# Marvell® ARMADA 16x Applications Processor Family

Version 3.2.x Boot ROM Reference Manual

Doc. No. MV-S301208-00, Rev. -

November 2010

PUBLIC RELEASE

Marvell. Moving Forward Faster

## Document Conventions

| | |
|---|---|
| ⇲ | **Note:** Provides related information or information of special importance. |
| ! | **Caution:** Indicates potential damage to hardware or software, or loss of data. |
| ϟ | **Warning:** Indicates a risk of personal injury. |

## Document Status

| | |
|---|---|
| Draft | For internal use. This document has not passed a complete technical review cycle and ECN signoff process. |
| Preliminary Tapeout (Advance) | This document contains design specifications for a product in its initial stage of design and development. A revision of this document or supplementary information may be published at a later date. Marvell may make changes to these specifications at any time without notice. Contact Marvell Field Application Engineers for more information. |
| Preliminary Information | This document contains preliminary specifications. A revision of this document or supplementary information may be published at a later date. Marvell may make changes to these specifications at any time without notice. . Contact Marvell Field Application Engineers for more information. |
| Complete Information | This document contains specifications for a product in its final qualification stages. Marvell may make changes to these specifications at any time without notice. Contact Marvell Field Application Engineers for more information. |

Milestone Indicator:
Draft = 0.xx
Advance = 1.xx
Preliminary = 2.xx
Complete = 3.xx

X . Y Z

**Work in Progress Indicator**
Zero means document is released.
Various Revisions Indicator

| | |
|---|---|
| Doc Status: PUBLIC RELEASE | Technical Publication: 3.00 |

For more information, visit our website at: www.marvell.com

# Table of Contents

# List of Figures

# List of Tables

# 1 Boot ROM Functional Overview

## 1.1 General Description

The ARMADA 16x Applications Processor Boot ROM software is preloaded into the processor internal ROM. No changes can be made to the Boot ROM because it resides in ROM.

## 1.2 Changes from Previous Stepping

**Table 1: ARMADA 16x A0 to B0 Differences**

| ARMADA 16x A0 Boot ROM | ARMADA 16x B0 Boot ROM |
|---|---|
| Boot ROM cannot resume from Hibernate mode directly to the operating system; it loads OBM, which resumes to OS. The resume package ID “0x52736D32” is NOT defined in ARMADA 16x Boot ROM. | Boot ROM resumes to OS directly without having to load OBM. The resume package ID “0x52736D32” is defined in ARMADA 16x B0 stepping. |
| For NAND/ONENAND only, Boot ROM searches for a valid Non-Trusted Image Module (NTIM) in the first 10 blocks. NTIM must be aligned to the start of each block. | For NAND, ONENAND, and SD_MMC only, Boot ROM searches for a valid NTIM in the first 10 blocks. NTIM has to be aligned to the start of each block. For SD/MMC, block size is hard-coded to be 512 bytes/block, which supports having the FAT partition table in address 0x0 of Partition 0 of an SD device while having the NTIM at a higher address. |
| Does not contain these return codes: 0xB, 0xC, 0x38, E5. See Appendix A for details. | Added these return codes: 0xB, 0xC, 0x38, E5. See Appendix A for details. |

Table 1: ARMADA 16x A0 to B0 Differences

| ARMADA 16x A0 Boot ROM | ARMADA 16x B0 Boot ROM |
|---|---|
| DDR configuration is not optimized. OBM should perform the DDR configuration. The CMCC package in the NTIM should read as follows:<br>CMCC_CONFIG_ENA_ID: 0x00000000<br>CMCC_MEMTEST_ENA_ID: 0x00000000<br>CMCC_CONSUMER_ID: 0x4F424D49 | ARMADA 16x B0 stepping supports resuming to OS code without having to load the OBM. This feature is called “QuickBoot.” Aspen A0 does **not** support QuickBoot. For QuickBoot to work, the Boot ROM must be able to read from the DDR device (that is, to read the ResumeFlag); the Boot ROM must reconfigure the ARMADA 16x DDR Controller before reading from the DDR device. Thus, the CMCC package in the NTIM needs to be as follows for QuickBoot to work in ARMADA 16x B0:<br><br>CMCC_CONFIG_ENA_ID: 0x00000001<br>CMCC_MEMTEST_ENA_ID: 0x00000000<br>CMCC_CONSUMER_ID: 0x54425249<br><br>If the CMCC_CONSUMER_ID is not 0x54425249 or if the CMCC_CONFIG_ENA_ID is not 1, then the Boot ROM will not be able to read from the DDR device. In that case, for ARMADA 16x B0, the NTIM needs to include a GPIO package which instructs the Boot ROM to take the DDR device out of Self-Refresh mode. This GPIO package is not necessary for ARMADA 16x A0 Boot ROM because it does not contain the QuickBoot feature. Since the Boot ROM cannot read from the DDR device in this case, it will not be able to resume to OS code; instead, it will load the OBM. The OBM will inspect the NTIM for a Resume package and resumes to OS code. The GPIO package looks as follows:<br><br>0x4750494F; GPIO Package<br>0x00000014; number of bytes in this package.<br>0x00000001; number of pairs: 1<br>0xB0000120; ddr command register address<br>0x00000001; initialize ddr command |
| Supports ONFI NAND MLC devices only. | Supports ONFI NAND MCL devices, Samsung non-ONFI MCL devices only. Tested with the following NAND MLC devices: Samsung device ID = 0xD5, 0xD7, and Hynix Device ID = 0xD5. |
| The Boot ROM does not boot from 4-bit SD/eSD cards/devices. See Errata DPF-641 in the Spec Update. | Fixed in ARMADA 16x B0 stepping. |
| The Boot ROM has a 4-second delay between each MMC/SD port if there is no card/device present. This affects SPI NOR devices. See Errata DPF-639 in the Spec Update. | Fixed in ARMADA 16x B0 stepping. |

## 1.3 Purpose Of This Document

This document covers the functional and operational details of the Marvell Boot ROM. Information in this document is required to understand the proper configuration, software requirements, and system requirements to deploy a platform. This document covers only the ARMADA 16x Applications Processor.

This document covers many important features, configurations, concepts, and requirements for proper operation with the Marvell Boot ROM. Topics covered include:

- Methods of booting the platform

- Memory options for system boot
- Requirements for non-trusted operation
- Marvell Boot ROM versions and feature sets
- Requirements for handling low-power modes
- Host tools required to generate boot information and collateral
- Methods for provisioning (initializing) a platform

## 1.4 ROM Location, Size, and Mapping

Table 2 describes the physical ROM size, physical ROM memory address, initial vector table mapping, and Marvel Boot ROM runtime address.

**Table 2: Marvell Boot ROM Physical Characteristics**

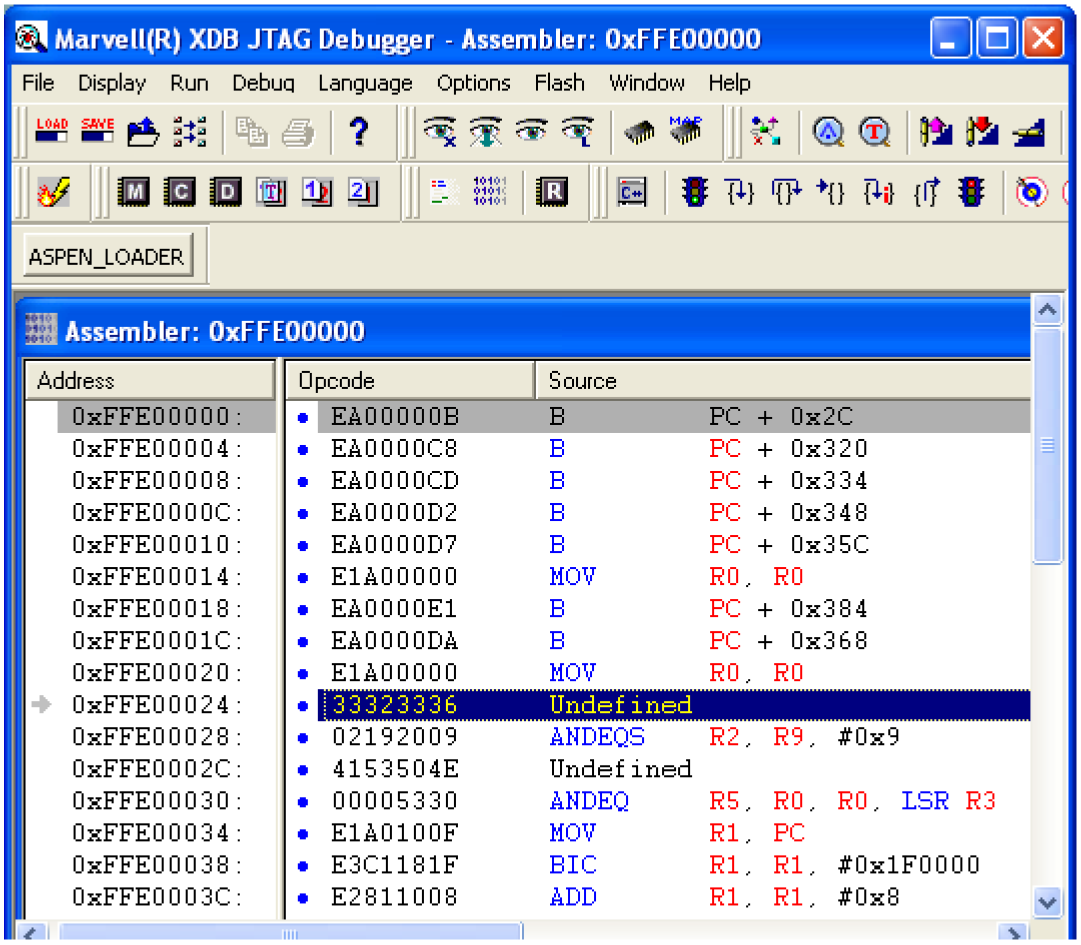
| Processor | ROM Size | ROM Memory Address | ROM Runtime Address | Vector Table Mapping |
|---|---|---|---|---|
| ARMADA 16x Applications Processor | 128KB | 0xFFE0_0000 - 0xFFE2_0000 | 0xFFE0_0000 - 0xFFE2_0000 | 0xFFFF_0000 |

## 1.5 Marvell Boot ROM Features

The ARMADA 16x Applications Processor Boot ROM version is identified by reading the memory location. The version includes four words (32 bits each) of data including the Boot ROM version, date, and two words to identify the processor stepping. Figure 1 provides an example of the version encoding for the ARMADA 16x Applications Processor. The version is decoded as follows:

- First word (32 bits) – ASCII encoded hex 0x33323434 = 3.2.44. All 3.2.x versions of the Marvell boot ROM have the Major.Minor.Build format, such as 0x33323132 = 3.2.12
- Second word (32 bits) – Unencoded date, where 0x02192009 = 2/19/2009.
- Third word (32 bits) – Processor family. These bytes are ASCII encoded hex for the processor family, where 0x4153504E = ASPN
- Fourth word (32 bits) – Processor stepping. In the example 0x00005330 = S0

Figure 1: Example of Marvell Boot ROM Version Located at 0x0000_0024


Marvell(R) XDB JTAG Debugger - Assembler: 0xFFE00000
File Display Run Debug Language Options Flash Window Help
ASPEN_LOADER
Assembler: 0xFFE00000

```
Address        Opcode      Source
0xFFE00000:  • EA00000B    B         PC + 0x2C
0xFFE00004:  • EA0000C8    B         PC + 0x320
0xFFE00008:  • EA0000CD    B         PC + 0x334
0xFFE0000C:  • EA0000D2    B         PC + 0x348
0xFFE00010:  • EA0000D7    B         PC + 0x35C
0xFFE00014:  • E1A00000    MOV       R0, R0
0xFFE00018:  • EA0000E1    B         PC + 0x384
0xFFE0001C:  • EA0000DA    B         PC + 0x368
0xFFE00020:  • E1A00000    MOV       R0, R0
0xFFE00024:  • 33323336    Undefined
0xFFE00028:  • 02192009    ANDEQS    R2, R9, #0x9
0xFFE0002C:  • 4153504E    Undefined
0xFFE00030:  • 00005330    ANDEQ     R5, R0, R0, LSR R3
0xFFE00034:  • E1A0100F    MOV       R1, PC
0xFFE00038:  • E3C1181F    BIC       R1, R1, #0x1F0000
0xFFE0003C:  • E2811008    ADD       R1, R1, #0x8
```

# 2 Marvell® ARMADA 16x Applications Processor Family Boot Feature Overview

Features are user-specified in the Non-Trusted Image Module (NTIM) header or defined by the processor.

## 2.1 Boot Memory

All processors use 128KB of internal memory for boot purposes. The memory is used for the Boot ROM data and stack, and also to load the first-level boot loader from flash memory. The location of the internal memory varies with processor implementation. The ARMADA 16x Applications Processor Boot ROM configures the L2 cache for use as a memory and is located at 0xD102_0000. Table 3 shows the available internal memory for each processor.

When the L2 cache is used, the Boot ROM disables, invalidates, and cleans L2 cache, then enables it as static RAM. Higher levels of software must re-enable L2 cache for use as a cache after the Boot ROM has been run on the system. If the L2 cache memory cannot be used, the ARMADA 16x Applications Processor tries a boot to CS0 by jumping to the base address of SMC_CS0 at 0x8000_0000.

**Table 3: Internal Memory Used by Boot ROM**

| Processor | Memory use by Boot ROM | Address Range |
|---|---|---|
| ARMADA 16x Applications Processor | L2 cache 128KB | 0xD102_0000 - 0xD103_FFFF |

Table 4 shows how the Boot ROM uses the memory as part of the boot process. The NTIM header is dynamically sized. See Chapter 3, “Image Modules” for details on sizing. The Bootloader can be loaded immediately following the NTIM header in the internal memory.

**Table 4: Boot Memory Layout by Version**

| Memory Usage | Start Address (ISRAM/L2) | Size |
|---|---|---|
| Vector Table for use by loadable image | Base Address (BA) from Table 4 | 64 bytes |
| Boot ROM Status Structure | BA + 0x40 | 192 bytes |
| Boot ROM Data and Stack | BA + 0x100 | 0x9F00 (39.75 KB) |
| NTIM | Defined by NTIM. Address must be BA + 0xA000 or higher. | Limited by Internal memory size described in Table 3 where NTIM + OBM must be less than Memory Size - 0xA000. |
| OEM Boot Module (OBM) | Defined by NTIM. Address must be BA + 0xA000 or higher. | |

## 2.2 Autoboot on an Uninitialized System

The Boot ROM is directed to probe a pre-determined address for each flash type that is supported for a valid NTIM header. If no header is found, the Boot ROM waits for a download from the USB port, if enabled (see the *Download Capabilities* section of the “Boot Operations” chapter.) The Boot ROM probes flash devices in the following order:

- XIP on SMEMC CS0
- x16 NAND on NFC CS0
- x8 NAND on NFC CS0
- OneNAND and Flex OneNAND on SMEMC CS0
- eSD or eMMC devices on MMC3
- Alternative eSD or eMMC devices on MMC3
- eSD or eMMC devices on MMC1
- SPI Flash

The current implementation of the ARMADA 16x Applications Processor probes the boot partition (Partition 1) of the eSD or eMMC device at offset 0x0 for the NTIM header. If the device does not support a physical boot partition, then the NTIM may be placed in the user partition and the probe process will find it.

The autoboot mechanism ends with the Boot ROM waiting for a download; if not, then the NTIM header is found for booting the system. Boot times can vary from one flash type to another because of the fixed-probe order.

## 2.3 Download Capabilities

The Boot ROM has the built-in capability to download and run an image over the USB OTG port. This mechanism uses the communication protocol defined in Chapter 9, “Communication Protocol”. The purpose of downloading is for manufacturing use on systems in the uninitialized platform boot state where the flash is not programmed, or for a boot failure. The intent is to allow an OEM to download software to program or debug the platform boot images.

The download mechanism is much slower than a boot from flash and is not intended as a normal boot option. The USB device driver in the Boot ROM can operate only as a device and must be attached to a Host PC running a utility that implements the communication protocol. The embedded device drivers cannot run in Host mode, which prevents plugging in USB mass storage devices such as a USB flash device.

# 3 Image Modules

An Image Module is a non-executable data header that contains a set of data structures that define flash information, binary image information, and OEM reserved data for non-trusted platforms. A Non-Trusted Image Module (NTIM) is used on non-trusted platforms and contains no security information. The details of the data structures for NTIM are presented in this chapter.

The Image Module format is described in Section 3.1, Image Module Format. The overall size of the Image Module is dynamic because it is a packed set of data structures of variable size, based on the information contained in all data structures.

The data structures that compose the Image Module are common between versions of the Image Module to allow for easy overlay and re-use of structures in software. Fields that are not used for non-trusted operation are considered reserved in the NTIM and should be set as specified in the sections that follow. All values in the Image Modules are 32 bits in size. The structures of the Image Module are shown in Section 3.1, Image Module Format and can be used as examples of software-based structures for creating an Image Module. The definition of each field is described in more detail in:

- Section 3.2, Version Information
- Section 3.2.1, Flash Information
- Section 3.2.2, Image Module Sizing Information
- Section 3.2.3, Image Information Array
- Section 3.2.4, OEM Reserved Area[SizeOfReserved]

The Image Module can be created by using the Marvell® Wireless Trusted Platform Tool Package or a custom tool created by the OEM. This section describes the structure and content of the binary header as it is programmed onto a flash device. These sections do not cover how to create the binary from user input—only how the Boot ROM interprets the data.

## 3.1 Image Module Format

The Image Module format for the boot operation consists of the structures described in this section. Table 5 provides information on required fields and optional fields for NTIM headers. The constant part of the Image Module (CTIM struct) is defined below. This structure is used to calculate dynamically the size of the NTIM. The structures IMAGE_INFO (or IMAGE_INFO_3_1_0 for NTIM v3.1), and KEY_MOD are stored as arrays of structures. The size of the array is specified by the NumImages and NumKeys variables, respectively, in the constant part of the NTIM. The SizeOfReserved variable is used to specify the number of reserved bytes stored in the NTIM header for optional parameters.

```
typedef struct{
{   VERSION_I VersionBind;
    FLASH_I FlashInfo;
    unsigned int NumImages;
    unsigned int NumKeys;
    unsigned int SizeOfReserved;
}CTIM, *pCTIM;
```

}CTIM, *pCTIM;

**Table 5: Non-Trusted Image Module Structures**

| Structure | Field | NTIM |
|---|---|---|
| VERSION INFORMATION | Version<br>Identifier<br>Trusted<br>IssueDate<br>OEMUniqueID | Y<br>Y<br>Y<br>Y<br>O |
| FLASH INFORMATION | Reserved[5]<br>BootFlashSign | R<br>Y |
| IMAGE MODULE SIZING INFORMATION | NumImages<br>SizeOfReserved | Y<br>O |
| IMAGE INFORMATION ARRAY IMAGE[NumImages] | ImageID<br>NextImage<br>FlashEntryAddr<br>LoadAddr<br>ImageSize<br>Partition Number | Y<br>Y<br>Y<br>Y<br>Y<br>Y |
| OEM RESERVED AREA[SizeOfReserved] | | O |

Based on the information in the CTIM structure, the NTIM can be sized using the following calculation:

- Size of NTIM = sizeof(CTIM) + (NumImages * sizeof(IMAGE_INFO)) + (NumKeys * sizeof(KEY_MOD)) + SizeOfReserved

Optionally, a master structure such as the NTIM structure below could be used to set pointers to the start of each section of the NTIM. The SetTIMPointers() below show C code examples. The SetTIMPointers() check the version of the NTIM to determine which Image information structure to use. Two versions of the NTIM header are currently supported: version 3.1.x (0x00030100) and version 3.2.x (0x00030200). The version identifier determines features supported in the header.

```
typedef struct
{
 pCTIM         pConsTIM;      // Constant part
 pIMAGE_INFO   pImg;          // Pointer to Images
 pKEY_MOD      pKey;          // Pointer to Keys
unsigned int   *pReserved;    // Pointer to Reserved Area
 pPLAT_DS       pTBTIM_DS;    // Pointer to Digital Signature
} TIM, *pTIM;

void SetTIMPointers( UINT8_T * StartAddr, TIM *pTIM_h)
{
    pTIM_h->pConsTIM = (pCTIM) StartAddr;// Overlap Contant Part of TIM with
actual TIM...
    pTIM_h->pImg = (pIMAGE_INFO) (StartAddr + sizeof (CTIM));
    if (pTIM_h->pConsTIM->VersionBind.Version >= TIM_3_2_00)
        pTIM_h->pKey = (pKEY_MOD) ((UINT_T)pTIM_h->pImg + \
                    ((pTIM_h->pConsTIM->NumImages) * sizeof (IMAGE_INFO)));
    else
        pTIM_h->pKey = (pKEY_MOD) ((UINT_T)pTIM_h->pImg + \
                    ((pTIM_h->pConsTIM->NumImages) * sizeof (IMAGE_INFO_3_1_0)));
    pTIM_h->pReserved = (PUINT) ((UINT_T)pTIM_h->pKey + \
                        ((pTIM_h->pConsTIM->NumKeys) * sizeof (KEY_MOD)));
    return;
}
```

**Figure 2: Sample Structures for C Code NTIM Implementation**

There are two versions of the Image Information structure based on the version of the NTIM being created.

```
typedef struct
 unsigned int Version;
 unsigned int Identifier;
 unsigned int Trusted;
 unsigned int IssueDate;
 unsigned int OEMUniqueID;
} VERSION_I, *pVERSION_I;

typedef struct{
 unsigned int Reserved[5];
unsigned int BootFlashSign;
} FLASH_I, *pFLASH_I;

typedef struct{
 unsigned intKeyID;
 unsigned int HashAlgorithmID;
 unsigned int ModulusSize;
 unsigned int PublicKeySize;
 unsigned int RSAPublicExponent[64];
 unsigned int RSAModulus[64];
 unsigned int KeyHash[8];
} KEY_MOD, *pKEY_MOD;

typedef struct{
 unsigned int ImageID;
 unsigned int NextImageID;
 unsigned int FlashEntryAddr;
 unsigned int LoadAddr;
 unsigned int ImageSize;
 unsigned int ImageSizeToHash;
 unsigned int HashAlgorithmID;
 unsigned int Hash[8];
 unsigned int PartitionNumber;
} IMAGE_INFO, *pIMAGE_INFO;

typedef struct{
 unsigned int ImageID;
 unsigned int NextImageID;
 unsigned int FlashEntryAddr;
 unsigned int LoadAddr;
 unsigned int ImageSize;
 unsigned int ImageSizeToHash;
 unsigned int HashAlgorithmID;
 unsigned int Hash[8];
} IMAGE_INFO_3_1_0, *pIMAGE_INFO_3_1_0;
```

These structures are further described in Table 5 and chapter sections. In Table 5, all field values are 32-bit unsigned integers. The NTIM columns indicate the fields that are required (Y), reserved default value (R), or optional (O) for the specific Image Module type. Some of the fields depend on the Version field, and some fields depend on the values of other fields. The specific details about each field follow Table 5.

## 3.2 Version Information

The Version Information structure provides information about the Image Module and platform.

- Version – Current version of the Image Module. Boot ROM currently supports two versions:
  - Version 3.1.x (0x0003010x)
  - Version 3.2.x (0x0003020x)
- Identifier – ASCII encoded hexadecimal value 0x54494d48 (“TIMH”) identifier used to locate the Image Module.
- `Trusted` – Identifier to distinguish between trusted and non-trusted platforms. A value of 0 indicates a Non-Trusted Image Module.
- `IssueDate` – Date this module was created in hexidecmal form (MMDDYYYY), that is, 0x08042008 represents a date of August 4, 2008.
- `OEMUniqueID` – OEM-specific identifier. OEM can assign any preferred coding to the value of this field.

### 3.2.1 Flash Information

The Flash Information substructure identifies boot flash properties a reserved field has a value of 0xFFFFFFFF.

- `Reserved[5]`
- `BootFlashSign` – Signature that determines from which flash the platform boots. The upper three bytes contains an ASCII-encoded hexadecimal value of `XIP` for XIP or `NAN` for NAND, OneNAND, and SanDisk flash. The lower byte contains the platform fuse encoding that determines the flash device being used. The BootFlashSign values are noted in Table 6.

**Table 6: Boot Flash Sign Definitions**

| Platform Boot Device | Encoded HEX Value* | BootFlashSign Value in the NTIM* |
|---|---|---|
| XIP on SMEMC CS0 | XIP'03 | 0x5849_5003 |
| x16 NAND on NFC CS0 | NAN'04 | 0x4E41_4E04 |
| x8 NAND on NFC CS0 | NAN'06 | 0x4E41_4E06 |
| OneNAND and Flex OneNAND on SMEMC CS0 | NAN'02 | 0x4E41_4E02 |
| eSD or eMMC devices on MMC3 | MMC'08 | 0x4D4D_4308 |
| eSD or eMMC devices on MMC3 (alternate configuration) | MMC'09 | 0x4D4D_4309 |
| eSD or eMMC devices on MMC1 | MMC'0B | 0x4DAD_430B |
| SPI Flash | SPI'0A | 0x5350_490A |
| Note: the Boot ROM uses the least significant byte of the BootFlashSign to determine the boot device. The most significant three bytes in the BootFlashSign are for readability only. | | |

## 3.2.2 Image Module Sizing Information

- `NumImages` – Number of "`IMAGE INFORMATION`" substructures in the Image Module.
- `NumKeys` – Number of "`KEY INFORMATION`" substructures in the Image Module.
- `SizeOfReserved` – Total Size of the OEM Reserved Area; values can range from 0 to 4 KB minus (size of other Image Module information). The value must be equal to the size of the header and all packages (which includes the termination package). See the details in Section 3.2.4, OEM Reserved Area[SizeOfReserved]

## 3.2.3 Image Information Array

The Image Information Array is a substructure that contains information about each image loaded into the boot flash. The number of substructures is determined by the `NumImages` field above.

- `ImageID` – A unique identifier for the image. Several predefined ASCII hexadecimal values are defined by the Boot ROM. The `"OBMI"` identifier (0x4F424D49) must be present in the array for the Boot ROM to correctly boot the platform. Other identifiers can be determined by the OEM, but are limited to 32 bits in size.
- `NextImageID` – ASCII hexadecimal value or OEM defined value for the next image that should be loaded from flash memory. If there is no next image, then NextImageID has a value of 0xFFFFFFFF.
- `FlashEntryAddr` – Offset from the start of the boot flash pointed to by the `BootFlashSign` field.
- `LoadAddr` – Absolute address for the image, which can be a DDR memory, internal SRAM, or XIP flash address.
- `ImageSize` – Size of the image in bytes.
- `ImageSizeToHash` – Number of bytes of the image that are included in the hash below. For NTIM, the `ImageSizeToHash` field has a reserved value of 0x0.
- `HashAlgorithmID` – Hashing algorithm that is used; values are:
  - 160 for SHA-1

    - 256 for SHA-2
    - For NTIM, the `HashAlgorithmID` field has a reserved value of 0x0.
- `Hash[8]` – Array that holds the hash of the image. For NTIM, the `Hash` array has a reserved value of all 0x0.
- `PartitionNumber` – Valid for NTIM version V3.2.x only. Specifies the physical or logical device partition where the image is located.

### 3.2.4 OEM Reserved Area[SizeOfReserved]

- `OEM Reserved Area[SizeOfReserved]` – Array of integers to be used by the OEM for value-added features. See section Section 3.3, Reserved Area Format for specific format details.

## 3.3 Reserved Area Format

The Reserved Area is a dedicated space in the NTIM that allows an OEM to add data that is targeted for specific use without altering the predefined layout of the Image Module. The reserved area is of variable size, which is tabulated in the `SizeOfReserved` field under the "`FlashInformation`" structure.

The content of the Reserved Area can be formatted as the OEM chooses, but to be compatible with the Wireless Trusted Platform Tool Package set of tools requires a predefined format, which consists of the Reserved Area Header and the Reserved Area Packages, as described in the following sections.

### 3.3.1 Reserved Area Header

The Reserved Area Header component spans eight bytes. Its primary purpose is to indicate to the interpreter of the NTIM that this portion of the reserved area complies with the format defined by the Wireless Trusted Platform Tool Package. It also indicates the number of packages to follow. The structure for the Reserved Area Header is as follows:

```
WTP_RESERVED_AREA:
    unsigned int WTPTP_Reserved_Area_ID;
    unsigned int NumReservedPackages;
```

- `WTPTP_Reserved_Area_ID` – indicates to the interpreting software that the reserved area complies with the format defined by the Wireless Trusted Platform Tool Package. This value should be the ASCII-encoded hexadecimal value `0x4F505448`, which represents `OPTH` in ASCII.
- `NumReservedPackages` – The number of packages to follow.This number should account for the termination package. For example, if there are two user packages, this number should be 3; two for the user packages and one for the termination package.

### 3.3.2 Reserved Area Packages

The Reserved Area Packages are the building blocks of the reserved area. Each package consists of a package header to identify the content, size, and payload data.

```
WTP_RESERVED_AREA_HEADER:
    unsigned int Identifier;
    unsigned int Size;
```

- `Identifier` – The identifier that defines the type of the package.
- `Size` – The total size of the package: four bytes for the identifier, four bytes for the size, plus the number of bytes of information in the payload that follows.

An unlimited number of Reserved Area Packages may be present as long as the size of the NTIM does not exceed 4 KB. The final Reserved Area Package should be the Termination Package with the Identifier value of `0x5465726D`, which is ASCII-encoded hexadecimal for “Term”.

## 3.4 Predefined Packages

A number of packages are defined for use with the Wireless Trusted Platform Tool Package tools. These predefined packages and their associated predefined header identifiers are described in the following sections.

Refer to the Wireless Trusted Platform documentation for more details on available packages.

### 3.4.1 GPIO Package

The GPIO package lets users set any memory space or address space to a preferred value, thereby allowing GPIOs to be configured by the Boot ROM.The header ID for this package is ASCII-encoded hexadecimal value 0x4750494F, which represents GPIO in ASCII. The number of pairs to set is defined by NumGpios, as shown in the following code. The number of GPIO_DEF structs defined is consistent with NumGpios.

```
OPT_GPIO_SET:
    WTP_RESERVED_AREA_HEADER WRAH;
    unsigned int NumGpios;
    pGPIO_DEF GPIO;


GPIO_DEF:
    volatile int *Addr;
    unsigned int Value;
```

### 3.4.2 UART/USB Protocol Packages

The UART/USB Protocol Packages allow overriding default USB and UART connection settings. The identifier for this package is `0x55415254` (for UART) or `0x00555342` (for USB). The port may also select (as appropriate) one of the following IDs:

**Table 7: UART/USB Package Identifiers**

| Package | Identifier Hex Value |
|---|---|
| FFIDENTIFIER: | `0x00004646` |
| ALTIDENTIFIER: | `0x00414C54` |
| DIFFIDENTIFIER: | `0x44696666` |
| SEIDENTIFIER: | `0x00005345` |
| U2DIDENTIFIER: | `0x55534232` |
| CI2IDENTIFIER | `0x00434932` |

```
OPT_PROTOCOL_SET:
    WTP_RESERVED_AREA_HEADER WRAH;
    unsigned int Port;
    unsigned int Enabled;
```

### 3.4.3 DDR Packages

The following optional DDR packages queue the Boot ROM to set up DDR based on the supplied parameters or register values.

#### 3.4.3.1 Configure Memory Controller Control (CMCC Package)

The Package ID (PID) for this package is ASCII-encoded hexadecimal value `0x434d4343`, which represents CMCC in ASCII.The number of bytes in the package is next which is 8, plus 8 bytes for each KeyID/value pair (see Table 8). Reference the following code.

```
CMCCSpecList_T:
    unsigned long PID;
    int NumBytes;
    CMCCSpec_T CMCCSpecs[1];

CMCCSpec_T:
    unsigned long KeyId;
    unsigned long KeyValue;
```

**Table 8: CMCC KeyId / Value pairs**

| KeyId | Value |
|---|---|
| CMCC_CONFIG_ENA_ID | 0 = Do Not Initialize DDR<br>1 = Initialize DDR |
| CMCC_MEMTEST_ENA_ID | 0 = Do Not Test Memory<br>1 = Test Memory |
| CMCC_CONSUMER_ID | This is the Consumer ID.<br>It must equal 0x54425249 “TBRI” to indicate that the Boot ROM is to look for and use any of the various DDR packages that may be present in the NTIM. |

#### 3.4.3.2 DDRC (Custom) Package

The DDR Custom package allows setting of selected MCU registers directly.

The Package ID (PID) for this package is ASCII-encoded hexadecimal value `0x44445243`, which represents DDRC in ASCII.The number of bytes in the package is next which is 8, plus 8 bytes for each KeyID/value pair. Reference the following code.

```
DDRCSpecList_T:
    unsigned long PID;
    int NumBytes;
    DDRCSpec_T DDRCSpecs[1];

DDRCSpec_T:
    unsigned long KeyId;
    unsigned long KeyValue;
```

Memory Controller registers that are supported in this Boot ROM are listed in Table 9 and show a Register ID in the right-hand column as used in the Marvell Boot Utility to select a register. The offset column contains the register offset from the MCU base address. All 32 bits of the KeyValue are written to the specified register.

There are two pseudo-register IDs in the last two entries of the table that are commands used to implement a delay (in µsec) and a specified register read. Refer to Section 7 on Boot ROM DRAM Initialization Details for more information on these commands.

**Table 9: DDRC (Custom) Package Register / KeyID**

| ARMADA 16x Register Name | Offset | Register ID | Numerical Value ID |
|---|---|---|---|
| Refer to Figure 3: Example DDR Package within the NTIM Binary File in the *ARMADA 16x Memory Controller Configuration and Tuning Application Note* for detailed information. | | | |
| Revision | n/a | ASPEN_SDRREVREG_ID | 0 |
| Address Decode | n/a | ASPEN_SDRADCREG_ID | 1 |
| SDRAM_CONFIG_0 | 0x020 | ASPEN_SDRCFGREG0_ID | 2 |
| SDRAM_CONFIG_1 | 0x030 | ASPEN_SDRCFGREG1_ID | 3 |
| SDRAM_TIMING_1 | 0x050 | ASPEN_SDRTMGREG1_ID | 4 |
| SDRAM_TIMING_2 | 0x060 | ASPEN_SDRTMGREG2_ID | 5 |
| SDRAM_TIMING_3 | 0x190 | ASPEN_SDRTMGREG3_ID | 6 |
| SDRAM_TIMING_4 | 0x1c0 | ASPEN_SDRTMGREG4_ID | 7 |
| SDRAM_TIMING_5 | 0x650 | ASPEN_SDRTMGREG5_ID | 8 |
| SDRAM_CNTRL_1 | 0x080 | ASPEN_SDRCTLREG1_ID | 9 |
| SDRAM_CNTRL_2 | 0x090 | ASPEN_SDRCTLREG2_ID | A |
| SDRAM_CNTRL_3 | 0x0F0 | ASPEN_SDRCTLREG3_ID | B |
| SDRAM_CNTRL_4 | 0x1A0 | ASPEN_SDRCTLREG4_ID | C |
| SDRAM_CNTRL_5 | 0x280 | ASPEN_SDRCTLREG5_ID | D |
| SDRAM_CNTRL_6 | 0x760 | ASPEN_SDRCTLREG6_ID | E |
| SDRAM_CNTRL_7 | 0x770 | ASPEN_SDRCTLREG7_ID | F |
| SDRAM_CNTRL_13 | 0x7D0 | ASPEN_SDRCTLREG13_ID | 10 |
| SDRAM_CNTRL_14 | 0x7E0 | ASPEN_SDRCTLREG14_ID | 11 |
| MMAP0_0 | 0x100 | ASPEN_ADRMAPREG0_ID | 13 |
| MMAP0_1 | 0x110 | ASPEN_ADRMAPREG1_ID | 14 |
| USER_INITIATED_COMMAND0 | 0x120 | ASPEN_USRCMDREG0_ID | 15 |
| DRAM_STATUS | 0x01B0 | ASPEN_SDRSTAREG_ID | 16 |
| PHY_CNTRL_3 | 0x140 | ASPEN_PHYCTLREG3_ID | 17 |

**Table 9: DDRC (Custom) Package Register / KeyID (Continued)**

| ARMADA 16x Register Name | Offset | Register ID | Numerical Value ID |
|---|---|---|---|
| PHY_CNTRL_7 | 0x1d0 | ASPEN_PHYCTLREG7_ID | 18 |
| PHY_CNTRL_8 | 0x1e0 | ASPEN_PHYCTLREG8_ID | 19 |
| PHY_CNTRL_9 | 0x1f0 | ASPEN_PHYCTLREG9_ID | 1A |
| PHY_CNTRL_10 | 0x200 | ASPEN_PHYCTLREG10_ID | 1B |
| PHY_CNTRL_11 | 0x210 | ASPEN_PHYCTLREG11_ID | 1C |
| PHY_CNTRL_12 | 0x220 | ASPEN_PHYCTLREG12_ID | 1D |
| PHY_CNTRL_13 | 0x230 | ASPEN_PHYCTLREG13_ID | 1E |
| PHY_CNTRL_14 | 0x240 | ASPEN_PHYCTLREG14_ID | 1F |
| PHY_DLL_CTRL_1 | 0xE10 | ASPEN_DLLCTLREG1_ID | 20 |
| TEST_MODE0 | 0x4C0 | ASPEN_TSTMODREG0_ID | 21 |
| TEST_MODE1 | 0x4D0 | ASPEN_TSTMODREG1_ID | 22 |
| MCB_CNTRL_1 (MCB_ARB_WT1) | 0x510 | ASPEN_MCBCTLREG1_ID | 23 |
| MCB_CNTRL_2 (MCB_ARB_WT2) | 0x520 | ASPEN_MCBCTLREG2_ID | 24 |
| MCB_CNTRL_3 (MCB_ARB_WT3) | 0x530 | ASPEN_MCBCTLREG3_ID | 25 |
| MCB_CNTRL_4 | 0x540 | ASPEN_MCBCTLREG4_ID | 26 |
| PERF_COUNT_CNTRL_0 | 0x0F00 | ASPEN_PRFCTLREG0_ID | 27 |
| PERF_COUNT_CNTRL_1 | 0x0F10 | ASPEN_PRFCTLREG1_ID | 28 |
| PERF_COUNT_STAT | 0x0F20 | ASPEN_PRFSTAREG_ID | 29 |
| PERF_COUNT_SEL | 0x0F40 | ASPEN_PRFSELREG_ID | 2A |
| PERF_COUNTER_SEL | 0xF50 | ASPEN_PRFCNTREG_ID | 2B |
| SDRAM_TIMING | 0x0660 | ASPEN_SDRTMGREG6_ID | 2C |
| PHY_CNTRL_TEST | 0xE80 | ASPEN_PHYCTLREGTEST | 2D |
| OPERATION_DELAY | n/a | ASPEN_OPDELAY_ID | 2F |
| OPERATION_READ | n/a | ASPEN_OPREAD_ID | 30 |

## 3.4.4 USB Vendor Request Package

The USB Vendor Request package is included in the reserved data when a special package requested by the vendor is required. This structure is the first word of any trailing data. There is no restriction that the data has to be 32-bit aligned.

The ID for this package is `0X56524551`, which represents `VREQ` in ASCII.

```
USB_VENDOR_REQ:
   WTP_RESERVED_AREA_HEADER WRAH;
```

```
    unsigned int bmRequestType;
    unsigned int bRequest;
    unsigned int wValue;
    unsigned int wIndex;
    unsigned int wLength;
    unsigned int wData; // First word of the proceeding Data.
                         // Note, there may be more trailing data
```

### 3.4.5 Resume From Hibernate Package

The Resume Package ID is used to indicate to the Boot ROM or OEM Boot Module that the boot cycle may be quickened by skipping certain operations that would normally be done at the initial power-on boot. The package identifier used to indicate a resume package is RESUMEBLID (0x52736D32) from Table 10 below.

The NTIM Resume package is defined as follows:

typedef struct

```
{
 WTP_RESERVED_AREA_HEADERWRAH;
 OPT_TIM_RESUME_DDR_INFO TimResumeDDRInfo;
}OPT_TIM_RESUME_SET, *pOPT_TIM_RESUME_SET;

typedef struct
{
 UINT_T DDRResumeRecordAddr;
 void*DDRScratchAreaAddr;
 UINT_T DDRScratchAreaLength;
}OPT_TIM_RESUME_DDR_INFO, *pOPT_TIM_RESUME_DDR_INFO;
```

The DDRResumeRecordAddr points to a structure defined as follows:

```
typedef struct
{
 UINT_T ResumeAddr;
 UINT_T ResumeParam;
 UINT_T ResumeFlag;
}OPT_RESUME_DDR_INFO, *pOPT_RESUME_DDR_INFO;
```

Before entering Hibernate, software must populate an OPT_RESUME_DDR_INFO structure and place it in the SDRAM location pointed to by “DDRResumeRecordAddr” of the NTIM DDR Resume package. The ResumeFlag must be 0x55AA55AA if the structure is valid. High-level operating system code should leave in SDRAM memory a filled out OPT_RESUME_DDR_INFO structure. Upon resumption of Hibernate mode (and also on “cold boot”), the Boot ROM inspects the Resume package, ensures the flat is 0x55AA55AA, and resumes directly to Operating System code available at ResumeAddr.

ARMADA 16x B0 stepping supports resuming to OS code without having to load the OBM. This feature is called *QuickBoot*. ARMADA 16x A0 does not support QuickBoot. For QuickBoot to work, the Boot ROM must be able to read from the DDR device (i.e. to read the ResumeFlag). This means the Boot ROM needs to re-configure the ARMADA 16x DDR Controller before reading from the DDR device. Thus, the CMCC package in the NTIM must be as follows for QuickBoot to work in ARMADA 16x B0:

CMCC_CONFIG_ENA_ID: 0x00000001
CMCC_MEMTEST_ENA_ID: 0x00000000
CMCC_CONSUMER_ID: 0x54425249

If the CMCC_CONSUMER_ID is not 0x54425249 or if the CMCC_CONFIG_ENA_ID is not 1, then the Boot ROM will not be able to read from the DDR device. In that case, for ARMADA 16x B0, the NTIM needs to include a GPIO package which instructs the Boot ROM to take the DDR device out of Self-refresh. This GPIO package is not needed on ARMADA 16x A0 Boot ROM as it does not contain the QuickBoot feature. Since the Boot ROM cannot read from the DDR device in this case, the Boot ROM will not be able to resume to OS code; instead, it will load the OBM. The OBM will inspect the NTIM for a resume package and resumes to OS code. The GPIO package looks as follows:

0x4750494F; GPIO Package
0x00000014; number of bytes in this package.
0x00000001; number of pairs: 1
0xB0000120; ddr command register address
0x00000001; initialize ddr command

## 3.4.6 Summary of Predefined Package IDs

Table 10 summarizes the predefined package IDs as indicated in the header of each package.

**Table 10: Pre-defined Package IDs**

| Name | Hex Word Value |
|---|---|
| DDRID | 0x44447248 |
| AUTOBIND | 0X42494e44 |
| TERMINATORID | 0x5465726D |
| GPIOID | 0x4750494F |
| UARTID | 0x55415254 |
| USBID | 0x00555342 |
| RESUMEID | 0x5265736D |
| USBVENDORREQ | 0x56524551 |
| USB_DEVICE_DESCRIPTOR | 0x55534200 |
| USB_CONFIG_DESCRIPTOR | 0x55534201 |
| USB_INTERFACE_DESCRIPTOR | 0x55534202 |
| USB_LANGUAGE_STRING_DESCRIPTOR | 0x55534203 |
| USB_MANUFACTURER_STRING_DESCRIPTOR | 0x55534204 |
| USB_PRODUCT_STRING_DESCRIPTOR | 0x55534205 |
| USB_SERIAL_STRING_DESCRIPTOR | 0x55534206 |
| USB_INTERFACE_STRING_DESCRIPTOR | 0x55534207 |

**Table 10: Pre-defined Package IDs**

| Name | Hex Word Value |
|---|---|
| USB_DEFAULT_STRING_DESCRIPTOR | 0x55534208 |
| USB_ENDPOINT_DESCRIPTOR | 0x55534209 |
| RESUMEBLID | 0x52736D32 |

# 4 Boot ROM DRAM Initialization Details

## 4.1 Default Operation

There is no default DDR configuration performed by the Boot ROM. Any initialization of DDR requires a Configure Memory Controller Control (CMCC) package and a DDR Custom (DDRC) package to be in the reserved area of the NTIM.

## 4.2 Configuring DDR

These two packages are required to be in the NTIM extended reserved area to initialize DDR.

- CMCC (Configure Memory Controller Control)
- DDRC (DDR Custom)

Both of these packages are optional, and the Boot ROM searches for them in the order listed above. The CMCC package in the NTIM specifies whether or not the Boot ROM consumes the DDRC package.

The Boot ROM first looks in the reserved area of the TIM for a CMCC package. This package contains three parameters consisting of Id / value pairs:

- CMCC_CONFIG_ENA_ID
- CMCC_MEMTEST_ENA_ID
- CMCC_CONSUMER_ID

The default values for the above flags are zero.

If a CMCC package is found and CMCC_CONSUMER_ID has an associated value "TBRI" (Trusted Boot ROM ID in ASCII), then the boot ROM looks for the other two flags and enables DDR initialization if CMCC_CONFIG_ENA_ID is found and has a value of 1, or enables the memory test if CMCC_MEMTEST_ENA_ID is found and has a value of 1.

If the DDR config flag is 1, then the Boot ROM searches for a DDRC package in the reserved area of the TIM. This package can modify the default register values for the Memory Controller.

There is no DDR initialization for situations where:

- No CMCC package is present, or
- CMCC package is found but no CMCC_CONSUMER_ID is present, or
- CMCC_CONSUMER_ID is not "TBRI"

See Section 3.4.3, DDR Packages for detailed information on the CMCC and DDRC package formats.

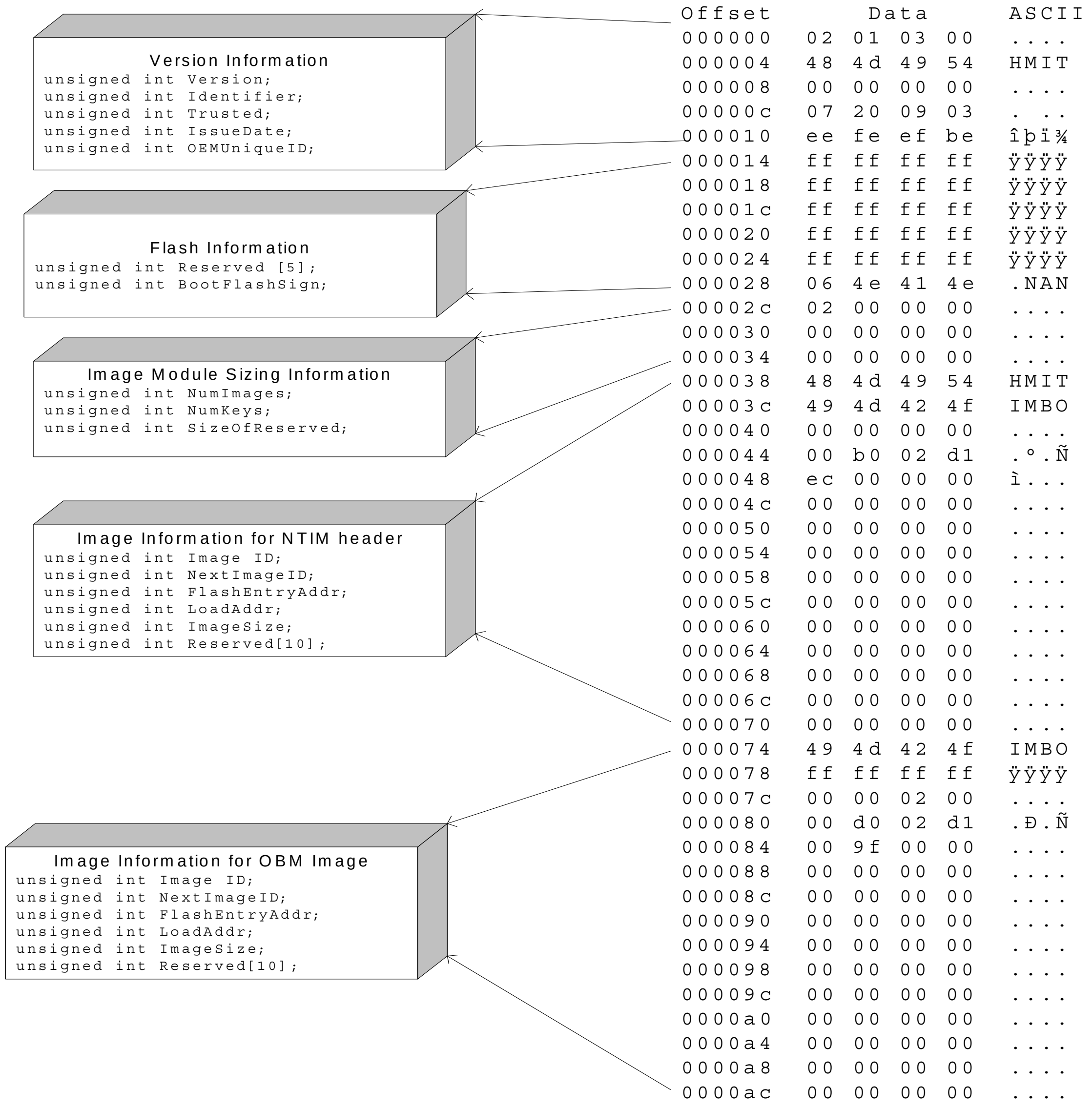
The memory test is optional once the DDR is initialized successfully. If the DDR Memory test is enabled, the Boot ROM tests the first 2 KB of SDRAM for read/write functionality. The memory test involves reading each memory location in the first 2048 bytes, inverting each bit, writing it back to memory and compared to what was written to determine if the memory is reliable.

If the memory initialization or memory test is unsuccessful, the memory is considered to be uninitialized. This situation has an impact on any image loads to SDRAM or transfer of control to any image that should have been loaded into SDRAM (the load/transfer of control will be unsuccessful).

# 5 Non-Trusted Image Module

The Non-Trusted Image Module (NTIM) is used for booting a non-trusted platform where security checking is not performed during the boot process. The minimum requirements for an NTIM are the Version Information, Flash Information, image module sizing parameters, and two image information structures. There are no maximum restrictions except that the size of the entire NTIM must be less than 8 KB. Figure 3 shows an example of a minimal NTIM that can be used to boot to an OEM boot module (first-level boot loader).

**Figure 3: Example of a Minimum Version 3.1.xx NTIM Header in Binary Format**



```
Version Information
unsigned int Version;
unsigned int Identifier;
unsigned int Trusted;
unsigned int IssueDate;
unsigned int OEMUniqueID;
```

```
Flash Information
unsigned int Reserved [5];
unsigned int BootFlashSign;
```

```
Image Module Sizing Information
unsigned int NumImages;
unsigned int NumKeys;
unsigned int SizeOfReserved;
```

```
Image Information for NTIM header
unsigned int Image ID;
unsigned int NextImageID;
unsigned int FlashEntryAddr;
unsigned int LoadAddr;
unsigned int ImageSize;
unsigned int Reserved[10];
```

```
Image Information for OBM Image
unsigned int Image ID;
unsigned int NextImageID;
unsigned int FlashEntryAddr;
unsigned int LoadAddr;
unsigned int ImageSize;
unsigned int Reserved[10];
```

```
Offset     Data          ASCII
000000   02 01 03 00     ....
000004   48 4d 49 54     HMIT
000008   00 00 00 00     ....
00000c   07 20 09 03     .  ..
000010   ee fe ef be     îþï¾
000014   ff ff ff ff     ÿÿÿÿ
000018   ff ff ff ff     ÿÿÿÿ
00001c   ff ff ff ff     ÿÿÿÿ
000020   ff ff ff ff     ÿÿÿÿ
000024   ff ff ff ff     ÿÿÿÿ
000028   06 4e 41 4e     .NAN
00002c   02 00 00 00     ....
000030   00 00 00 00     ....
000034   00 00 00 00     ....
000038   48 4d 49 54     HMIT
00003c   49 4d 42 4f     IMBO
000040   00 00 00 00     ....
000044   00 b0 02 d1     .°.Ñ
000048   ec 00 00 00     ì...
00004c   00 00 00 00     ....
000050   00 00 00 00     ....
000054   00 00 00 00     ....
000058   00 00 00 00     ....
00005c   00 00 00 00     ....
000060   00 00 00 00     ....
000064   00 00 00 00     ....
000068   00 00 00 00     ....
00006c   00 00 00 00     ....
000070   00 00 00 00     ....
000074   49 4d 42 4f     IMBO
000078   ff ff ff ff     ÿÿÿÿ
00007c   00 00 02 00     ....
000080   00 d0 02 d1     .Ð.Ñ
000084   00 9f 00 00     ....
000088   00 00 00 00     ....
00008c   00 00 00 00     ....
000090   00 00 00 00     ....
000094   00 00 00 00     ....
000098   00 00 00 00     ....
00009c   00 00 00 00     ....
0000a0   00 00 00 00     ....
0000a4   00 00 00 00     ....
0000a8   00 00 00 00     ....
0000ac   00 00 00 00     ....
```

# 6 Marvell® ARMADA 16x Applications Processor Boot ROM Operation Details

## 6.1 General Operation

Version 3.2.xx Boot ROM operates using a high-level state machine. The number of states varies depending on the Boot mode. Figure 4 shows the high-level state diadem used during a non-trusted boot. Table 11 provides a brief explanation of the function of each state.

After reset, the Boot ROM performs the essential initialization including:

- Reading the boot state configuration
- Programming the clocks
- GPIO settings
- Initializing the stack pointers
- Initializing heap pointers

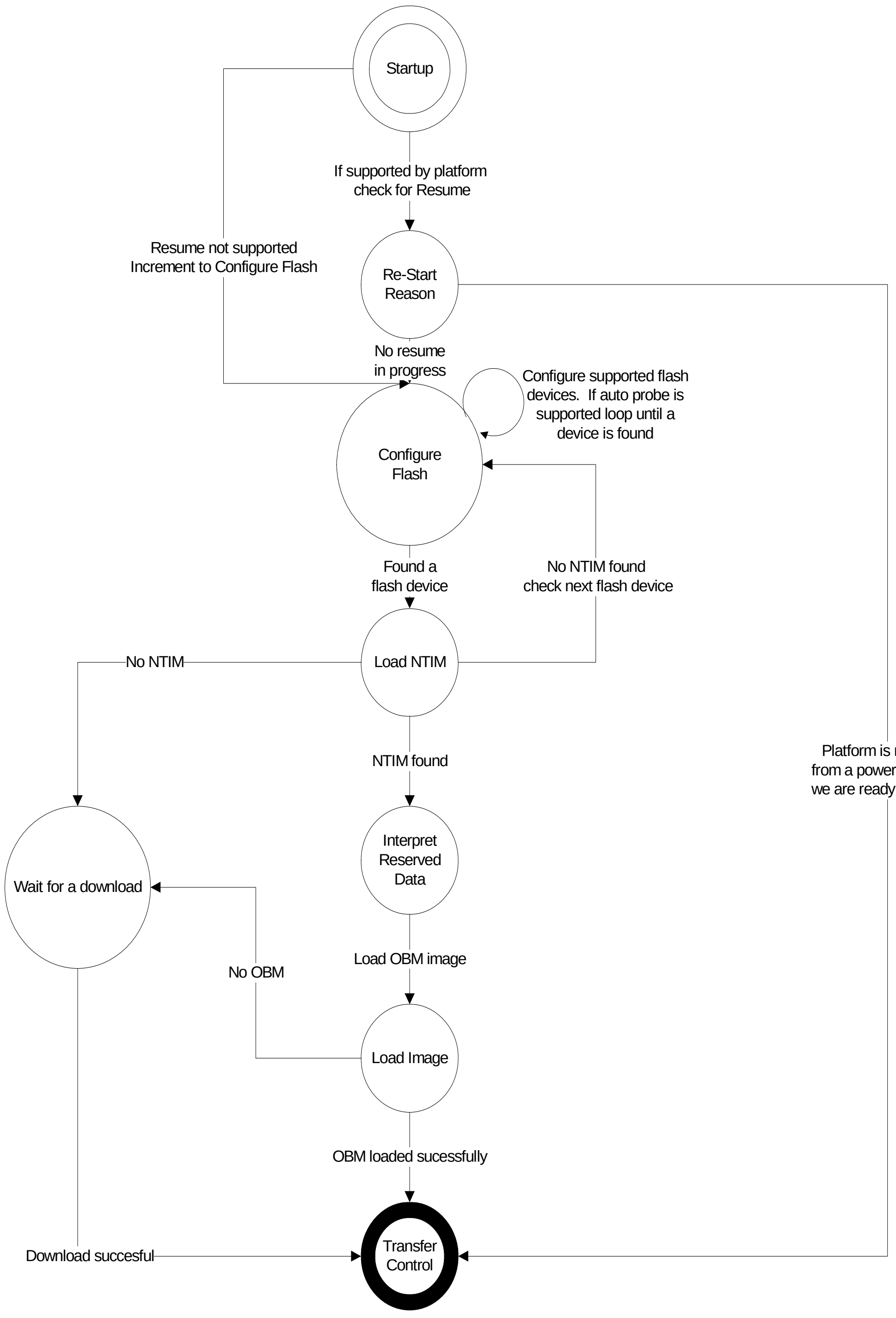
The ARMADA 16x Applications Processor is hard wired to jump to the Boot ROM after power-on, causing the core to execute instructions from the physical ROM space identified in Section 1.4 "ROM Location, Size, and Mapping" . The Boot ROM requires an NTIM header to boot to the next layer of software. This is true whether the next software image is loaded from a Flash device or over a port device, such as USB. Depending on the fuse configurations, the Boot ROM takes different actions as part of the boot process. Methods are in place to enable and disable certain features such as download capabilities over the ports.

The remainder of this section provides details of the operation based on Operational mode, Flash type, and download operation.

**Table 11: Description of States that the Boot ROM traverses**

| State | Description |
|---|---|
| STARTUP | Initial State when Boot ROM begins to execute and initializes the runtime environment. |
| RE-START REASON | State when Boot ROM check for the re-start reason. For ARMADA 16x only power on reset is required. |
| CONFIGUREFLASH | State where the Boot ROM configures Boot flash. |
| TIMLOAD | State where the Boot ROM loads NTIM from Boot flash. |
| RESERVEDDATA | State where Boot ROM analyzes optional reserved date in the NTIM to setup additional features. |
| IMAGELOAD | State where the Boot ROM loads the next boot image (OBM). |
| XFER | State right before Boot ROM hands control off to the OBM. |

Figure 4: Non-Trusted Boot State Diagram


Startup
If supported by platform check for Resume
Resume not supported Increment to Configure Flash
Re-Start Reason
No resume in progress
Configure supported flash devices. If auto probe is supported loop until a device is found
Configure Flash
Found a flash device
No NTIM found check next flash device
No NTIM
Load NTIM
NTIM found
Platform is resuming from a power mode, and we are ready to transfer
Wait for a download
Interpret Reserved Data
Load OBM image
No OBM
Load Image
OBM loaded sucessfully
Download succesful
Transfer Control

## 6.2 Flash Types Supported: NAND Flash

The Boot ROM for the Marvell® ARMADA 16x Applications Processor Family supports many different SLC NAND and MLC Nand devices such as:

- Large Block NAND x8 and x16
- Small Block NAND x8 and x16
- ONFI compliant NAND devices version 1.0

The Boot ROM supports booting from x8 or x16 NAND devices attached to Chip Select 0 of the processor NAND Flash Controller. Both large- and small-block devices, as well as ONFI 1.0 compliant devices are also supported. Contact a Marvell field representative for information about specific devices.

The image module (NTIM) should be located at offset 0x0 of any of the first 10 blocks of the NAND device. The Boot ROM searches for the “TIMH” identifier embedded in the version information of the image module (NTIM). If the structure is found, it is loaded into the internal SRAM of the system.

A NAND platform requires support for bad-block management, as well as error detection and correction. `ECC_EN` and `SPARE_EN` are enabled when programming NAND blocks using the NAND Flash controller. The Boot ROM makes use of the Marvell bad-block management scheme if the bad-block table is present. If the Marvell bad-block table is not present, the OEM Boot Module is limited in size as defined in Table 12.

**Table 12: OEM Boot Module Sizes Without Marvell Bad Block Management**

| Small Block NAND | Large Block NAND |
|---|---|
| Block 0 – 1 page (15.5 KB) | Block 0 -1 page (127 KB) |
| Block 0 – image module - 1 page (approximately 15 KB) | Block 0 – image module -1 page (approximately 126 KB) |

If the Marvell bad-block scheme is implemented, the OEM Boot Module size restriction does not exist and the OEM Boot Module can be any size. The size of the OEM Boot Module is determined from the image module as well as the starting location. The OEM Boot Module likely consumes contiguous blocks in the NAND device, (Blocks 1 through 3, for example). The image cannot be broken into non-contiguous blocks unless a block is relocated through the bad-block table. Refer to Section Section 6.6, Flash Management for more details on NAND bad-block management.

### 6.2.1 Boot ROM NAND Device Recognition

An algorithm to enable booting from different types of NAND devices works as follows:

1. Boot ROM issues the reset command 0xFF to the NAND device (a requirement of the ONFI standard). During the reset command, the NAND device toggles the Ready/Busy# when the reset command is issued. This mechanism determines if a NAND device is present. If Ready/Busy# is not toggled, then the Boot ROM does not attempt any additional commands to the NAND device.
2. Next, the Boot ROM issues several different READ ID commands to determine the device type. The first READ ID command is to check for the “ONFI” signature. If the ONFI signature is detected, the Boot ROM issues the Read Parameter Page command. The data from the Read Parameter Page command is then validated. If validation of the CRC passes, skip to Step 5 and configure the DFC controller for operation. If the CRC fails, continue with the next READ ID command in Step 3.
3. The Boot ROM now issues a 2-byte READ ID command to detect the manufacturer and device ID. Because some legacy small-block devices did not return device parameters, a lookup table is used to check for known small-block devices (see Table 13). If the device is a small-block device, skip to Step 5 and configure the DFC controller. If the device is not found in the look-up table, continue to Step 4 and the final READ ID command.

**Note**

The two bytes (manufacturer ID and device ID) are compared against the small block codes in Table 13. All Boot ROM versions support these devices.

- If the device is a small-block device, the Boot ROM configures the NAND controller for small-block operation with a 512-byte page size and 16 KB block size.

**Table 13: Small Block Devices**

| Manufacturer | Manufacturer Code | Device Codes |
|---|---|---|
| Samsung | 0xEC | 0x71, 0x78, 0x79, 0x72, 0x74, 0x36, 0x76, 0x46, 0x56, 0x35, 0x75, 0x45, 0x55, 0x33, 0x73, 0x43, 0x53, 0x39, 0xE6, 0x49, 0x59 |
| Toshiba | 0x98 | 0x46, 0x79, 0x75, 0x73, 0x72, 0xE6 |
| Hynix | 0xAD | 0x76, 0x56, 0x36, 0x46, 0x75, 0x55, 0x35, 0x45, 0x73, 0x53, 0x49 |
| ST Micro | 0x20 | 0x73, 0x35, 0x75, 0x45, 0x55, 0x76, 0x36, 0x46, 0x56, 0x79, 0x39, 0x49, 0x59 |
| | | |

4. The Boot ROM issues another READ ID command and retrieves 4 bytes of data from the device. The 4th byte of data from the device is used to interpret the device parameters and then configure the NFC. This example has a generic implementation where:
   - The Boot ROM uses the information returned in the 4th byte to determine the page and block size of the NAND device. Bits 1 and 0 for the page size and Bits 5 and 4 for the block size.
   - The Boot ROM does NOT use the manufacturer and device codes for large-block NAND configuration. The NFC is configured for large block based on the device parameters.
5. The Boot ROM then configures the command set for the appropriate NAND device (based on the above steps) and continues with normal Read operation.

**Note**

This operation was originally documented as the Boot ROM expecting 0x15 for x8 large block NAND and 0x55 for x16 large-block NAND devices. Although the entire 4th byte is read from the NAND device, the Boot ROM uses only bits 0, 1, 4, and 5, not the entire byte (all 8 bits) to configure the appropriate memory device.

The NAND Flash Controller (NFC) is set up for the appropriate NAND configuration as noted in Table 14. NDCR[DWIDTH_C] and NDCR[DWIDTH_M] are configured by the Boot ROM depending on the boot configuration SKU of the processor.

Table 14: NAND Flash Controller Initial Register Settings

| Register | Value for Small Block Operation | Value for Large Block Operation |
|---|---|---|
| NDCR[DWIDTH_C] and NDCR[DWIDTH_M] (bits27:26) initial settings are determined by the boot state fuses | 0xCC02_1FFF | 0xCD04_1FFF |
| Timing register 0 | 0x003F_3F3F | 0x003F_3F3F |
| Timing register 1 | 0x1FF0_C0FF | 0x1FF0_C0FF |

Table 15: NAND Command Set

| Command | Small Block Command Code | Large Block Command Code |
|---|---|---|
| Read | 0x0000_ | 0x3000 |
| Read Status | 0x0070 | 0x0070 |
| Read ID | 0x0090 address 0 | 0x0090 address 0 |
| Read ONFI ID | | 0x0090 address 0x20 |
| Read Parameter Page | | 0x00EC |

## 6.2.2 XIP Flash Support

The ARMADA 16x Applications Processor Boot ROM supports NOR Flash memory on the Data Flash interface (DFI) bus. Flash is supported through a command set, not through a particular JEDEC ID. Any device is supported, provided that the device complies with the commands as described in Table 16.

The processor natively can support AA/D muxed memories. Other memories may be connected and booted from but external latches are required. NOR-like NAND devices such as Samsung OneNAND are also supported via the DFI bus using the XIP data window for the device. The Boot ROM also has an integrated device driver for the Samsung OneNAND part, allowing use of the main memory array for boot images. Refer to Section 6.2 "Samsung OneNAND and FlexOneNAND" for more details.

Table 16: Flash Commands Supported by the Boot ROM

| Flash Command Name | Flash Command Data | Flash Type |
|---|---|---|
| Read Array | 0xFF | Intel StrataFlash® Wireless Memory (XIPA), Intel StrataFlash® Cellular Memory (M18) (XIPB) |
| Read Device Identifier | 0x90 | Intel StrataFlash® Wireless Memory (XIPA), Intel StrataFlash® Cellular Memory (M18) (XIPB) |
| Clear Status Register | 0x50 | Intel StrataFlash® Wireless Memory (XIPA), Intel StrataFlash® Cellular Memory (M18) (XIPB) |
| Word Program | 0x40 | Intel StrataFlash® Wireless Memory (XIPA) |

**Table 16: Flash Commands Supported by the Boot ROM (Continued)**

| | | |
|---|---|---|
| Word Program (Intel StrataFlash® Cellular Memory (M18)) | 0x41 | Intel StrataFlash® Cellular Memory (M18) (XIPB) |
| Unlock Block | 0x60/0xD0 | Intel StrataFlash® Wireless Memory (XIPA), Intel StrataFlash® Cellular Memory (M18) (XIPB) |
| **NOTE:** The command set supported by the Boot ROM is not specific to Intel StrataFlash®. Any NOR Flash device is supported if that device supports the same command sets as described in this table.<br><br>XIPA and XIPB are used for reference in other Boot ROM chapters. | | |

### 6.2.2.1 NOR Flash on Chip Select 0

The Boot ROM supports booting from an XIP device attached to Chip Select 0 of the processor Static Memory Controller. Several XIP devices are supported in AA/D muxed mode of operation; contact your local Marvell field engineering representative with questions about specific devices.

The image module (NTIM) should be located at offset `0x0` of the XIP device. For Chip Select 0, the XIP device is memory-mapped to `0x8000_0000`. The Boot ROM searches for the `"TIMH"` identifier embedded in the version information of the image module (NTIM). If the structure is found, it is loaded into the internal SRAM of the system. From this point, the Boot ROM uses the image module to load the OEM Boot Module.

The OEM Boot Module is described by the image information contained in the “IMAGE INFORMATION” array. It is identified by the `"OBMI"` image identifier, which is a required identifier for proper use with the Trusted Boot ROM. Using the information that describes the OEM Boot Module, the image is loaded from the offset pointed to by `FlashEntryAddr` to the location pointed to by `LoadAddr`. The `ImageSize` entry determines the number of bytes that are loaded.

### 6.2.2.2 Managed NAND on Chip Select 0

Many managed NAND hybrid devices have a NOR interface with an XIP area intended for booting. The Boot ROM can boot from these devices using the XIP area, provided that both the NTIM and first-level boot loader can be sized to fit into the XIP area of the device. On some devices, the XIP area is 1 KB in size, making the trusted boot operation impractical. For these devices, non- trusted operation is possible with a minimal NTIM and IPL initial program loader (IPL) that can read from the main memory array.

The IPL module is described by the image information contained in the “IMAGE INFORMATION” array. It is identified by the `"OBMI"` image identifier, which is a required identifier for proper use with the Boot ROM. Using the information that describes the OEM Boot Module, the image is loaded from the Flash offset pointed to by `FlashEntryAddr` to the location pointed to by `LoadAddr`. The number of bytes loaded is determined by the `ImageSize` entry. Here, the IPL module, acting as the OEM Boot Module, then loads the next software image from the main NAND array of the device.

This boot procedure is somewhat restrictive, and the NTIM and IPL module sizes must be calculated carefully to ensure they both fit into the XIP area of the device.

### 6.2.2.3 Samsung OneNAND and FlexOneNAND

When the Boot ROM is defined as having support for OneNAND and FlexOneNAND memory devices, this means that the Boot ROM has an embedded device driver that can access the main memory array of the device. Specific requirements are needed for accessing these memories and also the location of certain boot images.

The Boot ROM can program the Static Memory Controller and can directly access the correct memory locations to execute the OBM images and NTIM headers.

Even though the Boot ROM may have integrated drivers for these memories, it does not replace the necessity of OS-level Flash drivers. These drivers must be implemented in the OS.

The Samsung OneNAND and FlexOneNAND memory connects to the processor using the DFI bus with nCS0 as the boot chip select.

The NTIM can be placed at the start of any of the first ten blocks, which should be specified in the header itself where the header binary must also reside. The OBM location is also defined in the header and must be loaded at the indicated location.

For all subsequent Boot ROM versions, only the Manufacturer ID of 0xEC must be read. The Boot ROM then reads the Device and Generation IDs to determine whether the device is a OneNAND or FlexOneNAND device. The Boot ROM also calculates the density of the OneNAND without having them predefined as with previous generations.

### 6.2.3 SD/MMC Devices

The Boot ROM supports booting from certain SD and MMC protocol-based Flash devices attached to the MMC1 or MMC3 ports. Refer to Section 7.1 for Multi-Function Pins (MFPs) used for each port. The Boot ROM driver supports reading, writing, erasing and switching partitions. Boot ROM supports MMC specification V4.2 and V 4.3, and eSD specification V2.1.

For MMC-based devices, the Boot ROM searches for a NTIM at address 0x0 of Partition 1 (MMC V4.3 or later) or Address 0x0 (if no hardware partitions are supported by the MMC device; MMC V4.2)**.** For eSD-based devices, the Boot ROM searches for the NTIM at address 0x0 of the user partition. B0 Boot ROM checks up to 10 blocks (assumes 512 bytes/block) for the NTIM. Thus, the NTIM can be placed at offsets 0x000, 0x200, 0x400, 0x600, 0x800, 0xA00, 0xC00, 0xE00, 0x1000, or 0x1200. ARMADA 16x A0 Boot ROM checks offset 0x0 only.The OBM can then be found in a user choice partition and address as indicated in the NTIM.

The following SD/MMC devices are currently supported by the Boot ROM:

Samsung MoviNAND: KMAFN0000M, KMAKE0000M, KMBLE0000M, KMCME0000M

Sandisk iNAND based MCP: MCP211, MCP212, MCP214

### 6.2.4 SPI Flash Devices

The ARMADA 16x Applications Processor Boot ROM supports booting from many SPI devices attached to the SSP port. Booting from SPI flash is attempted as part of the autoboot process when NAND and eMMC devices are not bootable, or in the event of a boot failure.

The supported SPI devices can be found in Table 17:

**Table 17: Supported SPI Devices**

| Device Name | Manufacturer ID | Device ID 1 | Device ID 2 |
|---|---|---|---|
| Numonyx M25P40 | 0x20 | 0x20 | 0x13 |
| ST Micro M25P32 | 0x20 | 0x20 | 0x16 |
| Spansion S25FL016A | 0x01 | 0x02 | 0x14 |
| Atmel AT25FS040 | 0x1F | 0x66 | 0x04 |
| Atmel AT40DB642D | 0x1F | 0x28 | 0x00 |

#### 6.2.4.1 SPI Command Sets

**Table 18: SPI Command Sets**

| Device | Read | Read Status | Write Enable | Page Program | Program (stage 2) | Sector Erase |
|---|---|---|---|---|---|---|
| Numonyx M25P40 | 03h | 05h | 06h | 02h | N/A | D8h |
| ST Micro M25P32 | 03h | 05h | 06h | 02h | N/A | D8h |
| Spansion S25FL016A | 03h | 05h | 06h | 02h | N/A | D8h |
| Atmel AT25FS040 | 03h | 05h | 06h | 02h | N/A | D8h |
| Atmel AT40DB642D | 03h | D7h | N/A | 84h | 88h | 50h |

#### 6.2.4.2 SPI Device Detection

The Boot ROM first issues a Release from Power Down command (ABh) to the device, followed by a Read JEDEC ID command (9Fh). The ID returned is compared to those in Table 17, to determine which device is connected and which command set to use.

If an unknown device is detected, the Boot ROM assumes a 256-byte page size, and uses the command (03h) to read from the device.

## 6.3 Preprogrammed Flash Requirements

Preprogramming of Flash memory is supported for large-volume manufacturing and requires the following when using an image module (NTIM):

- Program the NTIM to the correct offset; contact your local Marvell field engineering representative for more information.
- Program the OEM Boot Module and any other image described in the image module to the address indicated by `FlashEntryAddr` of the image module.

The Boot ROM examines Flash memory and searches for the NTIM. After it is found, the image is loaded to the address specified in the NTIM.

## 6.4 Download Capability

The Boot ROM enables the USB 2.0 OTG port in device mode shortly after a power-on reset. An image can be downloaded over one of these ports by using the Communication protocol as described in Chapter 9, “Communication Protocol”.

### 6.4.1 USB Port

The default USB configuration is the USB 2.0 OTG port. This port is configured after a power-on reset and is run in Interrupt mode. Contact your local Marvell field engineering representative for more information about the communication protocol used by the target and host.

### 6.4.2 Error Reporting Capability

The V3.2.XX Boot ROM can tabulate in real-time a collection of information called the “Boot ROM Status Structure” (BRSS). The BRSS serves the following purposes:

1. Place holder for collecting information during run-time for error codes, traversed states and pointers to other relevant structures that are used during the boot process. This information

serves to debug potential problems by impaction or output through one of the enabled ports during debug with or without JTAG enablement.

2. Place holder for information passed up to higher levels of software, particularly aimed for the OBM. This information can be used to reduce boot-up times by not requiring duplicate work such as loading the NTIM, initializing DDR or even probing flash.

The location of the BRSS will be constant with respect to the start address of volatile memory (refer to section 0.1.2.) It is always the base address of ISRAM/L2 + 0x40 and has 192 bytes reserved for it. Table 19 depicts the layout of the BRSS.

**Table 19: Boot ROM Status Structure (BRSS)**

| Offset | Description of Content |
|---|---|
| 0x0 | Boot ROM Version |
| 0x4 | Boot ROM Build Date |
| 0x8 | Platform Type |
| 0xC | Platform SubType |
| 0x10 | Current State of the Boot ROM. (refer to tables 7 and 8) |
| 0x14 | The previous state of the Boot ROM. (refer to tables 7 and 8) |
| 0x18 | 32 bits of fuse relevant tabulation used by the Boot ROM |
| 0x1C | 32 bit error code. Should be used in conjunction with the state fields of the Boot ROM to pinpoint exact failure location. |
| 0x20 | Transfer Address of the next image. Note that on normal initialized boot, Transfer Address will contain an appropriate value only after IMAGELOAD state. |
| 0x24 | Probe Flash Index is used to determine at real-time inspection what flash has been configured in the probe mechanism. Probe Flash Index will change during run-time on platforms that require flash probing. |
| 0x28 | Pointer to the where the constant part of the NTIM (CTIM) that has been loaded in volatile memory. |
| 0x30 | Pointer to the first IMAGE_INFO structure of the NTIM in volatile memory, there may be multiple in an array fashion. |
| 0x38 | Pointer to the beginning of Reserved Area of the NTIM in volatile memory. |
| 0x48 | Flag indicates if the first of the security engines have been initialized. Platform dependent. |
| 0x4C | Flag indicates if the second of the security engines have been initialized. Platform dependent. |

## 6.5 Resume From Hibernate

The ARMADA 16x Applications Processor starting with the B0 stepping supports the capability to resume directly to a section of software left behind in DDR in instances where the DDR has been left in state-retentive mode. This feature can save critical boot time by aiding the Boot ROM to skip several time-intensive steps such as re-loading the OEM boot module from flash. To enter this mode of operation, several criteria must be met:

1. The operation must not be an initial power-on reset where critical boot images have not previously been loaded to DDR since the last power-on reset “cold boot”.
2. A NTIM with a resume package RESUMEBLID (0x52736D32) must be present in the resume area.
3. DDR must be initialized using the relevant DDR NTIM packages via the Boot ROM.

When the Boot ROM begins to execute, initially it has no indication if it is executing due to a power on reset, hardware reset, or Hibernate mode. The Boot ROM loads the NTIM, and looks for a package in the reserved area with the RESUMEBLID identifier. If the Boot ROM finds such a package it then examines the Resume Address field to find another copy the OPT_RESUME_DDR_INFO structure in the DDR. This structure in turn has the true address where the Boot ROM can directly jump without loading any additional images or further processing. The package in the DDR must also have a flag set to the value “0x55AA55AA” to indicate that the jump to image can handle a direct-resume process. However, before the Boot ROM transfers control, it inverts this flag and writes it out to the DDR at the same location (see ResumeFlag field in the OPT_RESUME_DDR_INFO package.)

**Note**

Software capable of handling the resume process after the Boot ROM must write out proper contents of the OPT_RESUME_DDR_INFO structure before going to Hibernate mode.

## 6.6 Flash Management

The Boot ROM supports two different Flash management schemes to accommodate the various restrictions on many Flash devices. The two schemes are detailed further in this section.

The two supported schemes for managing Flash are:

- Legacy Bad Block Management (BBM) for NAND and OneNAND devices
- Marvell Flash Management with Partitioning support (all supported Flash devices)

The Boot ROM determines which scheme is being used based on the version of the NTIM found in Flash. A NTIM with a version of 3.1.x (0x00030100) or lower forces the Boot ROM to use the backward-compatible mode. An NTIM with a version of 3.2.x (0x00030200) forces the Boot ROM to use the newer Marvell Flash Management scheme. See Chapter 3, “Image Modules” for information on NTIM versions. The Boot ROM does not create a BBT or update an existing BBT. The Boot ROM only reads a BBT from Block 0 to determine whether a block is good or bad and needs to be read from somewhere else.

### 6.6.1 Legacy Bad-Block Management

The bad-block management scheme consists of two components: the bad-block table and the pool of reserved relocatable blocks. The relocation table contains a list of bad blocks and their relocations. The table is stored in Block 0 of the Flash (see Section 6.6.1.1, Bad-Block Table (BBT) Location). The pool of reserved blocks is located at the end of the Flash device and consists of 2% of the device.

This management scheme supports NAND and OneNAND bad-block detection and relocation. Because of the physical limitations of NAND and OneNAND devices, any block (besides Block 0) may be bad (meaning, it cannot be written to reliably). The relocation table contains a list of these bad blocks and their relocations.

XIP, Managed NAND, and MMC devices can still be used as boot devices under this scheme; however, there is no support for partitioning and bad-block management is irrelevant.

### 6.6.1.1 Bad-Block Table (BBT) Location

The initial BBT typically is written to the page starting at offset 0x1000. The bad-block table requires exactly one page per block. If the bad-block table has to change at run time, each page is treated like a new slot for additional tables. Rather than erasing and creating a new table over the initial page each time, a new table is simply written to the next page after the current table, which reduces wear and tear on the block by reducing the number of erase cycles. This process of updating to the next page continues to the end of Block 0 and reduces the need to erase Block 0.

**Note**

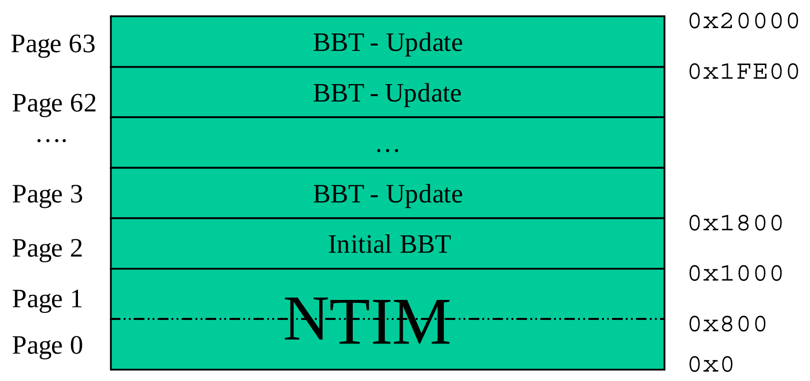
To support backward compatibility with older tools sets, including XDB and the Wireless Trusted Platform Tools Packages, the Boot ROM checks the last page of Block 0 for a BBT. It then uses the table with the most entries as the default BBT.

The maximum number of bad-block tables that can be written, before an erase is required, is defined by the size of the Flash device. To calculate this number, subtract 4 KB from the block size (this is the reserved size for the NTIM), and divide by the page size. (Example: 128 KB block with 2 KB pages. (128 - 4) / 2 = 62 pages). See Figure 5.

**Note**

The Boot ROM uses a binary search algorithm to find the most current table between the page starting at address 0x1000 (page 2 in Figure 5) and the last page in Block 0 (page 63 in Block 0). Figure 5 is an example of a typical Block 0 layout at run time indicating how the slot-based mechanism works.

**Figure 5: Block 0 Layout on a Micron MT29F2G08* with 128 KB Block Sizes and 2 KB Pages**



### 6.6.1.2 Bad Block Table Definition

Each bad-block table has a layout in Flash, as defined with the following structure:

```
Typedef struct S_Reloc
{
    USHORT Header;
    USHORT NumReloc;
    Rel_T  Relo[NAND_RELOC_MAX];
}Reloc_T;
```

The header is a fixed value of `0x524E` to identify the presence of a bad-block table; that is, if the header is valid as defined above, the initial block scan has been completed. Otherwise, the block scan has not been completed. The `NumReloc` parameter identifies the number of blocks that has currently been relocated and is followed by up to 127 relocation pairs.

```
Typedef struct S_Rel
{
    USHORT From;
    USHORT To;
}Rel_T;
```

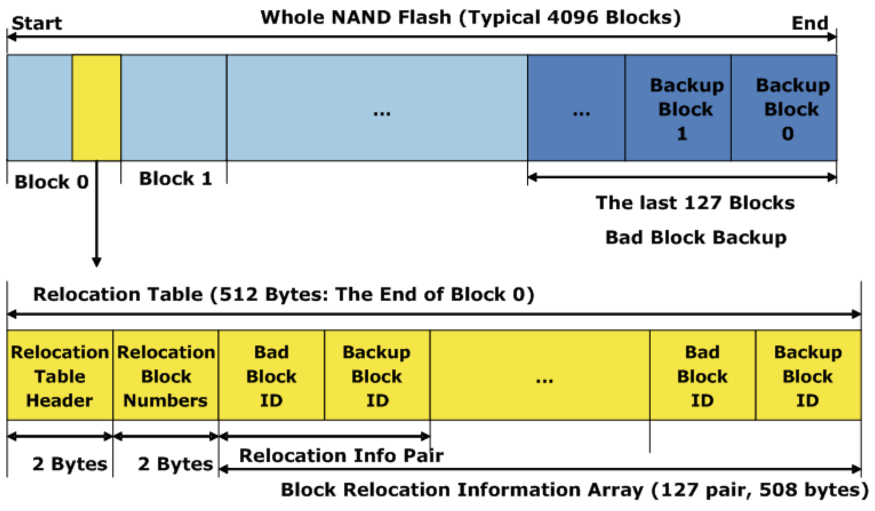
```
Const ULONG NAND_RELOC_MAX  = 127;
```

Each “From” entry identifies the block that has been relocated and the entry “To” identifies the relocated block number.

### 6.6.1.3 Bad Block Relocation Area

The last two percent of the blocks of the device are reserved for bad-block relocations. The first block that is relocated goes to the very last block of the device; the second block relocated goes to the second to the last block of the device, and so forth. This process effectively allows relocated blocks to grow from the highest address down. A block in the relocation pool itself may be relocated, so use caution when relocating to skip over these blocks. Figure 6 presents a typical Flash part layout and a relocation table layout to tie the concepts together.

**Figure 6: Example of Bad Block Table NAND Flash Mapping in Use — Small Block NAND Flash Type: Samsung K9K1216Q0C* (Device ID = 0x46)**



### 6.6.2 Marvell Flash Management with Partitioning Support

Marvell's current processor technology offers a flexible boot implementation allowing boot from NAND, NOR, and hybrid Flash devices. This variety of devices offers different features and implements differing technology. NAND and hybrid devices are moving from the current Single-Level Cell (SLC) technology towards the Multi-Level Cell (MLC) technology where multiple bits are stored per transistor. Newer devices, such as eMMC and eSD (embedded MMC and embedded SD), Flex OneNAND™, and mDoc devices also have partitioning capability at the hardware level. All of these features are making Flash management more complex for software developers.

The goal of the Marvell Flash management method is to align all of the software developed at Marvell with a common Flash management method for the entire software stack. This flexible method takes into consideration the varying features of both hardware and operating system software. The design provides a robust boot capability in the event of Flash device failures at the block level. The design also provides hooks for features such as Firmware-Over-the-Air (FOTA) updates and OEM customization.

The following NAND features have caused requirements to change:

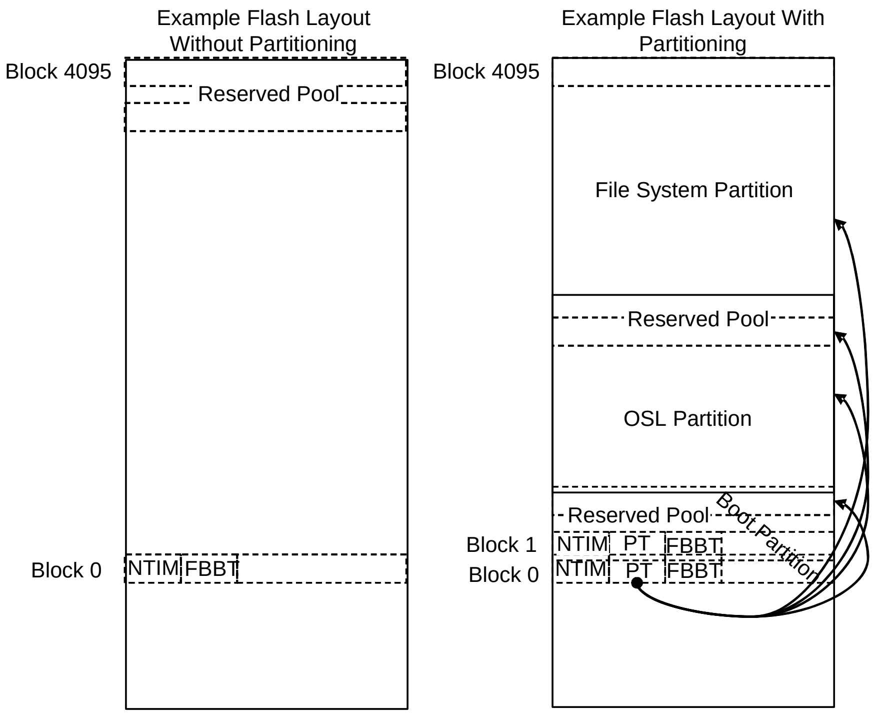
- Move to MLC technology. MLC technology has new restrictions for proper operation and different wear characteristics.
  - Linear write ordering requirement – All pages must be written in order within a block from the least significant page (LSP) to the most significant page (MSP).
  - Program/erase cycles are reduced to 10 K before the first failure is possible.
  - ECC requirements have increased dramatically.
  - "Read Disturb" phenomenon can occur when a Read to a page disturbs the data on an adjacent page, within a block.
  - Multiple Writes to an erased page are no longer allowed. The practice of writing to the data area and then later writing to the spare area is prohibited.
- Capability to implement hardware-based partitions. Many of the Flash devices entering the market have capabilities to create custom partitions or come partitioned from the factory. These include Flex-OneNAND™, mDoc, and eMMC/SD devices.

To handle these new requirements, the current implementation for bad-block management requires some changes. The management method incorporates several new structures and concepts to the overall Flash management infrastructure. The updates include:

- A backup boot block located in the next available block of the Flash device, normally Block 1.
- A partition table has been added to the boot block that defines the partitions within the device and their usage. Partitions can be either logical or hardware based. Their implementation is optional and the system still boots if no partition table is present.
- The bad-block table structure is updated to accommodate the new Flash management infrastructure.
- A factory bad-block table is located in Block 0 that stores only the factory marked bad blocks of a device.
- The runtime bad-block table resides in a separate block and is optional for each partition. The runtime bad-block table should be cumulative and incorporate the factory bad-block table information, as well as runtime bad-block information.
- Each defined partition has its own associated reserved pool and mechanism for implementing relocation to allow file system partitions to use the full block count of a partition for wear leveling without affecting relocated blocks.

Figure 7 shows the previous Flash bad-block management and the new Flash management method.

**Figure 7: Layout Change**



As an enhancement to the boot process, Marvell suggests passing information to the next level of software to optimize/reduce access to the Flash devices. In particular, the Non-Trusted Image Module (NTIM), Partition Table, and the Factory Bad-Block Table can remain in memory and a structure of pointers can be passed to the next layer of software. The location of the structure is implementation dependent.

**Example 1: Proposed Structure for Passing Boot Information**

```
struct {
    __int32* pNTIM;
    __int32* pPT;
    __int32* pFBBT;
    __int32 Reserved
}
```

The Marvell Flash management method is intended for use with all Flash types to provide for a consistent boot implementation independent of any operating system. The data found in the boot partition Block 0 is considered the master copy of the boot data for the device. All devices have a boot partition, whether it is logical or physical. This partition is designed for boot purposes and is used at a minimum by the Boot ROM and/or OEM Boot Module (OBM) firmware. Three important structures reside in the boot partition: the NTIM, the Factor Bad-Block Table (FBBT) for NAND devices, and the master partition table. These structures are described in detail in later sections.

Partitioning and bad-block management are considered optional by the embedded Boot ROM and the system can still boot without these features. However, there are side effects to device operation without them; software designers: consult vendor specifications to understand the technology being used. Also, the features implemented can vary according to the technology type being supported on the platform. NOR devices do not require a bad-block table, but can make use of partitioning. The Marvell Flash management method provides enough flexibility to boot using a wide variety of Flash devices.

### 6.6.2.1 Important Structures

Several important structures are involved in the Marvell Flash management method: the NTIM, the bad-block table(s), and the partition table. Only the NTIM structure is required for proper boot operation and the implementation of the bad-block table and partition table depends on the Flash device being used and OEM requirements.

In the absence of a partition table, the entire device is viewed as one logical boot partition and is expected to follow the format defined in section Section 6.6.2.3, Boot Partition and Boot Process. For NAND devices, the absence of a bad-block table forces a best effort type of boot, which means that the block defined in the NTIM is loaded and if an error occurs, the boot process may halt. Carefully consider such choices for each design.

The full NTIM structure is defined in the Boot ROM and Marvell Wireless Trusted Platform Tool Package (WTPTP) specifications. Some necessary modifications are being made to the Image section of the NTIM header to support partitioning. As a result of the changes, the version number for the NTIM is also incremented to version 3.2.0 (0x00030200). Within the NTIM, the following structures are modified:

- The Image Information structure (`IMAGE_INFO`) is modified to include a partition field.

**Example 2: Image Information Structure Changes in the NTIM**

```
struct {
unsigned int   ImageID;
unsigned int   NextImageID;
unsigned int   FlashEntryAddress;
unsigned int   LoadAddress;
unsigned int   ImageSize;
unsigned int   ImageSizetoHash;
unsigned int   HashAlgorithm;
unsigned int   Hash[8];
unsigned int   PartitionNumber;
}
```

The bad-block table structure has been updated to provide more relevant information. Two bad-block tables can now be defined: a factory bad-block table and a runtime bad-block table. The bad-block tables can exist with partitioning information or in the absence of partitioning information. This situation has the effect of requiring fields that may not always be used. These fields are treated as reserved when not in use. Example 3 captures the new factory bad-block table format. Example 4 captures the new runtime bad-block format.

**Example 3: Factory Bad Block Table Structure**

```
struct {
unsigned int   Identifier;
unsigned int   Version;
unsigned int   Type;
unsigned int   Reserved0;
unsigned int   Reserved1;
unsigned int   NumberofFactoryBadBlocks
unsigned int   RunTimeBBTLocationBootPartition;
unsigned int   Reserved2;
unsigned int   Reserved3;
unsigned int   Reserved4;
unsigned int   FactoryBadBlocks[NumberofFactoryBadBlocks];
}
```

The factory bad-block table is used to maintain the factory marked bad-block information on a NAND device. This table MUST be maintained for the life of the device as it is used as a reference in the event that errant software erases the factory information located in the spare area of the bad block. The fields used are as follows (also see Example 4):

- `Identifier` – 32-bit ASCII encoded hex identifier “MBBT” (0x4D424254). This identifier is used by the Boot ROM and must be set to “MBBT” for the Boot ROM to load and use the information in the table.
- `Version` – 32-bit version identifier currently set to 0x31303031. This field is used for tracking purposes and is not currently required by the Boot ROM.
- `Type` – 32-bit ASCII encoded hex identifier “Fact” (0x46616374). This identifier is used by the Boot ROM and must be set to “Fact” for the Boot ROM to recognize this table as the factory bad-block table.
- `NumberofFactoryBadBlocks` – Number of blocks identified as bad by the manufacturer.
- `RunTimeBBTLocationBootPartition` – Address offset from the base of the boot partition to the location of the runtime bad-block table used for the boot partition.
- `FactoryBadBlocks []` – Array of block numbers identified as bad from the manufacturer.

### Example 4: Runtime Bad Block Table Structure

```
struct {
unsigned int   Identifier;
unsigned int   Version;
unsigned int   Type;
unsigned int   Reserved;
unsigned int   PartitionID;
unsigned int   NumberofRleocationPairs;
unsigned int   Reserved;
unsigned int   Reserved;
unsigned int   BackupRuntimeBBTLocation;
unsigned int   Reserved;
unsigned int   RelocationPairs[NumberofRelocationPairs];
}
```

The runtime bad-block table is optional, but strongly suggested. At boot time, if the Boot ROM does not find the runtime bad-block table, it performs a best-effort boot. This means that it will read the first-level boot loader (OBM) according to information in the NTIM header and transfer control if no errors are encountered. For NAND devices, if an ECC error is encountered the boot process would be terminated. One option is to put the first-level boot loader into the first block of the NAND device and lock the block as a read-only block, which would minimize the need for having a runtime bad-block table for use by the Boot ROM.

When the runtime bad-block table is present, more flexibility is provided for Flash memory layout and size of the first-level boot loader. The fields in the runtime bad-block table are used to locate the images in the event a relocation operation occurred. The fields in the runtime bad-block table are used as follows:

- `Identifier` – The 32-bit ASCII encoded hex identifier “MBBT” (0x4D424254). This identifier is used by the Boot ROM and must be set to “MBBT” for the Boot ROM to load and use the information in the table.
- `Version` – The 32-bit version identifier currently set to 0x31303031. This field is used for tracking purposes and is not currently required by the Boot ROM.
- `Type` – The 32-bit ASCII encoded hex identifier “Runt” (0x52756E74). This identifier is used by the Boot ROM and must be set to “Runt” for the Boot ROM to recognize this table as the runtime bad-block table.
- `PartitionID` – Partition number in which this runtime bad-block table resides. Used as a check when partitioning is enabled.
- `NumberofRelocationPairs` – Number of blocks that have been relocated.
- `BackupRunTimeBBTLocation` – Address offset from the base of the partition to the location of the backup runtime bad-block table; used in the event of a runtime failure of the primary NAND block. This field is optional.
- `RelocationPairs[ ]` – Array of block number pairs with the [`FROM`, `TO`] format. The algorithm for creating relocation pairs is presented below.

A block is relocated when it is determined to have gone bad. In general, when a block has gone bad, it is defined by the manufacturer. Typically, it happens when an Erase or program failure occurs to the block. In this case, the block can be relocated to a reserved area of good blocks. The general format is to list the bad block first (the `FROM` block) and the good block in the reserved pool last (the `TO` block).

There are several special cases that must also be considered. The Marvell Flash management method does not chain relocated pairs. If Block 1 is relocated to Block 20, an entry of [`1`,`20`] is placed into the RBBT. At some later time, if Block 20 goes bad, the software does not create an entry of [`20`,`19`]. Because Block 20 was already used in a `TO` field, an entry is written into the table as [`20`, `-1`]. This means that Block 20 is not usable and cannot be relocated. The original entry [`1`, `20`] would be replaced with [`1`,`19`], assuming Block 19 is a good block in the reserved pool. This maintains one entry per relocated block.

The relocation of blocks may not be necessary for all partitions. If block-based wear leveling is incorporated into the NAND management scheme, then the use of a reserved pool and relocation may not be necessary. Reserved pool implementation is implementation defined and must be reviewed during the implementation phase. The Marvell Flash Management Method is flexible enough to allow for proprietary implementations in the management of the Flash device.

The partition structures are used to store the partition information of the Flash device. This structure consists of a partition table structure with an array of partition information structures. The high-level partition table structure is used for version information and to determine the number of partitions in the device. The partition information structure holds the specific parameters for a partition. See Example 5.

### Example 5: Partition Table and Partition Information Structures

```
struct {
unsigned int 64Identifier;
unsigned int   Version;
unsigned int   NumberofPartitions;
unsigned int   Reserved;
unsigned int   Reserved;
PartitionInfo  Partitions[NumberofPartitions];
}PartitionTable;


struct {
unsigned int   Type;
unsigned int   Usage;
unsigned int   Identifier;
unsigned int   PartitionAttributes;
unsigned int   StartingAddress;
unsigned int   Reserved1;
unsigned int   EndingAddress;
unsigned int   Reserved2;
unsigned int   RsvdPoolStartingAddr;
unsigned int   Reserved3;
unsigned int   RsvdPoolSize;
unsigned int   Reserved4;
unsigned int   RsvdPoolAlg;
unsigned int   RuntimeBBT_Type;
unsigned int   RuntimeBBTStartAddr;
unsigned int   Reserved5;
unsinged int   BackupRuntimeBBTStartAddr;
unsigned int   Reserved6;
unsigned int   Reserved7;
unsinged int   Reserved8;
}PartitionInformation;
```

Partitioning can be logical or physical, which allows hardware-based partition support for devices such as eMMC and eSD, or software-based partitioning on flat devices such as a raw NAND device. If a partition table is not present, the Boot ROM views the device as one flat boot partition. The fields for the `PartitionTable` are used as follows:

- `Identifier` – The 64-bit identifier which is ASCII encoded "MRVL MPT" (0x4D52564C, 0x204D5054).
- `Version` – The 32-bit version identifier used for tracking purposes. Not currently used by the Boot ROM.
- `Number of Partitions` – Number of partitions on the Flash device.
- `Partition[ ]` – Array of `PartitionInformation` structures.

The `PartitionInformation` structure is used to describe a particular partition. Many of the fields in this structure can be customized by an OEM. The fields are used as follows:

- `Type` – Identifies the partition as physical or logical using a 32-bit ASCII identifier of “Phys” (0x50687973) or “Logi” (0x4C6F6769).

- `Usage` – 32-bit identifier to specify the use of the partition. This can be a vendor-specific value or a predefined vlaue:
  - “Boot” – 0x424F4F54
  - “OSLD” – 0x4F534C44
  - “KRNL” – 0x4B524E4C
  - “FFOS” – 0x46464F53
  - “FSYS” – 0x46535953
- `Identifier` – The 32-bit identifier specified by the OEM to track the partition (if multiple partitions of the same type are used).
- `Partition attributes` – The 32-bit word of attribute flags used to define partition attributes.
- `Starting Address` – Absolute byte address for the LSB of the partition. For hard partitions, this value would normally be 0x0; for logical partitions, this value would be the offset from the base of the Flash.
- `Ending address` – Absolute byte address of the MSB in the partition.
- `RsvdPoolStartingAddr` – Absolute byte address for the starting block of the reserved pool, if used. If no reserved pool is present, this field is set to NULL.
- `RsvdPoolSize` – Size of the reseved pool, in bytes. If no reserved pool is present, set this field to NULL.
- `RsvdPoolAlg` – One of three 32-bit identifiers used to define the direction of the reserved pool growth:
  - NULL – Reserved pool not used.
  - “UPWD” – 0x55505744 – Reserve pool grows up.
  - “DNWD” – 0x444E5744 – Reserved pool grows down.
- `RuntimeBBT_Type` – The 32-bit identifier used to identify the runtime BBT type being used:
  - “MBBT” – 0x4D424254 – Marvell bad-block Table.
  - “WNCE” – 0x574e4345 – Microsoft® Windows® bad-block Table format.
  - “Linx” – 0x4C695E78 – Linux® bad-block table format.
  - NULL – Bad-block table NOT used.
  - Custom – Any custom 32-bit identifier.
- `RuntimeBBTStartAddr` – Absolute address of the runtime bad-block table Flash block, in bytes.
- `BackupRuntimeBBTStartAddr` – Absolute address of the backup runtime bad-block table Flash block, in bytes. This is optional and should be set to NULL if not used.

Reserved fields are intended for future use. This includes support for 64-bit addresses as Flash devices increase in size.

#### 6.6.2.2 Operation

This section describes the suggested operation of the software implementation. It is meant as a guide for software developers, but should not be considered the only approach. The goal is to define some common methods for Flash management implementations in software, and to point out where implementations can vary. In the end, a reliable Flash management implementation is required to assure consistent mobile device operation for the end user.

### 6.6.2.2.1 Common Software Implementations

Several concepts need to be common among all software implementations for the Flash management method to be successful. Many of the concepts are common because the Boot ROM implementation is the same for different Flash devices. This forces the lower levels of software to adhere to some components of the implementation. Other concepts and components can vary greatly at the OS level, where value add becomes more important.

First, there must be agreement on the general handling of Flash blocks. These concepts are not as critical for all Flash devices. NOR devices and managed NAND devices have better user visible characteristics than RAW NAND devices. In general, these concepts can apply to all devices.

A bad block is any block that has:

- Been marked bad by the device manufacturer
- Experienced a failure when an Erase command has been issued that results from bits that cannot be reset from 0 to 1. An Erase failure from a power outage should be re-tried.
- Experienced a program failure where the bits cannot be programmed correctly within an allowable ECC correction threshold. A program operation can be tried more than once but, in general, if two program attempts fail successively, the block should be marked as bad and no longer used.

A *corrupt block* is any block that has a data error on a Read operation. The error may not be due to block failure, but could be due to some other event. For Instance, and error can occur from a power failure during a Program operation, or improper programming procedure in software algorithms.

A *disturbed block* is any block that has experienced data errors due to a Read Disturb phenomenon, where the data is corrected by ECC and is usable, but the ECC threshold has been exceeded. The data must be refreshed on the block before the number of errors exceeds what can be corrected by the ECC algorithms.

A *good block* is any block that:

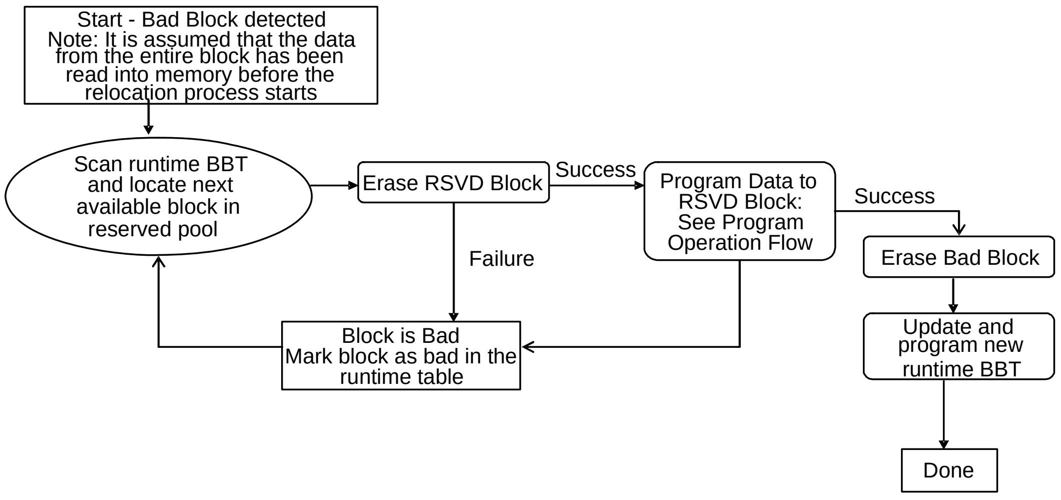
- Passes a Read operation with the number of ECC corrections below the threshold.
- Can be successfully erased and programmed with new data.

Having a common definition of various Flash block states helps understand why certain Flash operations are required. Primarily, these operations help manage raw NAND Flash devices. For other devices, some of the steps in the following flows never occur. For example, when reading a NOR device, getting an ECC error is not expected, and the Read should always be successful. But the basic flow operates the same way for both NAND and NOR.

All software that can program and erase Flash blocks must be able to detect a bad-block situation, for all Flash, and properly relocate the block on a NAND Flash device. Once a bad block is detected, several steps ensure there is no loss of data.The flows in Figure 8 through Figure 12 are suggested flows and should be customized based on the implementation requirements. The flows are also geared towards boot partitions where data reliability is the highest importance. For implementations that are performance sensitive or that incorporate wear-levelling features, these algorithms should be modified as needed.

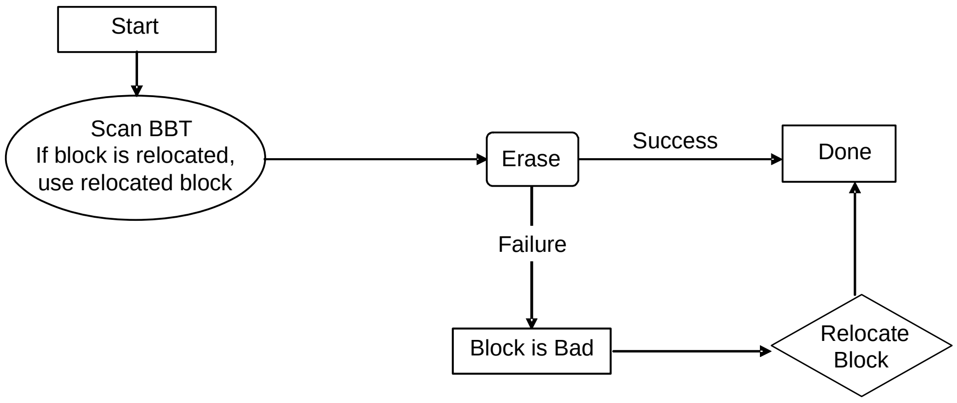
Figure 8 shows the general flow for handling a relocation of a block on a NAND device. The relocation flow may not be required on a managed NAND or NOR device. One key step is to erase the block that was found bad after the data has been relocated. This step is performed to avoid using stale data at a later time if the block is read by another software layer.

**Figure 8: Flow for Relocating a Block Found Bad During Runtime**


Start - Bad Block detected
Note: It is assumed that the data from the entire block has been read into memory before the relocation process starts
Scan runtime BBT and locate next available block in reserved pool
Erase RSVD Block
Success
Program Data to RSVD Block: See Program Operation Flow
Success
Erase Bad Block
Failure
Block is Bad
Mark block as bad in the runtime table
Update and program new runtime BBT
Done

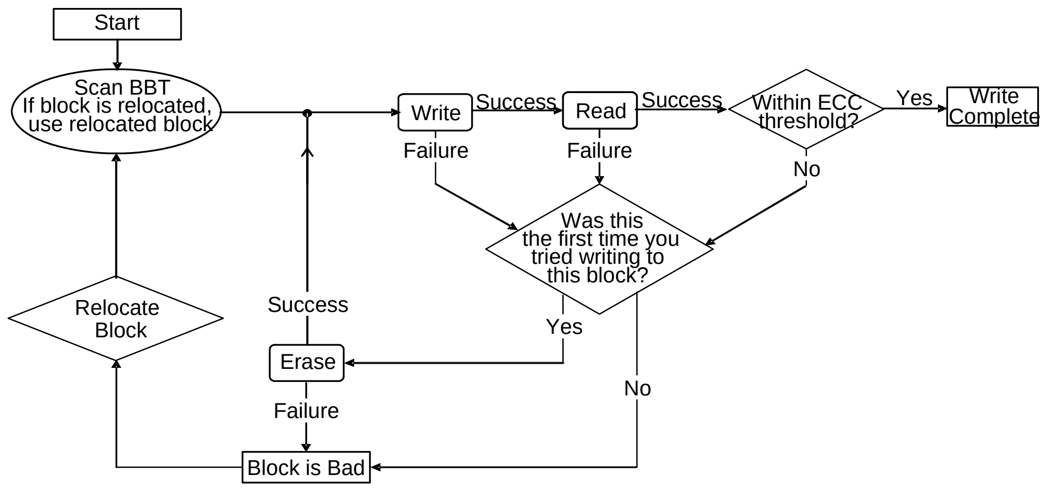
When erasing a block, the flow shown in Figure 9 should be used as a guideline. The one assumption is that the data resides in volatile memory before the Erase operation begins. To ensure the data is in volatile memory, use a Read/Modify/Erase/Write sequence for runtime operation. The details of this operation are at the software developer's discretion.

**Figure 9: Erase Operation Flow**



The program operation is detailed in Figure 10. The important step for this operation is to ensure that the data is within the ECC threshold after the program operation. ECC thresholds applies to MLC technologies where data errors are expected and ECC is used to correct most errors.

**Figure 10: Program Operation Flow**



The Read operation described in Figure 11 adds a check after the data is read to verify the number of ECC corrections that were required. This check is not needed for all devices, but for those that support ECC, it should be checked after every Read operation. MLC NAND has a high number of ECC bit corrections. Once the correction level exceeds a predefined threshold, the block must be refreshed using a Read/Modify/program operation as detailed in Figure 12. For MLC NAND, exceeding the ECC threshold occurs due to the Read phenomenon that is possible with this technology.

**Figure 11: Read Operation Flow**

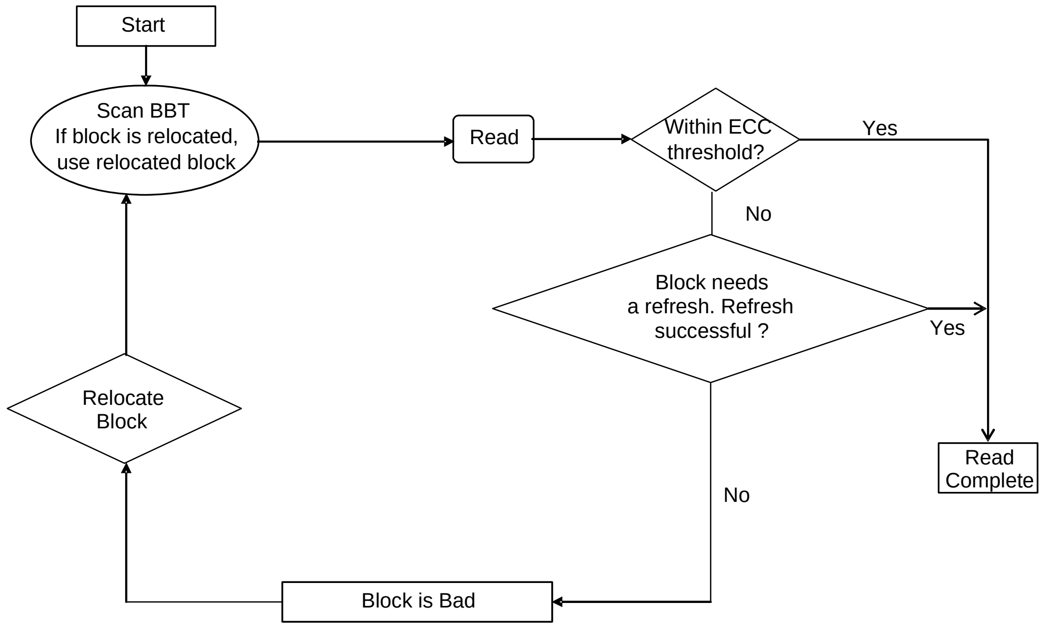
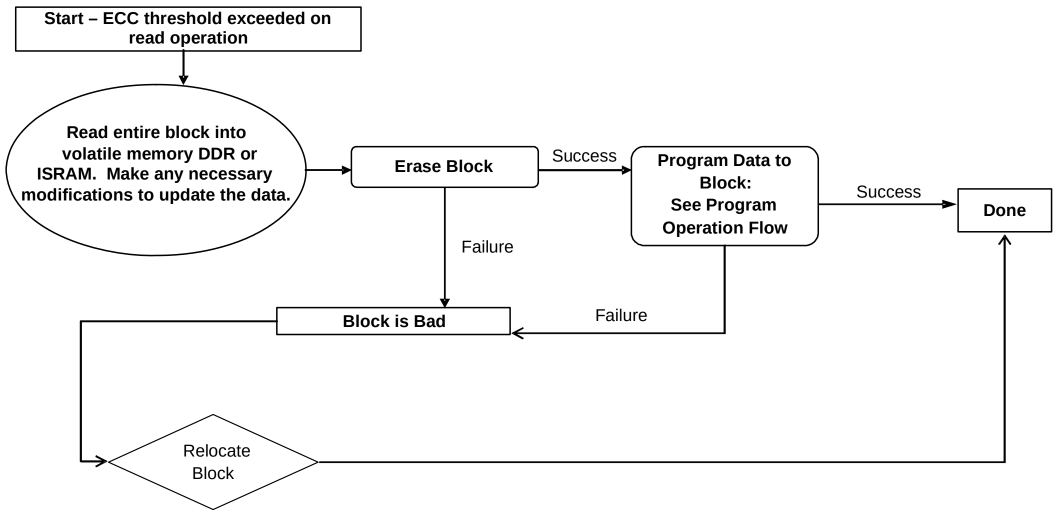


Figure 12: Refresh Operation for Blocks with Read Disturb


Start – ECC threshold exceeded on read operation
Read entire block into volatile memory DDR or ISRAM. Make any necessary modifications to update the data.
Erase Block
Success
Program Data to Block: See Program Operation Flow
Success
Done
Failure
Failure
Block is Bad
Relocate Block

### 6.6.2.3 Boot Partition and Boot Process

This section describes the methods and algorithms as well as the layout of the boot partition that is used by the Marvell Boot ROM. Additional information about the Boot ROM is provided in the Marvell Boot ROM specification.

#### 6.6.2.3.1 Boot ROM Flow and Expected Boot Partition Layout

The boot partition is a read-only partition that contains critical boot information and boot loaders. The level of protection to implement for the boot partition depends on the Flash device chosen and the needs of the OEM. Types of protection can include the following:

- Hardware locking mechanisms where the Flash provides the capabilities through hardware signalling to lock Write access to blocks in the Flash device.
- Software locking mechanisms where the Flash device offers commands issued through software to lock Flash device blocks.
- Logical separation where the Flash is partitioned logically through software, but no physical protection is implemented.

Figure 13 shows the expected layout of the boot partition. The critical boot data resides in Block 0 with a backup copy in Block 1 (or the first good block after Block 1). The boot layout can be applied to all Flash device types, not just NAND devices. This methodology can have benefits for FOTA even on NOR or managed NAND platforms. Even if a power failure occurs during an update of critical data, the backup block still books and continues the updates. The critical data needed to boot the platform includes the NTIM header, factory bad-block table for NAND platforms, and partitioning table (if multiple partitions are present). The next critical piece for boot operation is the OEM Boot Module. This first-level loader can reside in the same block as the header information, such as Block 0 and Block 1, if it is small enough to fit within the block size. Optionally, it can be placed elsewhere in the boot partition in a separate physical block.

The next important consideration is the location of the reserved pool for the boot partition. A reserved pool for the boot partition must reside in Blocks 2 through Block *n*. This location falls immediately after the backup block, Block 1, to streamline the boot process in the event of bad-block scenarios. When the platform is provisioned, there is no way to know which blocks might be bad. This scenario has the potential to make initial programming a cumbersome and complex task. If Block 1 is bad from the factory and Block 0 goes bad during a runtime operation, the location of the reserved pool must be in a location that the Boot ROM knows about. To avoid potential boot issues and provisioning issues, the reserved pool MUST be located starting at Block 2 and grow up from there. The algorithm performs the following steps during the boot process to locate the backup block in the event Block 0 becomes bad.

Start:

1. Read the critical boot data from Block 0. Check for the NTIM if the data is good. Perform a boot if the NTIM is found.

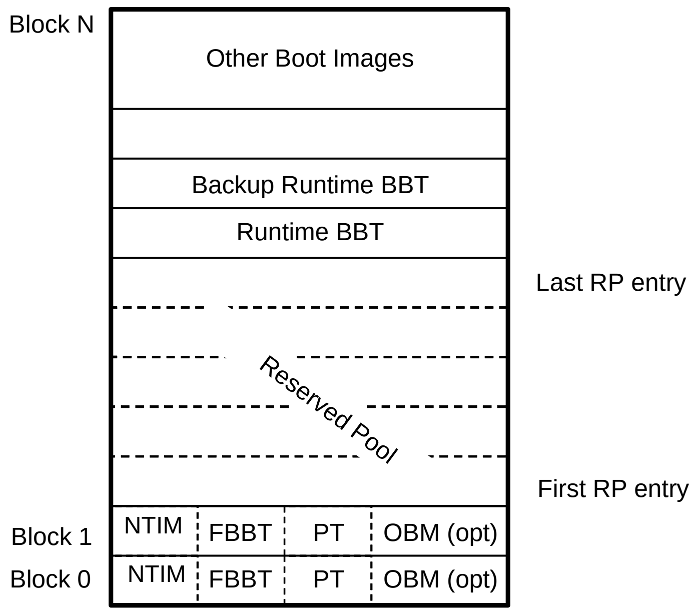
Loop:

2. Check for the backup block if the NTIM is not found or data is corrupt. Check for the NTIM if data is good. Perform a boot if the NTIM is found.
3. Increment the pointer to the next block. Now the blocks in the reserved pool are being consumed by boot code and the reserved pool usage must be adjusted to account for this modification in the reserved pool size.
4. Have all possible backup blocks been checked? If not, then Loop.
5. At least 10 blocks have been checked and no NTIM has been found. Wait for a download or terminate the boot process.

Boot:

Continue loading images.

**Figure 13: Boot Partition Layout**



The benefit of the reserved pool may not be immediately clear from the boot process alone. When the platform is first provisioned, the Flash programming tool must first gather the bad-block information for a NAND device. If Block 1 is bad, this could have a ripple effect on the system if images were placed in Block 2 instead of the reserved pool. Because of the algorithm used, in the event that Block 0 goes bad during a runtime operation, the backup block would force the Flash programming tool to perform unnecessary relocation operations until a good block was found for the backup block. To avoid the ripple effect at programming time, the reserved pool location is moved so that the reserved pool blocks can be consumed if Block 1 is bad from the factory. Then the Flash programming tool can simply make an adjustment to the reserved pool start and size.

To satisfy the Write ordering requirements of MLC NAND devices, all blocks are programmed in linear order starting with the least significant page of the block and ending with the most significant page of the block. From the Boot ROM perspective, only the NTIM and OBM are required to boot. The bad-block table and partition table are optional. However, if the bad-block table and partition table are missing, then the boot process becomes a best effort boot and may fail under certain circumstances, especially with NAND devices.

In the absence of the bad-block table, the Boot ROM relies on the NTIM specified addresses for booting the system. In the event of a data error, the boot process would be terminated. With a bad-block table in place, a data error would cause the Boot ROM to look for a relocated block and attempt to boot using data from the reserved pool.

If the partition table is not present, the entire device is considered to be the boot partition, where the layout of the Flash remains consistent with the boot partition layout. With low-level software viewing the entire flash as a boot partition, there may be some changes at the OS level where the reserved pool location is now at the beginning of the Flash. One option would be to reserve a few blocks in the beginning to satisfy the backup block requirements, then have another reserved pool located elsewhere in the Flash for use by the OS. The Boot ROM does not prevent this from happening and would still boot the system.

Under normal circumstances where all boot data is present, the boot would be as shown in Figure 14 for an initialized system. The Boot ROM stores Read Disturb information, for the block that it reads, to allow the OBM to handle any Read Disturb phenomenon. Figure 15 provides details of the uninitialized boot flow where a download happens. The uninitialized flow is expected to happen only when the Flash on the platform has not been programmed.

**Figure 14: Boot Flow for an Initialized Platform**

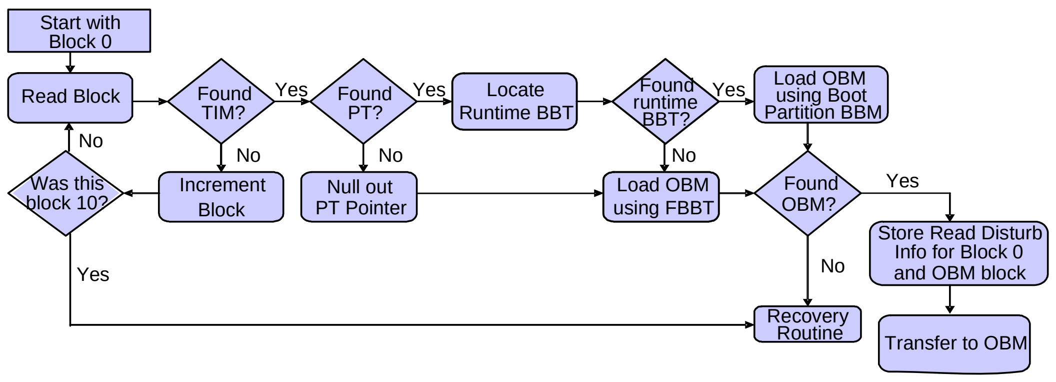
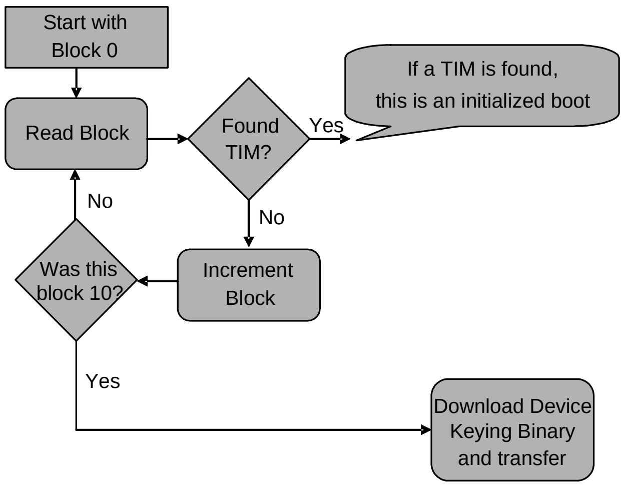


Figure 15: Boot Flow for an Uninitialized Platform


Start with Block 0
Read Block
Found TIM?
Yes
If a TIM is found, this is an initialized boot
No
Increment Block
Was this block 10?
No
Yes
Download Device Keying Binary and transfer

### 6.6.2.3.2 Boot Process for Higher Level Software (Post Boot ROM)

This section provides details of the general steps required to boot the system once the Boot ROM has transferred to the next layer of software. The implementation provides flexibility for OEMs to add value. Not all of the steps are required; whenever possible, optional steps are distinguished from required steps.

In general, all software in the software stack must be able to interpret the software structures defined in section Section 6.6.2.1, Important Structures. These structures provide necessary details about the system layout, image location, and other important details. These structures can be interpreted and then translated into customized formats, if necessary. For instance, the factory bad-block table could be translated into a different runtime format that is proprietary to the operating system, such as a bit map. However, the original factory bad-block table must not be overwritten as this would cause issues during the boot process when the Boot ROM is running.

The next general requirement is that all higher level software must be able to handle a “Bad Block,” “Disturbed Block,” and “Good Block” as described in section Section 6.6.2.2.1, Common Software Implementations. The higher level software may or may not be capable of handling a “corrupt block” as defined in section Section 6.6.2.2.1, Common Software Implementations. This is implementation dependent. To properly handle these types of blocks, the flows in Figure 8 through Figure 12 must be implemented. Also, the requirements of MLC NAND devices should be adhered to as described in the manufactures device specification.

At a high level, follow these steps as part of the boot process:

1. Locate the Read Disturb information remaining in memory by the Boot ROM. This information is gathered from the Flash device by reading the boot block, NTIM, bad-block, and partition table information.
2. Locate and read the next image in the boot process from Flash using the NTIM or by a proprietary method.
3. Check the ECC information on all blocks read from Flash to determine if the ECC is within the threshold.
4. Handle the disturbed blocks by implementing the refresh flow from Figure 12.
5. For bad blocks found during the refresh or normal boot process, relocate the block by implementing the flow found in Figure 8.

6. Once the next image has been loaded and maintenance tasks handled, prepare to transfer to the next image. This transfer includes gathering and storing any information required for the next layer of software.
7. Transfer control to the next image.

##### 6.6.2.3.3 Special Considerations for the OEM Boot Module

The OEM boot module has some special tasks in addition to those defined in the previous steps because the Boot ROM may not be able to fix Read Disturbs or other issues with Flash blocks. When the Boot ROM reads the blocks that contain the boot information and OBM binary, it checks the ECC threshold and stores the results. Upon transfer to the OBM, it becomes the duty of the OBM to check the results passed up from the Boot ROM and to refresh any blocks that were read by the Boot ROM and identified as out of the ECC threshold.

##### 6.6.2.3.4 Special Considerations for the Operating System Software

Once the platform is fully operational, there are special considerations at the OS level for MLC NAND devices to operate reliably. Because the majority of the Flash is reserved for operating system usage, the OS must periodically check ECC thresholds across any area of the device that it can read or write. This ECC threshold “check” should be a low-frequency algorithm that runs in the background as a “garbage collection” type of activity.

#### 6.6.2.4 Requirements for Flash Burning Utilities

Utilities that program the Flash have some additional duties to perform. The algorithms must consider development environments as well as manufacturing environments. Also, customer features for downloading versus preprogramming the Flash or parts of the Flash must be considered. The raw MLC NAND devices are the worst case for programming complexity. However, some hybrid devices require complex partitioning of the device.

In general, the Flash programming utilities must consider the following:

- The device has been programmed before. For a device already in service, the factory bad-block table must be preserved as well as any runtime updates. For a Flash device that has not previously been in service, the entire device must be scanned to capture and save the factory bad-block information.
- Algorithms are needed when bad blocks are found and relocations are necessary.
- The Boot ROM requires the relocation of the boot block and bad block to the appropriate areas for reliable boot operations.
- The operating system has requirements for partitioning or varying features, such as how to handle missing partition information, or bad-block table information that is considered optional.
- Backward compatibility issues. Several implementations already exist in products that cannot be modified. The PXA320 processor has a version 2.xx Boot ROM, which uses only the last page of Block 0 for a BBT. The PXA930 processor, PXA310 processor, and PXA300 processor have a version 3.xx Boot ROM that implements some basic wear leveling capabilities within the block. The PXA9XX processors have support for two wear levelling algorithms: one that grows up and one that grows down.

The Flash burning utilities must perform the following steps to successfully program a platform:

1. Download images and boot information in an implementation-specific manner. In general, the Flash Binary Format (FBF) must be supported for manufacturing environments. This format allows the OEM to download and program the entire software stack for a fully functioning system. In addition, support for development environments could be added that allows partial download of the software stack.
2. Determine the proper Flash based on information downloaded in the NTIM or Master Header.
3. Scan the Flash to determine if it has been in service previously. For NAND devices, the factory bad-block information and runtime bad-block information should be preserved. If this is the first

use of the Flash device, the factory information must be gathered and stored in the factory bad block table. This step can be skipped if it is not a NAND-based device.

4. Determine if the device must be partitioned based on the boot information downloaded. Partition as required. Treat the entire device as a boot partition if no partition information is downloaded.
5. Once the factory information and partitioning is completed, start programming data to the device.

   Boot block and backup boot block must be programmed first. If the backup Block 1 is marked bad from the factory, then use the first good block after Block 1. Also, create an entry in the bad-block table noting that the block in the reserved pool has been used.
6. Now program the images to the Flash. If a block has been marked as “bad” by the factory, it MUST NOT be erased as part of this process; skip it and relocate the block accordingly. If additional blocks are determined to be “bad” from the programming process, then create in memory a runtime bad-block table. This table is based off of the factory bad-block table, FBBT. Bad blocks accumulate in memory until all images have been programmed.
7. After all of the images are properly programmed, including any validation that may be required, transfer the runtime bad-block table to the Flash at the location indicated in the partition table for the boot partition.
8. If the Flash utility can load the OBM and then boot after complete programming, load the boot information into memory and pass to the OBM the `BootInfoPointer` structure.

# 7 Marvell® ARMADA 16x Applications Processor Family Implementation

The following tables provide implementation settings for the ARMADA 16x Applications Processor.

ARMADA 16x Applications Processor Boot ROM Register configurations:

- Table 20, "ARMADA 16x Applications Processor Pin Mux Settings"
- Table 21, "ARMADA 16x Applications Processor SMC register configuration for CS0 NOR and OneNand"

## 7.1 ARMADA 16x Applications Processor Register Settings

Table 20: ARMADA 16x Applications Processor Pin Mux Settings

| Ball Name | Signal Name | Address | Value |
|---|---|---|---|
| **Shared NFC and SMC Data Signals** | | | |
| MFP_0 | DF_IO15 | 0xD401_E04C | 0x8C0 |
| MFP_1 | DF_IO14 | 0xD401_E050 | 0x8C0 |
| MFP_2 | DF_IO13 | 0xD401_E054 | 0x8C0 |
| MFP_3 | DF_IO12 | 0xD401_E058 | 0x8C0 |
| MFP_4 | DF_IO11 | 0xD401_E05C | 0x8C0 |
| MFP_5 | DF_IO10 | 0xD401_E060 | 0x8C0 |
| MFP_6 | DF_IO9 | 0xD401_E064 | 0x8C0 |
| MFP_7 | DF_IO8 | 0xD401_E068 | 0x8C0 |
| MFP_8 | DF_IO7 | 0xD401_E06C | 0x880 |
| MFP_9 | DF_IO6 | 0xD401_E070 | 0x880 |
| MFP_10 | DF_IO5 | 0xD401_E074 | 0x880 |
| MFP_11 | DF_IO4 | 0xD401_E078 | 0x880 |
| MFP_12 | DF_IO3 | 0xD401_E07C | 0x880 |
| MFP_13 | DF_IO2 | 0xD401_E080 | 0x880 |
| MFP_14 | DF_IO1 | 0xD401_E084 | 0x880 |
| MFP_15 | DF_IO0 | 0xD401_E088 | 0x880 |
| **Shared SMC and NFC Control Signals** | | | |
| MFP_21 | ND_ALE<br>SMC_nWE | 0xD401_E0A0 | 0x880 |
| MFP_22 | ND_CLE<br>SMC_nOE | 0xD401_E0A4 | 0x880 |

Table 20: ARMADA 16x Applications Processor Pin Mux Settings (Continued)

| Ball Name | Signal Name | Address | Value |
|---|---|---|---|
| MFP_17 | ND_nWE<br>SMC_LUA | 0xD401_E090 | 0x880 |
| MFP_24 | ND_nRE<br>SMC_nLLA | 0xD401_E0AC | 0x880 |
| **Static Memory Controller SMC_CS0 Signals** | | | |
| MFP_18 | SMC_nCS0 | 0xD401_E094 | 0x883 |
| MFP_29 | SMC_SCLK | 0xD401_E0C0 | 0x880 |
| MFP_28 | SMC_RDY | 0xD401_E0BC | 0x2880 |
| **NAND Flash Controller Signals** | | | |
| MFP_16 | ND_nCS0 | 0xD401_E08C | 0x881 |
| MFP_26 | ND_RDY0 | 0xD401_E0B4 | 0x4881 |
| **SD/MMC3 Common Data Signals** | | | |
| MFP_0 | MMC3_DAT7 | 0xD401_E04C | 0x8C6 |
| MFP_1 | MMC3_DAT6 | 0xD401_E050 | 0x8C6 |
| MFP_2 | MMC3_DAT5 | 0xD401_E054 | 0x8C6 |
| MFP_3 | MMC3_DAT4 | 0xD401_E058 | 0x8C6 |
| MFP_4 | MMC3_DAT3 | 0xD401_E05C | 0x8C6 |
| MFP_5 | MMC3_DAT2 | 0xD401_E060 | 0x8C6 |
| MFP_6 | MMC3_DAT1 | 0xD401_E064 | 0x8C6 |
| MFP_7 | MMC3_DAT0 | 0xD401_E068 | 0x8C6 |
| **SD/MMC3 Primary Control Signals** | | | |
| MFP_8 | MMC3_CLK | 0xD401_E06C | 0x8C6 |
| MFP_9 | MMC3_CMD | 0xD401_E070 | 0x8C6 |
| **SD/MMC3 Secondary Control Signals** | | | |
| MFP_35 | MMC3_CMD | 0xD401_E0D8 | 0x8C6 |
| MFP_36 | MMC3_CLK | 0xD401_E0DC | 0x8C6 |
| **SD/MMC1 Signals** | | | |
| MFP_37 | MMC1_DAT7 | 0xD401_E000 | 0x8C1 |
| MFP_38 | MMC1_DAT6 | 0xD401_E004 | 0x8C1 |
| MFP_54 | MMC1_DAT5 | 0xD401_E044 | 0x8C1 |
| MFP_48 | MMC1_DAT4 | 0xD401_E02C | 0x8C1 |
| MFP_51 | MMC1_DAT3 | 0xD401_E038 | 0x8C1 |
| MFP_52 | MMC1_DAT2 | 0xD401_E03C | 0x8C1 |
| MFP_40 | MMC1_DAT1 | 0xD401_E00C | 0x8C1 |
| MFP_41 | MMC1_DAT0 | 0xD401_E010 | 0x8C1 |

Table 20: ARMADA 16x Applications Processor Pin Mux Settings (Continued)

| Ball Name | Signal Name | Address | Value |
|---|---|---|---|
| MFP_43 | MMC1_CLK | 0xD401_E018 | 0x8C1 |
| MFP_49 | MMC1_CMD | 0xD401_E030 | 0x8C1 |
| **SPI Signals** | | | |
| MFP_107 | SSP2_RXD | 0xD401_E1AC | 0x884 |
| MFP_108 | SSP2_TXD | 0xD401_E1B0 | 0x884 |
| MFP_110 | GPIO_110 | 0xD401_E1B8 | 0x880 |
| MFP_111 | SSP2_CLK | 0xD401_E1BC | 0x884 |
| MFP_112 | GPIO_112 | 0xD401_E1C0 | 0x880 |

Table 21: ARMADA 16x Applications Processor SMC register configuration for CS0 NOR and OneNand

| Register | Address | Value | Function |
|---|---|---|---|
| SMC_CSDFICFG0 | 0x51890009 | 0x00000000 | — |
| SMC_CSADRMAP0 | 0x518900C0 | 0x00000000 | — |

# 8 Methods for Platform Provisioning

The requirements for platform provisioning depend on the operational model selected. This chapter provides some guidance on provisioning a platform for operation with the Boot ROM. “Provisioning a platform” means performing the required steps to turn an uninitialized system into an initialized system capable of booting to an operating system. The provisioning process is a process of programming the Flash device and providing required information to allow the Boot ROM to boot the system on the next power-on reset. Therefore, the provisioning process takes a system from an uninitialized state to an initialized state, allowing the device to be deployed by an end user.

Support tools are required to properly provision a system depending on the Flash devices selected. Marvell provides the Marvell® Wireless Trusted Platform Tool Package as an example for OEMs. This package contains all of the host tools and middleware required for non-trusted boot. Contact your local Marvell field representative for more information.

Software and tools that may be required include:

- Firmware capable of running on the target system and initializing the Flash, such as programming images to the Flash at the proper locations, and initializing the Flash management infrastructure (for NAND devices).
- Host tools running on a PC that can generate the NTIM binary images, and communicate with the target for downloading images.

There are two basic methods of provisioning: pre-programming and downloading. Each method requires slightly different tools to accomplish the provisioning process. Pre-programming supports high-volume manufacturing processes. Downloading is development or manufacturing depending on the OEM requirements. Details of both methods are provided in the following sections. Differences between a development system and a manufacturing system are highlighted whenever possible.

## 8.1 Non-Trusted Provisioning

The non-trusted boot process occurs on the platform upon every reset of an initialized platform. The non-trusted boot processes use the information stored in the Non-Trusted Image Module (NTIM) to load the images from Flash memory before transferring control, if required. Use the provisioning process described in this section to store an NTIM on an uninitialized platform.

The first step toward provisioning a non-trusted system is to review the use cases to determine the requirements. During this time, consider the following:

1. How the Flash device is programmed and initialized. Several options are available:

- Pre-Programming:
  - Using the JTAG port using the JTAG software package
  - By a high volume Flash programming vendor
- Downloading using a separate software/firmware image. The Boot ROM supports downloading over the USB port. The “Device Keying Binary” is a reference software image that is provided in the Marvell® Wireless Trusted Platform Tool Package.

2. The Flash device that is used for booting the system. The options are:
   - X16 NAND device on data Flash Controller Chip Select 0
   - X8 NAND device on data Flash Controller Chip Select 0
   - XIP device on the Static Memory Controller Chip Select 0
   - OneNand/FlexOneNand device

- SD/MMC device

3. The size of the first boot loader binary. Size implications must be reviewed and are affected by processor internal SRAM available, Flash management implementation, and Flash device chosen. The “OEM Boot Module” is a reference first level boot loader image that is provided in the Marvell® Wireless Trusted Platform Tool Package.

### 8.1.1 Provisioning a Non-Trusted Boot Platform Using the Download Method

Complete these steps fully to provision an uninitialized platform using the download capabilities of the Boot ROM.

Preparation:

1. Decide on the usage model for booting the system.
2. Prepare a Device Keying Binary or other firmware image, a Non-Trusted Image Module binary to describe the Device Keying Binary, using the Marvell Wireless Trusted Platform Tool Package or a custom tool created by the OEM.
3. Prepare the OEM boot module and associated operating system images, as well as a second NTIM binary to describe them, using the Marvell Wireless Trusted Platform Tool Package or a custom tool created by the OEM.

Provisioning:

a) Boot the target platform and first download the Non-Trusted Image Module and associated Device Keying Binary created in Step 2 using the download tool available in the Marvell Wireless Trusted Platform Tool Package or a custom tool created by the OEM.
b) The Device Keying Binary runs on the system and must perform all of the requirements documented in Section 8.1.1.1
c) The Non-Trusted Image Module, OEM boot module, and associated OS images created in Step 3 are downloaded by the Device Keying Binary using the download tool available in the Marvell Wireless Trusted Platform Tool Package or a custom tool created by the OEM.
d) The Device Keying Binary must have the built-in capabilities to allow debug and testing of the Non-Trusted Image Module, OEM boot module, and associated OS images created in Step 3

Test:

4. As a last step, verify the non-trusted boot operation from a power-on reset.

#### 8.1.1.1 Device Keying Binary Requirements for an Unprogrammed Non-Trusted System

The Device Keying Binary is responsible for provisioning and preparing an uninitialized system for initial boot. It must determine the Flash used to boot, program the proper images to the Flash, and if the platform is a NAND platform, validate or create the relocation table.

An OEM may want to create multiple versions of the Device Keying Binary: one for use in manufacturing, and one for use in development. The development Device Keying Binary could be used to aid in platform debugging.

The Device Keying Binary is responsible for completing the following on non-trusted boot platforms:

- Provide an interface through the USB port to print messages.
- Provide an interface through the USB port to download binary images.
- Set up the DDR memory to temporarily hold images that are to be stored on the boot device.
- Set up all necessary Flashes to store the downloaded images. At a minimum, this would include the OEM boot module.

- Initialize the Flash management structures on the Flash device, see “Flash Management” for details.
- Verify images are downloaded error free using an implementation-dependent method such as CRC check, ECC check, or other method.
- Set the initial value of fuses that control processor features and boot options, if required.

### 8.1.2 Provisioning a Non-Trusted Boot Platform Using the Pre-Programming Method

Complete these steps fully to provision an uninitialized platform using the pre-programming capabilities of a Flash programming vendor or JTAG development tools.

Preparation:

1. Decide on the usage model for booting the system.
2. Prepare the OEM boot module and associated operating system images, as well as a second non-trusted image module binary to describe them, using the Marvell Wireless Trusted Platform Tool Package or a custom tool created by the OEM.

Provisioning:

a) The pre-programming software must perform all of the requirements documented in Section 8.1.2.1

b) The Non-Trusted Image Module, OEM boot module, and associated OS images created in Step 2 are programmed to the Flash according to the NTIM Flash load address for each image.

Test:

3. As a last step, verify the non-trusted boot operation from a power-on reset.
4. If required, the OBM must be able to program processor fuses on the first boot attempt.

#### 8.1.2.1 Pre-Programming Requirements for an Unprogrammed Non-Trusted System

The pre-programming software must complete the following tasks for the Boot ROM to boot the system:

- Set up all necessary Flashes to store the images. At a minimum, this would include the OEM boot module and NTIM.
- Initialize the Flash management structures on the Flash device, see “Flash Management” for details.
- Verify images are downloaded error free using an implementation-dependent method such as CRC check, ECC check, or other method.

# 9 Communication Protocol

The section describes the relevant details of the USB/UART communication protocol to allow OEMs to port their existing proprietary USB/UART applications to support communication with the Boot ROM.

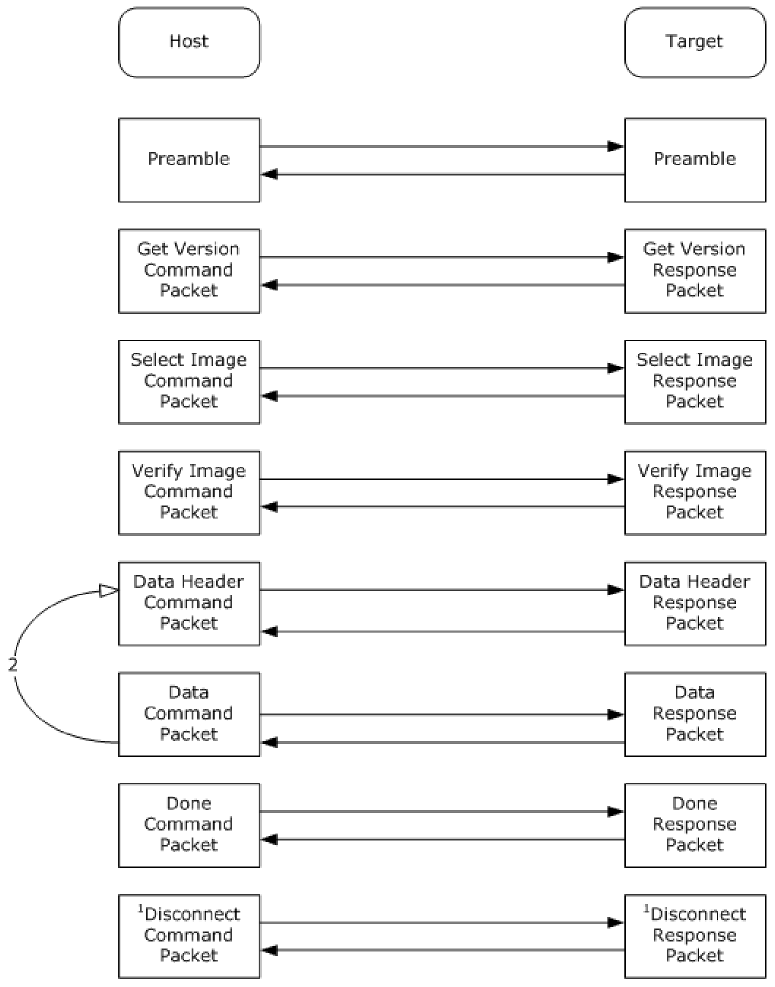
The communication protocol is used to download images during the device keying process, as well as for the JTAG re-enabling process.

Refer to Table 1 for support of this feature.

In this section, the "`Host`" refers to the `WTPTool.exe` application and "`Target`" refers to the Boot ROM.

The communication protocol follows a strict handshaking methodology, which is always initiated by the host. The host sends a command packet and the target responds with a status packet (response packet).

**Figure 16: Download Flow Diagram**


Host
Target
Preamble
Preamble
Get Version Command Packet
Get Version Response Packet
Select Image Command Packet
Select Image Response Packet
Verify Image Command Packet
Verify Image Response Packet
Data Header Command Packet
Data Header Response Packet
2
Data Command Packet
Data Response Packet
Done Command Packet
Done Response Packet
1Disconnect Command Packet
1Disconnect Response Packet

**NOTE:** 1. The disconnect command is only issued after the target has transmitted all of the files.
2. The data header and data command/response packets are sent continually until all data has been transmitted.

## 9.1 Preamble

The preamble data stream is a 4-byte data packet containing `0x00`, `0xD3`, `0x02`, and `0x2B`. Table 22 represents a 32-bit word: `0x2B02D300`. The preamble data stream requires that the bytes are in network byte ordering.

**Table 22: Preamble**

| Byte-3 | Byte-2 | Byte-1 | Byte-0 |
|---|---|---|---|
| 0x2B | 0x02 | 0xD3 | 0x00 |

The target responds to the preamble from the host with the same preamble.

## 9.2 Structure for Host Commands

The structure of all commands sent by the host follows this format:

```
struct Command
{
   Byte   CMD
   Byte   SEQ
   Byte   CID
   Byte   Flags
   Unsigned intLEN
   Byte [LEN]Data
}
```

- `CMD` (Command) – Contains the opcode that indicates the type of command being sent. The size is 1 byte.
- `SEQ` (Sequence) – Used during data transmission (when the data command is used) to ensure that the block of data that the host sends matches the block of data that the target is expecting. The sequence number is `0` for all other commands. The sequence number is 1 for the first data transmission, `2` for the second, and so on. Since the size of the sequence field is 1 byte, the sequence number rolls over after 255 data transmissions.
- `CID` (Command ID) – Specific number that relates all of the commands (and responses) of a single flow. A flow is the communication from the preamble to the “done” acknowledgement. The host defines the `CID` when it sends the first command after the preamble. The same `CID` is used until the done command after a download or a JTAG reenablement. If another download follows, the host must generate a new `CID` for the next download flow (after the next preamble).
- `Flags` – Bits [7:1] are reserved.
  Bit 0 – Endian format of the data. Once set, this flag must remain the same throughout the flow.
  - 1 = big Endian
  - 0 = little Endian
- `LEN` (Data Length in Bytes) – Number of bytes of the data field in the current command. This length does not include the `CMD`, `SEQ`, `CID`, `Flags`, or `LEN` fields. It is the total length (in bytes) of the data in the data field only. The `LEN` field itself is 4 bytes long, and is in little-endian format.
- `Data` – Data field associated with the current command. The number of bytes of this field must equal the `LEN` value above. If `LEN` is zero, then this field does not exist. On a word (32-bit) basis, the default configuration is to send the data in little-endian format.

## 9.3 List of Commands

Table 23 lists all commands sent by the host.

**Table 23: Host Commands**

| Commands | CMD | SEQ | LEN | Data | Comment |
|---|---|---|---|---|---|
| Public Key | 0x24 | 0 | 0 | None | Indicates that the next command is a data command containing the public key |
| Password | 0x28 | 0 | 0 | None | Tells the target to send a 64-bit password |
| Signed Password | 0x25 | 0 | 0 | None | Indicates that the next command is a data command containing the signed password |
| Get Version | 0x20 | 0 | 0 | None | Tells the target to send the version information |
| Select Image | 0x26 | 0 | 0 | None | Tells the target to respond with the image type to be downloaded |
| Verify Image | 0x27 | 0 | 1 | 0 = ACK<br>1 = NACK | Tells the target whether the image type asked for in Select Image is available |
| Data Header | 0x2a | y | 4 | Size | Tells the target how much data is left to be downloaded |
| Data | 0x22 | y | x | Data | Sends the target the next block of data |
| Message | 0x2b | 0 | 0 | None | Tells the target to send its message |
| Done | 0x30 | 0 | 0 | None | Tells the target that the current flow is complete, yet more images are available for download |
| Disconnect | 0x31 | 0 | 0 | None | Tells the target that the current flow is complete and there are no more images left to download |

x: `LEN` value is variable: `SEQ` number is incremental

**Note**

The `CID` is not listed in Table 23 because it is unique to each flow.

## 9.4 Structure of Status Responses

The structure of all status responses sent by the target follows this format:

```
struct Status
{
    ByteCMD
    ByteSEQ
    ByteCID
    ByteStatus
    ByteFlags
    ByteLEN
    Byte[Len]Data
}
```

- `CMD` – Same opcode as the command this response packet is acknowledging.
- `SEQ` (Sequence) – Used during data transmission in response to a data command to keep the host and target in synchronization. The sequence number is `1` for the first data transmission, `2`

for the second, and so on. Since the size of the sequence field is 1 byte, the sequence number rolls over after 255 data transmissions.

- `CID` (Command ID) – Specific number that relates all of the commands (and responses) of a single flow. A flow is the communication from the preamble to the “done” acknowledgement. The host defines the `CID` when it sends the first command after the preamble. The same `CID` is used until the “done” command after a download or a JTAG re-enablement. If another download follows, the host must generate a new `CID` for the next download flow (after the next preamble).
- `Status` – Status code of the target in response to the last command sent by the host.
- Flags – Bits [7:2] are reserved.
  Bit 0 – Message Flag. Tells the host that the target needs to send a message. The next command the host should send is a message command. The target lowers this flag when no messages remain in the queue.
  - `1 =` message waiting to be sent
  - `0 =` no messages

  Bit 1 – Message Type. This flag is applicable only when sent in a message response packet (the response packet `CMD` is `0x2B`). This flag tells the target whether data in the data field is an ASCII string or an integer value representing an error code.
  - `1 =` integer error code
  - `0 =` ASCII string

  For additional information about messaging, see Section 9.6, Messages.
- `LEN` (Data Length in Bytes) – Size of the data field of the current response. It is the total length (in bytes) of the data in the data field only. The maximum value of `LEN` is 255 bytes.
- `Data` – Data field associated with the current response. The number of bytes of this field must equal the `LEN` value above. If `LEN` is zero, then this field does not exist. On a word (32-bit) basis, this data is in little-endian format. The maximum size of the data field is 255 bytes.

## 9.5 Responses

Every command sent by the host requires the target to respond with a status packet. Some of the responses require data in the data field while others do not. Table 24 describes the contents of the data field for each response packet.

**Table 24: Target Responses**

| Commands | CMD | LEN | Data |
|---|---|---|---|
| Public Key | 0x24 | 0 | No data needed |
| Password | 0x28 | 8 | A 64-bit password |
| Signed Password | 0x25 | 0 | No data needed |
| Get Version | 0x20 | 12 | Version information<br>The first 4 bytes are ASCII characters and represent the target stepping version. The second 4 bytes is an integer capturing the date. The last 4 bytes are ASCII characters and represent the type of processor. |
| Select Image | 0x26 | 4 | Image Identifier |
| Verify Image | 0x27 | 0 | No data needed |
| Data Header | 0x2a | 4 | A 32-bit integer that tells the host how much data to send in the next Data command |

Table 24: Target Responses (Continued)

| Commands | CMD | LEN | Data |
|---|---|---|---|
| Data | 0x22 | 0 | No data needed |
| Message | 0x2b | x | ASCII string<br>This is a message that the target wants printed for the user. |
| Done | 0x30 | 0 | No data needed |
| Disconnect | 0x31 | 0 | No data needed |

## 9.6 Messages

At any time during the communication process, the target may send a text message to the host by the target raising Bit 0 of the flag field. The host should then send the message command as the following command.

**Note**

The host is not required to send the message command as soon as the message flag has been raised. The target keeps the message in the queue and the message flag bit raised until the message command is sent and the message has been handled.

## 9.7 Disconnect

After the target has finished downloading all of the images, the host issues the disconnect command. The target does not respond to the command until it has finished its operations, which allows the target to fill up the message queue with any messages needed to be sent to users.

Once the target issues the response packet to the disconnect command, the host must check the message flag. If the flag is not set, the host shuts down and the target transfers control. However, if the flag is set, the host must continue issuing message commands until the message flag is lowered. The host should ignore the status field during this sequence.

## 9.8 Status Codes

Table 25 describes the current status codes communicated back to the host application.

Table 25: Status Codes

| Error Code | Description |
|---|---|
| 0x00 | ACK |
| 0x01 | NACK |
| 0x02 | Sequence error |

# A Return Code Definitions

When the Boot ROM encounters an error, it logs it in Internal memory at address 0xD102005C**.**The following list contains the return codes and definitions.

| **/** General Error Code Definitions **/** | **0x0 - 0x1F** |
|---|---|
| NoError | 0x0 |
| NotFoundError | 0x1 |
| GeneralError | 0x2 |
| WriteError | 0x3 |
| ReadError | 0x4 |
| NotSupportedError | 0x5 |
| InvalidPlatformConfigError | 0x6 |
| PlatformBusy | 0x7 |
| PlatformReady | 0x8 |
| InvalidSizeError | 0x9 |
| ProbeListExhaustError | 0xA |
| DDR_NotInitializedError | 0xB |
| PlatformDisconnect | 0xC |
| **/** Flash Related Errors **/** | **0x20 - 0x3F** |
| EraseError | 0x20 |
| ProgramError | 0x21 |
| InvalidBootTypeError | 0x22 |
| ProtectionRegProgramError | 0x23 |
| NoOTPFound | 0x24 |
| BBTReadError | 0x25 |
| MDOCInitFailed | 0x26 |
| OneNandInitFailed | 0x27 |
| MDOCFormatFailed | 0x28 |
| BBTExhaustedError | 0x29 |
| NANDNotFound | 0x2A |
| SDMMCNotFound | 0x2B |
| FlexOneNANDNotFound | 0x2C |
| SDMMCReadError | 0x2D |
| XIPReadError | 0x2E |

| | |
|---|---|
| FlexOneNANDError | 0x2F |
| FlashDriverInitError | 0x30 |
| FlashFuncNotDefined | 0x31 |
| OTPError | 0x32 |
| InvalidAddressRangeError | 0x33 |
| FlashLockError | 0x34 |
| ReadDistrurbError | 0x35 |
| FlashReadError | 0x36 |
| SPIFlashNotResponding | 0x37 |
| ImageOverlayError | 0x38 |
| | |
| **/** NFC Related Errors **/** | **0x40 - 0x5F** |
| NFCDoubleBitError | 0x40 |
| NFCSingleBitError | 0x41 |
| NFCCS0BadBlockDetected | 0x42 |
| NFCCS1BadBlockDetected | 0x43 |
| NFCInitFailed | 0x44 |
| NFCONFIConfigError | 0x45 |
| NFC_WRREQ_TO | 0x46 |
| NFC_WRCMD_TO | 0x47 |
| NFC_RDDREQ_TO | 0x48 |
| NFC_RDY_TO | 0x49 |
| NFCCS0CommandDoneError | 0x4A |
| NFCCS1CommandDoneError | 0x4B |
| NFC_PGDN_TO | 0x4C |
| **/** NTIM Related Errors**/** | |
| InvalidTIMImageError | 0x6A |
| TIMNotFound | 0x6D |
| **/** Download Protocols **/** | **0x90 - 0xAF** |
| DownloadPortError | 0x90 |
| DownloadError | 0x91 |
| FlashNotErasedError | 0x92 |
| InvalidKeyLengthError | 0x93 |
| DownloadImageTooBigError | 0x94 |
| UsbPreambleError | 0x95 |
| TimeOutError | 0x96 |
| UartReadWriteTimeOutError | 0x97 |

| | |
|---|---|
| UnknownImageError | 0x98 |
| MessageBufferFullError | 0x99 |
| NoEnumerationResponseTimeOutError | 0x9A |
| UnknownProtocolCmd | 0x9B |
| UsbRxError | 0x9C |
| | |
| **/** JTAG ReEnable Error Codes **/** | **0xB0 - 0xCF** |
| JtagReEnableError | 0xB0 |
| JtagReEnableOEMPubKeyError | 0xB1 |
| JtagReEnableOEMSignedPassWdError | 0xB2 |
| JtagReEnableTimeOutError | 0xB3 |
| JtagReEnableOEMKeyLengthError | 0xB4 |
| **/** SD/MMC Error **/** | |
| SDMMC_SWITCH_ERROR | 0xD0 |
| SDMMC_ERASE_RESET_ERROR | 0xD1 |
| SDMMC_CIDCSD_OVERWRITE_ERROR | 0xD2 |
| SDMMC_OVERRUN_ERROR | 0xD3 |
| SDMMC_UNDERUN_ERROR | 0xD4 |
| SDMMC_GENERAL_ERROR | 0xD5 |
| SDMMC_CC_ERROR | 0xD6 |
| SDMMC_ECC_ERROR | 0xD7 |
| SDMMC_ILL_CMD_ERROR | 0xD8 |
| SDMMC_COM_CRC_ERROR | 0xD9 |
| SDMMC_LOCK_ULOCK_ERRROR | 0xDA |
| SDMMC_LOCK_ERROR | 0xDB |
| SDMMC_WP_ERROR | 0xDC |
| SDMMC_ERASE_PARAM_ERROR | 0xDD |
| SDMMC_ERASE_SEQ_ERROR | 0xDE |
| SDMMC_BLK_LEN_ERROR | 0xDF |
| SDMMC_ADDR_MISALIGN_ERROR | 0xE0 |
| SDMMC_ADDR_RANGE_ERROR | 0xE1 |
| SDMMCDeviceNotReadyError | 0xE2 |
| SDMMCInitializationError | 0xE3 |
| SDMMCDeviceVoltageNotSupported | 0xE4 |
| SDMMCWriteError | 0xE5 |

# B Acronyms and Abbreviations

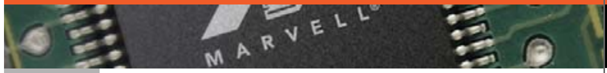
Table B-1: Acronyms and Abbreviations

| Acronym | Definition |
|---|---|
| BSP | Board Support Package |
| CRC | Cyclic Redundancy Check |
| DFI | Data Flash Interface |
| ECC | Error-Correcting Code |
| FFUART | Full-Featured Universal Asynchronous Receiver-Transmitter |
| HWR | Hardware Reset |
| NFC | NAND Flash Controller |
| NTIM | Non-Trusted Image Module |
| OBM | OEM Boot Module |
| OEM | Original Equipment Manufacturer |
| ONFI | Open NAND Flash Interface |
| OSBM | OEM System Boot Module |
| OTP | One Time Programmable |
| POR | Power On Reset |
| RNG | Random Number Generator |
| SBE | Secure Boot Enable |
| SDE | Secure Download Enable |
| TIM | Trusted Image Module |
| USB | Universal Serial Bus |
| WDR | Watchdog Reset |
| WTM | Wireless Trusted Module |
| WTP | Wireless Trusted Platform |
| WTPSP | Wireless Trusted Platform Service Package |

# C Revision History

Table C-1: Revision History

| Revision | Major Changes to Document |
|---|---|
| PUBLIC RELEASE | • Initial Release (October 2010) |



Marvell Semiconductor, Inc.
5488 Marvell Lane
Santa Clara, CA 95054, USA

Tel: 1.408.222.2500
Fax: 1.408.752.9028

www.marvell.com

**Marvell. Moving Forward Faster**