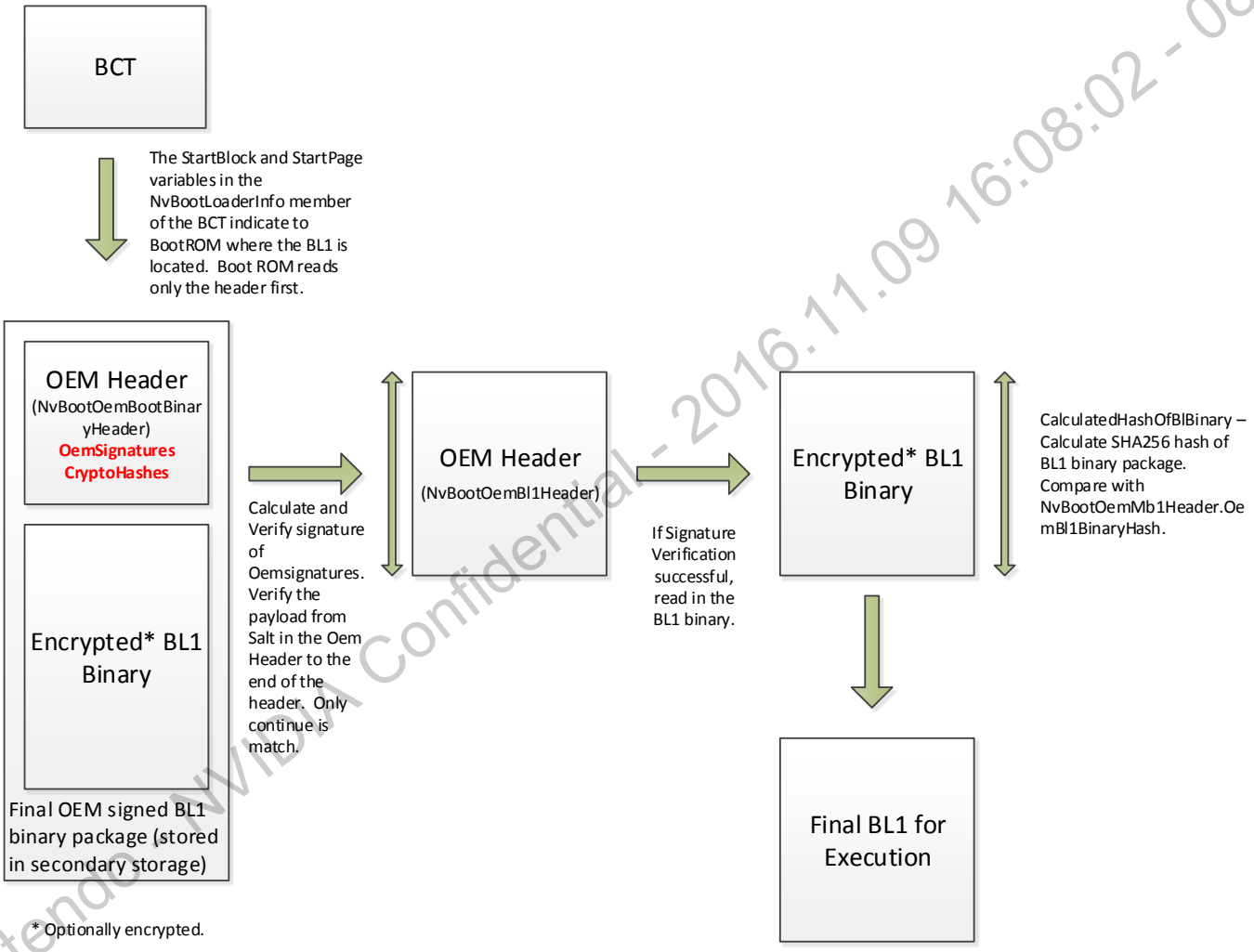
7.3.3.2 OEM BL Verification Flow

1. The BR uses the `StartBlock` and `StartPage` fields in the `NvBootLoaderInfo` member of the BCT to know where to read the BL and header from secondary storage. Because the BR does not know the size of the BL1 binary package, the BR reads the `NvBootOemBootBinaryHeader` (of which the BR knows the size) at `StartBlock+StartPage`.
2. The BR verifies the OEM cryptographic signature of the BL1 package using the authentication scheme selected by the OEM in fuses. Note that the OEM public key has already been verified and loaded before this step has been reached. To do so, the BR calculates the signature of the `NvBootOemBootBinaryHeader` from `Salt` to the end of the struct and compares against the signature stored in `OemSignatures`. If the authentication is successful, then the header is authentic. Otherwise, return a unique failure code.
3. At this point, the BR can trust the `Length` field of the `NvBootOemBootBinaryHeader`. The BR reads the BL1 Binary from secondary storage starting from where it left off reading the `NvBootOemBootBinaryHeader`.

> **Note:** The BR must check the value of `Length` and ensure it DOES NOT exceed the maximum IRAM buffer size allocated to store the BL header and the BL binary. If size exceeds the maximum buffer size, return a unique failure code.

4. Once the BL1 package is read into IRAM, the BR calculates its SHA256 hash. Let `CalculatedHashOfBL1Binary` be the name of this hash.
5. The BR compares `CalculatedHashOfBl1Binary` and `NvBootOemBootBinaryHeader.OemBl1BinaryHash`. If these hashes match, then the BR can trust that the BL1 binary package was signed by the OEM and authenticated with the OEM public key.
6. If OEM encryption is enabled, decrypt the whole BL1 binary package in place using the BEK.

OEM Boot Loader Verification Flow



8.0 SECURE SC7 / WARM BOOT RESUME

This section details the SC7 steps taken by the BR to authenticate and copy the SC7 firmware.

> **Note:** `NvBootWb0RecoveryHeader` is the struct defined as the SC7 firmware header.

1. Read AON Shadow TZRAM PG control at `APBDEV_PMC_TZRAM_PWR_CNTRL_0_TZRAM_SD`. If `APBDEV_PMC_TZRAM_PWR_CNTRL_0_TZRAM_SD == 0` (i.e., not powergated) trigger the TZRAM restore sequence and wait for the restore to complete before loading the SC7 resume FW. The programming sequence to trigger the TZRAM restore from the AON domain is as follows:
   a. SE_TZRAM_OPERATION ← 0x3 (REQ = INITIATE, MODE= RESTORE).
   b. BR polls SE_TZRAM_OPERATION_0_BUSY field until it becomes IDLE (from BUSY).
2. Read the address of the SC7 header and firmware from PMC_SECURE_SCRATCH119. The header and firmware are expected to contiguous.
3. Copy the SC7 firmware header only, from the address specified in PMC_SECURE_SCRATCH119 to a fixed location at the start of IRAM-B (0x40010000). The address specified in the scratch register can be in TZSRAM or SDRAM. The size of the SC7 header is fixed, so the BR knows how much to copy.
4. Read the `LengthInsecure` value in the SC7 firmware header. If `LengthInsecure` is greater than `(0x3_0000 - sizeof(NvBootWb0RecoveryHeader))` bytes, return a unique `NvBootError` code. This error shall propagate back to the secure dispatcher that executes the SC7 task list, and will eventually reset the chip. Note that 0x3_0000 bytes is the size of IRAM B, C, and D.
5. Copy the SC7 firmware to IRAM-B, immediately after where the SC7 firmware header was copied. Copy the lower of (`LengthInsecure` OR (0x30000 bytes-`sizeof(NvBootWb0RecoveryHeader)`). 0x30000 bytes is the size of IRAM B, C, and D.
6. Authenticate the SC7 header and firmware using the OEM's chosen authentication scheme. The authentication is run from `RandomAesBlock` in the SC7 firmware header to the end of the SC7 firmware. If the authentication operation fails, return a unique `NvBootError` code. This error shall propagate back to the secure dispatcher that executes the SC7 task list, and will eventually reset the chip.
7. If encryption is enabled by the OEM, decrypt the SC7 firmware header and SC7 firmware from `NvBootWb0RecoveryHeader.RandomAesBlocks` to the end of the SC7 firmware.
8. Do some additional sanity checks on the validated header:
   a. LengthInsecure == LengthSecure
      If this check fails, return an NvBootError code**.
   b. RecoveryCodeLength > (LengthSecure - sizeof(NvBootWb0RecoveryHeader)
      If this check fails, return an NvBootError code**.
   c. The EntryPoint is within the range
      [0x4001_0350*, Minimum(0x4001_0350*+RecoveryCodeLength, 0x4003_FFFF)]If this check fails, return an NvBootError code**.
      * 0x4001_0350 is IRAM-B + the size of the SC7 firmware header.
      ** An error here eventually propagates back to the secure dispatcher, where a full chip reset occurs.
9. If the BR reaches this step, the SC7 firmware is considered authenticated.

9.0 SECURE EXIT

9.1 Secure Exit Deltas from Tegra X1

- Tegra X1 BR exception vector redirection to IRAM is removed for Mariko. Exception vector always points to BR at reset, during execution and after BR exit. Only TZ can change the exception vector
- Default SE key derivation procedure is done in the cold boot path before exit.

9.2 Description

- All paths through the BR must go through the secure exit path. The only exception to this is in the preproduction or FA chip state, where it is possible to boot to a UART payload.
- Secure exit occurs at the end of the main() function in the boot ROM at `nvboot/startup/boot0c.c`. The tasks executed in the secure exit path are defined in the `NvBootTaskListId_SecureExit` dispatcher:

```
// Tasks related to BR Secure Exit flow
static const NvBootTask SecureExitBootTasks[] = {
    { &NvBootSeHousekeepingBeforeBRExit, 0x501},
    { &NvBootMainProcessScrDbgCtrl, 0x502}
};
```

The last function in the secure exit dispatcher does not return, as it hands off to the next firmware in the secure boot chain.

9.3 Secure Exit Tasks

This numbered list is associated with function names, when they are created/re-written for Mariko.

1. If the chip state is NOT “NV Production Mode”, FUSE_PRIVATEKEYDISABLE is set to KEY_INVISIBLE which hides the SBK, DK, and BEK fuses from software. BR does not set FUSE_PRIVATEKEYDISABLE to KEY_INVISIBLE in the NV Production Mode chip state so that the SBK/BEK/DK fuses can be read by the fuse provisioning software, to check if they have been burned correctly.
2. If the SECURE_PROVISION_INFO[0] fuse (the FSKP “KEY_HIDE” fuse bit), program FUSE_PRIVATEKEYDISABLE to KEY_INVISIBLE.
3. If this is an SC7 exit, zero out the BCT IRAM area.
4. Set the necessary permission bits of the SE, except in SC7 exit. During SC7 exit, the SE state is restored by the BR SE context restore code.
5. Clear out IRAM region from end of BCT area to 0x40010000.
6. IROM patch cleanup function (`NvBootIromPatchCleanup()`) is called.
   a. Invalidate all CAMS in IPATCH_ROM_OVERRIDE_VALID_0.
   b. Restore EVP_SWI_VECTOR to power on reset default.
   c. Set all ROM_OVERRIDE_CAM registers to power on reset values.
7. Jump to final exit code `NvBootMainAsmSecureExit()`. This code is written in assembly and is in file nvboot/startup/start.S. The exit code does the following:
   a. Clear IRAM from end of BCT to end of the stack.
   b. Log current microsecond timestamp into the BIT’s NvBootTimeLogExit field.
   c. Program SB_CSR_0 register, making BR the secure/protected region of IROM unreadable, as well as clearing the SECURE_BOOT_FLAG to 0 (DISABLE).
      Clearing the SECURE_BOOT_FLAG makes the debug policy as programmed into APBDEV_PMC_DEBUG_AUTHENTICATION_0 take effect. This register was programmed earlier in the boot

process, either after the BCT is authenticated in the cold boot path or after an authenticated “SetDebugFeatures” RCM message was sent in RCM.

d. Read back SB_CSR_0 for safety (Avoid any timing related issue with PIROM region not being read locked)..

e. Jump to the authenticated exit address.

9.4 Clearing of IRAM at Exit

Note: The addresses below are subject to change. They are dynamically generated at compile time in the linker script.

Table 5: Cold Boot and RCM

Description of IRAM area	Address Start (Hex)	Address End (Hex)	Criteria / Chip State	Action
BCT buffer in IRAM	40000464	40002C64	If forced RCM (strap or PMC)	Clear to 0
BCT end to end of stack	40002C64	40007A50	Always	Clear to 0

Table 6: SC7

Description of IRAM area	Address Start (Hex)	Address End (Hex)	Chip state	Action
Boot Config Table (BCT)	40000464	40002C64	Always	Clear to 0
BCT end to end of stack	40002C64	40007A50	Always	Clear to 0

10.0 SECURE DEBUG

The Mariko secure debug control code is needlessly complicated. For Mariko, the secure debug control code and API usage model will be greatly simplified. Since the `APBDEV_PMC_DEBUG_AUTHENTICATION_0` can be written at any time while BR is executing (with SECURE_BOOT_FLAG=1), and as many times as we need to, the secure debug control evaluation code need not be called just at secure exit time.

A new “ProcessSecureDebugControl” function will be created and exposed, with the input into the function to be the desired value for `APBDEV_PMC_DEBUG_AUTHENTICATION_0`. The aforementioned new function will then evaluate the input value according to the defined criteria (ECID, chip mode, etc.) and write the correct value to `APBDEV_PMC_DEBUG_AUTHENTICATION_0`.

Two places where the new function is called are:

- A `SecureDebugControl` message is received in RCM mode.
- At `NvBootColdBootEnableBctFlags`, when the BCT has been authenticated.

10.1 Data Structure Modifications

The SecureDebugControl field in the BCT and in the RCM header shall be separated into two fields. The debug bits that are subject to ECID check are placed into one field, and the debug bits that aren’t subject to ECID check in the other. The bit mapping shall be the same for both fields. This makes for a clear separation between the ECID vs non-ECID checked debug bits.

The prospective field names are:

- SecureDebugControl_Not_ECID_Checked: DBGEN, NIDEN in bit fields 5:4, bit fields 3:0 reserved.
- SecureDebugControl_ECID_Checked: SPIDEN, SPNIDEN, DEVICEEN, JTAG_ENABLE in bit fields 3:0, bit fields 5:4 reserved. The SecureDebugControl field will be marked with the “deprecated” attribute.

The same changes are made to the RCM header.

10.2 Changes in the Fuse Block for Mariko related to Debug Authentication signals

BR does not need to re-implement the debug authentication fuse redirection IROM patch for Mariko.

11.0 PLL / CLOCKS USAGE

PLL programming sequences will be implemented according to the programming guide in the PLL datasheets.

11.1 PLL Lock Timeout and Safety Margin

Per the clocks team, the following guidelines for PLL lock are used:

- Wait for lock bit to be asserted, OR timeout after 300us. (Value depends on worst case lock times of all PLL's used in the system).
- After lock is asserted, wait for an additional 10us for the PLL to be stable (Safety margin. Ideally we should be able to start running logic once lock is asserted).

11.2 Miscellaneous Notes

All of the following PLLs in this section have the same recommended PLL startup sequence per each PLL's respective datasheet:

Each step requires at least one update rate clock (CLOCKIN/M). Some steps require a larger delay which is defined inline

- After power up IDDQ=1
- With SETUP=0, transition IDDQ 1 ->0
- Wait 5 µs (program M/N and other registers any-time before asserting ENABLE)
- Transition ENABLE 0 -> 1 (clock in must be running before asserting ENABLE)

Wait for lock signal to assert before enabling spread spectrum.

Recommended to use ENABLE to stop or start PLL while keeping IDDQ=0.

> **Note:** With ENABLE=1, do NOT use transition of IDDQ=1->0 to stop/start PLL.

During the Tegra X1 development phase it was confirmed from the clocks team that the IDDQ sequence was not necessary, and since the default POR value of SETUP is 0, that step was skipped too. Lastly, if the IDDQ sequence and SETUP programming is skipped, then the 5us wait time is not necessary either.

11.3 Boot ROM PLLP Programming Sequence

1. Program CLK_RST_CONTROLLER_PLLP_OUTA_0_PLLP_OUT1_RSTN to 0.
2. Program CLK_RST_CONTROLLER_PLLP_OUTB_0 fields PLLP_OUT3_RSTN = 0, and PLLP_OUT4_RSTN = 0.
3. Program CLK_RST_CONTROLLER_PLLP_OUTC fields PLLP_OUT5_RSTN = 0.
4. Just prior to PLLP start, BR will have programmed the detected oscillator frequency into `CLK_RST_CONTROLLER_OSC_CTRL_0`. This information is used by hardware to auto setup the parameters (DIVN, DIVM, CPCON, LFCON, VCOCON, DCCON, OUT1_RATIO, DIVP, OUT3_RATIO, and OUT4_RATIO) to PLLP and its dividers.
5. Enable locked detection circuitry for safety (POR reset value should already be enabled), program PLLP_MISC's PLLP_EN_LCKDET field to ENABLE.
6. The IDDQ 0->1 sequence is not required (but listed in the PLL datasheet). Also, since SETUP is 0 by default, it is not programmed by BR.
7. BR then sets PLLP_ENABLE = 1 to start PLLP.
8. Wait for PLL lock or timeout at 300 µs.

11.4 Boot ROM PLLU Programming Sequence

1. Enable lock detect. Program PLLU_MISC's PLLU_EN_LCKDET field to ENABLE.
2. Set M, N, P based on oscillator frequency according to this mapping:

```
static const UsbPllClockParams s_UsbPllBaseInfo[NvBootClocksOscFreq_MaxVal] =
{
    //DivN, DivM, DivP
    {0x025, 0x01, 0x1}, // For NvBootClocksOscFreq_13,
    {0x01C,  0x1, 0x1}, // For NvBootClocksOscFreq_16_8
    {    0,    0,   0}, // dummy field
    {    0,    0,   0}, // dummy field
    {0x019, 0x01, 0x1}, // For NvBootClocksOscFreq_19_2
    {0x019, 0x02, 0x1}, // For NvBootClocksOscFreq_38_4,
    {    0,    0,   0}, // dummy field
    {    0,    0,   0}, // dummy field
    {0x028, 0x01, 0x1}, // For NvBootClocksOscFreq_12
    {0x028, 0x04, 0x1}, // For NvBootClocksOscFreq_48,
    {    0,    0,   0}, // dummy field
    {    0,    0,   0}, // dummy field
    {0x025, 0x02, 0x1}  // NvBootClocksOscFreq_26
};
```

3. Set PLLU_ENABLE to ENABLE.
4. Wait for PLL lock or timeout at 1300 µs.

11.5 Boot ROM PLLM Programming Sequence

1. Disable PLL as precaution, set PLLM_ENABLE to DISABLE.
2. Enable lock detect using PLLM_MISC2_0_PLLM_EN_LCKDET.
3. Program PLLM_MISC1, PLLM_SETUP to value from BCT.
4. Program PLLM_MISC2_0_PLM_KVCO to value from BCT.
5. Program PLLM_MISC2_0_PLLM_KCP to value from BCT.
6. Program M, N, P dividers from BCT values.
7. Set PLLM_BASE_0_PLLM_ENABLE to ENABLE.
8. Poll for PLL lock or timeout after 300 µs.

11.6 Boot ROM PLLC4 Programming Sequence

1. Disable LCKDET and set PLLC4_ENABLE to DISABLE.
2. Set IDDQ from 1 to 0.
3. Wait 5 µs.
4. Program LCKDET to ENABLE, and also setup the MISC parameters: KCP = 0, KVCO = 0.
5. Program M = 2, N = 0x32, P = 0. See NvBootClocksGetPllMiscParams.
6. Set PLLC4_ENABLE to ENABLE.
7. Wait for PLL lock or timeout after 300 µs.

Notice

The information provided in this specification is believed to be accurate and reliable as of the date provided. However, NVIDIA Corporation ("NVIDIA") does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This publication supersedes and replaces all other specifications for the product that may have been previously supplied.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and other changes to this specification, at any time and/or to discontinue any product or service without notice. Customer should obtain the latest relevant specification before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer. NVIDIA hereby expressly objects to applying any customer general terms and conditions with regard to the purchase of the NVIDIA product referenced in this specification.

NVIDIA products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on these specifications will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this specification. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this specification, or (ii) customer product designs.

No license, either express or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this specification. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA. Reproduction of information in this specification is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

ARM

ARM, AMBA and ARM Powered are registered trademarks of ARM Limited. Cortex, MPCore and Mali are trademarks of ARM Limited. All other brands or product names are the property of their respective holders. "ARM" is used to represent ARM Holdings plc; its operating company ARM Limited; and the regional subsidiaries ARM Inc.; ARM KK; ARM Korea Limited.; ARM Taiwan Limited; ARM France SAS; ARM Consulting (Shanghai) Co. Ltd.; ARM Germany GmbH; ARM Embedded Technologies Pvt. Ltd.; ARM Norway, AS and ARM Sweden AB

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA, the NVIDIA logo, and Tegra are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright