Intel® Axxia™ AXM5600 Communication Processor Boot Process

Application Note

Rev. 1.1

May 2016

Doc. No.: 561644, Rev.: 1.1

Legal and Disclaimers

Contents

Revision History

Date	Revision	Description
May 2016	1.1	2.2.2 BL2 on page 10: Updated graphic. 2.2.4 BL3-3 on page 13: Updated graphic. 9.0 Downloading and Building Boot Loader Images on page 37: Updated file paths, software version, and graphic information. 9.1 Prerequesites on page 39: Updated file paths and software version information. 9.2 Download the U-Boot Source Code on page 40: Updated file paths and software version information. 9.3 Download the ARM Trusted Firmware Source Code on page 40: Updated file paths and software verison information. 9.5 Build the ARM Trusted Firmware on page 41: Updated file paths and software version information. 10.1 Creating a Text Version of the Parameter File Using ASE on page 44: Updated publication version information.
January 2016	1.0	: Applied Intel branding. 2.2.4 BL3-3 on page 13: Updated list of U-Boot commands. 3.2 BL2/BL3-1 and BL3-3 on page 16: Changed tool for generating Secure Boot header/ trailer to `sbb_mk_img` and made some other updates to this section. 3.3 mkimage Header Options For BL2/BL3-1 Image on page 18: Added this section. 4.5 Enabling and Disabling Redundant Boot Features on page 23: Added information about Redundant Boot selection using the RS2B bit. 5.0 Secure Boot Process on page 24: Updated section. 9.0 Downloading and Building Boot Loader Images on page 37: Reorganized section. 9.1 Prerequesites on page 39: Updated tags for u-boot and for ARM Trusted Firmware. 9.2 Download the U-Boot Source Code on page 40: Updated file paths and software version information. 9.3 Download the ARM Trusted Firmware Source Code on page 40: Added step about creating a local copy of the tagged version. 9.5 Build the ARM Trusted Firmware on page 41: Updated several steps. 9.6 Build U-Boot and u-boot-spl on page 42: Updated several steps. 9.6 Build U-Boot and u-boot-spl on page 42: Updated file paths and software version information. 10.1 Creating a Text Version of the Parameter File Using ASE on page 44: Updated *ASE Screen Capture* and *config-uboot.txt File* figures. 10.2 Creating a Binary Version of the Parameter File Using RTE on page 46: Changed tool used to generate boot image files to ncpBootParamGen. 11.3 Programming Serial Flash Using DStream on page 50: Added placeholder for this section.
February 2015	0.1	Initial document release.

1.0 Boot Process Introduction

The Intel® Axxia™ AXM5600 communication processor includes an ARM* Cortex*-A57 CPU complex with up to 16 internal CPUs. The AXM5600 device uses a multi-stage process to boot these internal CPUs. This document describes the typical boot process when booting by using the internal Cortex-A57 CPUs. The internal boot flow is the most commonly used method in customer applications. The AXM5600 device also supports booting from an external host processor connected through the PCIe* or sRIO interfaces. This mode is not described in detail in this document. Refer to the *Intel Axxia AXM5600 Communication Processor CPU Complex Hardware Reference Manual* for more details on additional boot modes.

Intel provides all software packages required to boot the AXM5600 from a device reset to loading of a Linux* OS. Key features of the boot software for AXM5600 include the following.

- Uses the U-Boot boot loader
- Uses the ARM Trusted Firmware (ATF) framework
- Provides Redundant Boot Image support using two copies of each boot image for failsafe boot and in-field software updates
- Provides Secure Boot Image support via image authentication, decryption, and revision checking
- Supports DDR Data Retention
- Includes a Secure Monitor

2.0 Boot Flow Overview

The boot flow for the AXM5600 is based on the ATF framework. U-Boot and the ATF reference implementation code are used to implement the various firmware packages required for device boot. The following sections describe the AXM5600 boot flow.

2.1 Mapping ATF Stages to the Boot Flow

The ARM Trusted Firmware (ATF) framework provides an implementation framework for "secure world" software on ARMv8-A processors, such as the Cortex-A57 used in the AXM5600. This includes various bootloader software stages and a Secure Monitor. Figure 1 on page 7 shows a block diagram of ATF stages relevant to the AXM5600 bootloader. Figure 2 on page 8 shows these stages mapped to a complete boot flow from device reset until applications are running in an OS environment.

Figure 1. ARM Trusted Firmware - AXX5600 Bootloader Stages

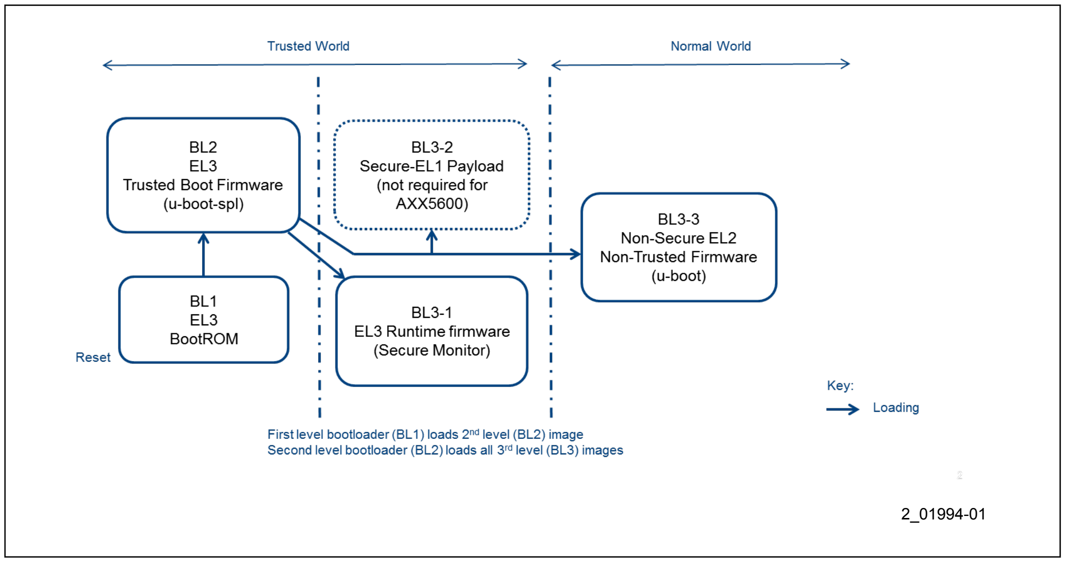
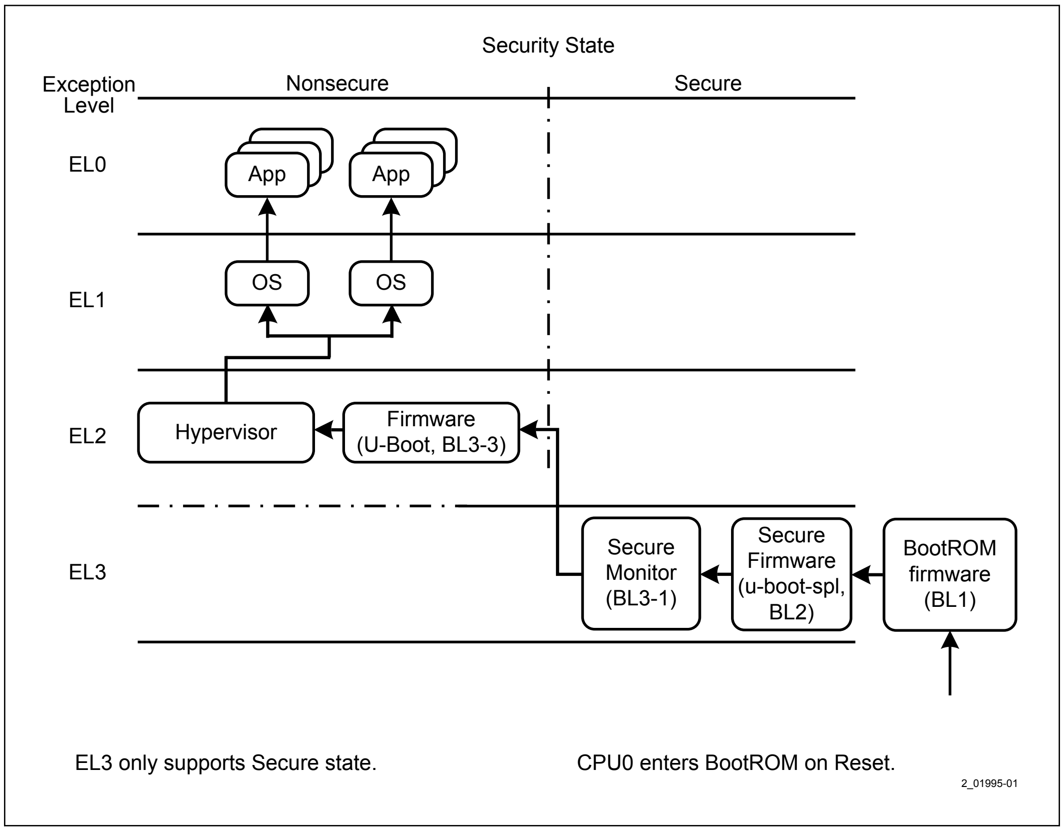


Figure 2. AXX5600 Boot Flow



ATF defines multiple Bootloader Levels. For AXM5600, 3 levels are used, BL1 – BL3. Within BL3, multiple stages are possible. On AXM5600, stages BL3-1 and BL3-3 are used. Each level is implemented using the following firmware packages.

- BL1 – On-chip bootROM
- BL2 – u-boot-spl
- BL3-1 – Secure Monitor firmware
- BL3-3 – U-Boot

2.2 Bootloader Levels

A brief description of the different Bootloader Levels as implemented for the AXM5600 is given below. BL1, BL2, BL3-1 and BL3-3 are executed by CPU0, and all other CPUs are held in reset. All other CPU cores are released from reset by a subsequent software stage, for example an OS image or a hypervisor. Typically, a hypervisor or an OS would make a request to the Secure Monitor using a Secure Monitor Call to release other CPU cores.

All bootloader firmware uses the Aarch64 instruction set and CPU0 is in the Aarch64 execution state.

2.2.1 BL1

BL1 is the first boot step executed after a power-on reset, a System Reset or a Chip Reset on AXM5600. It is stored in an on-chip bootROM. It contains low-level code to initialize the AXM5600 and load the next boot stage (BL2). BL1 can operate in all customer applications regardless of differences in system configuration, such as DDR memory size and PLL speed settings. BL1's main task is to load BL2 from a serial flash device connected to chip select 0 of the AXM5600's Synchronous Serial Port (SSP) into Local Secure Memory, an on-chip 256KB RAM block.

To enter BL1, the AXM5600 device is configured for an Internal Boot with the CPU reset vector set to address 0x80FFFF0000. Internal Boot is selected if the bootMode bit in the Latch On Reset Configuration Register (LOR) is set to 0. The CPU reset vector is set to address 0x80FFFF0000 if the bootTarget bit in the LOR is set to 1. After a power-on reset or a System Reset, these bits contain the values of the BOOT_MODE and BOOT_TARGET pins.

The following conditions must be met to enable the AXM5600 device to execute the bootROM code after a power-on or System Reset.

- The BOOT_MODE pin must be tied to logic 0.
- The BOOT_TARGET pin must be tied to logic 1.
- A 125-MHz clock signal must be present on clk_ref inputs.
- A serial flash device must be connected to chip select 0 of the SSP. Intel uses a 25FL128 serial flash device in the AXM5600 Evaluation Kit. Any serial flash that uses standard 25-series commands for reading serial flash contents can be used.

The bootMode and bootTarget bits in the LOR can be modified by software during runtime. This can be used to change the chip configuration after a Chip Reset event – Chip Reset does not latch the BOOT_MODE and BOOT_TARGET pins, instead it reads the bootMode and bootTarget bits from LOR directly.

The bootROM performs the following sequence of tasks.

1. Zero the contents of Local Secure Memory (LSM).
2. Initialize the Secure Boot Block (SBB) and eFuse Controller Module (ECM) peripherals.
3. Enable the synchronous serial port (SSP).
4. Load a combined BL2/BL3-1 image from serial flash connected to SSP chip select 0 into LSM.
    a. If the Redundant Boot feature is enabled, BL2/BL3-1 images A and B will be verified and the chosen file will be loaded into LSM.
    b. If Secure Boot is enabled, BL2/BL-3 image will be processed by the SBB peripheral.
5. CPU0 jumps to the BL2 image entry point at the first address in LSM.

Note: On AXM5600, the Secure Monitor (BL3-1) binary is loaded into LSM at the same time as u-boot-spl(BL2) binary – both binaries are merged at build time. See Build U-Boot and u-boot-spl on page 42. This reduces the number of separate images that must be loaded.

2.2.2 BL2

BL2 is implemented using the U-Boot Secondary Program Loader (`u-boot-spl`) software package. This is a cut-down version of the U-Boot bootloader. Its main tasks are to initialize system (DDR) memory, system clocks and PLLs, and load the BL3-3 image into system memory starting at address 0x00000000. `U-boot-spl` executes in an on-chip SRAM block, the Local Secure memory (LSM). The size of this memory block is 256KB, hence `u-boot-spl` must fit within a small memory footprint. `U-boot-spl` is personalized to individual applications and systems using a parameter file and U-Boot environment variables, which are both stored in serial flash. The parameter file is loaded after the `u-boot-spl` binary image has been loaded and started.

At time of publication, the following settings can be specified in the parameter file.

- Clock PLL speed settings
- PCIe* and sRIO SerDes lane mapping
- AVS (operating voltage level)
- DDR memory parameter settings (size, speed, etc.)
- DDR memory retention enable/disable
- DDR memory range and MBIST test feature selection
- UART baud rate selection. This value is used to set the baud rate for BL3-3 (U-Boot). BL2 (u-boot-spl) uses a baud rate defined in the U-Boot source code.
- Redundant Boot Sequence number. Each parameter file has a sequence number that should be increased by one each time a new parameter file is programmed into serial flash. When the Redundant Boot feature is enabled, `u-boot-spl` uses the sequence number to determine if the parameter file being loaded is a newer version or an older version.
- Description Field. Each parameter file can contain a Description field that contains ascii text. U-Boot will print this text at boot time.

If the Redundant Boot feature is enabled, `u-boot-spl` can select between two different parameter files – A and B.

A text version of the parameter file is generated using the ASE software tool. This text file can be edited in any text editor. The file is then compiled using the `ncpBootMem` command-line tool, supplied in the RTE software package. See Creating a Parameter File on page 44 for a description of the steps required to create and compile a parameter file.

`u-boot-spl` executes using CPU0, and all other CPUs are held in reset. `u-boot-spl` performs the following sequence of tasks.

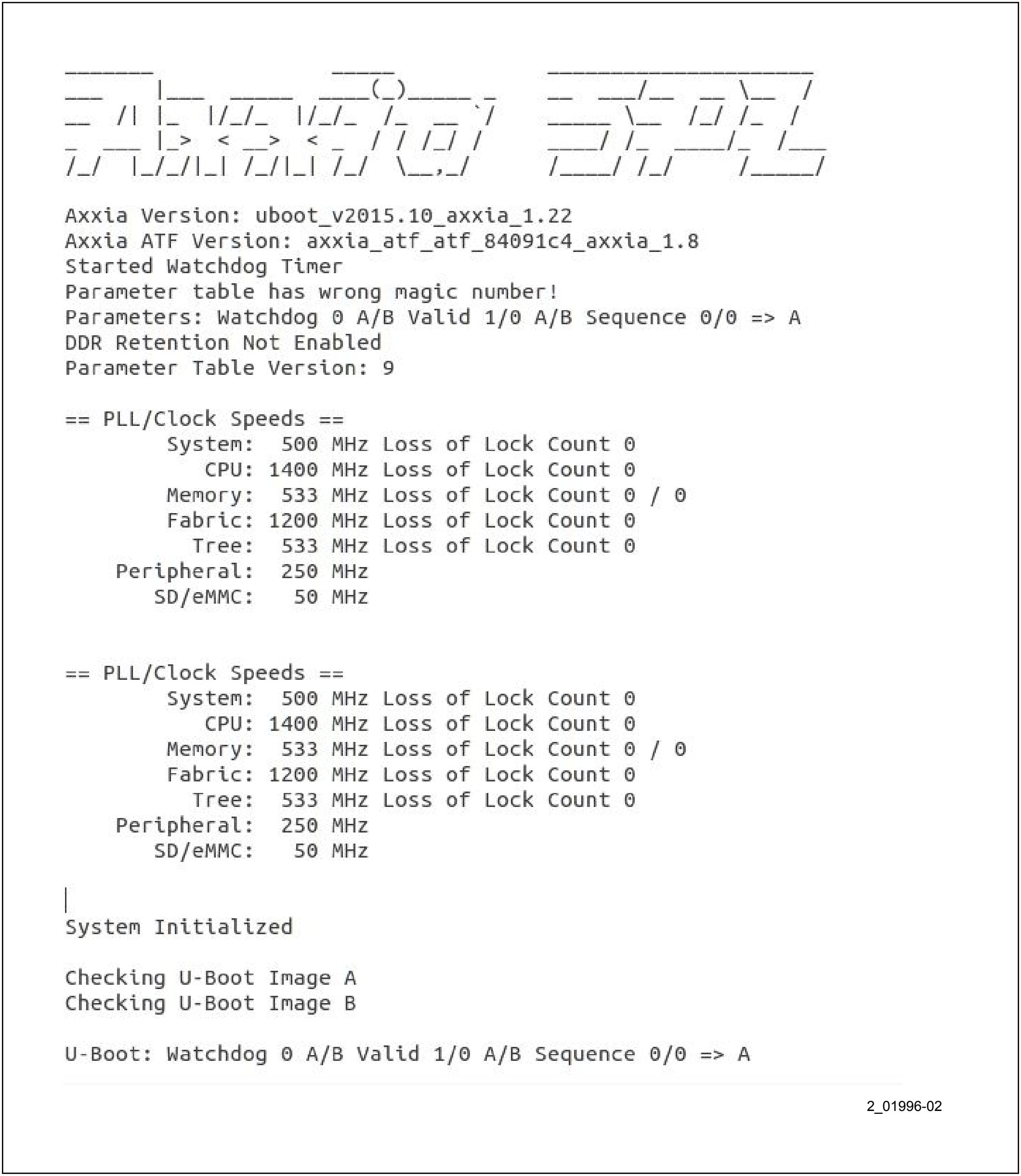
1. Initializes UART0 to allow printing of the boot text. (Note that `u-boot-spl` does not have a command prompt, therefore, it does not accept command-line instructions.)
2. Reads the parameter file from the serial flash device connected to chip select 0 of SSP.
3. If the Redundant Boot feature is enabled, `u-boot-spl` selects one of two available parameter files.
4. Sets the AXM5600 device operating voltage level (AVS).

5. Initializes the system clocks and the associated PLLs.
6. Initializes the DDR system memory.
7. Configures the PCIe, sRIO, and USB3 SerDes lanes.
8. Initiates any system memory tests that are specified in the parameter file.
9. Loads the BL3-3 image from a serial flash and stores it in system memory starting at address 0x00000000. If the Redundant Boot feature is enabled, `u-boot-spl` selects one of two available BL3-3 images (A and B).
10. Optionally requests Secure Boot processing of the BL3-3 image from the Secure Boot Block (SBB) peripheral.
11. Jumps to the BL3-1 Secure Monitor which is also loaded in LSM.

If the DDR memory retention feature has been selected in the parameter file, and the last reset event was a Chip Reset, `u-boot-spl` will not initialize the attached external system memory SDRAMs (SMEM0 and SMEM1), their associated memory controllers, and DDR3/DDR4 PHY logic. This allows the contents of system memory to be retained after a Chip Reset. This can be useful for debugging reset events, or recovering from an unexpected reset.

The following figure shows the boot text from a typical Stage 2 process.

Figure 3. BL2 `u-boot-spl` Boot Text



```
_______              _____        _____________________
___    |___  _____  ____(_)_____ _   __  ___/__  __ \__  /
__  /| |_  |/_/_  |/_/_  /_  __ `/   _____ \__  /_/ /_  /
_  ___ |_>  <  __>  <  _  / / /_/ /   ____/ /_  ____/_  /___
/_/  |_/_/|_| /_/|_| /_/  \__,_/    /____/ /_/     /_____/

Axxia Version: uboot_v2015.10_axxia_1.22
Axxia ATF Version: axxia_atf_atf_84091c4_axxia_1.8
Started Watchdog Timer
Parameter table has wrong magic number!
Parameters: Watchdog 0 A/B Valid 1/0 A/B Sequence 0/0 => A
DDR Retention Not Enabled
Parameter Table Version: 9

== PLL/Clock Speeds ==
        System:  500 MHz Loss of Lock Count 0
           CPU: 1400 MHz Loss of Lock Count 0
        Memory:  533 MHz Loss of Lock Count 0 / 0
        Fabric: 1200 MHz Loss of Lock Count 0
          Tree:  533 MHz Loss of Lock Count 0
    Peripheral:  250 MHz
       SD/eMMC:   50 MHz


== PLL/Clock Speeds ==
        System:  500 MHz Loss of Lock Count 0
           CPU: 1400 MHz Loss of Lock Count 0
        Memory:  533 MHz Loss of Lock Count 0 / 0
        Fabric: 1200 MHz Loss of Lock Count 0
          Tree:  533 MHz Loss of Lock Count 0
    Peripheral:  250 MHz
       SD/eMMC:   50 MHz

|
System Initialized

Checking U-Boot Image A
Checking U-Boot Image B

U-Boot: Watchdog 0 A/B Valid 1/0 A/B Sequence 0/0 => A
```

2_01996-02

2.2.3 BL3-1

BL3-1 is the Secure Monitor firmware. It includes code which allows switching CPU cores between Secure and Non-Secure states, and routines to allow powerup/powerdown of CPU cores and other device functions using the ARM Power State Coordination Interface (PSCI). Referring to Figure 2 on page 8, note that BL1 – BL2 execute in Secure State, whereas BL3-3 executes in the Non-Secure state. The Secure Monitor provides the switch from Secure to Non-Secure that is required for the boot flow to move from BL2 to BL3-3. The Secure Monitor is the only software on the target system that can perform the Secure/Non-Secure transition. After the transition, CPU0 will begin executing code from address 0x0000000000.

Note: On AXM5600, the Secure Monitor (BL3-1) binary is loaded into LSM at the same time as the `u-boot-spl` (BL2) binary – both binaries are merged at build time. See Build U-Boot and u-boot-spl on page 42. This reduces the number of separate images that must be loaded.

The Secure Monitor remains resident in LSM after the boot sequence has completed. This allows hypervisors and OS's to make calls to the Secure Monitor to request features such as CPU core powerup/powerdown using the ARM Secure Monitor Call (SMC) calling convention.

2.2.4 BL3-3

BL3-3 is implemented using a full-featured version of the U-Boot bootloader, which executes from system memory. Typically, a system will have megabytes or gigabytes of system memory, so U-Boot can have many more features than BL2 (`u-boot-spl`). U-Boot's main task is to allow loading of subsequent software stages (OS images, hypervisor images, etc.)

The following U-Boot features are included in the AXM5600 release.

- A command-line interface through UART0. (See the list of U-Boot commands below for details.)
- Environment variables for storing command sequences and configuration parameters. Two copies of the environment variables may be used to provide redundant back-up in case one set becomes corrupted.
- Support for booting SMP Linux* images.
- Support for loading files and images using GEMAC (10/100/1G) and EIOA (10/100/1000/10G) Ethernet ports.
- Support for loading images from serial flash and USB devices.
- Support for Device Tree files (read/modify/pass to OS, etc.)
- PCIe* bus configuration space read/write functions.
- I^{2}C bus read/write functions.

The following U-Boot commands are supported with the command-line interface via UART0.

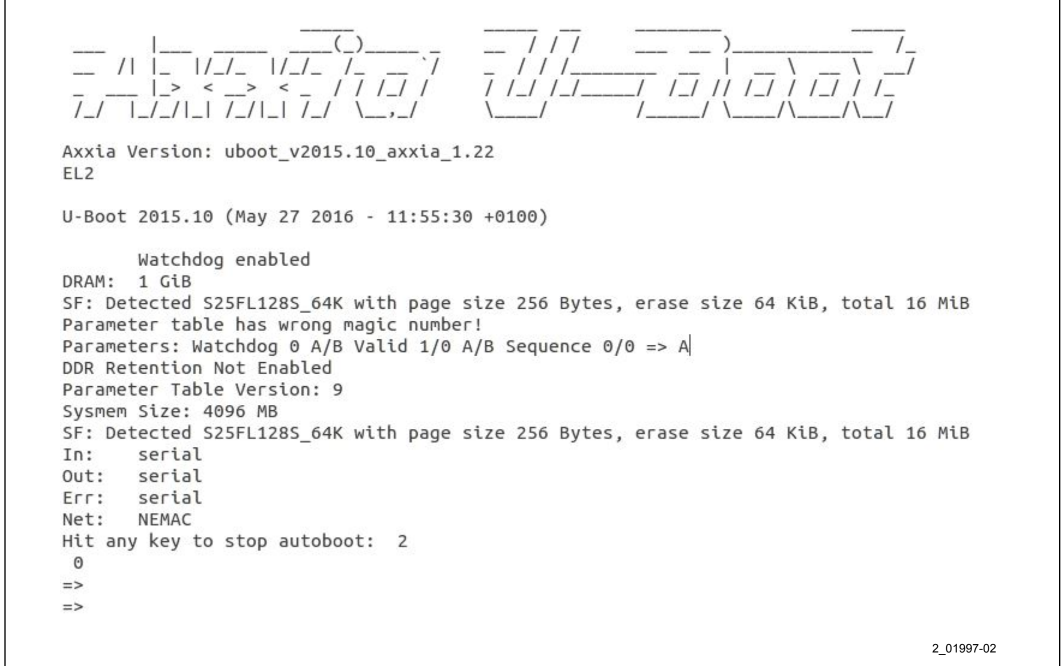
```
?       - alias for 'help'
base    - print or set address offset
bdinfo  - print Board Info structure
boot    - boot default, i.e., run 'bootcmd'
bootd   - boot default, i.e., run 'bootcmd'
bootm   - boot application image from memory
bootp   - boot image via network using BOOTP/TFTP protocol
cmp     - memory compare
coninfo - print console devices and information
cp      - memory copy
crc32   - checksum calculation
dhcp    - boot image via network using DHCP/TFTP protocol
echo    - echo args to console
editenv - edit environment variable
env     - environment handling commands
erase   - erase FLASH memory
```

```
ext2load- load binary file from a Ext2 filesystem
ext2ls  - list files in a directory (default /)
ext4load- load binary file from a Ext4 filesystem
ext4ls  - list files in a directory (default /)
ext4size- determine a file's size
fdt     - flattened device tree utility commands
flinfo  - print FLASH memory information
go      - start application at address 'addr'
gpio    - query and control gpio pins
help    - print command description/usage
i2c     - I2C sub-system
iminfo  - print header information for application image
imxtract- extract a part of a multi-image
itest   - return true/false on integer compare
loadb   - load binary file over serial line (kermit mode)
loads   - load S-Record file over serial line
loadx   - load binary file over serial line (xmodem mode)
loady   - load binary file over serial line (ymodem mode)
loop    - infinite loop on address range
md      - memory display
mdio    - MDIO utility commands
mii     - MII utility commands
mm      - memory modify (auto-incrementing address)
mtest   - simple RAM read/write test
mw      - memory write (fill)
nm      - memory modify (constant address)
pci     - list and access PCI Configuration Space
ping    - send ICMP ECHO_REQUEST to network host
printenv- print environment variables
protect - enable or disable FLASH write protection
reset   - Perform RESET of the CPU
run     - run commands in an environment variable
saveenv - save environment variables to persistent storage
sbb     - sbb command source destination [size (encrrypt only)] [safe]
setenv  - set environment variables
setexpr - set environment variable as the result of eval expression
sf      - SPI flash sub-system
sleep   - delay execution for some time
source  - run script from memory
sspi    - SPI utility command
tftpboot- boot image via network using TFTP protocol
usb     - USB sub-system
usbboot - boot from USB device
version - print monitor, compiler and linker version
```

U-Boot executes from main system memory, that is, DDR memory, starting from physical address 0x00_0000_0000.

The following figure shows the boot text from U-Boot.

Figure 4. U-Boot Boot Text



```
                     _____            _____  __     ________           _____
 ___    |___  _____  ____(_)_____ _   __  / / /     ___  __ )____________  /_
 __  /| |_  |/_/_  |/_/_  /_  __ `/   _  / / /_________  __  |  __ \  __ \  __/
 _  ___ |_>  <  __>  <  _  / / /_/ /   / /_/ /_/_______/  /_/ // /_/ / /_/ / /_
 /_/  |_/_/|_| /_/|_| /_/  \__,_/    \____/          /_____/ \____/\____/\__/

Axxia Version: uboot_v2015.10_axxia_1.22
EL2

U-Boot 2015.10 (May 27 2016 - 11:55:30 +0100)

       Watchdog enabled
DRAM:  1 GiB
SF: Detected S25FL128S_64K with page size 256 Bytes, erase size 64 KiB, total 16 MiB
Parameter table has wrong magic number!
Parameters: Watchdog 0 A/B Valid 1/0 A/B Sequence 0/0 => A
DDR Retention Not Enabled
Parameter Table Version: 9
Sysmem Size: 4096 MB
SF: Detected S25FL128S_64K with page size 256 Bytes, erase size 64 KiB, total 16 MiB
In:    serial
Out:   serial
Err:   serial
Net:   NEMAC
Hit any key to stop autoboot:  2
 0
=>
=>
```

2_01997-02

3.0 Boot Image File Formats

The AXM5600 boot flow requires the following three different types of files to be programmed into a serial flash device.

- BL2 (`u-boot-spl`) parameter file
- Combined BL2/BL3-1 (`u-boot-spl`/Secure Monitor) image
- BL3-3 (U-Boot) image

3.1 Parameter File

The parameter file contains board specific settings (clock speeds, memory sizes, etc) that `u-boot-spl` uses to initialize the system. It is created using the ASE software tool for AXM5600, supplied by Intel. ASE allows the user to create a text file version of the parameter file which is then converted to a binary file using the RTE software package. The steps to generate this file are described in Creating a Parameter File on page 44.

The parameter file format includes a header which contains CRC and Magic Number files that can be used to determine if the file is valid. The header is added during file creation using the RTE software package.

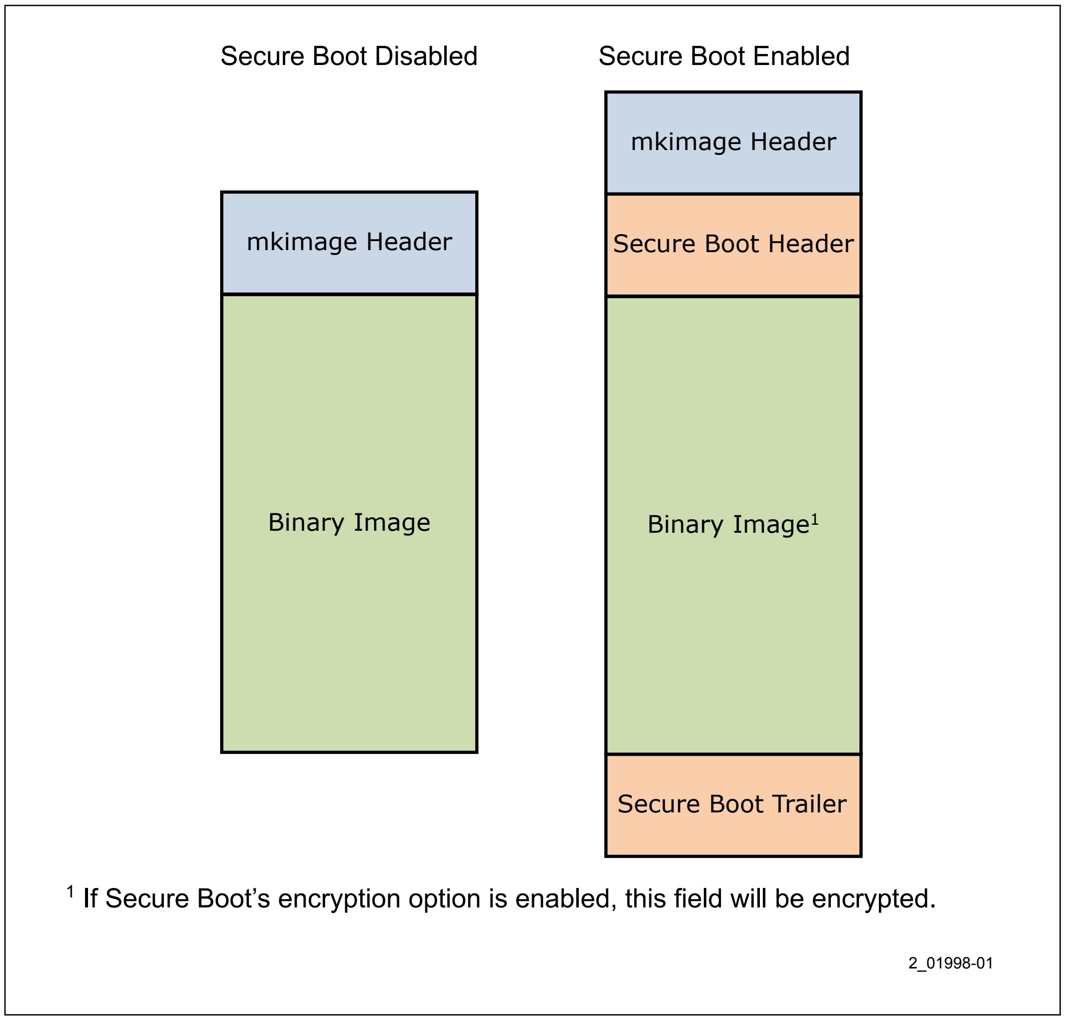
3.2 BL2/BL3-1 and BL3-3

BL2/BL3-1 and BL3-3 bootloader images files can be in one of the two following formats, dependent on whether Secure Boot is enabled or disabled.

Secure Boot Disabled	If Secure Boot is disabled, the image will consist of a binary image with a 64-byte `mkimage` header prepended by the `mkimage` tool.
Secure Boot Enabled	If Secure Boot is enabled, the image will consist of a binary image wrapped in a Secure Boot header/trailer as generated by the `sbb_mk_img` tool (part of the `sbbimg` tool package used to generate SBB keys and images) , with a 64-byte `mkimage` header prepended by the `mkimage` tool.

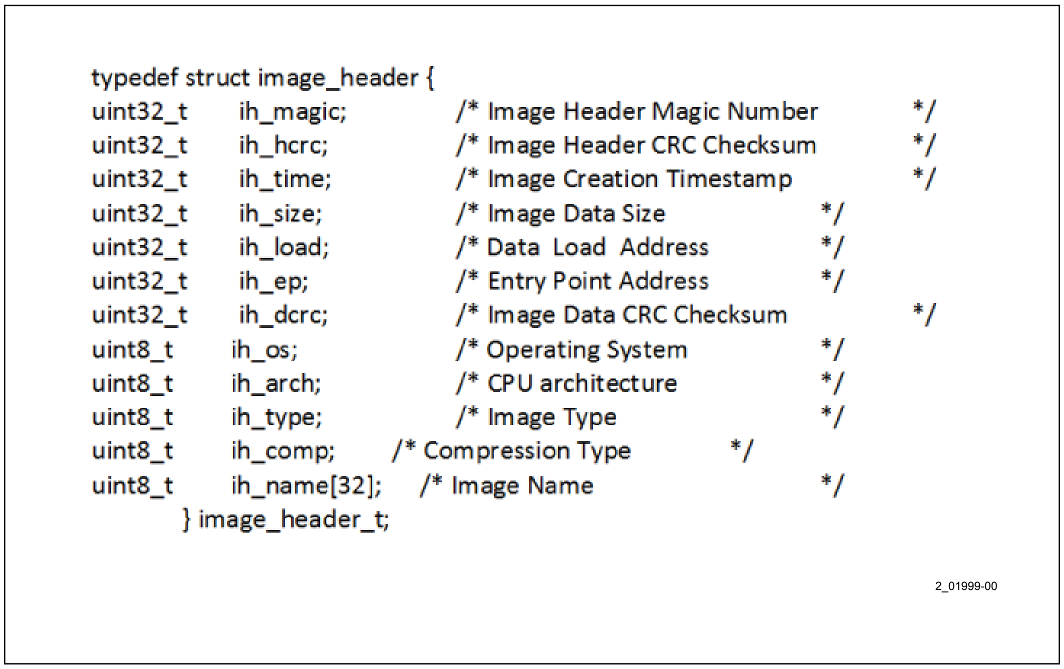
The following figure shows the layout of the two Stage 3 image formats.

Figure 5. Image Formats



The `mkimage` header contains various fields that can be checked by u-boot-spl/u-boot code to determine if the image is valid. Specifically, the `Magic Number`, `CPU Architecture`, and `Image Data CRC Checksum (dcrc)` fields can be checked. The header contents are specified in the include/image.h file in the U-Boot source code and also shown in the following figure.

Figure 6. mkimage Header Definition



```
typedef struct image_header {
uint32_t     ih_magic;          /* Image Header Magic Number          */
uint32_t     ih_hcrc;           /* Image Header CRC Checksum          */
uint32_t     ih_time;           /* Image Creation Timestamp           */
uint32_t     ih_size;           /* Image Data Size             */
uint32_t     ih_load;           /* Data  Load  Address         */
uint32_t     ih_ep;             /* Entry Point Address         */
uint32_t     ih_dcrc;           /* Image Data CRC Checksum            */
uint8_t      ih_os;             /* Operating System            */
uint8_t      ih_arch;           /* CPU architecture            */
uint8_t      ih_type;           /* Image Type                  */
uint8_t      ih_comp;       /* Compression Type          */
uint8_t      ih_name[32];     /* Image Name                  */
        } image_header_t;
```

The Secure Boot header and trailer contain information that can be used to authenticate and optionally decrypt and revision check the boot images. A full description of the Secure Boot header and trailer is given in the *Image Varification and Image Verification Double Read Functions* section of the Secure Boot Block chapter within the *Intel Axxia AXX5600 Communication Processor Security Subsystem Hardware Reference Manual*.

To support Redundant Boot, two copies of each of the above files can be used (A and B). This allows the system to boot if a particular image becomes corrupted. See Redundant Boot Support on page 20.

Programming Boot Images Into Serial Flash on page 48 shows the addresses in serial flash where the various boot images should be stored.

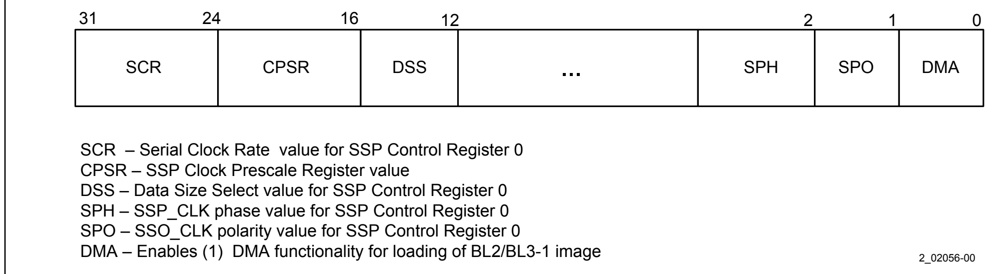
3.3 mkimage Header Options For BL2/BL3-1 Image

The standard definition for an mkimage header is shown in Figure 6 on page 18. The AXM5600 boot flow can use two of the fields in the header for alternate functions for the combined BL2/BL3-1 image:

- The Data Load Address field is used to hold a Sequence Number value for combined BL2/BL3-1 images when Redundant Boot Support is enabled. See Redundant Boot Support chapter for more details on Redundant Boot and Setting Redundant Boot Sequence Numbers on page 21 for details on Sequence Numbers.
- The Entry Point Address field is used to hold configuration information for the SSP. This allows the user to change the speed and configuration of the SPP from the default values used by the boot ROM. The following figure shows the mapping of bits in the 32-bit Entry Point Address field to SSP control fields. Most of the fields are values placed into SSP control registers. The DMA bit allows the user to use

the SSP's DMA function to load BL2/BL3-1 images for higher, or use CPU reads from SSP FIFO to load the image. The DMA option can be faster, but when using the SSP DMA, the file size must be a multiple of 4 bytes and the source and target addresses must be 4-byte aligned.

Figure 7. SSP Control Fields Mapping To Entry Address Field


31
24
16
12
2
1
0
SCR
CPSR
DSS
...
SPH
SPO
DMA
SCR – Serial Clock Rate value for SSP Control Register 0
CPSR – SSP Clock Prescale Register value
DSS – Data Size Select value for SSP Control Register 0
SPH – SSP_CLK phase value for SSP Control Register 0
SPO – SSO_CLK polarity value for SSP Control Register 0
DMA – Enables (1) DMA functionality for loading of BL2/BL3-1 image
2_02056-00

At time of publication, the default value for SSP configuration is 0x00027001 (DMA =On, SPH=0, SPO=0, DSS=7, CPSR=2, SCR=0).

4.0 Redundant Boot Support

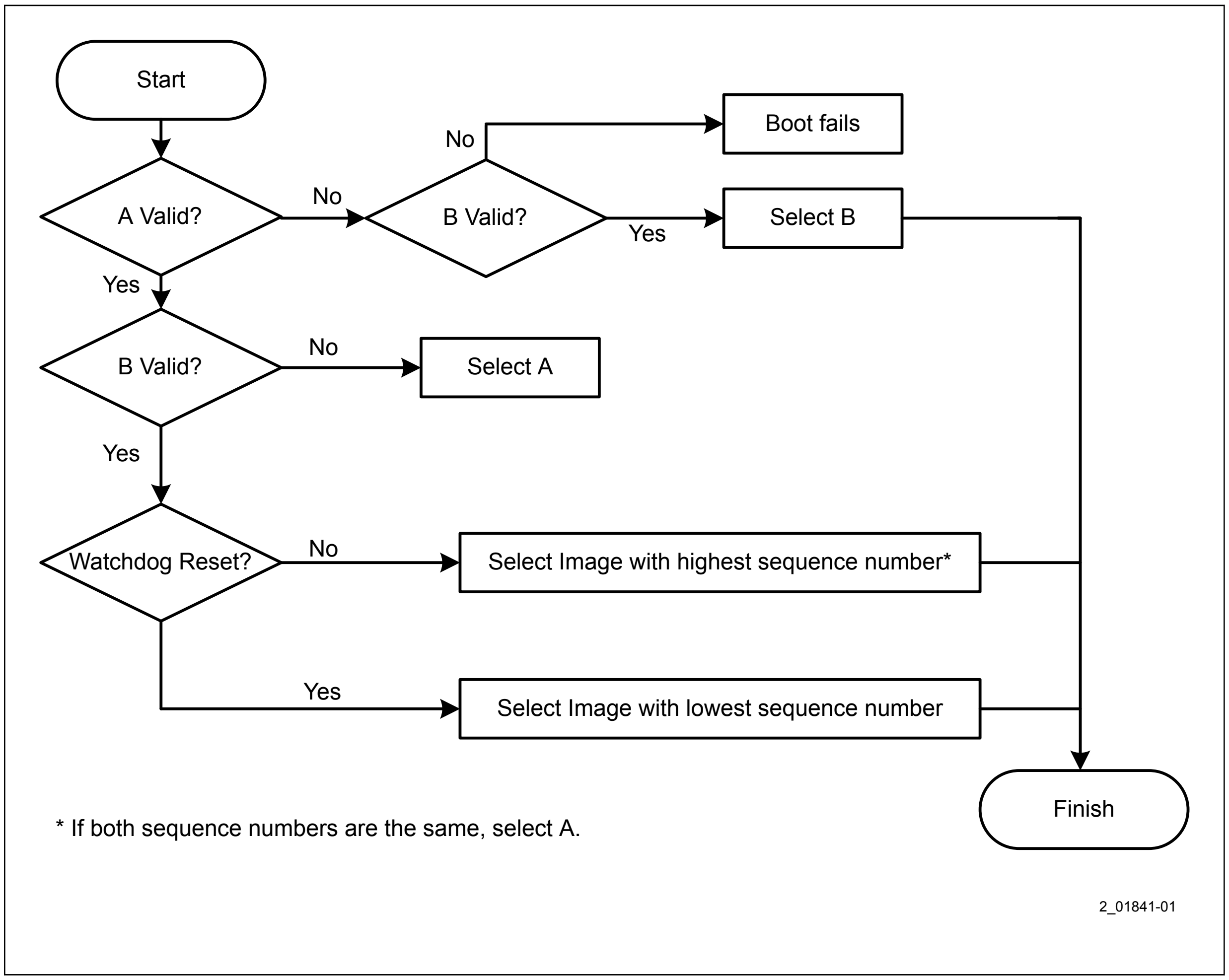
4.1 Redundant Boot Overview

The AXM5600 bootROM, U-Boot and `u-boot-spl` include an optional Redundant Boot Image feature that provides secondary copies of the combined BL2/BL3-1 image (`u-boot-spl.img`), the BL2 parameter file, and the BL3-3 image (`u-boot.img`). It also includes a watchdog capability to check for run-time failures. This provides a fail-safe mechanism for updates to these files, including in-service updates, and allows fallback to a secondary image in case of image corruption or run-time failure. Redundant Boot does not provide extra copies of the U-Boot environment area, because the standard U-Boot already has support for maintaining two copies of the U-Boot environment area.

4.2 Redundant Image Selection Algorithm

Redundant boot uses the algorithm shown in Figure 8 to select between the two copies of combined BL2/BL3-1 image , BL2 parameter file or BL3-3 image. The same algorithm is used each time any of these files needs to be selected. The two copies of each file are labelled ‘A’ and ‘B’. The algorithm checks if each file is valid using values embedded in the files header (see Boot Image File Formats on page 16 for details). Each copy also has an associated Sequence Number which is used to decide which image should be used.

Figure 8. Image Selection Algorithm


Start
A Valid?
No
B Valid?
No
Boot fails
Yes
Select B
Yes
B Valid?
No
Select A
Yes
Watchdog Reset?
No
Select Image with highest sequence number*
Yes
Select Image with lowest sequence number
Finish
* If both sequence numbers are the same, select A.
2_01841-01

The algorithm will produce one of the following results.

- If both images are not valid, the boot will fail.
- If only one image is valid, that image is selected.
- If both images are valid, and the last reset was not caused by a watchdog timeout, the image with the newer sequence number is selected. If both sequence numbers are the same, image A is selected.
- If both images are valid, and the last reset was caused by a watchdog timeout, the image with the older sequence number is selected. If both sequence numbers are the same, image A is selected.

4.3 Setting Redundant Boot Sequence Numbers

Sequence Numbers are 32-bit unsigned hexadecimal numbers with a leading '0x'. Examples of valid numbers are 0x00000001, 0x0000000f, 0xffffffff. U-boot-spl and U-Boot prints the Sequence Number information in boot text. It is recommended to increment the Sequence Number for each new version of a boot image file.

Parameter files have a Sequence Number field embedded in the file. The value of this field can be set using the Axxia Software Environment (ASE) tool during the creation of the parameter file. Refer to Creating a Parameter File on page 44 for details.

Combined BL2/BL3-1 files have a 64 byte `mkimage` header at the start of the file. The `Data Load Address` field in the header contains the Sequence Number (this image type does not require the `Data Load Address` field to contain an address, so it has been reused to hold the Sequence Number).

BL3-3 boot images do not contain any Sequence Number fields. Instead, the sequence numbers for the two Stage 3 boot images (A and B) are stored in two U-Boot environment variables – `uboot_a_sequence` and `uboot_b_sequence`. These variables can be set and stored from the U-Boot command line console using the following commands.

- `setenv uboot_a_sequence <value>`
- `setenv uboot_b_sequence <value>`
- `save`
- `save` *

Note: The second `save` command ensures that the environment variables are stored in both copies of U-Boot's environment variables. It maintains primary and secondary copies of all environment variables.

A special case exists if an image has the Sequence Number 0xffffffff. In this case, the next highest Sequence Number is 0x00000000. This allows the user to rollover the Sequence Numbers and start at 0x00000000 again.

4.4 Watchdog Operation

The AXM5600 device has a system watchdog timer capability. A 32-bit countdown timer (Timer 5 in the Timer Block) can be used to generate System Resets or Chip Resets if the timer is allowed to count down to zero (a watchdog timeout). The `u-boot-spl` and `u-boot` boot loaders use the watchdog hardware to generate a Chip Reset if the watchdog timer's count value is not periodically updated to avoid a watchdog timeout.

The `u-boot-spl` and `u-boot` boot loaders include code to periodically update the watchdog timer. If a watchdog timeout occurs, this indicates that the `u-boot-spl` or `u-boot` boot loader operation has been disrupted and the update code has either failed to execute or has taken too long to perform the update.

The `u-boot-spl` and `u-boot` boot loaders use bit 2 of the AXM5600 device's Persistent Scratch Register (a register than can maintain its contains thru a Chip Reset operation) to log if the last reset was due to a watchdog timeout. Value '1' indicates that a watchdog timeout has occurred. Value '0' indicates that another reset type has occurred.

U-Boot also has an option to leave the watchdog running on entry to the next software stage (typically an OS image or hypervisor image), or to disable the watchdog prior to entry to the next software stage. The method to enable/disable watchdog features is described in Enabling and Disabling Redundant Boot Features on page 23.

4.5 Enabling and Disabling Redundant Boot Features

Redundant Boot selection between the two copies of `u-boot-spl.img` is enabled using a one time programmable eFuse (the RS2B bit) in the AXX56xx device. For more details, refer to the Redundant Boot section of *Intel Axxia AXX5600 Communication Processor Security Subsystem Hardware Reference Manual*. When this bit is enabled, the bootROM selects one of the two images using the algorithm detailed earlier.

Redundant Boot selection between the two parameter files and between the two `u-boot.img` files is controlled by the following #defines in the `include/configs/axm5600.h` file

#define CONFIG_REDUNDANT_PARAMETERS

- This enables/disables the redundant parameter file support. By default this option is enabled.

#define CONFIG_REDUNDANT_UBOOT

- This enables/disables redundant boot support for `u-boot.img` files. By default this option is enabled.

#define CONFIG_HW_WATCHDOG

- This enables/disables the Axxia device's watchdog timer. By default, this option is enabled.

  Note: If Redundant Boot for `u-boot-spl.img` has been enabled using the RS2B eFuse, the watchdog is also enabled by the bootROM.

#define LEAVE_WATCHDOG_ON

- This option allows the watchdog timer to be disabled before jumping to OS image from Stage 3. By default, the watchdog is disabled prior to the jump to OS.

#define WATCHDOG_TIMEOUT_SECS

- This sets the period of the watchdog timer.

5.0 Secure Boot Process

The Axxia device supports a Secure Boot process that enables the user to specify optional authentication, decryption, and revision checks on the BL2/BL3-1 and BL3-3 images, and by using various features of the Secure Boot Block (SBB) and settings in the eFuse Controller Module (ECM).

The goal of the Secure Boot flow is to provide software and configuration integrity, authentication, version anti-rollback, and anti-cloning. The Secure Boot process on the Axxia device provides hierarchical digital signature verification, optional anti-rollback, and optional decryption of the software image prior to accepting and executing the image.

Each stage of the boot process that implements the Secure Boot functionality can go through any of the following three processes:

- Software image integrity and authentication – minimum required functionality
- Software revision rollback checking
- Software image decryption

The following security algorithms are used during the Secure Boot process:

- SHA-256, SHA-384, or SHA-512/256 for the integrity algorithm
- ECDSA-256 or ECDSA-384 for the digital signature algorithm with four selectable curves
- AES-256 CBC for the encryption algorithm
- AES-WRAP (AES-256 Key Wrap) for the decryption process

The SBB Image Verification firmware functions accomplish all these secure boot features.

5.1 Enabling Secure Boot

The Stage BL-1 boot ROM is the first software module that runs on the Axxia device when it is released from System Reset or Chip Reset and configured to boot by using the internal Cortex-A57 MPCore processors. The boot ROM loads the next boot stage from a serial flash memory connected to chip select 0 of the SSP interface into the Local Secure Memory (LSM) on the Axxia device. The bootROM reads the Secure Boot Enable (SBE) eFuse bit to determine if this boot image must now be processed with the Secure Boot functionality. If SBE is enabled, CPU0 requests the SBB process the next boot stage image by using the SBB image verification firmware function (see Image Integrity and Authentication on page 25 for more details on SBB image verification firmware). If the image passes the SBB processing, CPU0 jumps to the next boot stage, if it fails, the device boot stops, or, if the bootloader is configured to support a Redundant Boot, a backup copy of the boot image may be loaded and SBB processing repeated.

The SBE eFuse bit only enables Secure Boot testing for the software image loaded by the BL-1 booROM. The use of Secure Boot functionality on all subsequent boot stages is under software control, and the SBE bit is not used. In the Intel reference boot software, stage BL2 can be configured to perform Secure Boot processing on the BL3-3 image and BL3-3 can be configured to perform Secure Boot processing on any subsequent boot image. The SBB Image verification firmware can be used by any software to perform Secure Boot processing.

For test purposes, to enable the Secure Boot functionality without programming the SBE eFuse, software can set the secure_boot bit in the Force Fuse register in the SysCon block and write the accompanying keys into the Key Fuse Force registers in the ECM. Setting this bit and then performing a Chip Reset resets the Axxia device and causes the boot ROM to expect the next boot image to have secure boot headers.

Secure Boot functions add header and signature sections to boot images, as described in the following sections. As a result, a boot image with Secure Boot functions has a different layout compared to a boot image without Secure Boot functions. It is possible in the AXM5600 to perform a nonsecure boot by using a secure boot image if encryption of the boot image is not used. The Boot ROM code checks the SBE eFuse and fuse force register, and if not set, jumps over the secure boot header and begins executing at the start of regular code.

5.2 Secure Boot Features

Secure Boot encompasses three different features:

- Image Integrity and Authentication
- Software Revision Rollback Checking
- Software Image Encryption

The Image Integrity and Authentication feature is always present in a Secure Boot software image. Software Revision Checking and Software Image Encryption are optional features that can be included or excluded from the image.

The three Secure Boot features are described in subsequent sections.

5.3 Image Integrity and Authentication

Image integrity and authentication is the process of guaranteeing that a software image has been created by a known source and has not been subsequently modified or corrupted. Figure 9 shows the image integrity and authentication process.

Figure 9. Image Integrity and Authentication

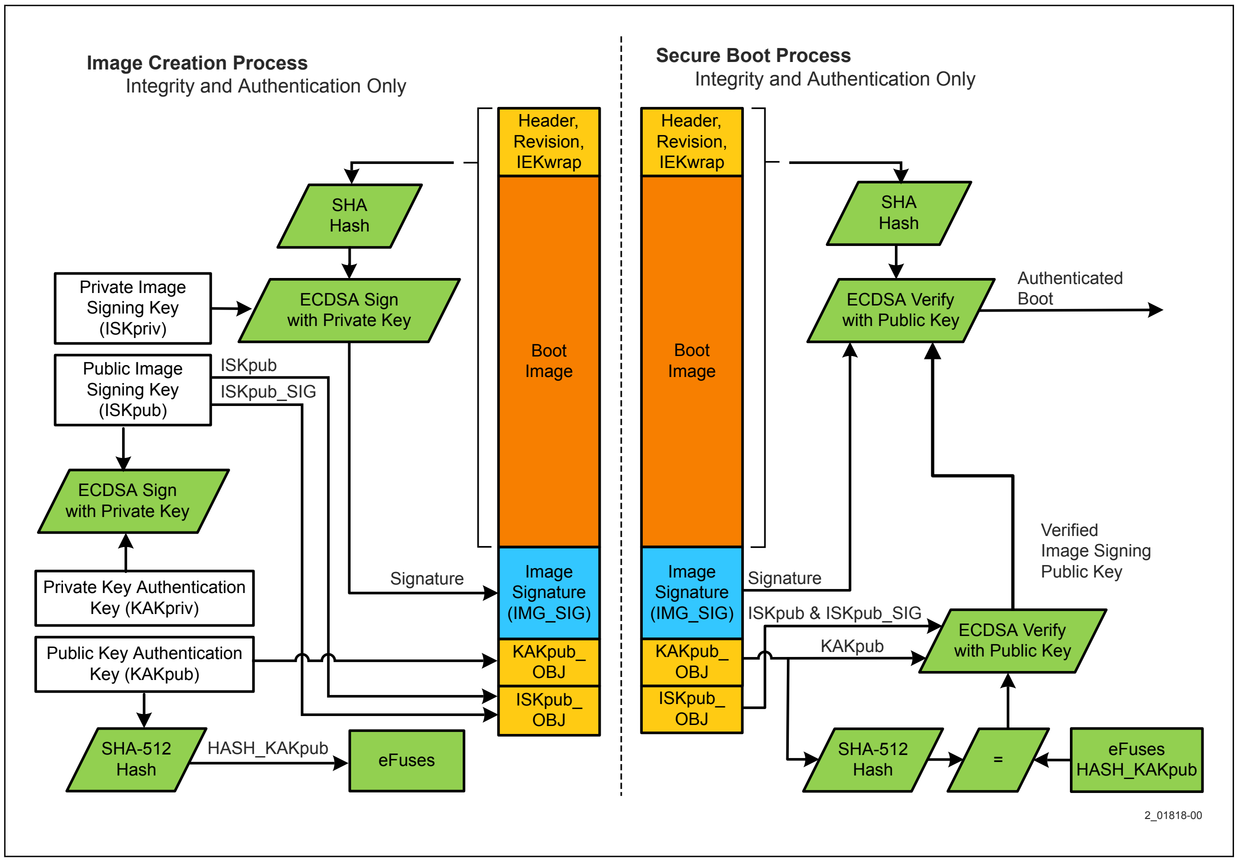


Image Creation Process
Integrity and Authentication Only
Secure Boot Process
Integrity and Authentication Only
Header, Revision, IEKwrap
SHA Hash
Private Image Signing Key (ISKpriv)
ECDSA Sign with Private Key
Public Image Signing Key (ISKpub)
ISKpub
ISKpub_SIG
Private Key Authentication Key (KAKpriv)
Public Key Authentication Key (KAKpub)
SHA-512 Hash
HASH_KAKpub
eFuses
Signature
Boot Image
Image Signature (IMG_SIG)
KAKpub_ OBJ
ISKpub_ OBJ
ECDSA Verify with Public Key
Authenticated Boot
Verified Image Signing Public Key
ISKpub & ISKpub_SIG
KAKpub
=
eFuses HASH_KAKpub
2_01818-00

For the image creation process, the raw software image is presented to a Code Signing tool, along with the Private Authority Key and an Image Signing ECDSA key pair (ISKpriv and ISKpub). The Authority Key may be the Private Key Authentication Key (KAKpriv) if no intermediate keys (IKs) are used. Otherwise, it may be a Private Intermediate Key (IKpriv). The preceding example diagram does not use intermediate keys. The Code Signing tool creates an ECDSA signature of the ISKpub by using a Private Authority Key (KAKpriv in the example diagram). This signature (ISKpub_SIG) and the ISKpub are part of the ISKpub_OBJ, which is stored at the end of the secure boot image. Immediately preceding the ISKpub_OBJ is the KAKpub_OBJ and any IKpub_OBJ that are used. The Code Signing tool prepends the necessary headers, which define the options used with this image. The headers and the original image are then run through a SHA hash operation that creates a 256/384-bit hash value. The headers and boot image are integrity protected from modification in this scheme. The 256/384-bit hash value is then encrypted by using ECDSA with the Private Image Signing Key (ISKpriv), resulting in a 512/768-bit signature value. The signature is appended to the end of the image.

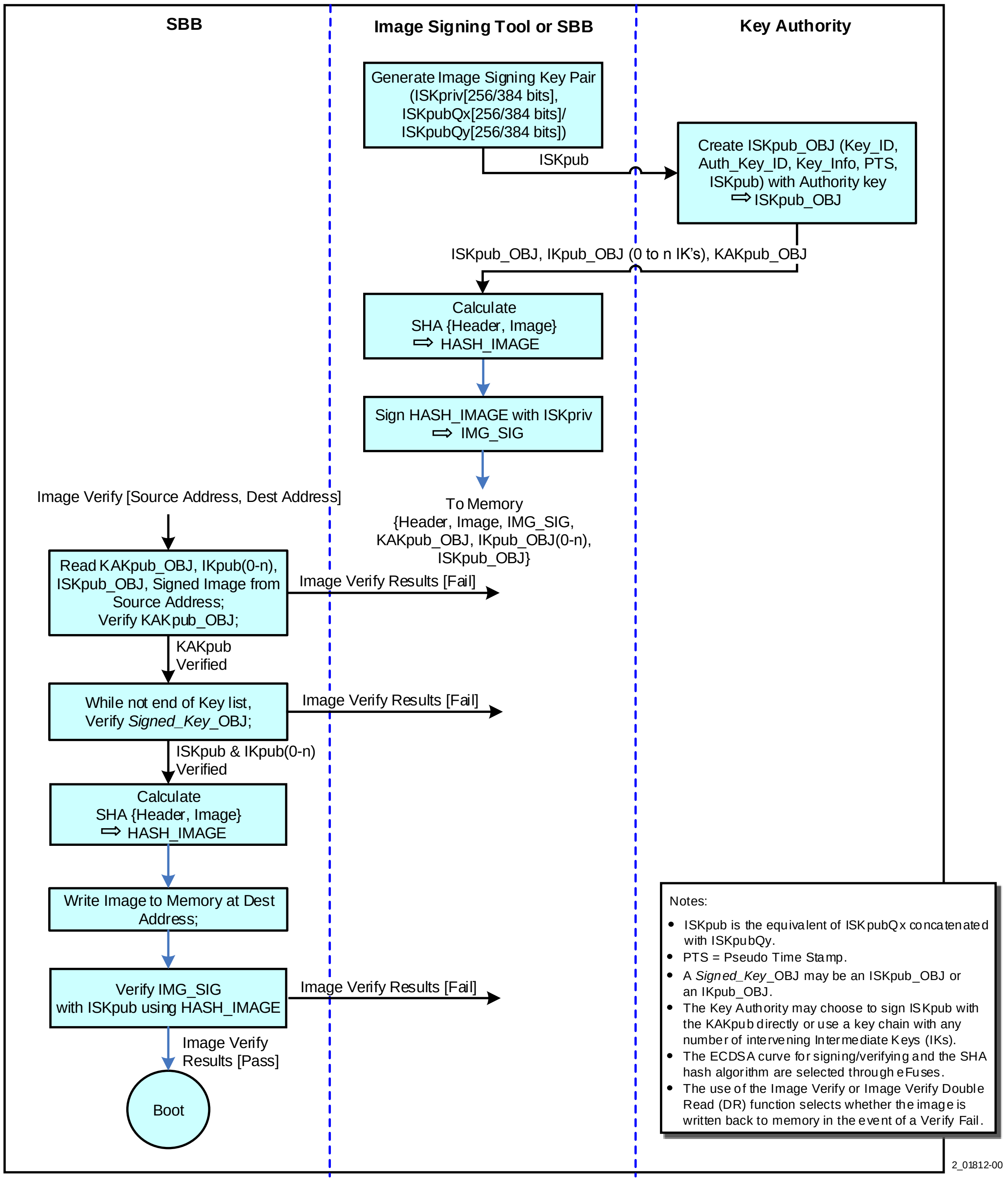
For the image verification process, the header is retrieved and evaluated to determine the boot options active in this image. The KAKpub_OBJ is retrieved from near the end of the image. A SHA hash of this object is created by the SBB and compared to all programmed non-revoked HASH_KAKpub_OBJs stored in the eFuses on the chip. This action verifies the Public Key Authentication Key. If any Public Intermediate Key Objects (IKpub_OBJs) are used, those objects are retrieved and verified against the KAKpub and then the next Public Authority Key in the key chain. The Public Image

Signing Key (ISKpub) and its signature ISKpub_SIG are retrieved from ISKpub_OBJ. The ISKpub is verified by using the ISKpub_SIG and the Public Authority Key. All keys are verified on every secure boot operation all the way back to the HASH_KAKpub_OBJ in the eFuses, creating a chain of trust. The header and the original image are then run through an SHA hash operation, which creates a 256/384-bit hash value. This hash value is then decrypted by using ECDSA with ISKpub, resulting in a 512/768-bit signature value. This calculated signature value is then compared to the original signature value that was appended to the end of the image to determine if they are equal. If they are equal, the image has not been modified and could have been created only by the owner of the Image Signing Key, Key Authentication Key, and any Intermediate Keys used in between. If the signatures are equal, the software image is accepted. Otherwise, the SBB issues an interrupt to indicate the failure of the image verification.

Note: ECDSA public and private key pairs have exact mathematical relationships and cannot be chosen individually. Intel provides a Key Generation tool (ncpSbb) for this purpose. Additionally, the key owner must securely keep the private keys used in this process. After an ECDSA key pair is generated, the private key is typically stored on a secure server.

Figure 10 shows a flowchart of the different steps to generate an image for Secure Boot and the steps the SBB uses to verify the image.

Figure 10. Image Verification Process


SBB
Image Signing Tool or SBB
Key Authority
Generate Image Signing Key Pair (ISKpriv[256/384 bits], ISKpubQx[256/384 bits]/ ISKpubQy[256/384 bits])
ISKpub
Create ISKpub_OBJ (Key_ID, Auth_Key_ID, Key_Info, PTS, ISKpub) with Authority key ⇨ ISKpub_OBJ
ISKpub_OBJ, IKpub_OBJ (0 to n IK's), KAKpub_OBJ
Calculate SHA {Header, Image} ⇨ HASH_IMAGE
Sign HASH_IMAGE with ISKpriv ⇨ IMG_SIG
To Memory {Header, Image, IMG_SIG, KAKpub_OBJ, IKpub_OBJ(0-n), ISKpub_OBJ}
Image Verify [Source Address, Dest Address]
Read KAKpub_OBJ, IKpub(0-n), ISKpub_OBJ, Signed Image from Source Address; Verify KAKpub_OBJ;
Image Verify Results [Fail]
KAKpub Verified
While not end of Key list, Verify *Signed_Key*_OBJ;
Image Verify Results [Fail]
ISKpub & IKpub(0-n) Verified
Calculate SHA {Header, Image} ⇨ HASH_IMAGE
Write Image to Memory at Dest Address;
Verify IMG_SIG with ISKpub using HASH_IMAGE
Image Verify Results [Fail]
Image Verify Results [Pass]
Boot
2_01812-00

Notes:

- ISKpub is the equivalent of ISKpubQx concatenated with ISKpubQy.
- PTS = Pseudo Time Stamp.
- A *Signed_Key*_OBJ may be an ISKpub_OBJ or an IKpub_OBJ.
- The Key Authority may choose to sign ISKpub with the KAKpub directly or use a key chain with any number of intervening Intermediate Keys (IKs).
- The ECDSA curve for signing/verifying and the SHA hash algorithm are selected through eFuses.
- The use of the Image Verify or Image Verify Double Read (DR) function selects whether the image is written back to memory in the event of a Verify Fail.

5.4 Software Revision Rollback Checking

Software revision rollback checking makes sure that the software has a minimum revision number. Use this capability to prevent the loading of old or obsolete software images. The revision information is placed in one of the headers created in the previous image integrity and authentication process. You can add software revision rollback checking to the image integrity and authentication process described in the preceding section. The image integrity and authentication process and the software revision rollback checking process can be performed in a single operation if they are both enabled.

The minimum version permitted to boot on the Axxia device must have been previously programmed into the Software Revision eFuse fields. During image validation, the SBB checks that the software revision is greater than or equal to the minimum revision number that is programmed into the eFuses. Software can update the eFuses as required to specify a new minimum software revision.

The SBB supports independent revision checking on four different software levels, where a level is defined as a software image. For example, you can use revision checking for two boot loader stages and two subsequent software stages.

Figure 11 shows the software revision rollback checking process.

Figure 11. Software Revision Rollback Checking

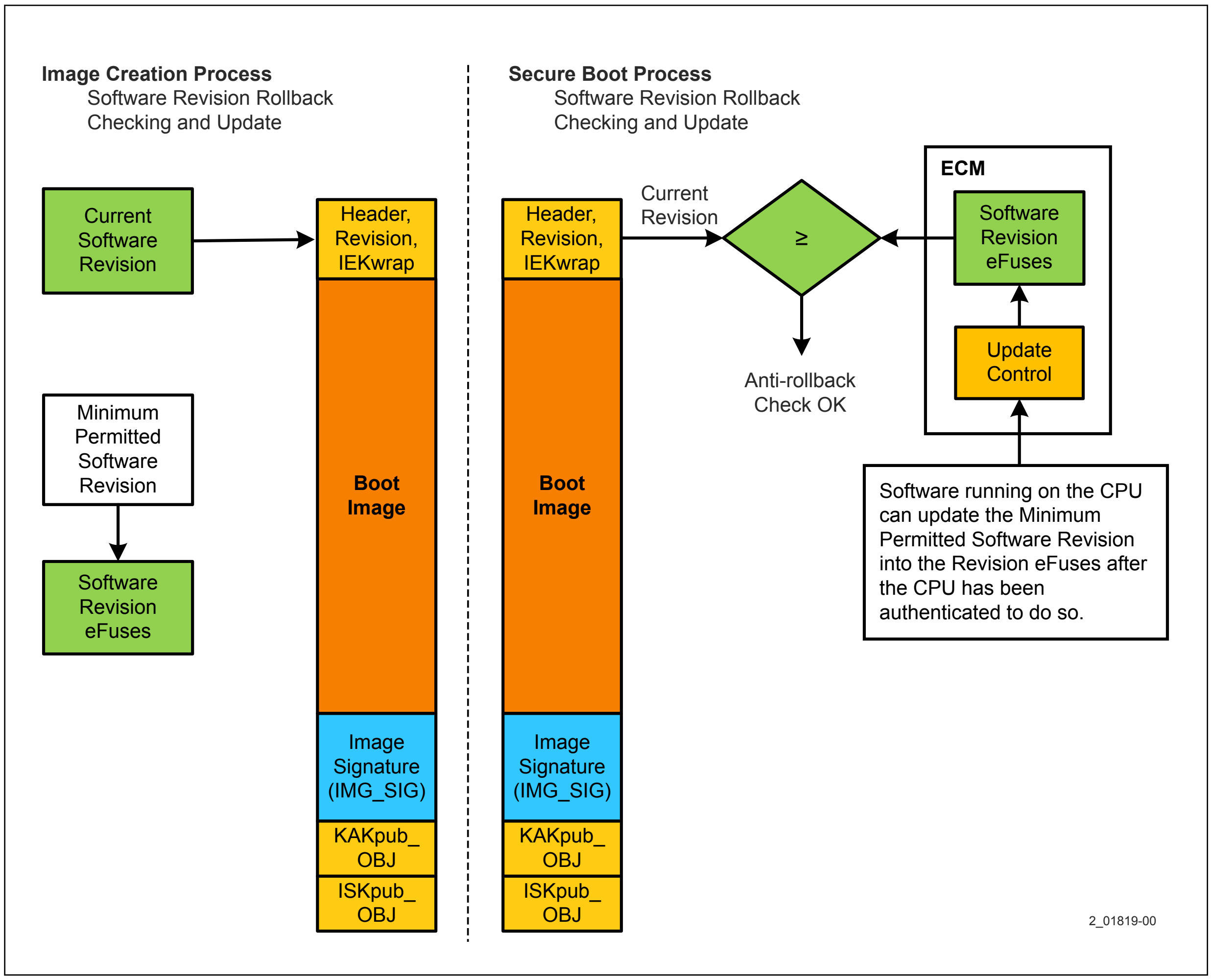


Image Creation Process
Software Revision Rollback Checking and Update
Current Software Revision
Header, Revision, IEKwrap
Boot Image
Image Signature (IMG_SIG)
KAKpub_ OBJ
ISKpub_ OBJ
Minimum Permitted Software Revision
Software Revision eFuses
Secure Boot Process
Software Revision Rollback Checking and Update
Current Revision
≥
Anti-rollback Check OK
ECM
Software Revision eFuses
Update Control
Software running on the CPU can update the Minimum Permitted Software Revision into the Revision eFuses after the CPU has been authenticated to do so.
2_01819-00

During the image creation process, the current software revision and software image are both presented to the Code Signing tool. The Code Signing tool prepends the necessary headers to the image. One of these headers is the current software revision. This header stores the software level affected and the image's current software revision. The current software revision is integrity protected through the image integrity and authentication process so that an attacker cannot modify its value.

During the image validation process, the SBB checks whether software revision checking is enabled in the boot image header. If so, the current revision is retrieved and compared to the revision that is programmed into the eFuses. If the current revision is greater than or equal to the programmed eFuse revision, the boot image is accepted. Otherwise, the software image is rejected and an interrupt is generated.

5.5 Software Image Encryption

Software images can be encrypted, which protects the contents of the image from observation or cloning. You can add image encryption to the image integrity and authentication process described in the preceding section. The image integrity and authentication process and image encryption process can be performed in a single operation if they are both enabled.

The raw software image is encrypted by using the AES-256 cipher block chaining (CBC) algorithm with an image encryption key (IEK) generated by the host software or Intel tools. The headers are not encrypted. The image encryption key is then wrapped by using the AES-WRAP algorithm also using a 256 bit key. This algorithm uses the Secure Boot Key Encryption Key (SBKEK), that is programmed into eFuses on the Axxia device. The wrapped encryption key (IEKwrap) is placed in the software image headers.

To decrypt the image, the SBB uses unwraps the Image Encryption Key (IEK), an AES-256 CBC key, in accordance with the AES-WRAP algorithm. This returns the IEK and an ICV. The SBB checks the ICV. If the ICV value is 0xa6a6a6a6a6a6a6a6, the IEK is correct. If a different value is found, the SBB repeats the process using the next programmed non-revoked SBKEK. If no valid ICV is found, secure boot fails. This key is then used to decrypt the software. If the decryption is successful and the image passes integrity and authentication checks, the software image can be executed. Otherwise, the software image is rejected and an interrupt is generated.

Figure 12 shows the image encryption process.

Figure 12. Software Image Encryption

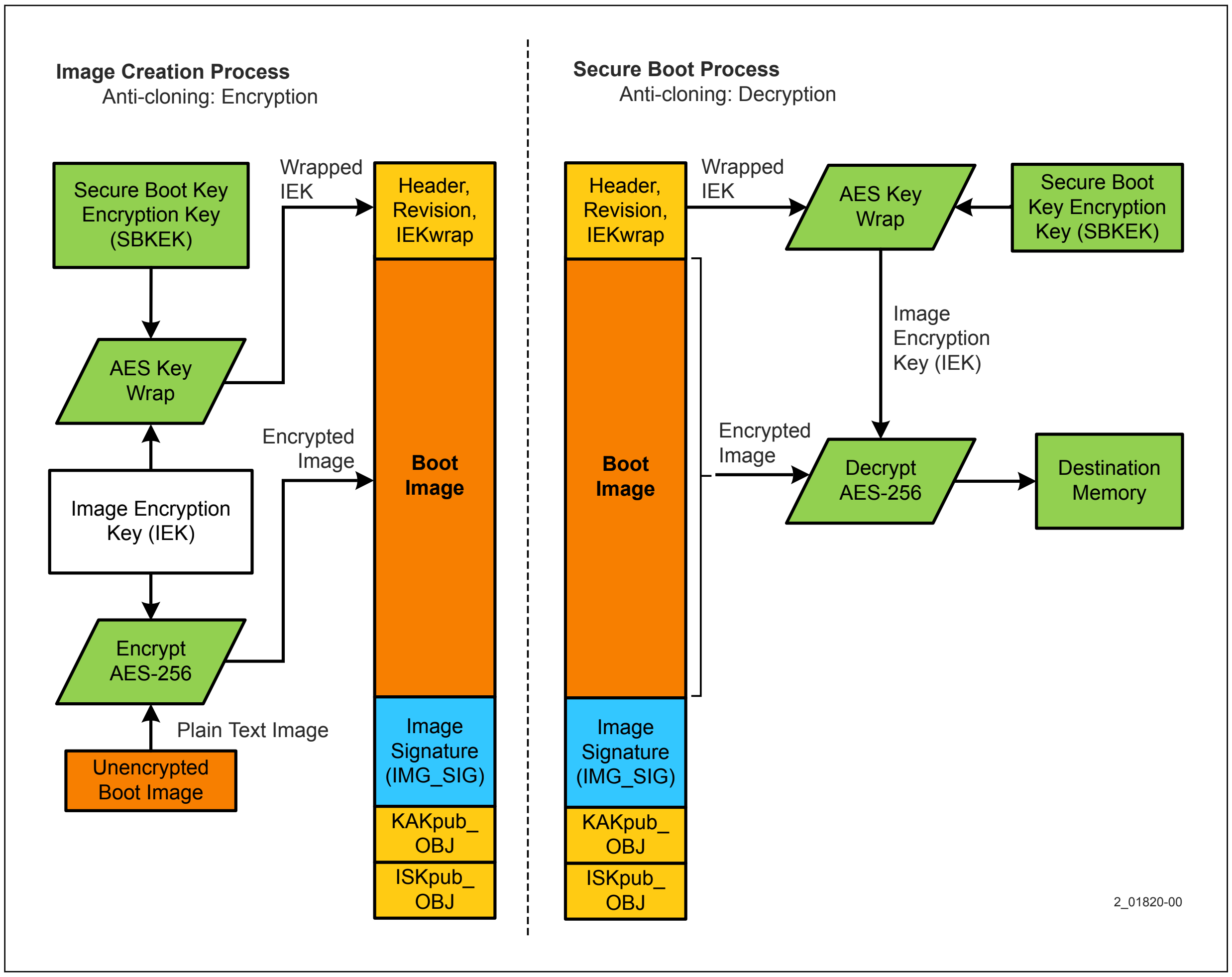
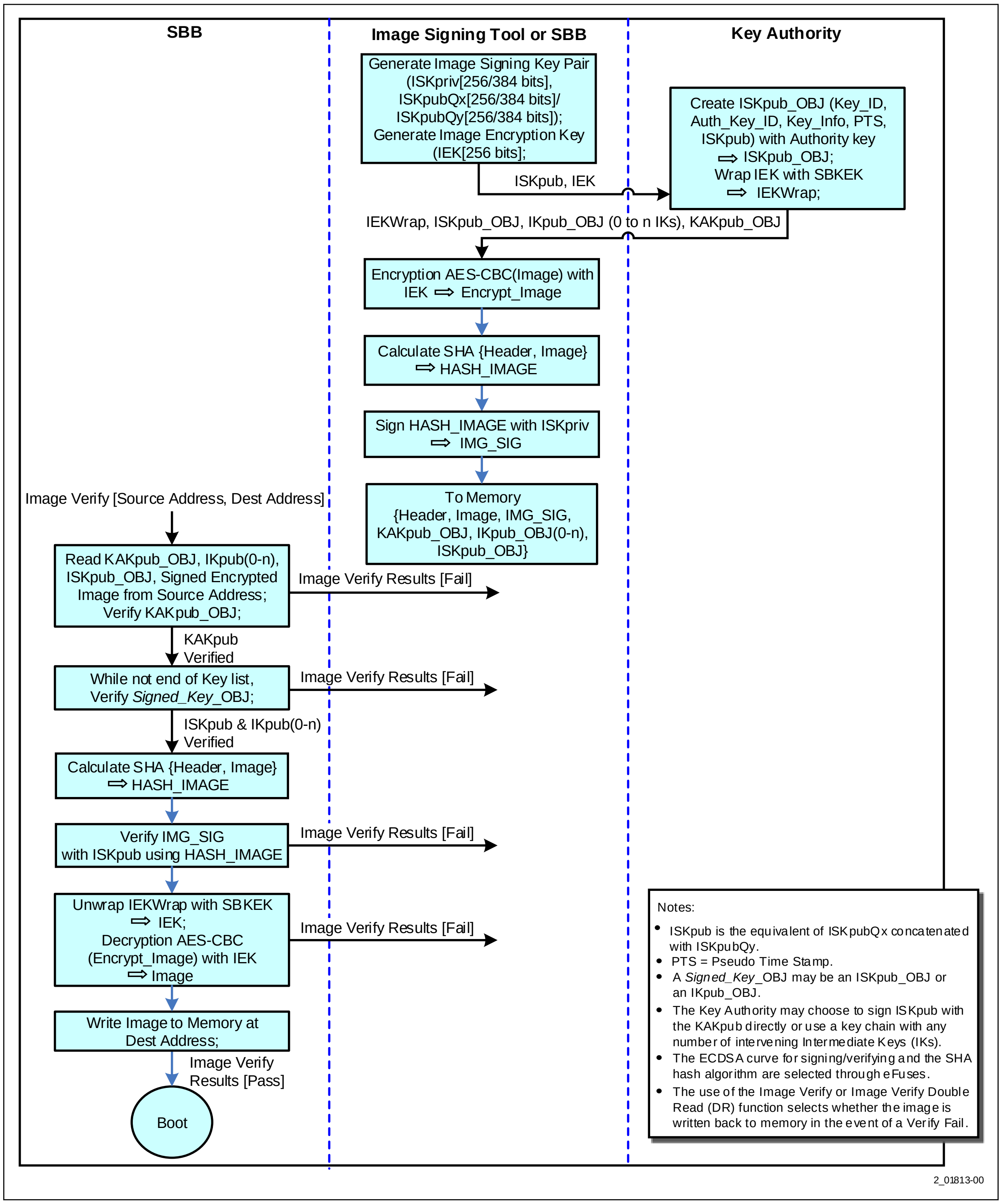


Figure 13 shows a flowchart of the different steps to generate an image for secure boot and the steps the SBB uses to decrypt and verify the image.

Figure 13. Software Image Verification Process with Decryption


SBB
Image Signing Tool or SBB
Key Authority
Generate Image Signing Key Pair (ISKpriv[256/384 bits], ISKpubQx[256/384 bits]/ ISKpubQy[256/384 bits]); Generate Image Encryption Key (IEK[256 bits];
ISKpub, IEK
Create ISKpub_OBJ (Key_ID, Auth_Key_ID, Key_Info, PTS, ISKpub) with Authority key ⇨ ISKpub_OBJ; Wrap IEK with SBKEK ⇨ IEKWrap;
IEKWrap, ISKpub_OBJ, IKpub_OBJ (0 to n IKs), KAKpub_OBJ
Encryption AES-CBC(Image) with IEK ⇨ Encrypt_Image
Calculate SHA {Header, Image} ⇨ HASH_IMAGE
Sign HASH_IMAGE with ISKpriv ⇨ IMG_SIG
To Memory {Header, Image, IMG_SIG, KAKpub_OBJ, IKpub_OBJ(0-n), ISKpub_OBJ}
Image Verify [Source Address, Dest Address]
Read KAKpub_OBJ, IKpub(0-n), ISKpub_OBJ, Signed Encrypted Image from Source Address; Verify KAKpub_OBJ;
Image Verify Results [Fail]
KAKpub Verified
While not end of Key list, Verify Signed_Key_OBJ;
Image Verify Results [Fail]
ISKpub & IKpub(0-n) Verified
Calculate SHA {Header, Image} ⇨ HASH_IMAGE
Verify IMG_SIG with ISKpub using HASH_IMAGE
Image Verify Results [Fail]
Unwrap IEKWrap with SBKEK ⇨ IEK; Decryption AES-CBC (Encrypt_Image) with IEK ⇨ Image
Image Verify Results [Fail]
Write Image to Memory at Dest Address;
Image Verify Results [Pass]
Boot
2_01813-00

Notes:

- ISKpub is the equivalent of ISKpubQx concatenated with ISKpubQy.
- PTS = Pseudo Time Stamp.
- A *Signed_Key*_OBJ may be an ISKpub_OBJ or an IKpub_OBJ.
- The Key Authority may choose to sign ISKpub with the KAKpub directly or use a key chain with any number of intervening Intermediate Keys (IKs).
- The ECDSA curve for signing/verifying and the SHA hash algorithm are selected through eFuses.
- The use of the Image Verify or Image Verify Double Read (DR) function selects whether the image is written back to memory in the event of a Verify Fail.

6.0 DDR Retention

BL2 uboot-spl implements a DDR retention feature which allows the DDR memory contents to be retained in the event of a device Chip Reset. This is useful for diagnosing the cause of resets, because it allows reading of any application logs or code that were present in DDR prior to the reset.

The AXM5600 device has hardware support for DDR Retention through a Chip Reset sequence. This can be used to retain DDR contents for analysis after a fault condition or software controlled reset. U-Boot for the AXM5600 provides support for this feature.

DDR Retention is a compile-time option. Its operation is controlled by the following `#define` directive in the U-Boot file `include/configs/axm56xx.h;`

#define CONFIG_MEMORY_RETENTION

- If defined, this enables the DDR retention feature.

7.0 Boot Image Storage

All `u-boot-spl`/secure monitor images, parameter file images and U-Boot images must be stored in a serial flash device connected to chip select 0 of the SSP interface. Table 2 on page 48 lists the locations in serial flash assigned to each image.

When U-Boot (BL3-3) is running, subsequent boot images— Linux OS images, hypervisor images, etc. —can be read from different locations, because U-Boot supports a wider range of device interfaces compared to the earlier boot stages. Options include the following:

- Serial flash connected to chip select 0 of SSP interface.
- USB storage device connected to the USB port.
- Software images can be loaded using the GEMAC (10/100/1000) and EIOA (1G/10G) ethernet interfaces. Protocols supported include tftp, dhcp and bootp.

8.0 Memory Mapping

All boot stages run on CPU0. The bootROM executes with the CPU's MMU disabled. u-boot-spl, Secure Monitor and U-Boot execute with the MMU enabled. The following table shows the physical address to virtual address mappings and attributes that are configured during operation.

Table 1. U-Boot Memory Mapping

Virtual Address Range (64-bit)	Physical Address Range (40-bit)	Size	Attributes	Memory Type	Comments
BL2 u-boot-spl					
0x0000000000000000 - 0x0000007fffffff	0x0000000000 - 0x7fffffff	512KB	Non-shareable, Non-cacheable, non-bufferable	Strongly ordered	System (DDR) Memory[1]
0x0000008000000000 - 0x000000ffffffffff	0x8000000000 - 0xffffffffff	512KB	Non-shareable, Non-cacheable, non-bufferable	Strongly ordered	Local Secure Memory (LSM) and Memory mapped register space
BL3-3 (u-boot)					
0x0000000000000000 - 0x0000007fffffffff	0x0000000000 - 0x7fffffffff	512KB	Shareable, cacheable (Write back, no write allocate), bufferable	Normal	System (DDR) Memory[1]
0x0000008000000000 - 0x000000ffffffffff	0x8000000000 - 0xffffffffff	512KB	Non-shareable, Non-cacheable, non-bufferable	Strongly ordered	Memory mapped register space
TBD	TBD	TBD	TBD	TBD	TBD

Notes:
1. An application board can implement only 0- 64GB of DDR memory. U-boot-spl and u-boot only require a small amount of DDR, typically less than 4MB.

9.0 Downloading and Building Boot Loader Images

The BL2 (u-boot-spl) and BL3-3 (U-Boot) images are based on the open-source U-Boot boot loader, and Intel provides the full source code for these boot loaders as a reference. The BL3-1 Secure Monitor is based on the ARM Trusted Firmware source code.

The code is distributed using `git` repositories. `git` is a distributed version control system used in many software development environments involving multiple developers in different locations. `git` users can create and manage repositories containing the source code files for a project.

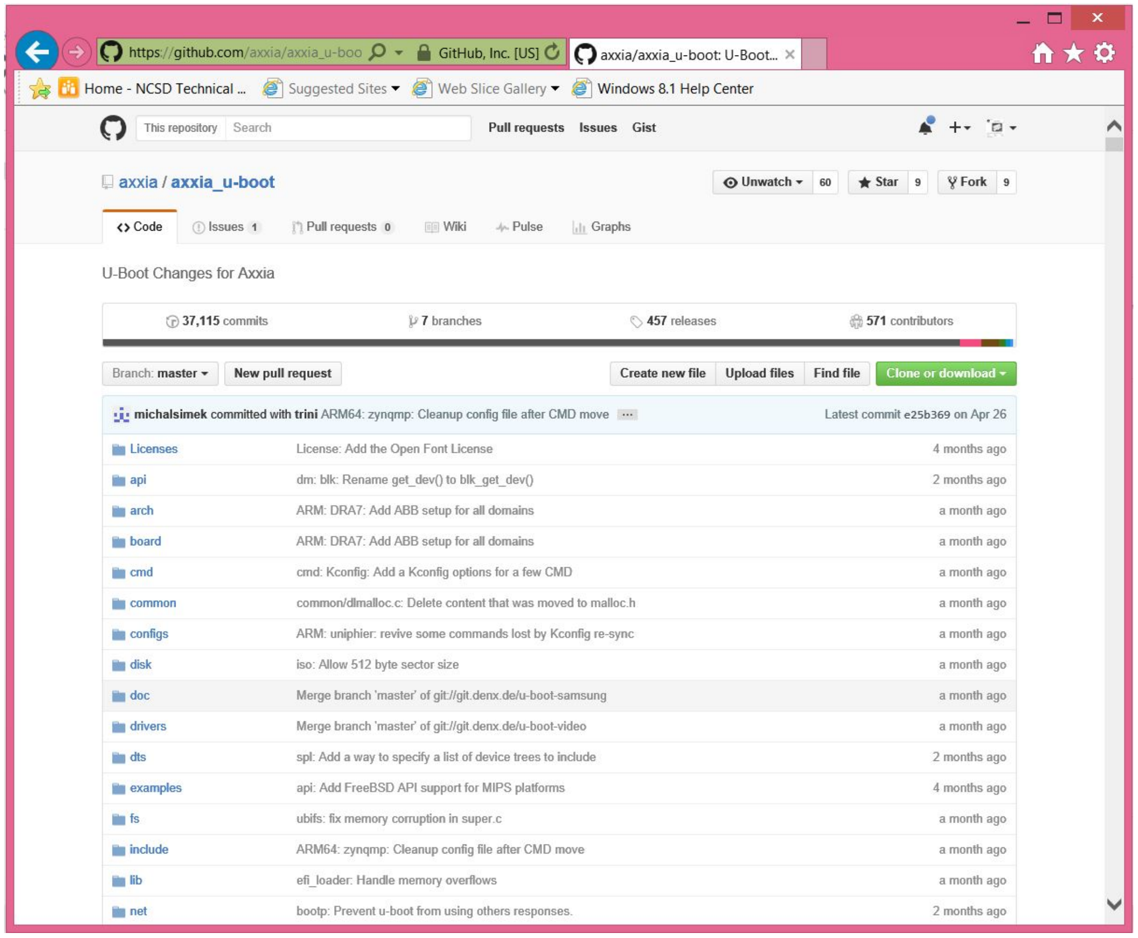
Intel distributes U-Boot and ARM Trusted Firmware for the AXM5600 device using `git` repositories located at the GitHub* website.

To access the repositories, follow these steps:

1. Go to https://github.com.
2. Create an account.
3. Search for and add the `axxia/axxia_u-boot` and `axxia/axxia_atf` repositories to your account.

See Figure 14 and Figure 15 for the GitHub website.

Figure 14. U-Boot Repository on GitHub



U-Boot Changes for Axxia

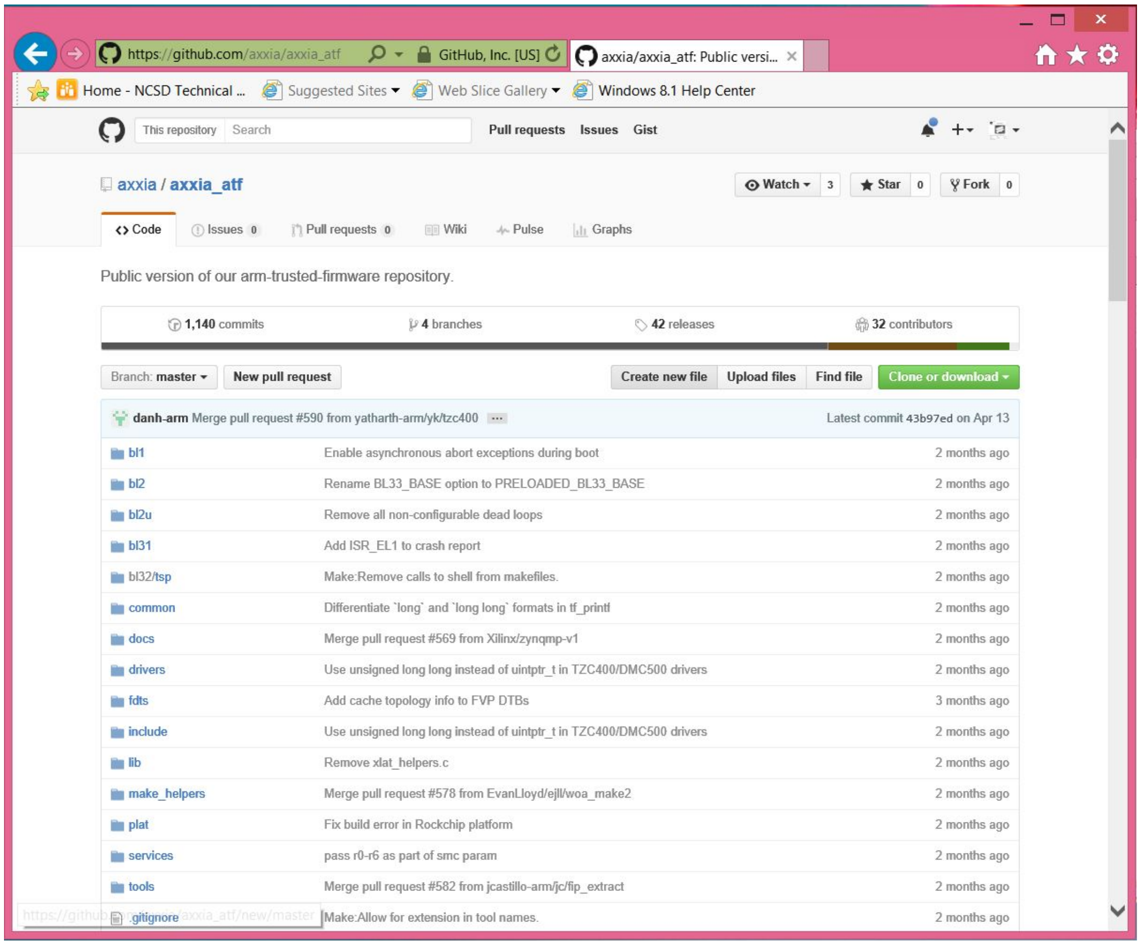
37,115 commits 7 branches 457 releases 571 contributors

michalsimek committed with trini ARM64: zynqmp: Cleanup config file after CMD move ... Latest commit e25b369 on Apr 26

Licenses	License: Add the Open Font License	4 months ago
api	dm: blk: Rename get_dev() to blk_get_dev()	2 months ago
arch	ARM: DRA7: Add ABB setup for all domains	a month ago
board	ARM: DRA7: Add ABB setup for all domains	a month ago
cmd	cmd: Kconfig: Add a Kconfig options for a few CMD	a month ago
common	common/dlmalloc.c: Delete content that was moved to malloc.h	a month ago
configs	ARM: uniphier: revive some commands lost by Kconfig re-sync	a month ago
disk	iso: Allow 512 byte sector size	a month ago
doc	Merge branch 'master' of git://git.denx.de/u-boot-samsung	a month ago
drivers	Merge branch 'master' of git://git.denx.de/u-boot-video	a month ago
dts	spl: Add a way to specify a list of device trees to include	2 months ago
examples	api: Add FreeBSD API support for MIPS platforms	4 months ago
fs	ubifs: fix memory corruption in super.c	a month ago
include	ARM64: zynqmp: Cleanup config file after CMD move	a month ago
lib	efi_loader: Handle memory overflows	a month ago
net	bootp: Prevent u-boot from using others responses.	2 months ago

2_01744-01

Figure 15. ARM Trusted Firmware Repository on GitHub


https://github.com/axxia/axxia_atf — axxia/axxia_atf: Public versi...

axxia / axxia_atf

Watch 3 · Star 0 · Fork 0

Code · Issues 0 · Pull requests 0 · Wiki · Pulse · Graphs

Public version of our arm-trusted-firmware repository.

1,140 commits · 4 branches · 42 releases · 32 contributors

Branch: master · New pull request · Create new file · Upload files · Find file · Clone or download

danh-arm Merge pull request #590 from yatharth-arm/yk/tzc400 — Latest commit 43b97ed on Apr 13

bl1	Enable asynchronous abort exceptions during boot	2 months ago
bl2	Rename BL33_BASE option to PRELOADED_BL33_BASE	2 months ago
bl2u	Remove all non-configurable dead loops	2 months ago
bl31	Add ISR_EL1 to crash report	2 months ago
bl32/tsp	Make:Remove calls to shell from makefiles.	2 months ago
common	Differentiate `long` and `long long` formats in tf_printf	2 months ago
docs	Merge pull request #569 from Xilinx/zynqmp-v1	2 months ago
drivers	Use unsigned long long instead of uintptr_t in TZC400/DMC500 drivers	2 months ago
fdts	Add cache topology info to FVP DTBs	3 months ago
include	Use unsigned long long instead of uintptr_t in TZC400/DMC500 drivers	2 months ago
lib	Remove xlat_helpers.c	2 months ago
make_helpers	Merge pull request #578 from EvanLloyd/ejll/woa_make2	2 months ago
plat	Fix build error in Rockchip platform	2 months ago
services	pass r0-r6 as part of smc param	2 months ago
tools	Merge pull request #582 from jcastillo-arm/jc/fip_extract	2 months ago
.gitignore	Make:Allow for extension in tool names.	2 months ago

2_02000-01

Each `git` repository has a link to a wiki page that provides links to `Readme` and `Changes` files for different builds of U-Boot and the ARM Trusted Firmware. The following sections include instructions for downloading and building the AXM5600 bootloader images. Check the GitHub wiki pages to determine if the instructions have changed for more recent U-Boot versions.

9.1 Prerequesites

The following instructions describe how to download (or clone) a copy of the U-Boot and ARM Trusted Firmware repositories and build the various bootloader images for the AXM5600 device. To complete these instructions, you must have the following items:

- A PC running Linux. At the time of publication, a system running `Ubuntu* 14.04 64-bit` was used.
- An installed copy of `git` on Linux.

In `Ubuntu`, use the following command to install `git`:

```
sudo apt-get install git
```

- An installed copy of the `gcc` compiler that supports the ARMv8 architecture. You can download a prebuilt version of `gcc` from the following Linaro* website:

```
http://releases.linaro.org/14.11/components/toolchain/binaries/aarch64-linux-gnu
```

At time of publication, the author used the following version of `gcc`:

```
http://releases.linaro.org/14.11/components/toolchain/binaries/aarch64-linux-gnu/gcc-linaro-4.9-2014.11-x86_64_aarch64-linux-gnu.tar.xz
```

- Tag names for the `u-boot` and ARM Trusted Firmware versions that you want to build. `git` uses tags to specify a particular point in the development of a project. Intel updates `u-boot` and ARM Trusted Firmware code regularly and uses tags to identify a particular version or build. Intel updates customers with the latest tag for any devices they use.

  If you do not have the tag value for your particular device, contact your Intel Field Representative for more details.

  At the time of publication, the tag for an AXM5600 build were:

  — `u-boot_v2015.10_axxia_1.22` (for u-boot/u-boot-spl)

  — `atf_84091c4_axxia_1.8` (for ARM Trusted Firmware)

9.2 Download the U-Boot Source Code

Perform the following steps to download or clone a copy of the U-Boot and u-boot-spl source code:

1. Clone the U-Boot repository from the GitHub server.

```
git clone https://github.com/axxia/axxia_u-boot.git
```

2. Check out the `axxia-dev` branch.

```
cd axxia_u-boot
git checkout --track -b axxia-dev origin/axxia-dev
```

3. Create a local copy of the tagged version required, and checkout.

```
git branch local_build u-boot_v2015.10_axxia_1.22
git checkout local_build
```

9.3 Download the ARM Trusted Firmware Source Code

Perform the following steps to download or clone a copy of the ARM Trusted Firmware source code:

1. Clone the ARM Trusted Firmware repository from the GitHub server.

```
git clone https://github.com/axxia/axxia_atf.git
```

2. Check out the axxia-dev branch.

```
cd axxia_atf
git checkout --track -b axxia-dev
```

3. Create a local copy of the tagged version required, and checkout.

```
git branch local_build atf_84091c4_axxia_1.8
git checkout local_build
```

9.4 Configure Host Environment Variables

Use the following commands on the Linux host machine to configure gcc compiler-related environment variables.

```
export PATH=<path-to-ARM-gcc-compiler>:$PATH
export CROSS_COMPILE=aarch64-linux-
export ARCH=arm64
```

9.5 Build the ARM Trusted Firmware

Perform the following steps to build the ARM Trusted Firmware:

1. Go to the ARM Trusted Firmware source code folder.

```
cd axxia_atf
```

2. Run make to build BL31. Add 'PLAT=axxia USE_COHERENT_MEM=0', and optionally, 'DEBUG=1' to the make command line. If 'DEBUG=1' is added, there will be more output on the console, and more assertions done in the code.

```
make DEBUG=1 PLAT=axxia USE_COHERENT_MEM=0 bl31
```

3. Create an Elf file containing the BL3-1 Secure Monitor. If not using DEBUG=1 option, use command:.

```
${CROSS_COMPILE}objcopy -I binary -O elf64-littleaarch64 -B aarch64 \ --
rename-section .data=.monitor build/axxia/release/bl31.bin \ build/axxia/
release/bl31.o
```

If using DEBUG = 1, use the following command:

```
${CROSS_COMPILE}objcopy -I binary -O elf64-littleaarch64 -B aarch64 \ --
rename-section .data=.monitor build/axxia/debug/bl31.bin \ build/axxia/debug/
bl31.o
```

After building, the `axxia_atf` or axxia_atf/build/axxia/debug folder will contain the bl31.o file, which is an Elf file containing the BL3-1 Secure Monitor. You must copy this file into the U-Boot source code as described in the following section.

9.6 Build U-Boot and u-boot-spl

1. Go to the U-Boot source code folder.

```
cd axxia_uboot
```

2. Configure the U-Boot makefiles for the AXM5600 using one of the commands below. The three listed commands allow configuration for three different targets: AXM5600 silicon device, AXM5600 emulator, or AXM5600 simulation.

```
make axm5600_defconfig
make axm5600_emu_defconfig  #(for emulation system)
make axm5600_sim_defconfig  #(for simulation)
```

3. Copy BL3-1 Secure Monitor object code from the ARM Trusted Firmware build into the u-boot-spl folder. If Secure Monitor was built without DEBUG=1 option, use command:

```
cp ~/acp/axxia_atf/build/axxia/release/bl31.o  ./spl/
```

   If Secure Monitor was built with DEBUG=1 option, use command:

```
cp ~/acp/axxia_atf/build/axxia/debug/bl31.o  ./spl/
```

4. Build U-Boot and u-boot-spl.

```
make
```

   A simple `make` command builds u-boot and u-boot-spl. Version strings can be added to the make command to define the u-boot/u-boot-spl version number and the ATF Secure Monitor version number. Typically, information from the git tags for each package would be used. For example:

```
make  AXXIA_VERSION=u-boot_v2015.10_axxia_1.22
AXXIA_ATF_VERSION=axxia_atf_atf_84091c4_axxia_1.8
```

   u-boot-spl prints these version numbers to console when it starts.

5. Use mkimage to add a header to the `u-boot-spl` binary.

   For this step, choose a a Sequence Number and optional SSP configuration value for the u-boot-spl header using the `–a` and `–e` command line options. If SSP configuration does not need to be updated, just omit the `–e` option

```
./tools/mkimage -A arm64 –T firmware –C none –a <Sequence_Number> -e
<SSP_options> -n XLOADER -d spl/u-boot-spl.bin spl/u-boot-spl.img
```

   For example, to set Sequence number to 0 and SSP options to 0x00027001 :

```
./tools/mkimage -A arm64 -T firmware -C none -a 0 -e 0x00027001 -n u-boot-spl
-d spl/u-boot-spl.bin spl/u-boot-spl.img
```

After U-Boot has been built, the two files of interest are:

- spl/u-boot-spl.img

- u-boot.img

10.0 Creating a Parameter File

The BL2 (`u-boot-spl`) uses a parameter file to obtain values and options that can vary on different application boards. This allows `u-boot-spl` to support a wide range of application boards without the need to modify the `u-boot-spl` source code.

A list of the options that can be set in a parameter file is shown in BL2/BL3-1 and BL3-3 on page 16.

The parameter file is initially generated as a text file using the Application Software Environment (ASE) software package. The text file contains a human-readable description of each option selected. This text file is then converted to a binary format using the run-time environment (RTE) software package, specifically the RTE `ncpBootMem` command.

10.1 Creating a Text Version of the Parameter File Using ASE

The Axxia Software Environment (ASE) is an Eclipse-based tool that enables you to create, define, simulate, and test configurations and applications for Axxia communication processors, in addition to generating U-Boot parameter files. A full description of ASE installation and usage is available in the *Intel Axxia Communication Processor Axxia Software Environment (ASE) User Guide*. The guide, and the ASE installer can be downloaded from the Intel website. At time of publication, version 1.4.5.018 was used..

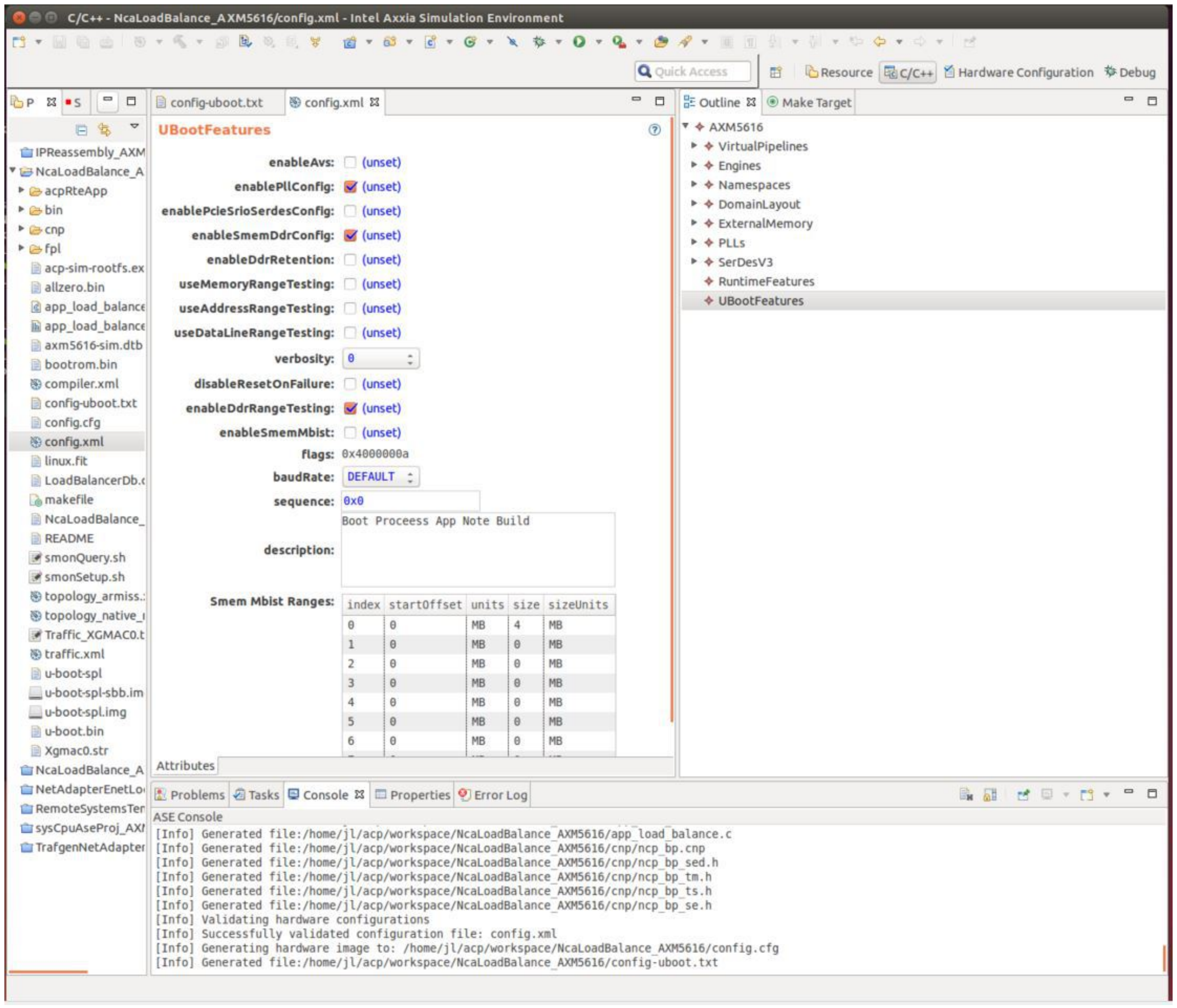
Figure 16 on page 45 shows a screen capture of ASE. An ASE project is based on a configuration file, typically called `config.xml`. This file contains various settings and options for peripherals and accelerator engines in the Axxia device, including options used by `u-boot-spl`. Selecting `config.xml` in the Project Explorer pane (the left pane) opens an Outline view in the Outline pane (the right pane). Clicking on the various fields in the Outline pane expands the view and shows individual options in the middle pane. The following fields contain options relevant to the boot loader:

- **External Memory ➤ System Memory** (DDR and DDR phy options)
- **PLLs** (PLL and clock options)
- **SerDes** (SerDes lane configuration and mapping for PCIe and sRIO)
- **UBootFeatures** (enable/disable flags for various options, Stage 3 baud rate selection, and memory MBIST test ranges)

Note: The exact contents of these fields may change in newer versions of ASE.

You can modify the various settings to match the requirements for your application.

Figure 16. ASE Screen Capture



C/C++ - NcaLoadBalance_AXM5616/config.xml - Intel Axxia Simulation Environment

UBootFeatures

enableAvs:	☐ (unset)
enablePllConfig:	✓ (unset)
enablePcieSrioSerdesConfig:	☐ (unset)
enableSmemDdrConfig:	✓ (unset)
enableDdrRetention:	☐ (unset)
useMemoryRangeTesting:	☐ (unset)
useAddressRangeTesting:	☐ (unset)
useDataLineRangeTesting:	☐ (unset)
verbosity:	0
disableResetOnFailure:	☐ (unset)
enableDdrRangeTesting:	✓ (unset)
enableSmemMbist:	☐ (unset)
flags:	0x4000000a
baudRate:	DEFAULT
sequence:	0x0
description:	Boot Proceess App Note Build

Smem Mbist Ranges:

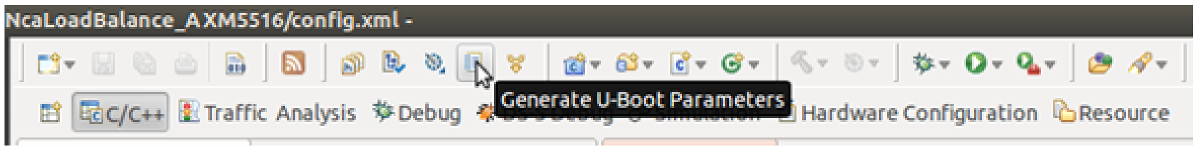
index	startOffset	units	size	sizeUnits
0	0	MB	4	MB
1	0	MB	0	MB
2	0	MB	0	MB
3	0	MB	0	MB
4	0	MB	0	MB
5	0	MB	0	MB
6	0	MB	0	MB

ASE Console

```
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/app_load_balance.c
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/cnp/ncp_bp.cnp
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/cnp/ncp_bp_sed.h
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/cnp/ncp_bp_tm.h
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/cnp/ncp_bp_ts.h
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/cnp/ncp_bp_se.h
[Info] Validating hardware configurations
[Info] Successfully validated configuration file: config.xml
[Info] Generating hardware image to: /home/jl/acp/workspace/NcaLoadBalance_AXM5616/config.cfg
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/config-uboot.txt
```

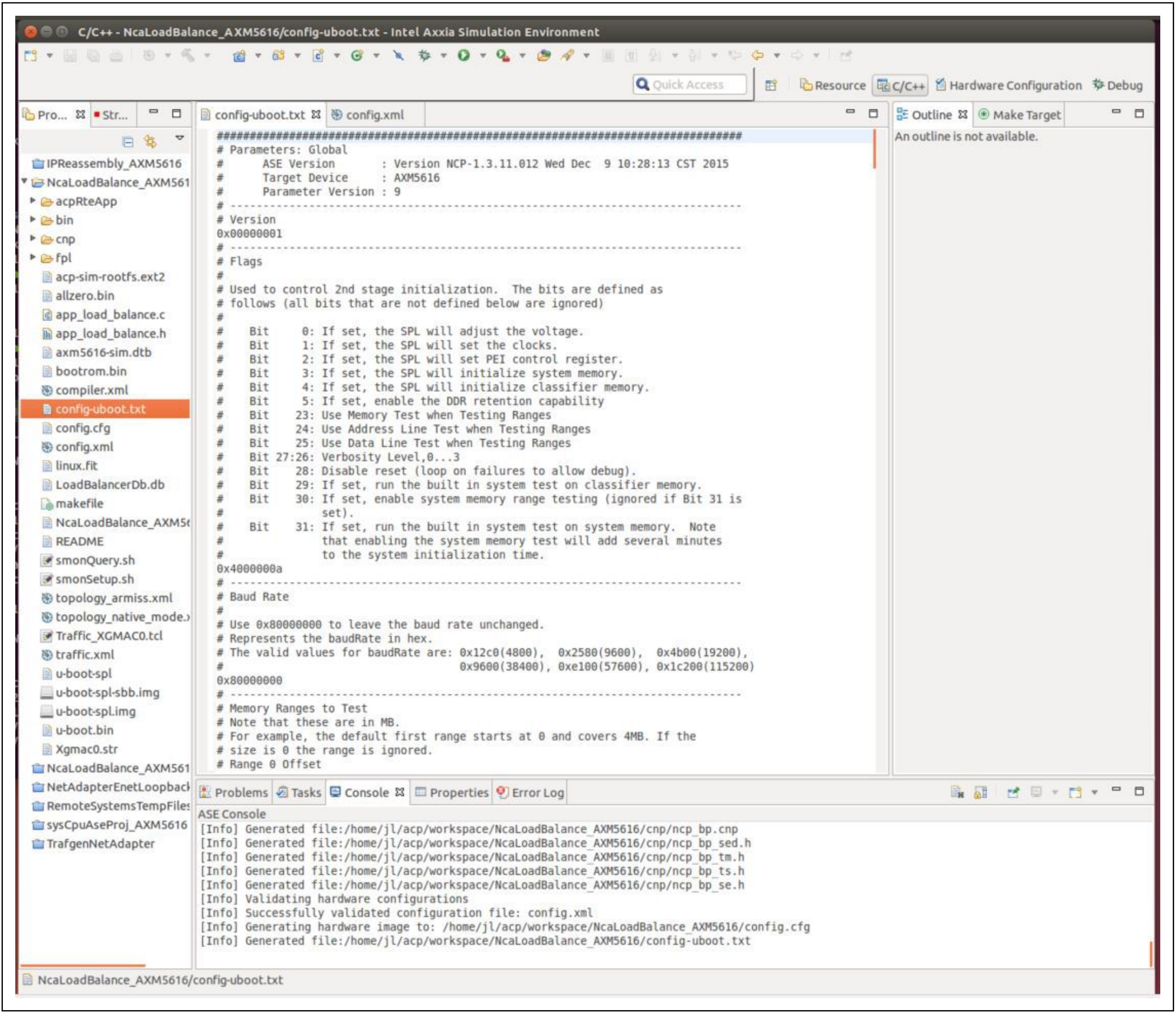
After the correct settings have been selected in `config.xml`, ASE can generate a text parameter file, typically called `config-uboot.txt`, using the Generate U-Boot Parameters icon. See Figure 17 for details.

Figure 17. Generating a Parameter File


NcaLoadBalance_AXM5516/config.xml -
Generate U-Boot Parameters
C/C++
Traffic Analysis
Debug
Hardware Configuration
Resource

Double clicking `config-uboot.txt` in the Project Explorer pane opens the parameter file in the middle pane. See Figure 18 on page 46 for details. The contents are in human-readable ASCII text. The contents of the file can also be edited manually, if required.

Figure 18. config-uboot.txt File



```
################################################################################
# Parameters: Global
#      ASE Version       : Version NCP-1.3.11.012 Wed Dec  9 10:28:13 CST 2015
#      Target Device     : AXM5616
#      Parameter Version : 9
# ------------------------------------------------------------------------------
# Version
0x00000001
# ------------------------------------------------------------------------------
# Flags
#
# Used to control 2nd stage initialization.  The bits are defined as
# follows (all bits that are not defined below are ignored)
#
#    Bit     0: If set, the SPL will adjust the voltage.
#    Bit     1: If set, the SPL will set the clocks.
#    Bit     2: If set, the SPL will set PEI control register.
#    Bit     3: If set, the SPL will initialize system memory.
#    Bit     4: If set, the SPL will initialize classifier memory.
#    Bit     5: If set, enable the DDR retention capability
#    Bit    23: Use Memory Test when Testing Ranges
#    Bit    24: Use Address Line Test when Testing Ranges
#    Bit    25: Use Data Line Test when Testing Ranges
#    Bit 27:26: Verbosity Level,0...3
#    Bit    28: Disable reset (loop on failures to allow debug).
#    Bit    29: If set, run the built in system test on classifier memory.
#    Bit    30: If set, enable system memory range testing (ignored if Bit 31 is
#               set).
#    Bit    31: If set, run the built in system test on system memory.  Note
#               that enabling the system memory test will add several minutes
#               to the system initialization time.
0x4000000a
# ------------------------------------------------------------------------------
# Baud Rate
#
# Use 0x80000000 to leave the baud rate unchanged.
# Represents the baudRate in hex.
# The valid values for baudRate are: 0x12c0(4800),  0x2580(9600),  0x4b00(19200),
#                                    0x9600(38400), 0xe100(57600), 0x1c200(115200)
0x80000000
# ------------------------------------------------------------------------------
# Memory Ranges to Test
# Note that these are in MB.
# For example, the default first range starts at 0 and covers 4MB. If the
# size is 0 the range is ignored.
# Range 0 Offset
```

```
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/cnp/ncp_bp.cnp
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/cnp/ncp_bp_sed.h
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/cnp/ncp_bp_tm.h
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/cnp/ncp_bp_ts.h
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/cnp/ncp_bp_se.h
[Info] Validating hardware configurations
[Info] Successfully validated configuration file: config.xml
[Info] Generating hardware image to: /home/jl/acp/workspace/NcaLoadBalance_AXM5616/config.cfg
[Info] Generated file:/home/jl/acp/workspace/NcaLoadBalance_AXM5616/config-uboot.txt
```

10.2 Creating a Binary Version of the Parameter File Using RTE

The `config-uboot.txt` file generated in the previous section must be converted to a binary format, which can then be programmed into a serial flash device on the application board. The conversion is performed using the `ncpBootMem` command-line tool supplied in the Run Time Environment (RTE) software packages for the AXM5600 family of devices. RTE can be downloaded from the website.

RTE is supplied as a source code tarball, which must be compiled. It can be compiled to run on the AXM5600 device, or to run on a Linux PC. Build instructions are supplied in the tarball.

RTE includes the `ncpBootParamGen` command-line tool, which can be used to generate boot image files. It can be used to convert the `config-uboot.txt` file to a binary format. To produce a `parameters.bin` binary file from a `config-uboot.txt` text file, use the following command:

```
ncpBootParamGen -v 9 -c 56xx -a image -r parameters  config-
uboot.txt  parameters.bin
```

Note: This command is likely to change during development of AXM5600 RTE software package. Please refer to RTE documentation in case of any error messages.

This command will create a binary file of size 4096 bytes. This file can be programmed into a serial flash device connected to chip select 0 of the AXM5600 device’s SSP.

11.0 Programming Boot Images Into Serial Flash

All u-boot-spl/secure monitor images, parameter file images and u-boot images must be stored in a serial flash device connected to chip select 0 of the SSP interface.

The bootROM requires u-boot-spl A and B images to be at the locations specified in Table 1 on page 36. All other offsets are defined in a header file in the u-boot source code. Therefore these locations can be changed if required.

At time of publication, the following header files were used.

- `include/configs/axm56xx.h` for the AXM5600 device
- `include/configs/axm56xx_emu.h` for the AXM5600 emulation system
- `include/configs/axm56xx_sim.h` for the AXM5600 simulation

Table 2 lists the locations and sizes of various regions defined in the serial flash.

Table 2. Serial Flash Regions

Address Offset	Region Size	Name	Contents
0x00000000	256 KB	`u-boot-spl A`	BL2/BL3-1 image A
0x00040000	256 KB	u-boot-spl A	BL2/BL3-1 image B
0x00080000	64 KB (4 KB used)	Parameter file A	Parameter file binary[1]
0x00090000	64 KB (4 KB used)	Parameter file B	Parameter file binary[1]
0x000a0000	64 KB	Env(1)	`u-boot` environment variables, primary copy
0x000b0000	64 KB	Env(2)	`u-boot` environment variables, secondary copy
0x00100000	2 MB	`u-boot` image A	Stage 3 (`u-boot`) binary image
0x00300000	2 MB	`u-boot` image B	Stage 3 (`u-boot`) binary image

Notes:
1. The parameter file size is 4 KB. A region of size 64 KB is allocated for each parameter file because the typical minimum block erase size of a serial flash device used is 64 KB.

There are a number of ways to program the images into serial flash. This document describes the following three methods:

- Use the `ncpBootMem` command to update images on a system that is currently running RTE software.
- Use the U-Boot commands to update images on a system that is currently running `u-boot`.
- Use a DStream JTAG debug cable to program images into a system with a blank serial flash.

Other methods (for example, programing the serial flash in a standalone flash programming device) are also possible.

As a prerequisite for all programming options, you must do the following:

- Create the `u-boot.img` and `spl/u-boot-spl.img` files as described in Build U-Boot and u-boot-spl on page 42.
- Create the parameter file as described in Creating a Parameter File on page 44.

If using the Redundant Boot feature, you must program two versions (A and B) of each file into serial flash. The two versions can be different or identical. In the following sections, it is assumed that there are two versions of each file. The suffixes '_A' and '_B' are used to differentiate the two versions.

11.1 Programming Serial Flash Using the ncpBootMem Command

On a live application board that is currently running a Linux OS and has RTE installed, it is possible to update the images in the serial flash using the `ncpBootMem` command-line tool with the following steps:

1. Copy the `u-boot.img` files, the `u-boot-spl.img` files and the parameter files to a folder accessible by Linux on the target system.
2. Use the following commands to program a minimum set of images into serial flash:

```
ncpBootMem -a write -r prom -o 0x00000000 u-boot-spl_A.imgncp
BootMem -a write -r prom -o 0x00080000 parameters_A.bin
ncpBootMem -a write -r prom -o 0x00100000 u-boot_A.img
```

3. When using the Redundant Boot feature, you must program two copies of each file into serial flash. The two copies can either be identical or different. The following extra commands are required to program the second set of files.

```
ncpBootMem -a write -r prom -o 0x00040000  u-boot-spl_B.img
ncpBootMem -a write -r prom -o 0x00090000 parameters_B.bin
ncpBootMem -a write -r prom -o 0x00300000 u-boot_B.img
```

11.2 Programming Serial Flash Using U-Boot

On a system that has an existing installation of U-Boot, it is possible to update the images in serial flash using the U-Boot `sf` serial flash commands. The following sequence of commands will update the minimum set of images. This sequence assumes the images are stored on a tftp server that can be accessed by U-Boot running on the target board. U-Boot can use `tftp` to read each image. However, if `tftp` is not available, it is possible to read the images from other locations (for example, from a USB flash disk), or to load the images into RAM using a JTAG debug cable.

```
sf probe 0
sf erase 0 40000
tftp $loadaddr u-boot-spl_A.img
sf write $loadaddr 0 40000

sf erase 80000 10000
```

```
tftp $loadaddr parameters_A.bin
sf write $loadaddr 80000 10000

sf erase 100000 200000
tftp $loadaddr u-boot_A.img
sf write $loadaddr 100000 200000
```

When using the Redundant Boot feature, you must program two copies of each file into serial flash. The two copies can either be identical or different. The following extra commands are required.

```
sf erase 40000 40000
tftp $loadaddr u-boot-spl_B.img
sf write $loadaddr 40000 40000

sf erase 90000 10000
tftp $loadaddr parameters_B.bin
sf write $loadaddr 50000 10000

sf erase 300000 200000
tftp $loadaddr u-boot_B.img
sf write $loadaddr 300000 200000
```

11.3 Programming Serial Flash Using DStream

To be added in future release.