



REALTEK

AN0500

Ameba-ZII application note

Abstract

Ameba-ZII is a high-integrated IC. Its features include 802.11 Wi-Fi, RF, Bluetooth and configurable GPIOs.

This manual introduce users how to develop Ameba-ZII, including SDK compiling and downloading image to Ameba-ZII.

COPYRIGHT

©2011 Realtek Semiconductor Corp. All rights reserved. No part of this document may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form or by any means without the written permission of Realtek Semiconductor Corp.

DISCLAIMER

Realtek provides this document “as is”, without warranty of any kind. Realtek may make improvements and/or changes in this document or in the product described in this document at any time. This document could include technical inaccuracies or typographical errors.

TRADEMARKS

Realtek is a trademark of Realtek Semiconductor Corporation. Other names mentioned in this document are trademarks/registered trademarks of their respective owners.

USING THIS DOCUMENT

This document is intended for the software engineer’s reference and provides detailed programming information.

Though every effort has been made to ensure that this document is current and accurate, more information may have become available subsequent to the production of this guide.

REVISION HISTORY

Revision	Release Date	Summary
0.1	2018/11/27	Initial draft
0.2	2019/01/09	Add OTA part
0.3	2019/02/25	Add EVB V2.0 build and image tool environment
0.4	2019/02/28	Add How to enable secure boot & boot time
0.5	2019/03/01	Add DEV_2V0 board user guide
0.6	2019/03/06	Add trust zone project
0.7	2019/03/12	Update OTA implementation method
0.8	2019/03/27	Update TrustZone layout
0.9	2019/04/02	Add How to generate flash image combines both firmware1 and firmware2

COPYRIGHT	2
DISCLAIMER	2
TRADEMARKS	2
USING THIS DOCUMENT	2
REVISION HISTORY	3
1 Demo Board User Guide	9
1.1 Demo board overview (DEV_1V0)	9
1.2 Demo board overview (DEV_2V0)	11
1.3 Pin Mux Alternate Functions (DEV_1V0)	13
1.3.1 Pin mux table	13
1.3.2 Pin-Out Reference.....	14
1.4 Pin Mux Alternate Functions (DEV_2V0)	15
1.4.1 Pin mux table	15
1.4.2 Pin-Out Reference.....	16
1.5 Module Features	17
2 SDK Build Environment Setup	19
2.1 Introduction.....	19
2.2 Debugger Settings	19
2.2.1 J-Link	19
2.2.2 OpenOCD/CMSIS-DAP (TBD).....	22
2.3 Log UART Settings	22
2.3.1 EVB V1.0.....	22
2.3.2 EVB V2.0.....	23
2.4 IAR Environment	24
2.4.1 Install and setup IAR IDE in Windows	24
2.4.2 IAR Project introduction	24
2.4.3 IAR memory configuration.....	28
2.4.4 IAR memory overflow	29
2.5 GCC Environment (TBD)	29
3 Image Tool.....	30
3.1 Introduction.....	30
3.2 Environment Setup.....	30

3.2.1	Hardware Setup	30
3.2.2	Software Setup.....	32
3.3	Image download.....	32
4	Memory Layout.....	33
4.1	Memory type.....	33
4.2	Flash memory layout.....	33
4.2.1	Partition table	34
4.2.2	System data.....	36
4.2.3	Calibration data.....	37
4.2.4	Boot image	37
4.2.5	Firmware 1/Firmware 2	38
4.3	SRAM layout	40
4.4	TrustZone Memory Layout.....	41
5	Boot process	42
5.1	Boot flow	42
5.2	Secure boot	42
5.2.1	Secure boot flow	42
5.2.2	Partition table and Boot image decryption flow	43
5.2.3	Secure boot use scenario	45
5.2.4	How to enable secure boot.....	46
5.3	Boot time (TBD).....	57
6	Secure JTAG/SWD	58
6.1	Functional description (TBD).....	58
6.2	How to enable and use Secure JTAG/SWD (TBD)	58
7	Over-the-Air (OTA) Firmware Update.....	59
7.1	OTA operation flow	60
7.2	Boot process flow	61
7.3	Upgraded partition.....	62
7.4	Firmware image output.....	63
7.4.1	OTA firmware swap behavior	63
7.4.2	Configuration for building OTA firmware	64
7.5	Implement OTA over Wi-Fi.....	65

7.5.1	OTA using local download server base on socket	65
7.5.2	OTA using local download server based on HTTP	68
7.6	OTA signature.....	71

List of Figures

Figure 1-1 Top View of Ameba-ZII 1V0 Demo Board	9
Figure 1-2 Ameba-ZII 1V0 Demo Board PCB Layout	10
Figure 1-3 Top View of Ameba-ZII 2V0 Dev Board	11
Figure 1-4 Ameba-ZII 2V0 Dev Board PCB Layout.....	12
Figure 1-5 Pin Out Reference	14
Figure 1-6 Pin Out Reference for DEV_2V0	16
Figure 2-1 Connection between J-Link Adapter and Ameba-ZII SWD connector.....	19
Figure 2-2 Outlook of J-Link adapter and Ameba-ZII SWD connection	20
Figure 2-3 J-Link GDB server UI under Windows	21
Figure 2-4 J-Link GDB server connect under Windows.....	22
Figure 2-5 Log UART via FT232 on EVB V1.0.....	23
Figure 2-6 Log UART via FT232 on EVB V2.0.....	23
Figure 3-1 AmebaZII Image Tool UI.....	30
Figure 3-2 Ameba-ZII EVB V1.0 Hardware Setup	31
Figure 3-3 Ameba-ZII EVB V2.0 Hardware Setup	31
Figure 4-1 Address Allocation of Different Memories on Ameba-ZII	33
Figure 4-2 Flash memory layout	34
Figure 4-3 TrustZone memory layout	41
Figure 5-1 Overview of boot flow	42
Figure 5-2 Secure boot flow	43
Figure 5-3 Partition table and boot image decryption flow	44
Figure 5-4 secure boot use scenario	45
Figure 7-1 Methodology to Update Firmware via OTA	59
Figure 7-2 OTA Process Flow	60
Figure 7-3 Boot Process Flow.....	61
Figure 7-4 OTA update procedure	62
Figure 7-5 OTA Firmware SWAP Procedure.....	63

List of Table

Table 1-1 GPIOA Pin MUX: DEV_1V0 Board	13
Table 1-2 GPIOA Pin MUX: DEV_2V0 Board	15
Table 1-3 RTL8720C Features & Specifications.....	18
Table 4-1 Size of Different Memories on Ameba-ZII	33
Table 4-2 Description of flash layout	34
Table 4-3 The layout of Partition table	35
Table 4-4 OTA/MP TRAP map	35
Table 4-5 Layout of system data	36
Table 4-6 Definition for OTA section in system data	36
Table 4-7 Definition for Flash section in system data.....	37
Table 4-8 Definition for Log UART section in system data	37
Table 4-9 The layout of boot image	37
Table 4-10 The layout of firmware image.....	38
Table 4-11 AmebaZII DTCM (256KB) memory layout	40
Table 4-12 Description of RAM layout	41

1 Demo Board User Guide

1.1 Demo board overview (DEV_1V0)

RTL8720C embedded on Ameba-ZII DEV demo board, which consists of various I/O interfaces. The corresponding HDK (Hardware Development Kit) documents are available at:

- Module HDK version: HDK-XXXX
- DEV HDK version: RTL-AMEBAZII_DEV01_1V0

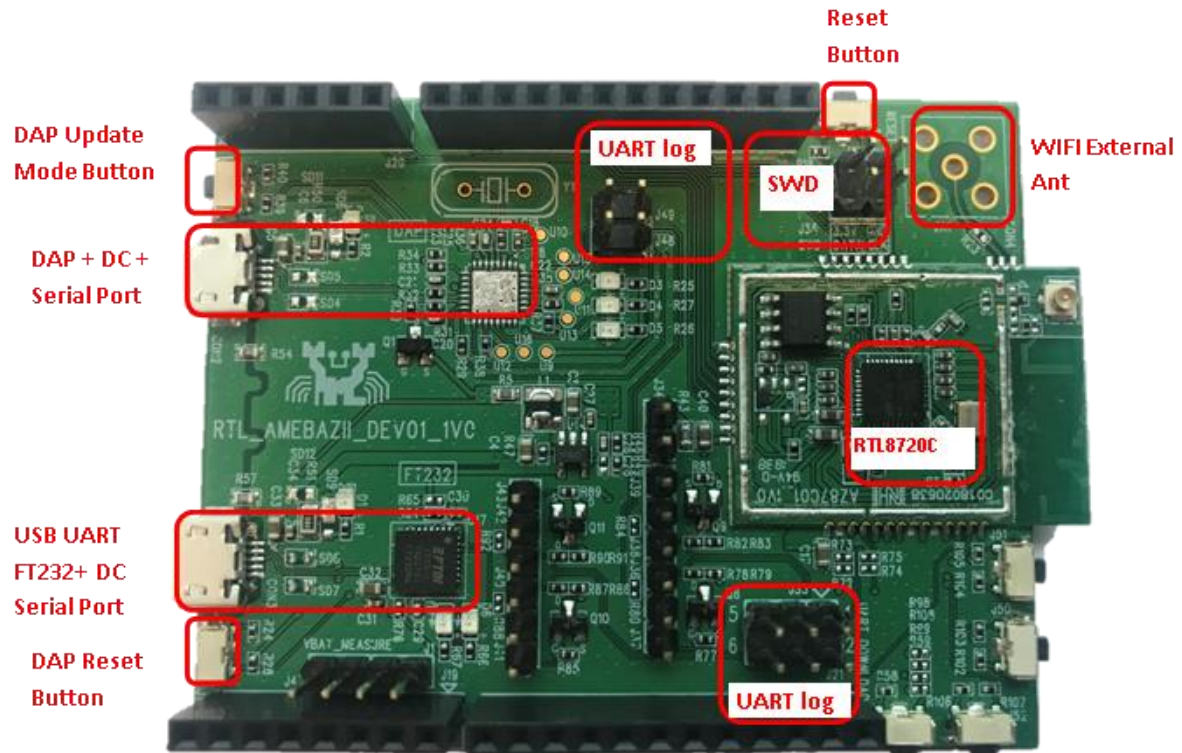


Figure 1-1 Top View of Ameba-ZII 1V0 Demo Board

1.2 Demo board overview (DEV_2V0)

The corresponding HDK (Hardware Development Kit) documents are available at:

- Module HDK version: HDK-XXXX
- DEV HDK version: RTL-AMEBAZII_DEV_2V0

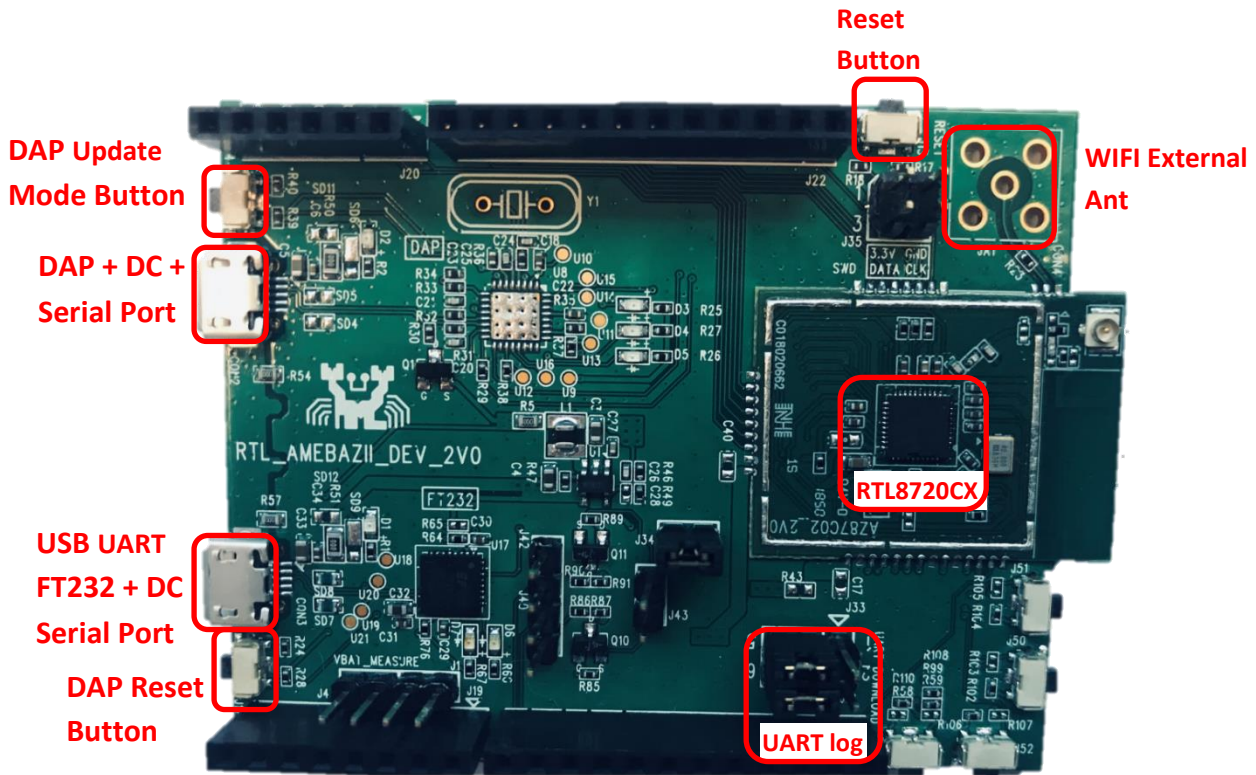


Figure 1-3 Top View of Ameba-ZII 2V0 Dev Board

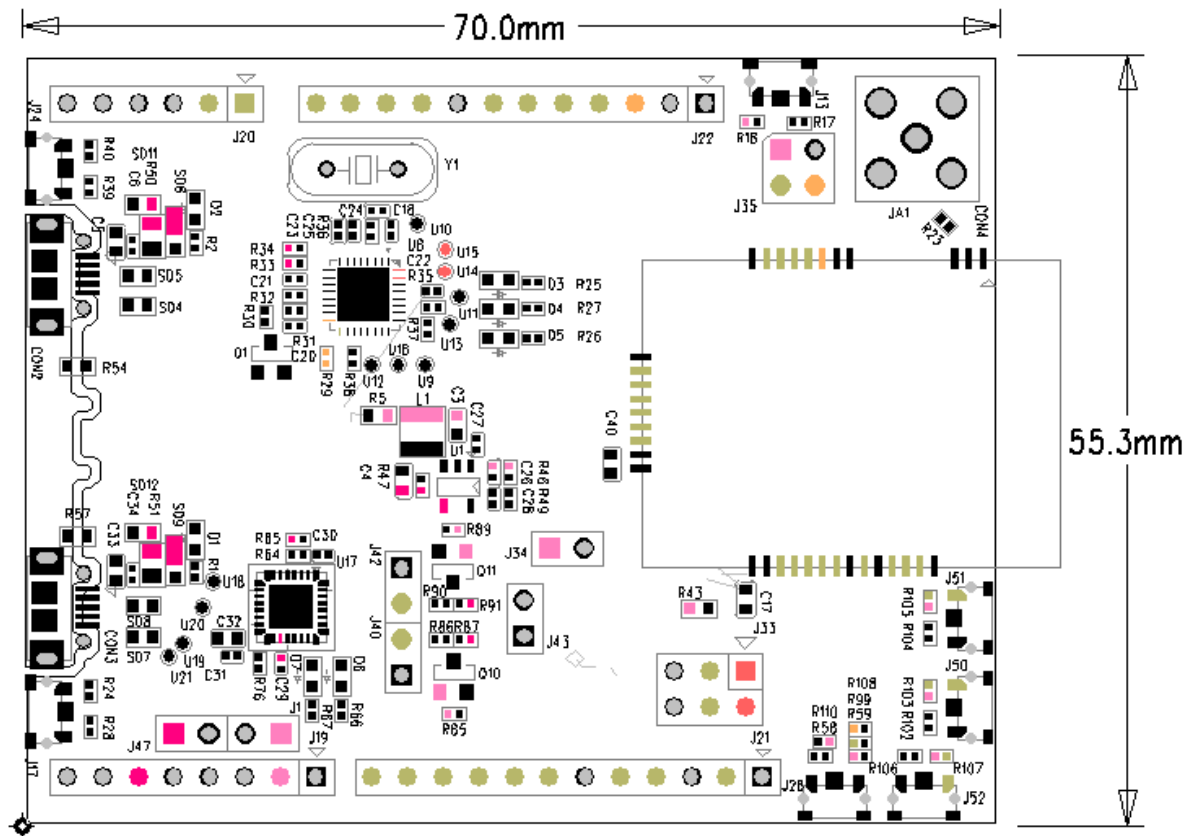


Figure 1-4 Ameba-ZII 2V0 Dev Board PCB Layout

1.3 Pin Mux Alternate Functions (DEV_1V0)

1.3.1 Pin mux table

<i>Pin Name</i>	<i>SPIC-Flash/SDIO</i>	<i>JTAG</i>	<i>UART</i>	<i>SPI/WL_LED/EXT_32K</i>	<i>I2C</i>	<i>PWM</i>
<i>GPIOA_0</i>		JTAG_CLK	UART1_IN	EXT_32K		PWM[0]
<i>GPIOA_1</i>		JTAG_TMS	UART1_OUT	BT_LED		PWM[1]
<i>GPIOA_2</i>		JTAG_TDO	UART1_IN	SPI_CS _n	I2C_SCL	PWM[2]
<i>GPIOA_3</i>		JTAG_TDI	UART1_OUT	SPI_SCL	I2C_SDA	PWM[3]
<i>GPIOA_4</i>		JTAG_TRST	UART1_CTS	SPI_MOSI		PWM[4]
<i>GPIOA_5</i>			UART1_RTS	SPI_MISO		PWM[5]
<i>GPIOA_6</i>						PWM[6]
<i>GPIOA_7</i>	SPI_M_CS			SPI_CS _n		
<i>GPIOA_8</i>	SPI_M_CLK			SPI_SCL		
<i>GPIOA_9</i>	SPI_M_DATA[2]		UART0_RTS	SPI_MOSI		
<i>GPIOA_10</i>	SPI_M_DATA[1]		UART0_CTS	SPI_MISO		
<i>GPIOA_11</i>	SPI_M_DATA[0]		UART0_OUT		I2C_SCL	PWM[0]
<i>GPIOA_12</i>	SPI_M_DATA[3]		UART0_IN		I2C_SDA	PWM[1]
<i>GPIOA_13</i>			UART0_IN			PWM[7]
<i>GPIOA_14</i>	SDIO_INT		UART0_OUT			PWM[2]
<i>GPIOA_15</i>	SD_D[2]		UART2_IN	SPI_CS _n	I2C_SCL	PWM[3]
<i>GPIOA_16</i>	SD_D[3]		UART2_OUT	SPI_SCL	I2C_SDA	PWM[4]
<i>GPIOA_17</i>	SD_CMD					PWM[5]
<i>GPIOA_18</i>	SD_CLK					PWM[6]
<i>GPIOA_19</i>	SD_D[0]		UART2_CTS	SPI_MOSI	I2C_SCL	PWM[7]
<i>GPIOA_20</i>	SD_D[1]		UART2_RTS	SPI_MISO	I2C_SDA	PWM[0]
<i>GPIOA_21</i>			UART2_IN		I2C_SCL	PWM[1]
<i>GPIOA_22</i>			UART2_OUT	LED_0	I2C_SDA	PWM[2]
<i>GPIOA_23</i>				LED_0		PWM[7]

Table 1-1 GPIOA Pin MUX: DEV_1V0 Board

1.3.2 Pin-Out Reference

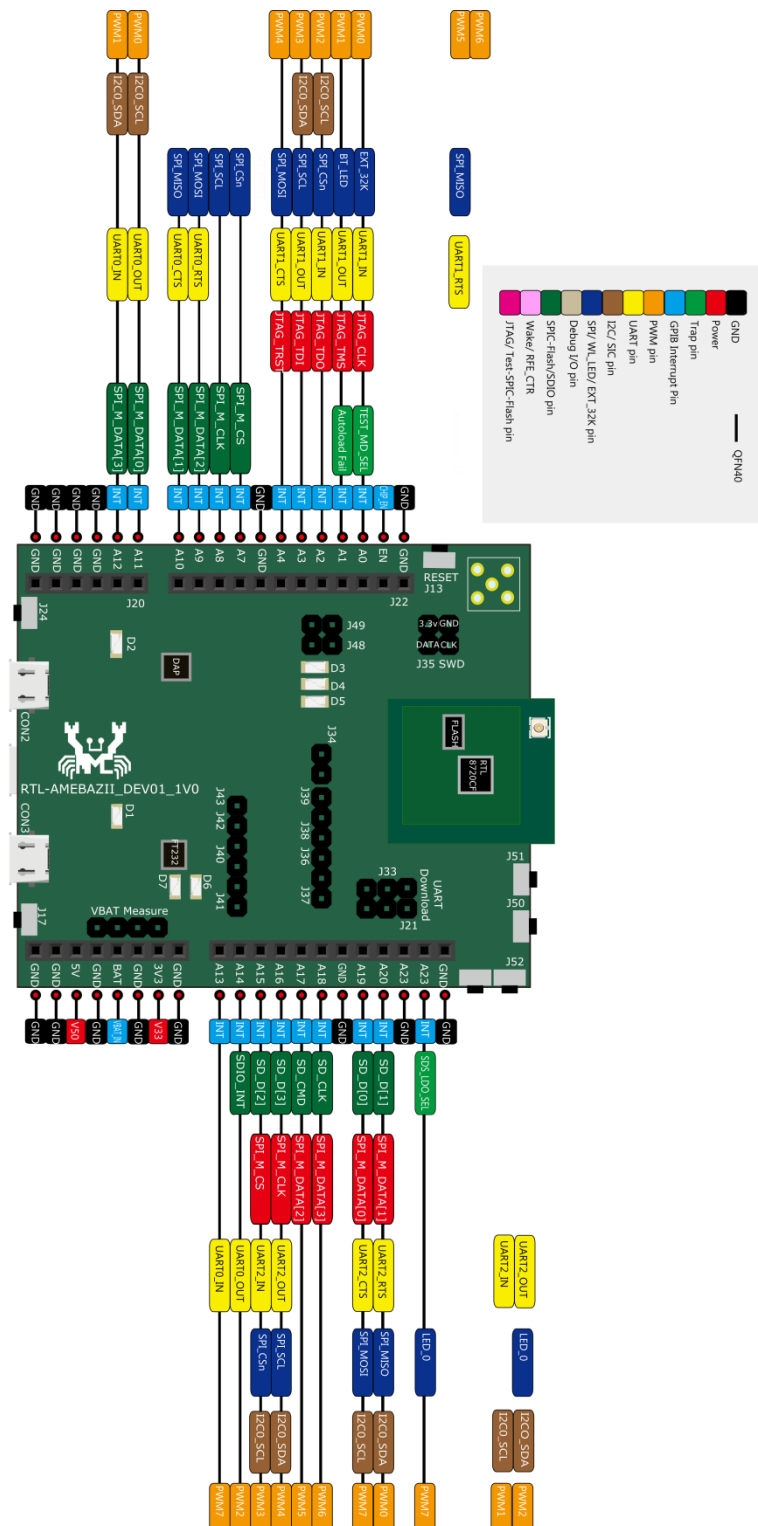


Figure 1-5 Pin Out Reference

1.4 Pin Mux Alternate Functions (DEV_2V0)

1.4.1 Pin mux table

Pin Name	SPIC-Flash/SDIO	JTAG	UART	SPI/WL_LED/EXT_32K	I2C	PWM
GPIOA_0		JTAG_CLK	UART1_IN	EXT_32K		PWM[0]
GPIOA_1		JTAG_TMS	UART1_OUT	BT_LED		PWM[1]
GPIOA_2		JTAG_TDO	UART1_IN	SPI_CS _n	I2C_SCL	PWM[2]
GPIOA_3		JTAG_TDI	UART1_OUT	SPI_SCL	I2C_SDA	PWM[3]
GPIOA_4		JTAG_TRST	UART1_CTS	SPI_MOSI		PWM[4]
GPIOA_5			UART1_RTS	SPI_MISO		PWM[5]
GPIOA_6						PWM[6]
GPIOA_7	SPI_M_CS			SPI_CS _n		
GPIOA_8	SPI_M_CLK			SPI_SCL		
GPIOA_9	SPI_M_DATA[2]		UART0_RTS	SPI_MOSI		
GPIOA_10	SPI_M_DATA[1]		UART0_CTS	SPI_MISO		
GPIOA_11	SPI_M_DATA[0]		UART0_OUT		I2C_SCL	PWM[0]
GPIOA_12	SPI_M_DATA[3]		UART0_IN		I2C_SDA	PWM[1]
GPIOA_13			UART0_IN			PWM[7]
GPIOA_14	SDIO_INT		UART0_OUT			PWM[2]
GPIOA_15	SD_D[2]		UART2_IN	SPI_CS _n	I2C_SCL	PWM[3]
GPIOA_16	SD_D[3]		UART2_OUT	SPI_SCL	I2C_SDA	PWM[4]
GPIOA_17	SD_CMD					PWM[5]
GPIOA_18	SD_CLK					PWM[6]
GPIOA_19	SD_D[0]		UART2_CTS	SPI_MOSI	I2C_SCL	PWM[7]
GPIOA_20	SD_D[1]		UART2_RTS	SPI_MISO	I2C_SDA	PWM[0]
GPIOA_21			UART2_IN		I2C_SCL	PWM[1]
GPIOA_22			UART2_OUT	LED_0	I2C_SDA	PWM[2]
GPIOA_23				LED_0		PWM[7]

Table 1-2 GPIOA Pin MUX: DEV_2V0 Board

1.4.2 Pin-Out Reference

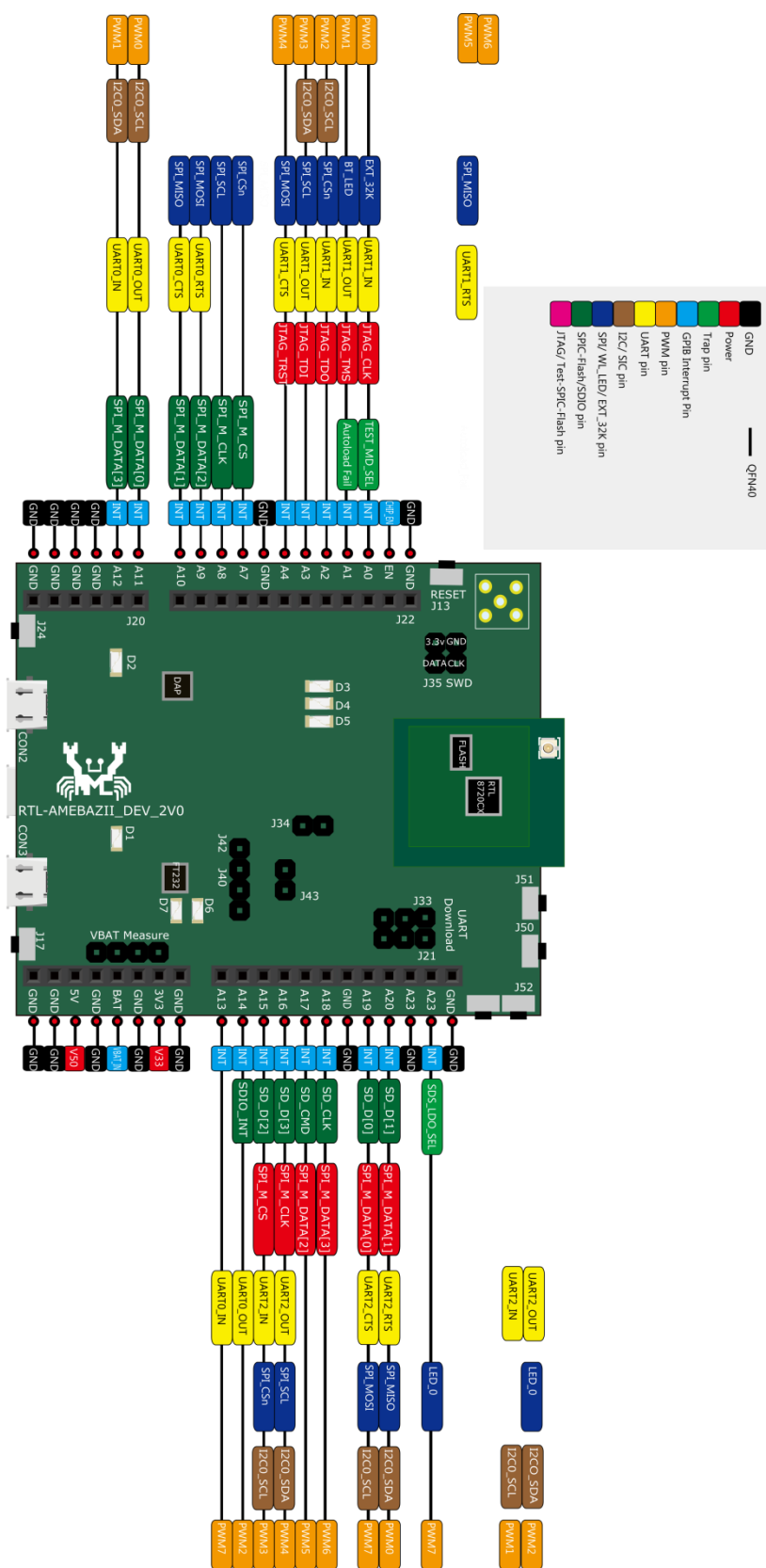


Figure 1-6 Pin Out Reference for DEV_2V0

1.5 Module Features

Feature list		RTL8720CF	RTL8720CM	RTL8710CX
Package	trays and tape-in-reel	(5x5mm ²) QFN40	(5x5mm ²) QFN40	(5x5mm ²) QFN40
Integrated core	Core type	KM4	KM4	KM4
	Core clock maximum freq.	100MHz	100MHz	100MHz
Memory	Internal ROM	384KB	384KB	384KB
	Internal SRAM	256KB	256KB	256KB
	Flash (2MB)	Yes	No	No
	pSRAM (4MB)	No	Yes	No
FPU	Floating point unit	No	No	No
SWD/JTAG		SWD/JTAG	SWD/JTAG	SWD/JTAG
Secure boot		Yes	Yes	Yes
Trust-zone-M		Yes	Yes	Yes
		(Fixed size: 32KB)	(Fixed size: 32KB)	(Fixed size: 32KB)
Backup register	Backup register for power save	No	No	No
F/W protection		No	No	No
WIFI	802.11 b/g/n	Yes	Yes	Yes
Bluetooth		Yes	Yes	No
BOR	BOR Detection	Yes	Yes	Yes
Peripherals	UART	3	3	3
	SPI Master	Max. 20Mbps	1	1
	SPI Slave	Max. 4Mbps	1	1
	I2C	Max. 400Kbps	1	1
	GDMA	2 channel	1	1
	GPIO	IN/OUT/INT	20	14
	I2S		0	0
	RTC	D/H/M/S	0	0
	OUTPUT	0	0	0

	Timer	Basic timer use 32K	1	1	1
		Advanced timer use 40M	8	8	8
	PWM	OUTPUT	8	8	8
		INPUT Capture	0	0	0
	WDG		1	1	1
	USB device		0	0	0
	SDIO 2.0 Device		1	1	1
External 32K	External 32K		1	1	1
Dsleep Wakepin	Deep sleep wake pin		20	14	14

Table 1-3 RTL8720C Features & Specifications

2 SDK Build Environment Setup

2.1 Introduction

This document illustrates how to build Realtek WiFi SDK. We will focus on both Windows platform and Linux distribution in this document. For Windows, we use Windows 7 64-bit as our platform.

And this SDK supports two compiling toolchains, both IAR and GCC.

2.2 Debugger Settings

To download code or debug on Ameba-ZII, user needs to make sure the debugger is setup properly first.

Ameba-ZII supports J-Link and CMSIS-DAP for code download and entering debugger mode. The settings are described below.

2.2.1 J-Link

Ameba-ZII supports J-Link debugger. We need to connect the SWD connector to J-Link debugger. The SWD pin definitions can refer to section 1.2.2 and the connection is shown as below. After finished these configuration, please connect it to PC side. Note that if you are using Virtual Machine as your platform, please make sure the USB connection setting between VM host and client is correct so that the VM client can detect the device.

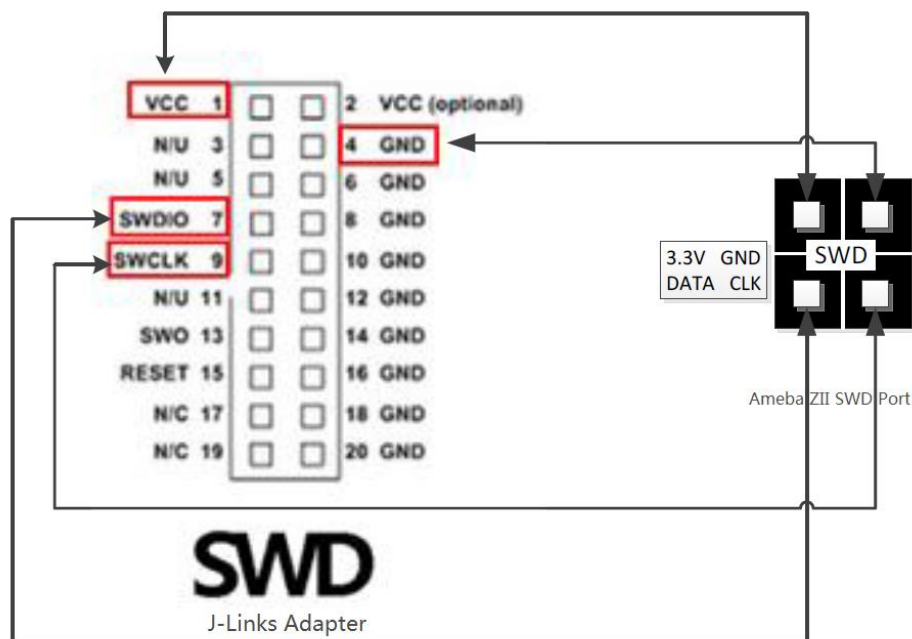


Figure 2-1 Connection between J-Link Adapter and Ameba-ZII SWD connector

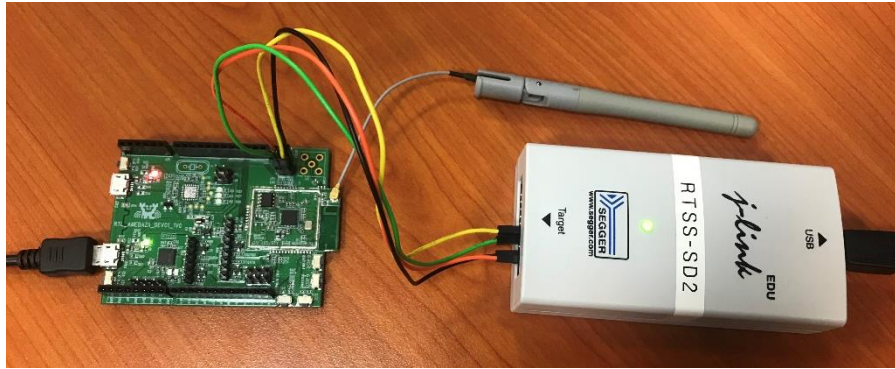


Figure 2-2 Outlook of J-Link adapter and Ameba-ZII SWD connection

Note: To be able to debug Ameba-ZII which is powered by Cortex-M33, user needs a J-Link debugger with the latest hardware version (Check [https://wiki.segger.com/Software and Hardware Features Overview](https://wiki.segger.com/Software_and_Hardware_Features_Overview) for details). J-Link EDU with hardware version V10 is used to prepare this document.

2.2.1.1 Windows

To be able to use J-Link debugger, user needs install J-Link GDB server first. For Windows, please check <http://www.segger.com> and download “J-Link Software and Documentation Pack” (<https://www.segger.com/downloads/jlink>).

Note: to support TrustZone feature, it’s better to download the latest version of J-Link Software. Version 6.40 is used to prepare this document.

To check whether the connection works fine, user can go to the location of SEGGER J-Link tool and run “JLinkGDBServer.exe”. Make sure the configuration is correct and click “OK”.

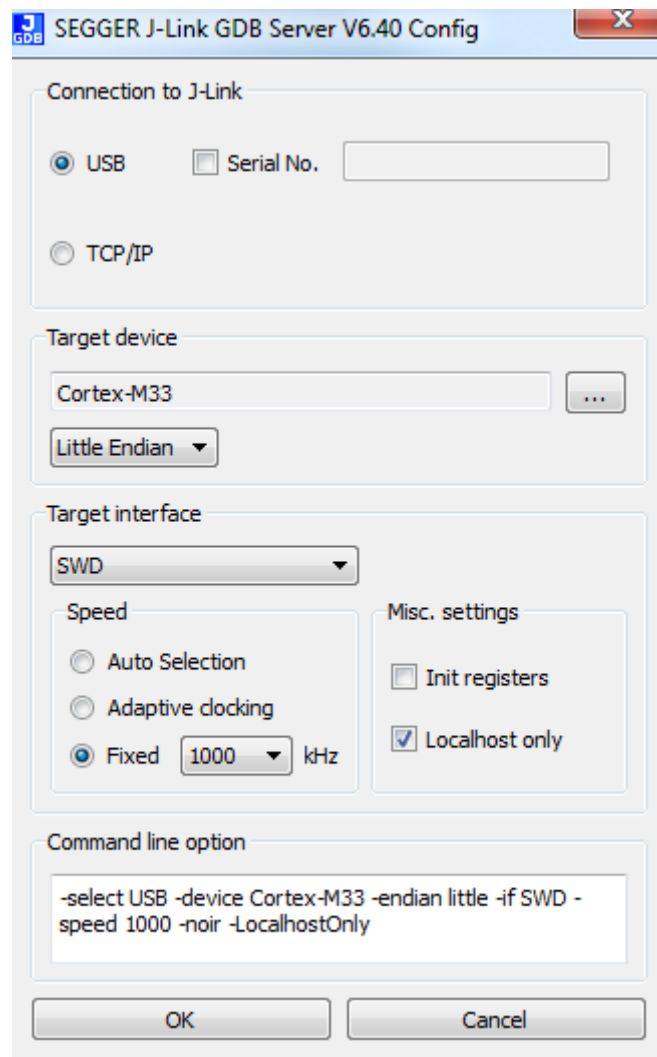


Figure 2-3 J-Link GDB server UI under Windows

Please check and make sure below information is shown properly.

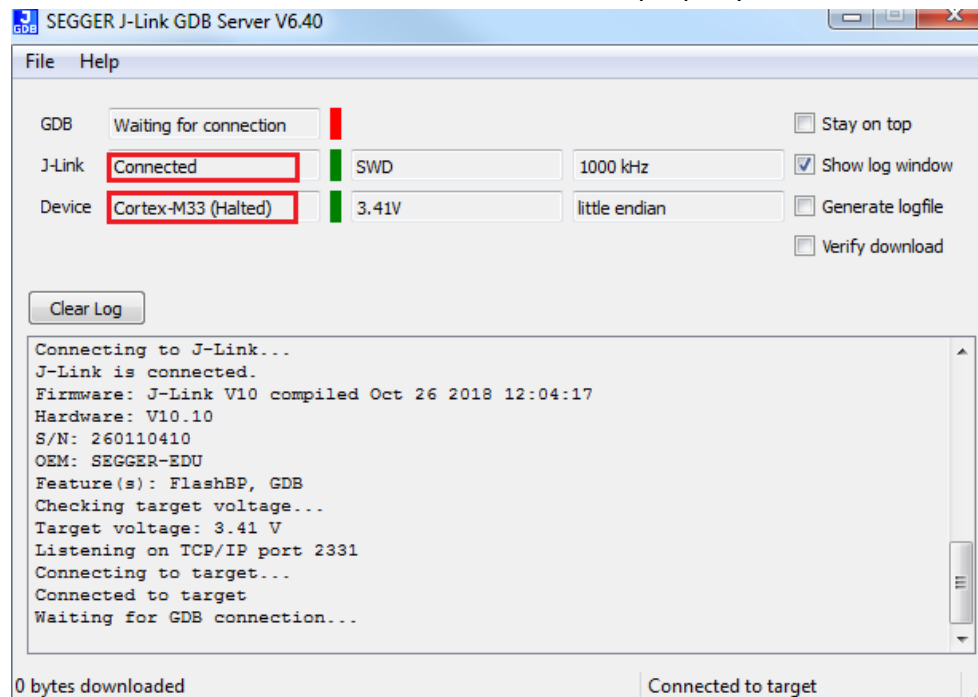


Figure 2-4 J-Link GDB server connect under Windows

2.2.1.2 Linux (TBD)

2.2.2 OpenOCD/CMSIS-DAP (TBD)

2.2.2.1 Windows (TBD)

2.2.2.2 Linux (TBD)

2.3 Log UART Settings

To be able to start development with the demo board, Log UART must be connected properly. Different versions of EVBs have different connections.

2.3.1 EVB V1.0

By default, UART2 (GPIOA_15/GPIOA_16, check figure 1-3) is used as system log UART. User needs to connect UART2 to CON3 (FT232) or CON2 (DAP).

- 1) Connection to log UART via FT232
PA15 <-> J41-1, PA16 <-> J43-1

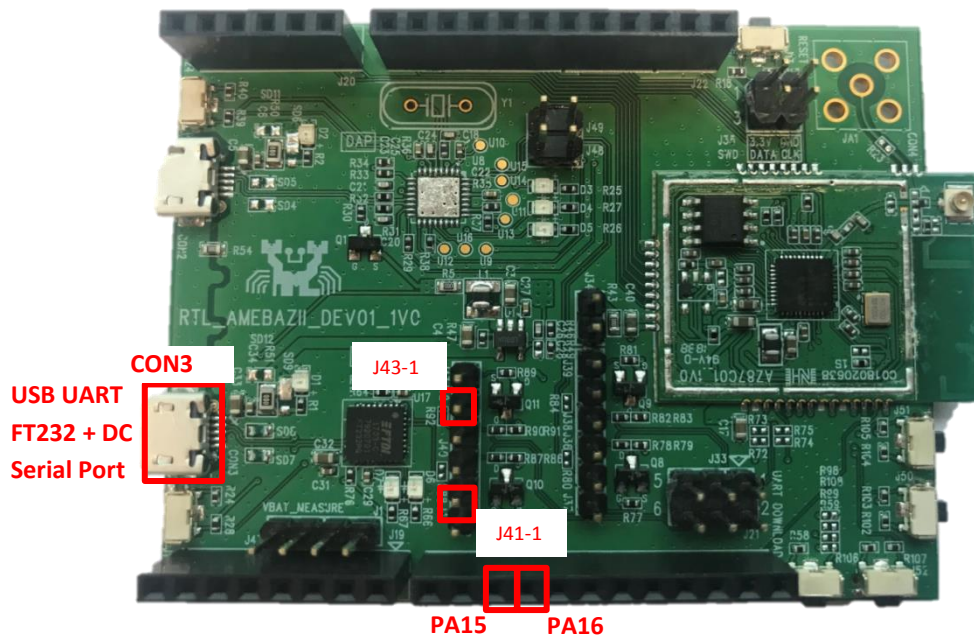


Figure 2-5 Log UART via FT232 on EVB V1.0

2) Connection to log UART via DAP (TBD)

2.3.2 EVB V2.0

By default, UART2 (GPIOA_15/GPIOA_16, check figure 1-3) is used as system log UART. User needs to connect jumpers to J33 for CON3 (FT232) or CON2 (DAP).

1) Connection to log UART via FT232

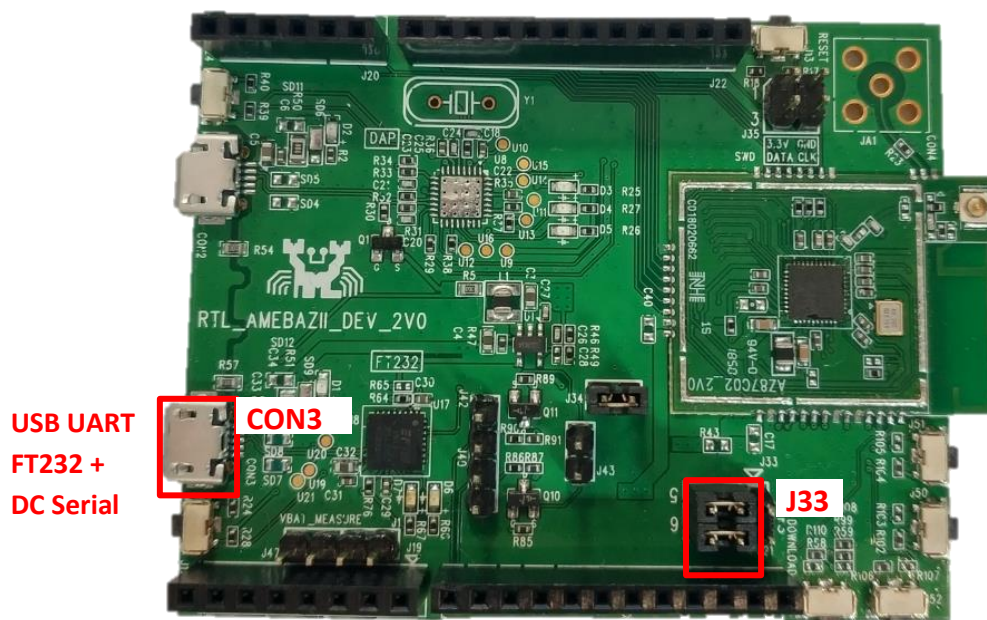


Figure 2-6 Log UART via FT232 on EVB V2.0

- 2) Connection to log UART via DAP (TBD)

2.4 IAR Environment

IAR IDE supports only Windows, so this section is applicable for Windows only.

2.4.1 Install and setup IAR IDE in Windows

IAR IDE provides the toolchain for Ameba-ZII. It allows users to write programs, compile and upload them to your board. Also, it supports step-by-step debug.

User can visit the official website of IAR Embedded Workbench, and install the IDE by following its instructions.

Note: Please use IAR version 8.30 or above.

2.4.2 IAR Project introduction

2.4.2.1 Ignore secure project

Currently users can use ignore secure mode. Project_is (ignore secure) is the project without TrustZone configuration. This project is easier to develop and suit for first-time developer.

2.4.2.1.1 Compile

- 1) Open SDK/project/realtek_amebaz2_v0_example/EWARM-RELEASE/Project_is.eww.
- 2) Confirm application_is in Work Space, right click application_is and choose “Rebuild All” to compile.
- 3) Make sure there is no error after compile.

2.4.2.1.2 Generating image binary

After compile, the images partition.bin, bootloader.bin, firmware_is.bin and flash_is.bin can be seen in the EWARM-RELEASE\Debug\Exe.

- 1) partition.bin stores partition table, recording the address of Boot image and firmware image;
- 2) bootloader.bin is bootloader image;
- 3) firmware_is.bin is application image;
- 4) flash_is.bin links partition.bin, bootloader.bin and firmware_is.bin. Users need to choose flash_is.bin when downloading the image to board by PG Tool.

2.4.2.1.3 Download

After a successfully compilation and flash_is.bin is generated without error, user can either

- 1) Directly download the image binary on to demo board from IAR IDE (as below)

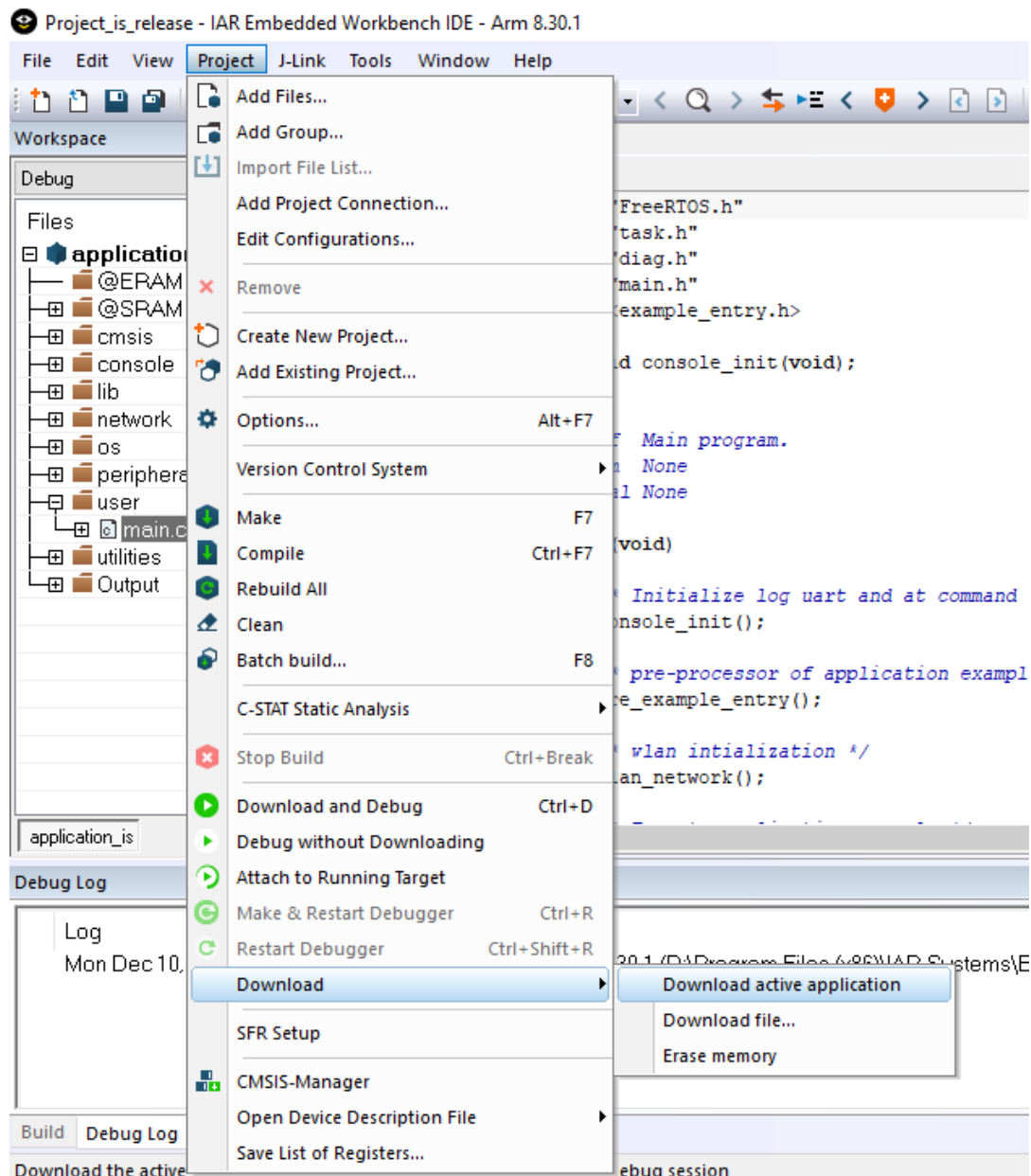


Figure 2-7 IAR download binary on flash

Note: Please make the project first when some code is modified before download the bin file on the board, otherwise the download will fail and below logs will be shown.

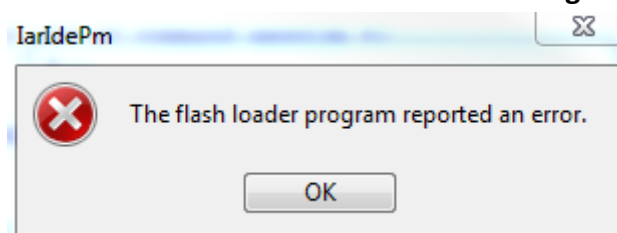


Figure 2-8 IAR download code on flash error message on IDE

```
Realtek Ameba-ZII Flash Loader Build @ 19:38:43, Nov 28 2018
DownloadingImage size (8b80980f) is invalid! Make sure the image is generated before the download
```

Figure 2-9 IAR download code on flash error message on Log UART

- 2) Or using the PG tool for Ameba-ZII (Will not be shown here, please check chapter 3 for details).

2.4.2.2 TrustZone project (TBD)

If it is in trust zone mode, “*project_tz.eww*” (tz means trust zone) is the project with trust zone configuration. You can decide your code is secure or not by putting your code to *application_s* or *application_ns*. Please note that this release version only has simple demo for Trust Zone, the complete version will be released in next version.

2.4.2.2.1 Compile

- 1) Open SDK/project/realtek_amebaz2_v0_example/EWARM-RELEASE/Project_tz.eww.
- 2) Confirm *application_ns* and *application_s* are in Work Space.
- 3) Right click *application_s* and click “Rebuild All” to compile *application_s* first. If *application_s* is compiled successfully, it will generate a file named *application_s_import_lib.o* and the file will be put in “lib” folder of *application_ns*, shown in Figure 2-10.

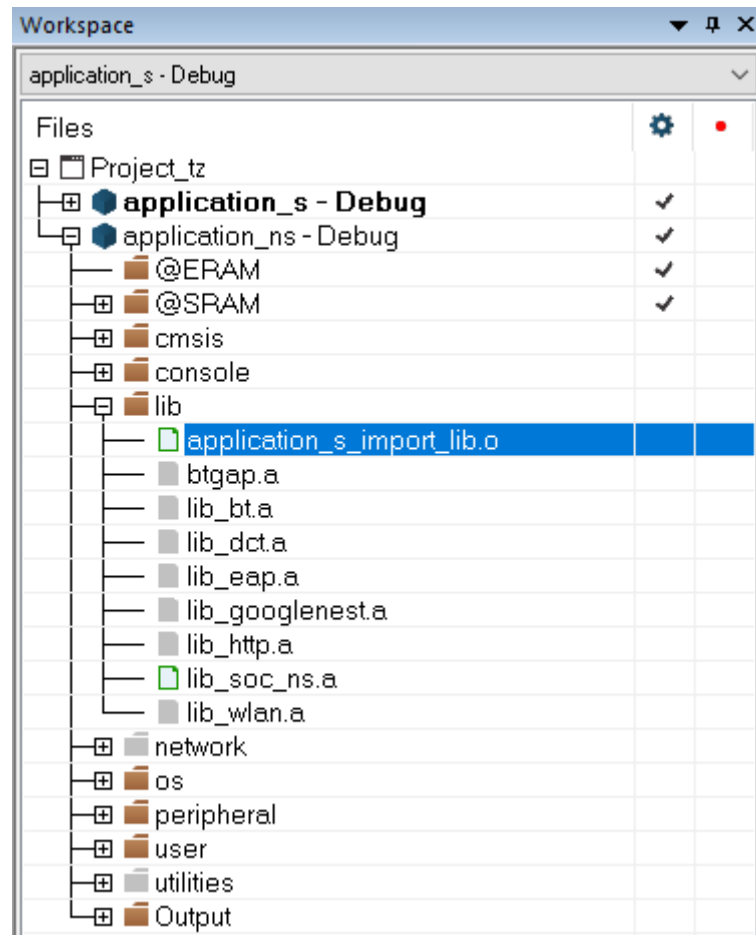


Figure 2-10 application_s compile result

- 4) Make sure *application_s* is compiled successfully and the file *application_s_import_lib.o* has been put under “lib” in *application_ns*.
- 5) Right click *application_ns* and click “Rebuild All” to build *application_ns*.
- 6) Make sure the *application_ns* is compiled successfully.

2.4.2.2.2 Generating image binary

After compile, the images *partition.bin*, *bootloader.bin*, *firmware_tz.bin* and *flash_tz.bin* can be seen in the EWARM-RELEASE\Debug\Exe.

- 1) *partition.bin* stores partition table, recording the address of Boot image and firmware image;
- 2) *bootloader.bin* is bootloader image;
- 3) *firmware_tz.bin* is application image;
- 4) *flash_tz.bin* links *partition.bin*, *bootloader.bin* and *firmware_tz.bin*. Users need to choose *flash_tz.bin* when downloading the image to board by PG Tool.

2.4.2.2.3 Download

After a successfully compilation and *flash_tz.bin* is generated without error, user can either

- 1) Directly download the image binary on to demo board from IAR IDE (as below)

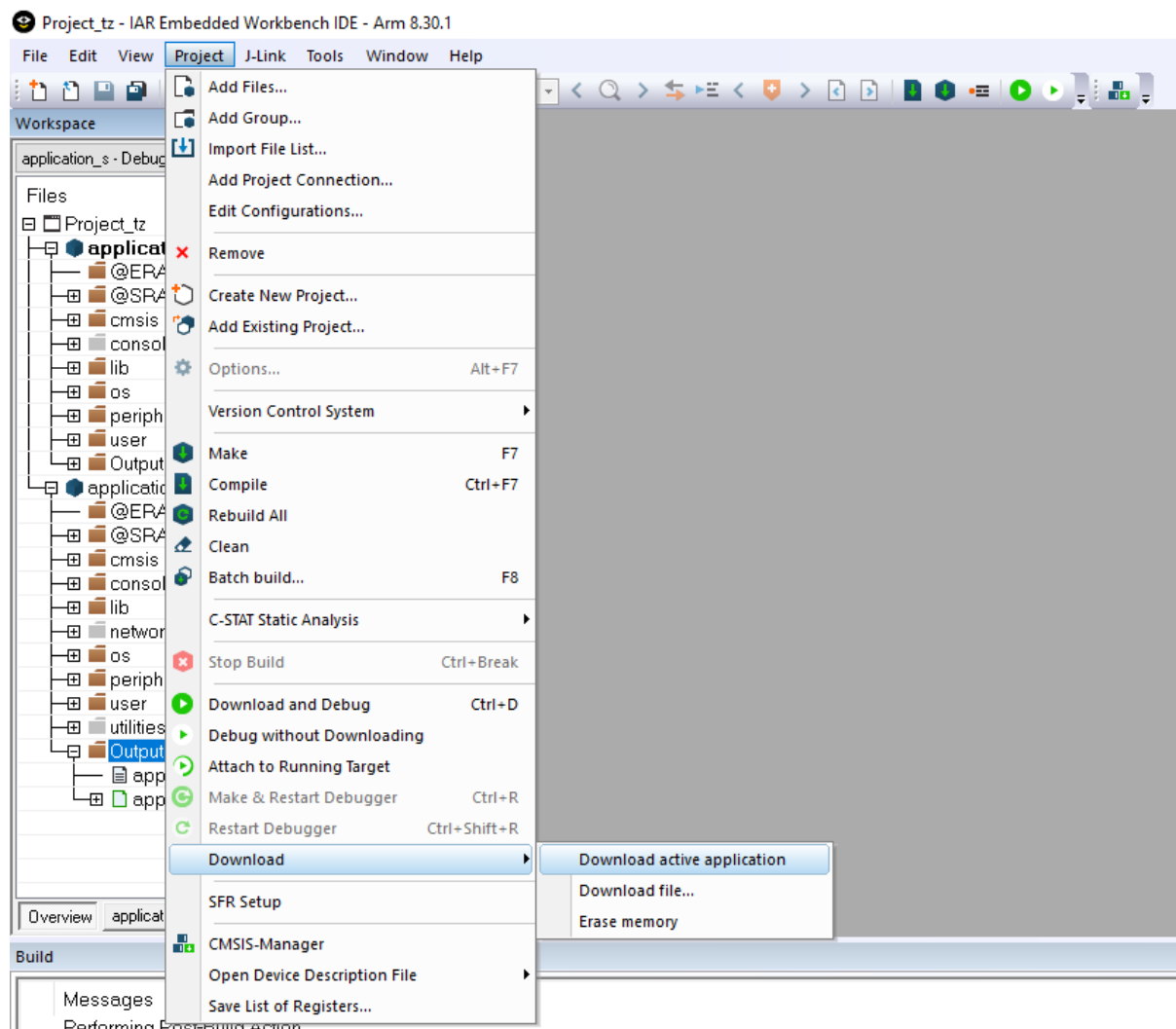


Figure 2-11 IAR download binary on flash

- 2) Or using the PG tool for Ameba-ZII (Will not be shown here, please check chapter 3 for details).

2.4.3 IAR memory configuration

The whole memory layout of Ameba-ZII can refer to chapter Memory Layout.

And there will be some extra configurations user needs to do if they want to put some code to certain memory region.

2.4.3.1 Configure memory from IAR IDE

In IAR Workspace (figure 2-8), there are “@ERAM” and “@SRAM” group. “@ERAM” represents external RAM, which can be known as PSRAM. “@SRAM” represents SRAM. Except “@ERAM” and “@SRAM” group, the rest of read-only section (TEXT and RODATA) will be placed on XIP.

If users want to link a particular file into PSRAM (need to make sure that PSRAM is available, refer to section 4.1.), users can drag-and-drop the files into “@ERAM”. For example, “test.c” will be linked to PSRAM as the graph above. If users want to place specific source file into SRAM, users can drag-and-drop the files into “@SRAM”. For example, “flash_api.c”, “flash_fatfs.c”, “hal_flash.c” is placed in SRAM as the graph above.

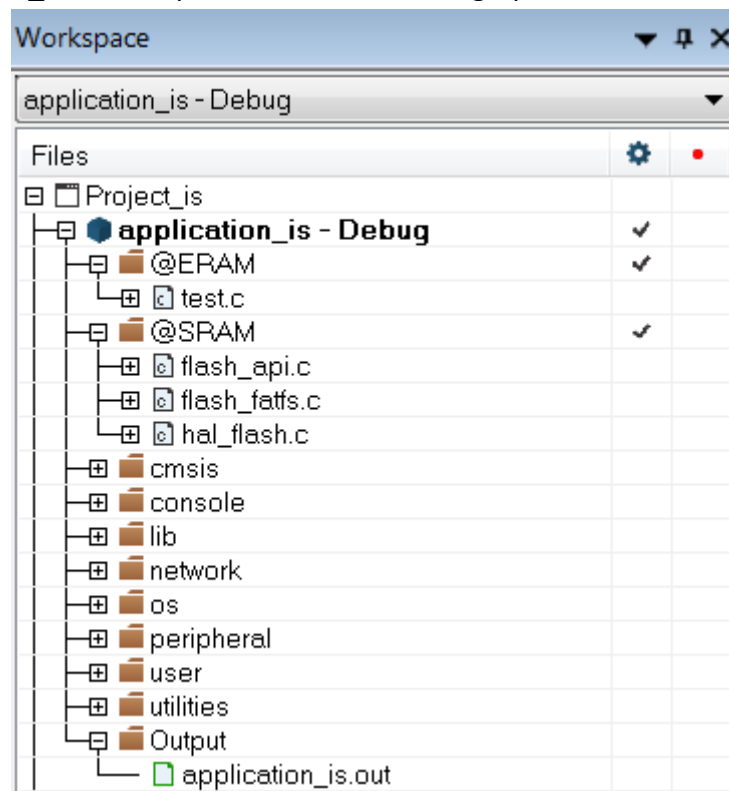


Figure 2-12 Overview of IAR Workspace

Note: There are some files which must NOT be linked to XIP section. Please keep the default settings of the IAR project if one doesn't know about this.

2.4.3.2 Configure memory from ICF file

IAR uses ICF (IAR Configuration File) to configure memory allocation so users can configure memory allocation by ICF file.

ICF file of Ameba-ZII locates: “SDK/project/realtek_amebaz2_v0_example/EWARM-RELEASE/ application_is.icf”

Open “application_is.icf” with any text editor. There are some memory regions in it, which is:

- DTCM_RAM_region
- ERAM_region
- XIP_FLASH_region

Users can reference IAR document if they don’t know the format of ICF.

2.4.4 IAR memory overflow

In default, Ameba-ZII place read-only (TEXT and RODATA) section in XIP area. If XIP does not have enough space, it will show the errors as below while linking.

Error[Lp011]: section placement failed
unable to allocate space for sections/blocks with a total estimated minimum size of 0x110’3e5e bytes (max align 0x8) in <[0x9b00’0140-0x9bff’ffff]> (total uncommitted space 0xff’fec0).

The solution is to either minimize the code or move the code to other memory region. Same rule applies to SRAM and PSRAM if it’s available.

2.5 GCC Environment (TBD)

3 Image Tool

3.1 Introduction

This chapter introduces how to use Image Tool to generate and download images. As show in Figure 3-1, Image Tool has two menu pages:

- Download: used as image download server to transmit images to Ameba through UART.
- Generate: contact individual images and generate a composite image.

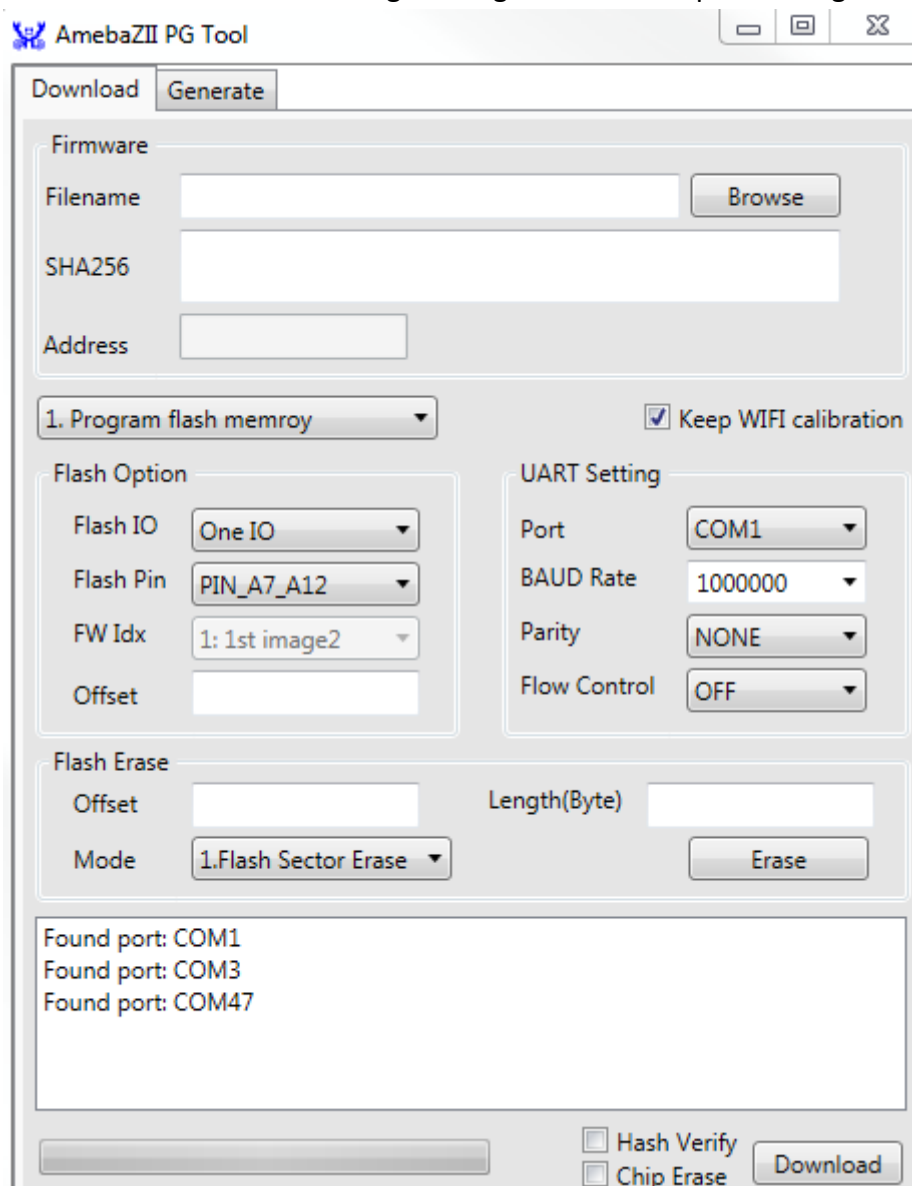


Figure 3-1 AmebaZII Image Tool UI

3.2 Environment Setup

3.2.1 Hardware Setup

There is difference for hardware setup of different EVB versions.

3.2.1.1 EVB V1.0

User needs to connect CON3 to user's PC via a Micro USB cable.

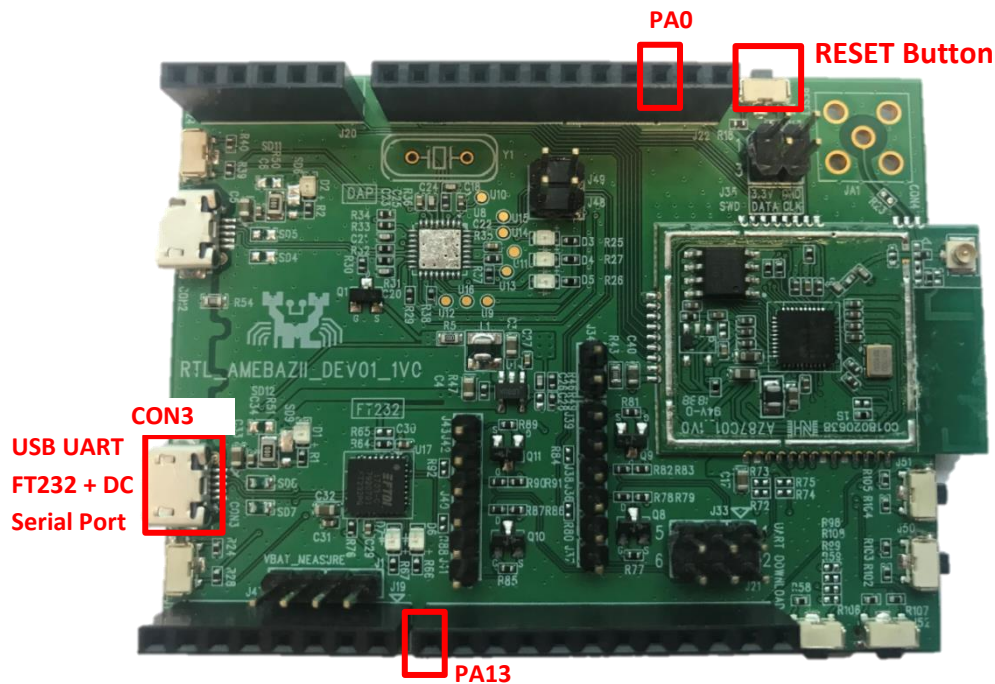


Figure 3-2 Ameba-ZII EVB V1.0 Hardware Setup

3.2.1.2 EVB V2.0

User needs to connect CON3 to user's PC via a Micro USB cable. Add jumpers for J34 and J33 (J33 is for log UART which has two jumpers) if there is no connections.

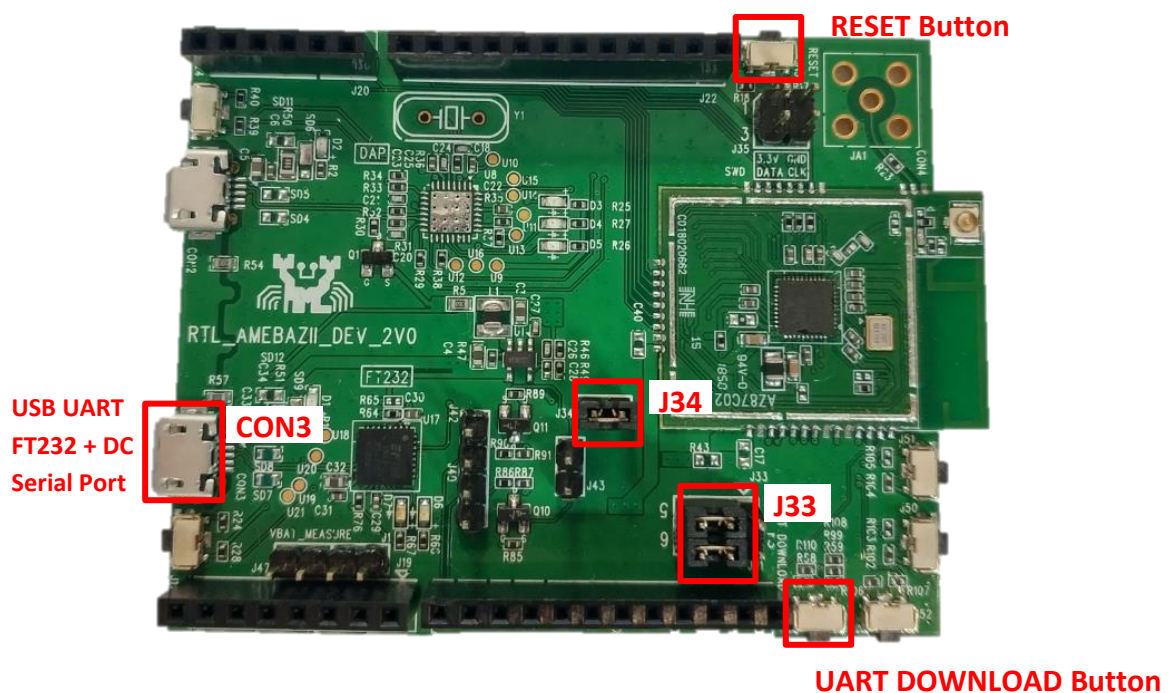


Figure 3-3 Ameba-ZII EVB V2.0 Hardware Setup

3.2.2 Software Setup

- Environment Requirements: EX. WinXP, Win 7 Above, Microsoft .NET Framework 3.5
- AmebaZII_PGTool_v1.0.1.exe

3.3 Image download

User can download the image on demo board by following below steps:

- 1) Trigger Ameba-ZII chip enter UART download mode by
 - a. Different version of EVBs has different ways to enter UART download mode
 - (a) EVB V1.0
Connect both PA0 and PA13 to 3.3v, and make sure the log UART is connected properly (refer to section 2.3).
 - (b) EVB V2.0
Press and hold the UART DOWNLOAD button then press the RESET button and release both buttons. And make sure the log UART is connected properly (refer to section 2.3).
 - b. Press RESET button and release, then below log should be shown on log UART console. (Please remember to disconnect the log UART before using Image Tool to download, because the tool will also need to connect to this log UART port)

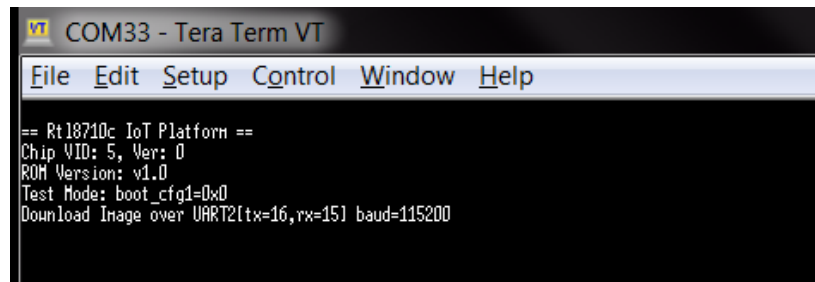


Figure 3-4 Ameba-ZII UART download mode

- 2) Choose the correct UART port (use rescan to update the port list)
- 3) “Browse” to choose the image to be downloaded (flash_xx.bin)
- 4) Choose “Mode” “1. Program flash memory”
- 5) Choose correct “Flash Pin” according to the IC part number

Flash Pin	IC part number
PIN_A7_A12	RTL8710CX/RTL8720CM
PIN_B6_B12	RTL8720CF

- 6) Click “Download” to start downloading image. While downloading, the status will be shown on the left bar.

Note: it’s recommended to use the default settings unless user is familiar with them.

4 Memory Layout

This chapter introduces the memory in Ameba-ZII, including ROM, RAM, SRAM, PSRAM and Flash. Also, this chapter provides the guide that users can place their program to the specific memory to fit user's requirement. However, some of program is fixed in specific memory and cannot be moved. This program is also discussed in this chapter.

4.1 Memory type

The size and configuration is as shown below

	Size(bytes)	Description
ROM	384K	Reserved
RAM	256K	Internal DTCM
PSRAM	4M	MCM PSRAM, only available on RTL8720CM
XIP	16M	Execute in Place, section TEXT and RODATA, virtual address remapped by SCE (Secure Code Engine). Physical address of Flash starts from 0x98000000 which can refer to the datasheet for more details.

Table 4-1 Size of Different Memories on Ameba-ZII

And the graph of configuration on Ameba-ZII is as shown below:

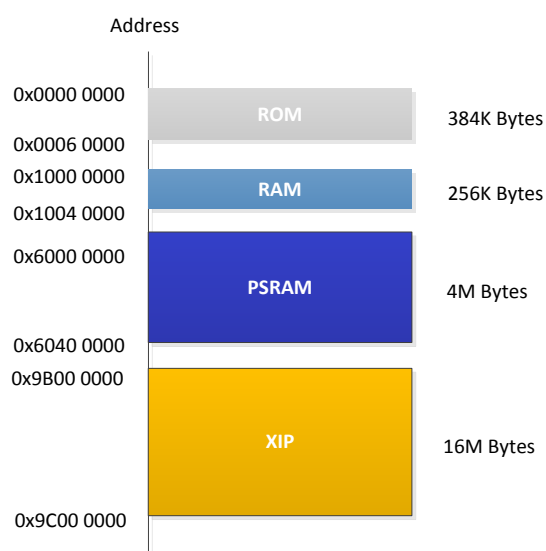


Figure 4-1 Address Allocation of Different Memories on Ameba-ZII

4.2 Flash memory layout

The default flash layout used in SDK is shown in below figure.

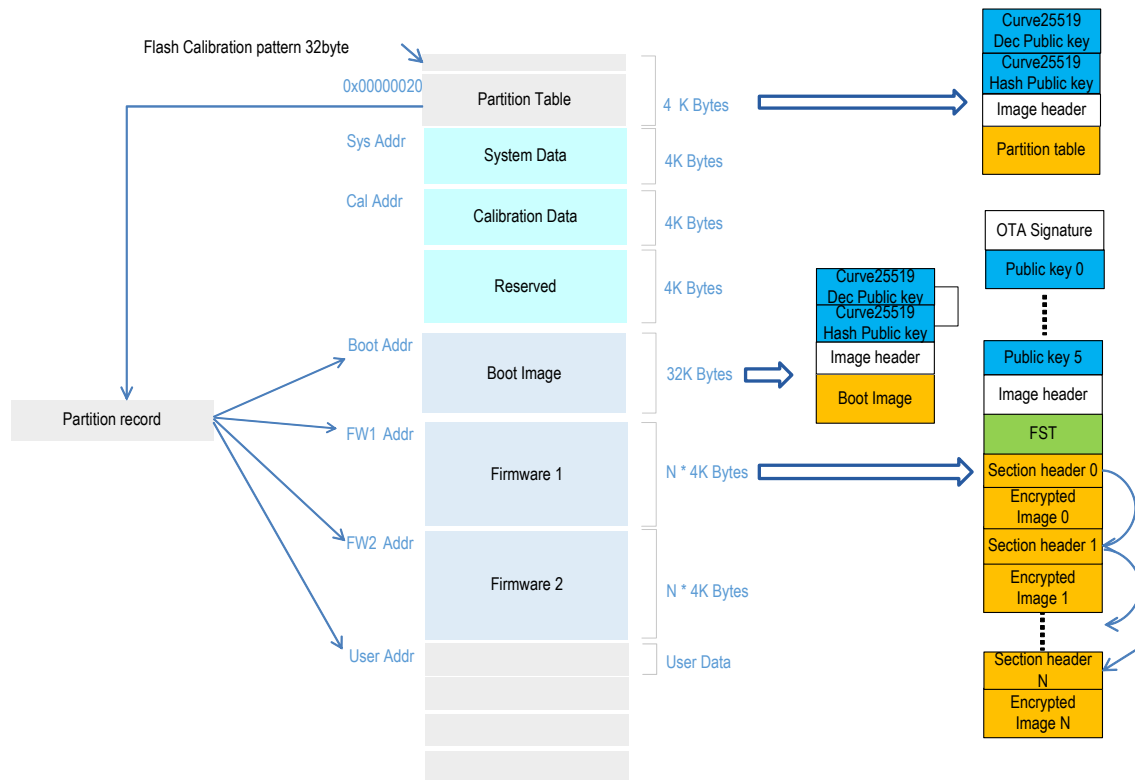


Figure 4-2 Flash memory layout

And the description of each block is in below Table 4-2.

Items	Start Offset Address	Limit Offset Address	Address adjustable	Size	Description	Mandatory
Partition table	0x00000000	0x00001000-1	N	4KB	The first 32 bytes is flash calibration pattern. The actual partition table starts from 0x20	Y
System data	0x00001000	0x00002000-1	N	4KB	To store some system settings	Y
Calibration data	0x00002000	0x00003000-1	N	4KB	RESERVED, user don't need to configure this portion	Y
Reserved	0x00003000	0x00004000-1	N	4KB	RESERVED, user don't need to configure this portion	Y
Boot image	0x00004000	0x0000C000-1	Y	32KB	Bootloader code/data	Y
Firmware 1	0x0000C000	0x0000C000 + N*4KB-1	Y	N*4KB	Application image	Y

Table 4-2 Description of flash layout

4.2.1 Partition table

4.2.1.1 The layout of partition table

The partition table stores below information

- Start address: the offset address on Flash for the image, align to 4K
- Length: the length of the image, align to 4K
- TYPE: type of the image (Pt=0/boot/sys/cal/user/fw1/fw2/var/mp/rdp/resv)
- DBG_SKIP : skip download to ram from flash when debug mode enable
- HKeyValid: indicates the Hash Key is valid(bit[0]!=0) or not(bit[0]=0)
- Hash key : to do all firmware validation (from first byte to end)
- User Data: user secret data
- Hash: from the first byte of partition table to the end of user data (two public keys + Header + partition info + partition records + user data), calculated before encryption if the encryption is on

4.2.1.2 How to configure partition table (TBD)

4.2.2 System data

System data section is the one which stores some system settings, including OTA section, Flash section, Log UART section etc... The size of system data section is 4KB.

	0x00	0x04	0x08	0x0C
0x00	RSVD	RSVD	Force old OTA	RSVD
0x10	RSVD	RSVD	RSVD	RSVD
0x20	WORD1: RSVD WORD0: SPI Mode	WORD1: Flash Size WORD0: Flash ID	RSVD	RSVD
0x30	ULOG Baudrate	RSVD	RSVD	RSVD
0x40 ~ 0x70	RSVD (SPIC calibration setting)			
.....	RSVD			

Table 4-5 Layout of system data

4.2.2.1 OTA section

Offset	Bit	Function	Description
0x00	[31:0]	RSVD	RSVD
0x04	[31:0]	RSVD	RSVD
0x08	[31:8]	RSVD	RSVD
	[7:0]	Force old OTA	Select GPIO to force booting from old OTA image. Available GPIO pins may vary from different Chip part number. (GPIOA2~6, GPIOA13) BIT[7]: active_state, 0 or 1 BIT[6]: RSVD BIT[5]: port BIT[4:0]: pin number
0x0C	[31:0]	RSVD	RSVD

Table 4-6 Definition for OTA section in system data

4.2.2.1.1 Force old OTA

The platform provides a “Force old OTA” option to let user roll back to the previous OTA image by using a GPIO pin as trigger.

4.2.2.2 Flash section

Offset	Bit	Function	Description
0x20	[31:16]	RSVD	RSVD
	[15:0]	SPI IO Mode	0xFFFF: Quad IO mode 0x7FFF: Quad Output mode 0x3FFF: Dual IO mode 0x1FFF: Dual Output mode 0x0FFF: One IO mode
0x24	[31:16]	Flash Size	0xFFFF: 2MB 0x7FFF: 32MB 0x3FFF: 16MB 0x1FFF: 8MB 0x0FFF: 4MB 0x07FF: 2MB 0x03FF: 1MB
	[15:0]	Flash ID	Use it only if the flash ID cannot get by flash ID cmd
0x28	[31:0]	RSVD	RSVD
0x2C	[31:0]	RSVD	RSVD

Table 4-7 Definition for Flash section in system data

4.2.2.3 Log UART section

Offset	Bit	Function	Description
0x30	[31:0]	Baudrate	0xFFFFFFFF: 115200 110~40000000
0x34	[31:0]	RSVD	RSVD
0x38	[31:0]	RSVD	RSVD
0x3C	[31:0]	RSVD	RSVD

Table 4-8 Definition for Log UART section in system data

4.2.3 Calibration data

Reserved

4.2.4 Boot image

4.2.4.1 The layout of boot image

The format of the boot image is as below

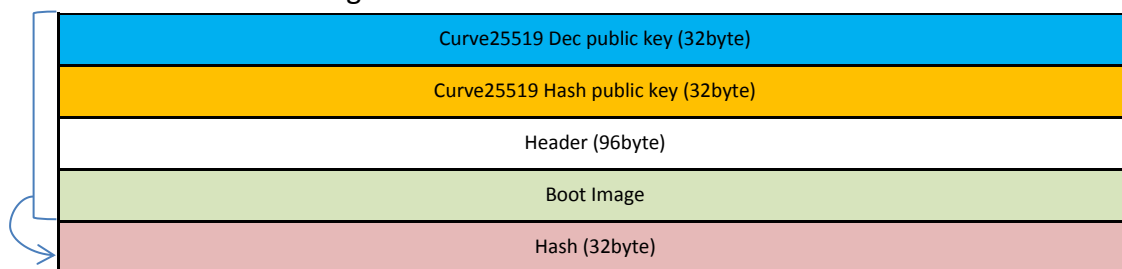


Table 4-9 The layout of boot image

- Curve25519 Dec public key: the public key used to generate AES key to decrypt image
- Curve25519 Hash public key: the public key used to generate Hash key to validate the hash value
- Header: boot image header
- Boot Image: boot image body (TEXT+DATA), will be padded with 0 to make its size is multiple of 32 bytes.

- Hash: two public keys + Header + Boot Image, calculated before encryption if image encryption is on

4.2.4.2 How to configure boot image (TBD)

4.2.5 Firmware 1/Firmware 2

4.2.5.1 The layout of firmware image

The format of the Firmware image is as below



Table 4-10 The layout of firmware image

- OTA signature: The hash result of the 1st Image header “Sub FW Image 0 Header”
- Public key 0 ~ 5: Encryption key
 - key 0 is dedicated to enc/dec all “OTA signature/Header/FST”
 - key 1~5 are reserved
- Sub-image x Header: image header of FW sub-image x
- Sub-image x FST: Firmware Security Table of FW sub-image x

- Each sub-image has image sections which have a section header and a section image body
- Hash: calculated with Encrypted FW image if image encryption is on
 - The 1st sub-image
 - From OTA Signature to the last image section, including all padding bytes
 - Other sub-image
 - From the sub image header to the last image section, including all padding bytes

4.2.5.2 How to configure firmware image (TBD)

4.2.5.3 How to generate flash image combines both firmware1 and firmware2

There may be requirements need to generate flash image combines both firmware1 and firmware2. The default set of AmebaZ2 flash image contains partition table, boot image, and firmware1 image. In order to add firmware2 image to flash image, extra configurations need to be done.

- Prepare firmware2 image.
Please note that there is serial number needs to be customized for firmware2 image. Refer to “7.4.2 Configuration for building OTA firmware” for details of the serial number setting.
Please note that default serial number is “100”, any number larger than “100” will set firmware2 image as default for flash image otherwise firmware1 image is default.
Add firmware2 image at “project\realtek_amebaz2_v0_example\EWARM-RELEASE\Debug\Exe”
- Update “postbuild_is.bat” for combines firmware1 and firmware2
Add “FW2=Debug\Exe\firmware2_is.bin” after “FW1=Debug\Exe\firmware_is.bin” at “component\soc\realtek\8710c\misc\iar_utility\postbuild_is.bat” refers the following table

```

::generate flash image, including partition + bootloader + firmware
%tooldir%\elf2bin.exe combine Debug/Exe/flash_is.bin
PTAB=Debug\Exe\partition.bin,BOOT=Debug\Exe\bootloader.bin,FW1=Debug\Exe\firmware_is.bin,FW2=Debug\Exe\firmware2_is.bin >> postbuild_is_log.txt
if not exist Debug\Exe\flash_is.bin (
    echo flash_is.bin isn't generated, check postbuild_is_log.txt
    echo flash_is.bin isn't generated > postbuild_is_error.txt
    goto error_exit
)

```

Please note that firmware2 image must have the same name as the code added in “postbuild_is.bat”.

- Generate and download flash image

Please refer to “2.4.2 IAR Project introduction” for generate flash image. The firmware1 image is auto generated when generating flash image. The generated image (“flash_is.bin”) is the flash image combines both fimware1 and firmware2.

Please note that the flash image is only downloadable by Image Tool. Refer to “3.3 Image download” for details of using Image Tool downloading.

- Switch between firmware1 image and firmware2 image

To switch firmware1 image and firmware2 image please refer to ATCMD “ATSC” and “ATSR” which is detailed in “AN0075 Realtek Ameba-all at command v2.0.docx”

4.3 SRAM layout

The range of DTCM is from 0x10000000 to 0x10040000. The layout of this memory region is illustrated below.

Note: the layout may be changed according to actual application, please refer to the linker file for exact layout details.

0x10000000	Vector Table
0x100000A0	Reserved for ROM
0x10000480	RAM Entry Table
0x100004F0	RAM Image Signature
0x10000500	Image2 RAM
0x10030000	RAM Bootloader
0x1003EA00	MSP
0x1003FA00	Reserved for ROM
0x1003FFFF	

Table 4-11 AmebaZII DTCM (256KB) memory layout

Items	Start Offset Address	Limit Offset Address	Address adjustable	Size	Description	Mandatory
Vector Table	0x10000000	0x100000A0-1	N	160B	The vector table	Y
Reserved for ROM	0x100000A0	0x10000480-1	N	992B	Reserved for ROM code	Y
RAM Entry Table	0x10000480	0x100004F0-1	N	112B	Entry function table of Image 2	Y
RAM Image Signature	0x100004F0	0x10000500-1	N	16B	RTK pattern for RAM Image	Y
Image2 RAM	0x10000500	0x10030000-1	Y	~190KB	User application image (TEXT+DATA+BSS+HEAP)	Y
RAM Bootloader	0x10030000	0x1003EA00-1	N	~58KB	RAM boot image, will be recycled by Image2 for BSS and HEAP	Y
MSP	0x1003EA00	0x1003FA00-1	Y	4KB	CPU main stack	Y

Reserved for ROM	0x1003FA00	0x1003FFFF	N	1535B	Reserved for ROM(NS) code	Y
------------------	------------	------------	---	-------	---------------------------	---

Table 4-12 Description of RAM layout

4.4 TrustZone Memory Layout

By default the memory is marked as secure, unless the address matches a region defined in SAU (Security Attribution Unit) which is used to define a number of memory regions as non-secure or NSC (Non-Secure Callable).

AmebaZII has 4 SAUs, but they are already used by the system and not open to users. Below is the default configuration of each SAU.

- SAU 0: 0x00019000 ~ 0x0005ffff as Non-Secure
- SAU 1: 0x10008000 ~ 0x4ffffff as Non-Secure
- SAU 2: 0x60100000 ~ 0x9bfeffff as Non-Secure
- SAU 3: 0x9bfc0000 ~ 0x9bffffff as Secure(NSC)

Thus, if the TrustZone is enabled, the system memory layout will become as below.

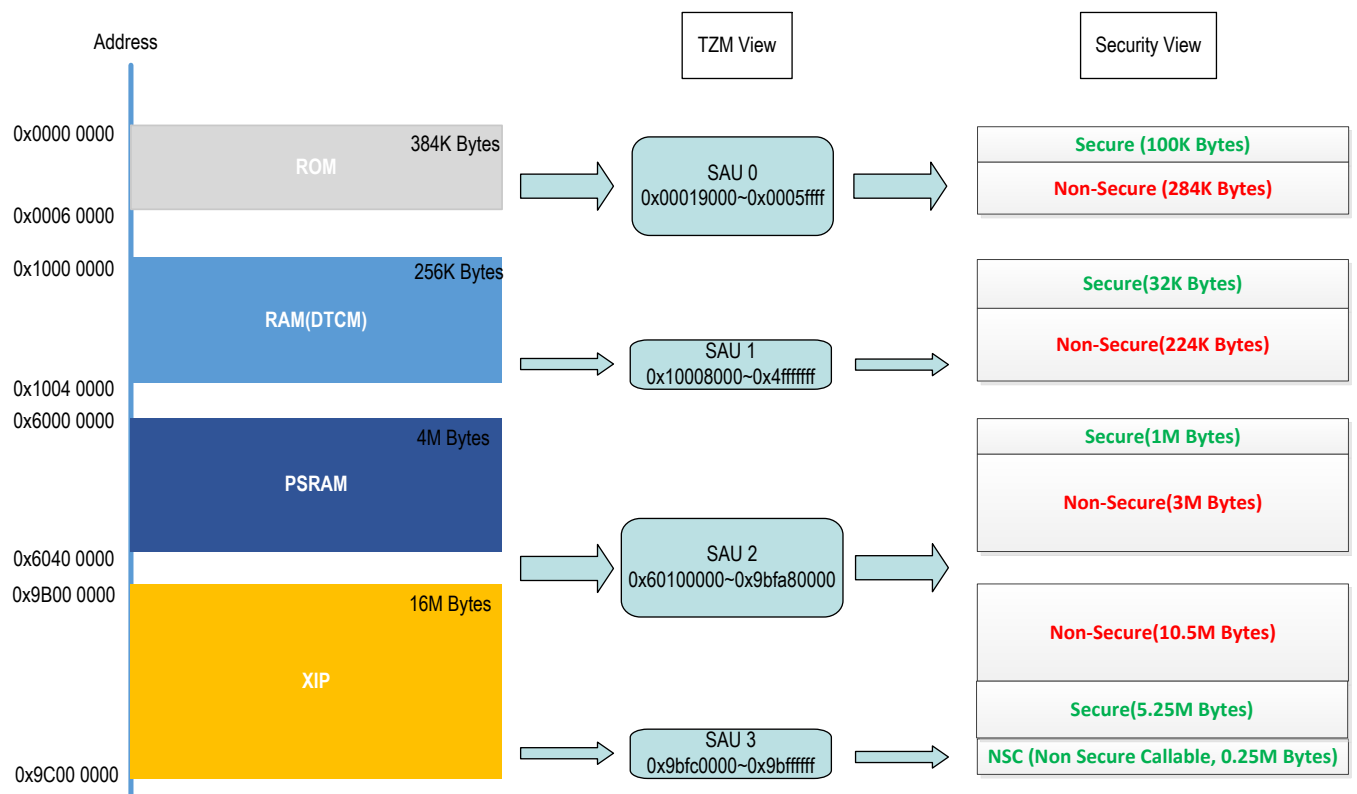


Figure 4-3 TrustZone memory layout

5 Boot process

This chapter describes the boot process of AmebaZII platform.

5.1 Boot flow

While booting, the system will firstly load the partition table which has all image information, such as the image address, keys, user data etc... Then from the partition table, boot image will be loaded, and firmware image will be loaded at the end of the boot process.

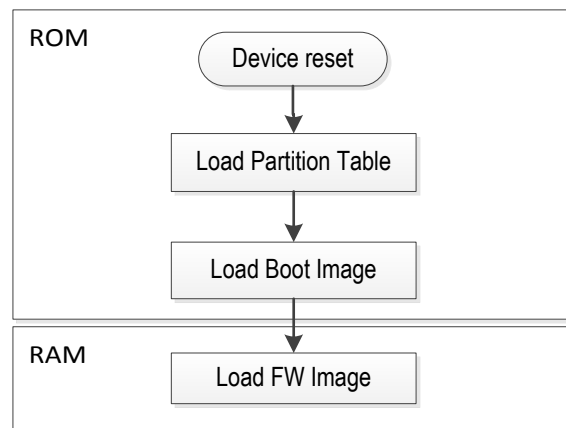


Figure 5-1 Overview of boot flow

5.2 Secure boot

Secure boot aims at firmware protection, which prevents attackers from modifying or replacing firmware maliciously. When the chip is power on, the ROM security boot executes to check the validation of each image.

If the image is valid, then the authentication will be successful, which means the firmware is safe. And the subsequent process can be continued. Otherwise, the SoC will go into infinite loop.

5.2.1 Secure boot flow

While booting, the system will use the encryption private key which locates in super secure efuse, and the public key which locates in flash, to generate AES keys and Hash keys, then use them to decrypt and verify each image.

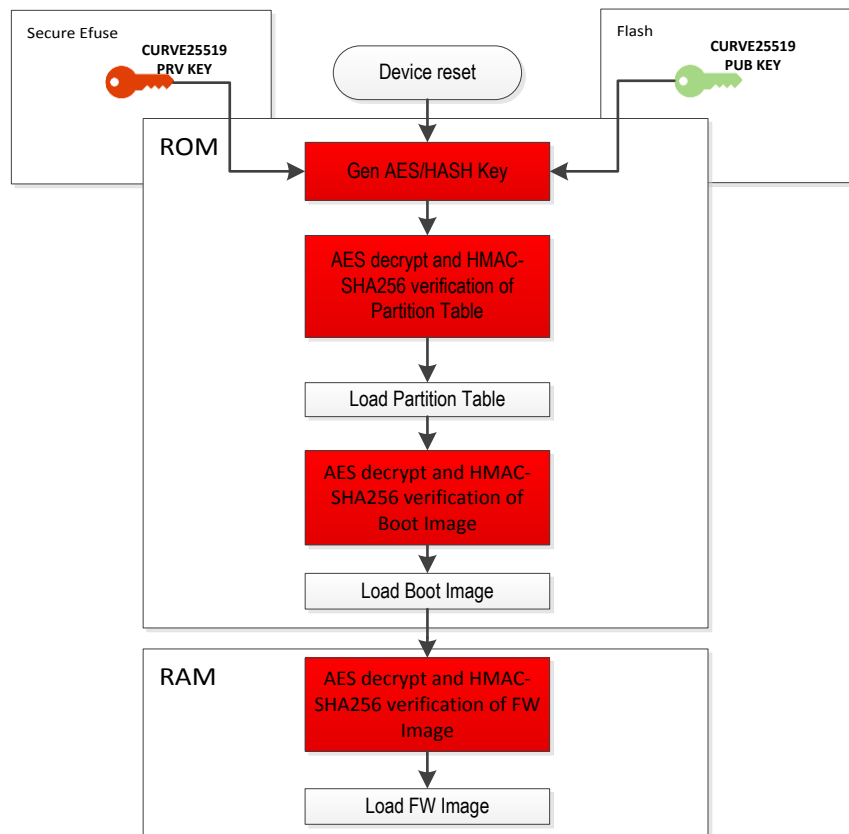


Figure 5-2 Secure boot flow

5.2.2 Partition table and Boot image decryption flow

Figure 5-3 illustrates the decryption flow of partition table and boot image in secure boot process.

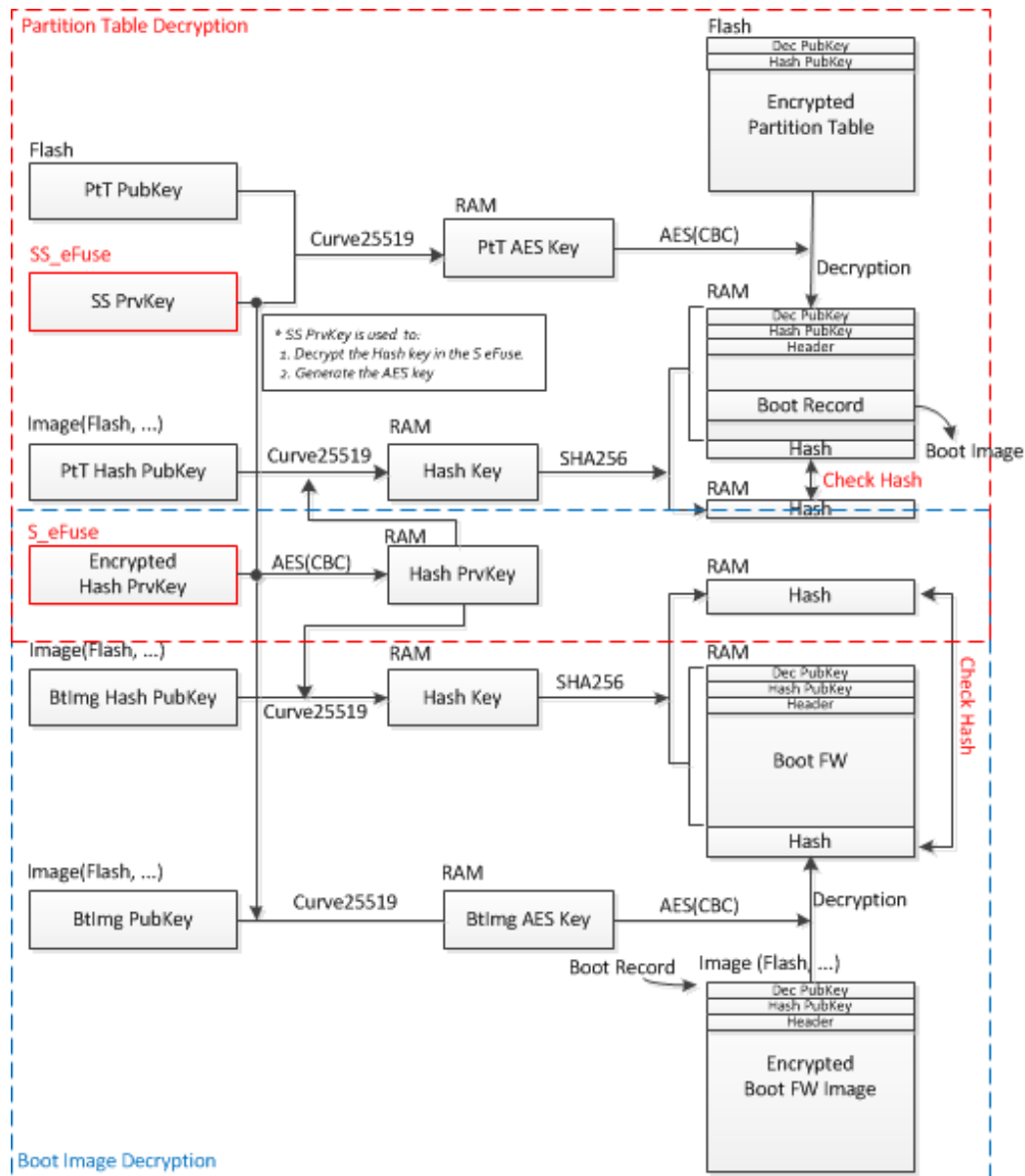


Figure 5-3 Partition table and boot image decryption flow

5.2.3 Secure boot use scenario

Figure 5-4 illustrates the use scenario of secure boot in a real product development. Firstly software developer needs to generate key pairs, and encrypt and hash the firmware properly. And when it comes to mass production, manufacturing facility will program the encrypted firmware along with a reference hash value on device, and program the private key in super secure efuse zone. While device boots up, it will use the super secure private key and public key to verify the hash value and decrypt each image. If case of any error or failure in the image validation process, the device won't boot.

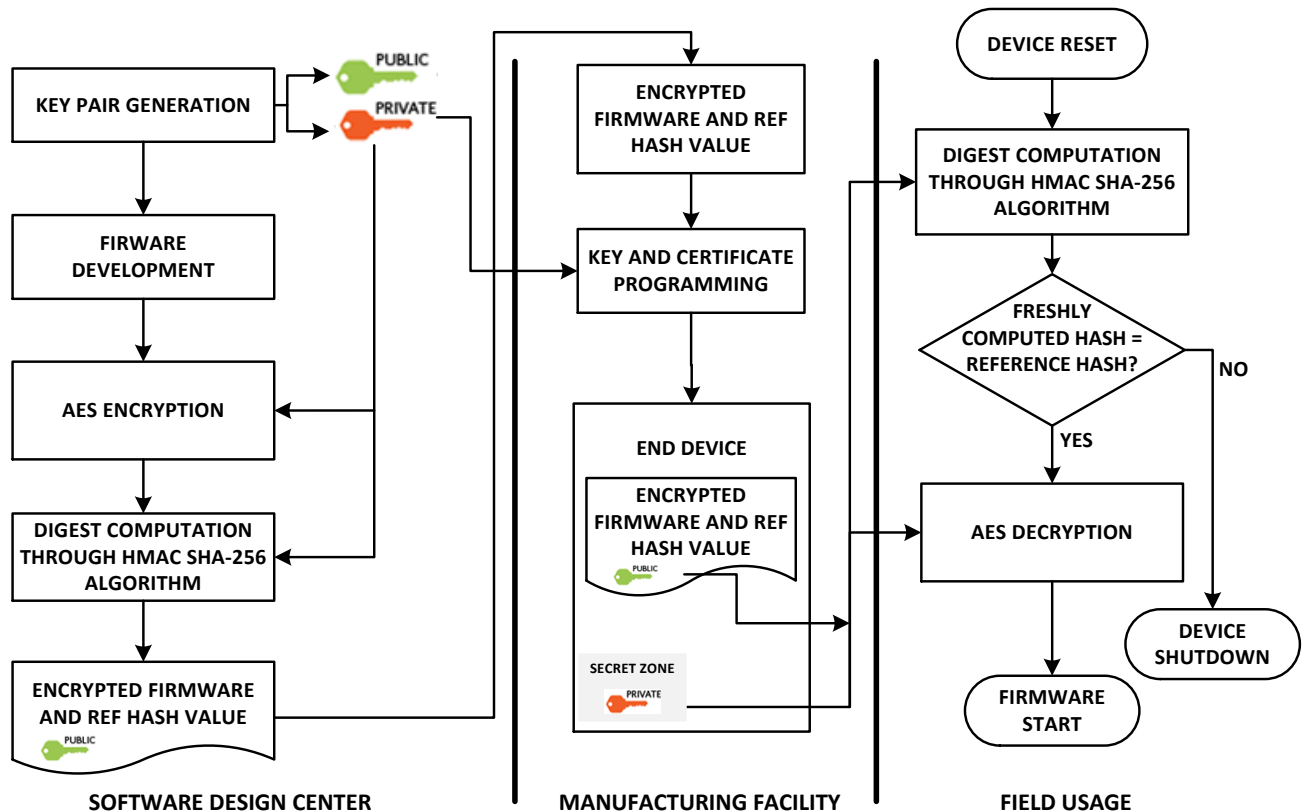


Figure 5-4 secure boot use scenario

5.2.4 How to enable secure boot

This chapter shows how to enable the secure boot.

5.2.4.1 Keys configuration

As mentioned above, while booting, the system will use the encryption private key which locates in super secure efuse, and the public key which locates in flash, to generate AES keys and Hash keys, then use them to decrypt and verify each image.

The super secure private key (named as “privkey_enc”) and the encrypted hash private key (named as “privkey_hash”) are configured in *keycfg.json* under *project\realtek_amebaz2_v0_example\EWARM-RELEASE*.

What shows following is the default value. You can configure the keys by yourself in *keycfg.json*.

```
{
  "__comment_0": "configuration for private key, use auto to generate random key",
  "__comment_1": "private key maybe different from your desired input, program will
modify first byte and last byte of key",
  "__comment_2": "to get actual private key, please open key.json after key generated.",
  "__comment_3": "hash private key in key.json will be encrypted",

  "privkey_enc": "000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1
D1E5F",
  "privkey_enc1": "auto",

  "privkey_hash": "000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C
1D1E5F",
  "privkey_hash1": "auto"
}
```

5.2.4.2 Keys programming in Efuse

After configuring the keys in *keycfg.json*, rebuild the whole project in IAR. The *key.json* file under *project\realtek_amebaz2_v0_example\EWARM-RELEASE* will be updated after compiling. The “privkey_enc” is the same with the “privkey_enc” in *keycfg.json*, but the “privkey_hash” has been changed after *Hash* algorithm.

```
{
  "__comment_EFUSE": "should keep these two key in safe place, or secure firmware
protection is useless",
  "EFUSE": {
    "__comment_privkey_enc": "this key in EFUSE SUPER SECURE ZONE",

    "privkey_enc": "000102030405060708090A0B0C0D0E0F101112131415161718191A
1B1C1D1E5F",
    "__comment_privkey_hash": "this key in EFUSE SECURE ZONE, encrypted by
upper key",
  }
}
```

```

    "privkey_hash":"64A7433FCF027D19DDA4D446EEF8E78A22A8C33CB2C337C07366
C040612EE0F2"
  },
  "TOOL":{
    "__comment_pubkey_enc":"public key for encryption",

    "pubkey_enc":"8F40C5ADB68F25624AE5B214EA767A6EC94D829D3D7B5E1AD1BA
6F3E2138285F",
    "__comment_pubkey_hash":"public key for hash, only for partition table
and bootloader",

    "pubkey_hash":"8F40C5ADB68F25624AE5B214EA767A6EC94D829D3D7B5E1AD1B
A6F3E2138285F"
  }
}

```

“example_secure_boot.c” is an example provided for enabling secure boot. To write the keys to efuse, firstly, change CONFIG_EXAMPLE_SECURE_BOOT to 1 in platform_opts.h.

```

/*For secure boot example */
#define CONFIG_EXAMPLE_SECURE_BOOT 1

```

Secondly, modify the super secure key “susec_key[]” and the secure key “sec_key[]” to correspond with “privkey_enc” and “privkey_hash” in *key.json* file under *project\realtek_amebaz2_v0_example\EWARM-RELEASE* in “example_secure_boot.c”. In another word, “susec_key[]” should be the same with “privkey_enc” in *key.json*, and “sec_key[]” should be the same with “privkey_hash” in *key.json*.

```

//these two keys are the same default keys used in SDK
const uint8_t susec_key[PRIV_KEY_LEN] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x5F
};
const uint8_t sec_key[PRIV_KEY_LEN] = {
    0x64, 0xA7, 0x43, 0x3F, 0xCF, 0x02, 0x7D, 0x19,
    0xDD, 0xA4, 0xD4, 0x46, 0xEE, 0xF8, 0xE7, 0x8A,
    0x22, 0xA8, 0xC3, 0x3C, 0xB2, 0xC3, 0x37, 0xC0,
    0x73, 0x66, 0xC0, 0x40, 0x61, 0x2E, 0xE0, 0xF2
};

```

Thirdly, modify 0 to 1 to enable write super secure key function and write secure key function to write keys to efuse. What needs to be noted is, the efuse is one-time writable, please make sure the key is correct before programming in efuse.

```

// write SS key
memset(write_buf, 0xFF, PRIV_KEY_LEN);
if(1){ // fill your data
    for(i=0; i<PRIV_KEY_LEN; i++)
        write_buf[i] = susec_key[i];
}
if(1){ // write
    device_mutex_lock(RT_DEV_LOCK_EFUSE);
    ret = efuse_susec_key_write(write_buf);
    device_mutex_unlock(RT_DEV_LOCK_EFUSE);
    if(ret < 0){
        dbg_printf("efuse SS key: write address and length error\r\n");
        goto exit;
    }
    dbg_printf("\r\nWrite Done.\r\n");
}else{
    dbg_printf("\r\nPlease make sure the key is correct before programming in
efuse.\r\n");
}
    dbg_printf("\r\n");
// write S key
memset(write_buf, 0xFF, PRIV_KEY_LEN);
if(1){ // fill your data
    for(i=0; i<PRIV_KEY_LEN; i++)
        write_buf[i] = sec_key[i];
}
if(1){ // write
    device_mutex_lock(RT_DEV_LOCK_EFUSE);
    ret = efuse_sec_key_write(write_buf, 0);
    device_mutex_unlock(RT_DEV_LOCK_EFUSE);
    if(ret < 0){
        dbg_printf("efuse S key: write address and length error\r\n");
        goto exit;
    }
    dbg_printf("\r\nWrite Done.\r\n");
}else{
    dbg_printf("\r\nPlease make sure the key is correct before programming in
efuse.\r\n");
}
    dbg_printf("\r\n");

```

The SS key locker can be enabled by changing 0 to 1 marked as yellow here. If the locker is enabled, the SS key turns to be unreadable forever. So this configuration is irreversible, please do it only if you are certain about SS key.


```

/*
Step 3: lock and protect the SS key from being read by CPU
*/
// lock SS key, make SS key unreadable forever.
// this configure is irreversible, so please do this only if you are certain about SS key
if(1){
    device_mutex_lock(RT_DEV_LOCK_EFUSE);
    ret = efuse_lock_susec_key();
    device_mutex_unlock(RT_DEV_LOCK_EFUSE);
    if(ret < 0){
        dbg_printf("efuse SS key lock error\r\n");
        goto exit;
    }
}
}

```

Enable secure boot by setting the flag to 1 in step 4. After enabling the secure boot, the device will only boot with encrypted image. The configure is also irreversible, so please do this if you are certain that the firmware image is encrypted and hashed with the correct SS key and S key.

```

/*
Step 4: enable the secure boot so that device will only boot with encrypted image
*/
// enable secure boot, make device boot only with correctly encrypted image
// this configure is irreversible, so please do this only if you are certain that the fw image
is encrypted and hashed with the correct SS key and S key
if(1){
    device_mutex_lock(RT_DEV_LOCK_EFUSE);
    ret = efuse_fw_verify_enable();
    device_mutex_unlock(RT_DEV_LOCK_EFUSE);
    if(ret < 0){
        dbg_printf("efuse secure boot enable error\r\n");
        goto exit;
    }
}
device_mutex_lock(RT_DEV_LOCK_EFUSE);
ret = efuse_fw_verify_check();
device_mutex_unlock(RT_DEV_LOCK_EFUSE);
if(ret)
    dbg_printf("secure boot is enabled!");
}

```

What needs to highlight is, chapter 5.2.4.1 and chapter 5.2.4.2 aim to write keys to efuse and enable secure boot. Before enabling secure boot, the Ameba-ZII is in non-secure boot mode, that means encrypted image cannot boot. So till now, the application is built to enable secure boot. After enabling, encrypt the image and then the Ameba-ZII can boot with encrypted image.

5.2.4.3 Encrypt the image

The boot/firmware 1/firmware 2 addresses are stored in partition records, defined in *partition.json* under *project\realtek_amebaz2_v0_example\EWARM-RELEASE*. The addresses can be modified if needed.

```
"boot":{
  "start_addr": "0x4000",
  "length": "0x8000",
  "type": "BOOT",
  "dbg_skip": false,

  "hash_key": "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF",
  },
  "fw1":{
    "start_addr": "0x10000",
    "length": "0x80000",
    "type": "FW1",
    "dbg_skip": false,

    "hash_key": "000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E5F",
    },
    "fw2":{
      "start_addr": "0x90000",
      "length": "0x80000",
      "type": "FW2",
      "dbg_skip": false,

      "hash_key": "000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E5F"
    }
  }
```

Ameba-ZII is defaulted as non-secure boot mode, secure boot asks encrypted firmware. The keys to enable firmware encryption are defined in *amebaz2_bootloader.json/amebaz2_firmware_is.json* under *project\realtek_amebaz2_v0_example\EWARM-RELEASE*. “enc” can be changed from *false* to *true* in following code to enable corresponding module encryption. In *amebaz2_bootloader.json*,

```
"PARTAB": {
  "source": null,
  "header": {
    "next": null,
    "__comment_type": "Support
Type: PARTAB, BOOT, FWHS, FWLS, ISP, VOE, WLN, DTCM, ITCM, SRAM, ERAM, XIP, MO, CPFW",
    "type": "PARTAB",
    "enc": true,
```

```

        "serial": 0
    },
    "list" : ["partab"],
    "partab": {
        "__comment_ptable": "move to partition.json",

        "__comment_file": "TODO: use binary file directly",
        "file": null
    }
},
"BOOT": {
    "source": "Debug/Exe/bootloader.out",
    "header": {
        "next": null,
        "__comment_type": "Support
Type: PARTAB, BOOT, FWHS, FWLS, ISP, VOE, WLN, DTCM, ITCM, SRAM, ERAM, XIP, MO, CPFW",
        "type": "BOOT",
        "enc": true,

"user_key1": "AA0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1
E1F",
        "serial": 0
    },
    "list" : ["sram"],
    "sram": {
        "__comment_option": "TODO: not ready",
        "option": null,

        "__comment_entry": "startup function table symbol",
        "entry": "gRamStartFun",

        "start": "RAM_FUNTAB$$Base",
        "end": "RAM_RODATA$$Limit",

        "__comment_file": "TODO: use binary file directly",
        "file": null
    }
}
}

```

In *amebaz2_firmware_is.json*, “*enc*” also is set to *true* to encrypt the image. But what needs to be noted is, the “*enc*” in “*XIP_FLASH_P*” could not be set to *true* because this section is reserved for plain data.

```

"FWHS": {
    "source": "Debug/Exe/application_is.out",
    "header": {
        "next": null,
        "__comment_type": "Support
Type:PARTAB,BOOT,FWHS_S,FWHS_NS,FWLS,ISP,VOE,WLN,DTCM,ITCM,SRAM,ERAM,XIP,
M0,CPFW",
        "type": "FWHS_S",
        "enc": true,

        "user_key2": "BB0102030405060708090A0B0C0D0E0F101112131415161718191A1
B1C1D1E1F",
        "__comment_pkey_idx": "assign by program, no need to
configure",
        "serial": 107
    },
    "FST": {
        "__comment_FST0": "enc_algorithm: cbc/ecb with cipher key",
        "__comment_FST1": "validpat is used for section header validation",
        "__comment_FST2": "hash_en/enc_en?",
        "enc_algorithm": "cbc",
        "hash_algorithm": "sha256",
        "part_size": "4096",

        "__comment_validpat": "use auto or dedicated value",
        "validpat": "0001020304050607",
        "hash_en": true,
        "enc_en": true,
        "cipherkey": null,
        "cipheriv": null
    },
}

"XIP_FLASH_C": {
    "source": "Debug/Exe/application_is.out",
    "header": {
        "next": null,
        "__comment_type": "Support
Type:PARTAB,BOOT,FWHS_S,FWHS_NS,FWLS,ISP,VOE,WLN,DTCM,ITCM,SRAM,ERAM,XIP,
M0,CPFW",
        "type": "XIP",
        "enc": true,
        "__comment_pkey_idx": "assign by program, no need to
configure",
        "serial": 0
    },
    "FST": {

```

```

        "__comment_FST0": "enc_algorithm: cbc/ecb with cipher key",
        "__comment_FST1": "validpat is used for section header validation",
        "__comment_FST2": "hash_en/enc_en?",
        "enc_algorithm": "cbc",
        "hash_algorithm": "sha256",
        "part_size": "4096",

        "__comment_validpat": "use auto or dedicated value",
        "validpat": "0001020304050607",
        "hash_en": true,
        "enc_en": true,
        "cipherkey": null,
        "cipheriv": null
    },
}

"XIP_FLASH_P": {
    "source": "Debug/Exe/application_is.out",
    "header": {
        "next": null,
        "__comment_type": "Support
Type:PARTAB,BOOT,FWHS_S,FWHS_NS,FWLS,ISP,VOE,WLN,DTCM,ITCM,SRAM,ERAM,XIP,
MO,CPFW",
        "type": "XIP",
        "enc": false,
        "__comment_pkey_idx": "assign by program, no need to
configure",
        "serial": 0
    },
    "FST": {
        "__comment_FST0": "enc_algorithm: cbc/ecb with cipher key",
        "__comment_FST1": "validpat is used for section header validation",
        "__comment_FST2": "hash_en/enc_en?",
        "enc_algorithm": "cbc",
        "hash_algorithm": "sha256",
        "part_size": "4096",

        "__comment_validpat": "use auto or dedicated value",
        "validpat": "0001020304050607",
        "hash_en": true,
        "enc_en": false,
        "cipherkey": null,
        "cipheriv": null
    },
}

```

5.2.4.4 Open “secure_bit”

“secure_bit” in *postbuild_is.bat* under *component\soc\realtek\8710c\misc\iar_utility* needs to be set to 1.

```
%tooldir%\elf2bin.exe convert amebaz2_bootloader.json BOOTLOADER secure_bit=1 >>
postbuild_is_log.txt
if not exist Debug\Exe\bootloader.bin (
    echo bootloader.bin isn't generated, check postbuild_is_log.txt
    echo bootloader.bin isn't generated > postbuild_is_error.txt
    pause
    exit 2 /b
)

::generate partition table
%tooldir%\elf2bin.exe convert amebaz2_bootloader.json PARTITIONTABLE secure_bit=1 >>
postbuild_is_log.txt
if not exist Debug\Exe\partition.bin (
    echo partition.bin isn't generated, check postbuild_is_log.txt
    echo partition.bin isn't generated > postbuild_is_error.txt
    pause
    exit 2 /b
)

::generate firmware image
if not exist amebaz2_firmware_is.json (
    echo amebaz2_firmware_is.json is missing
    echo amebaz2_firmware_is.json is missing > postbuild_is_error.txt
    pause
    exit 2 /b
)
%tooldir%\elf2bin.exe convert amebaz2_firmware_is.json FIRMWARE secure_bit=1 >>
postbuild_is_log.txt
if not exist Debug\Exe\firmware_is.bin (
    echo firmware_is.bin isn't generated, check postbuild_is_log.txt
    echo firmware_is.bin isn't generated > postbuild_is_error.txt
    pause
    exit 2 /b
)
```

5.2.4.5 Secure boot execution

Set the “privkey_enc” and “privkey_hash” in *keycfg.json* as shown below, in this example, just change the last character of default value from *F* to *0*.

```
{
  "__comment_0": "configuration for private key, use auto to generate random key",
  "__comment_1": "private key maybe different from your desired input, program will
modify first byte and last byte of key",
  "__comment_2": "to get actual private key, please open key.json after key generated.",
  "__comment_3": "hash private key in key.json will be encrypted",

  "privkey_enc": "000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1
D1E50",
  "privkey_enc1": "auto",

  "privkey_hash": "000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C
1D1E50",
  "privkey_hash1": "auto"
}
```

After rebuilding the project, the keys are updated in *key.json*.

```
{
  "__comment_EFUSE": "should keep these two key in safe place, or secure firmware
protection is useless",
  "EFUSE": {
    "__comment_privkey_enc": "this key in EFUSE SUPER SECURE ZONE",

    "privkey_enc": "000102030405060708090A0B0C0D0E0F101112131415161718191A
1B1C1D1E50",
    "__comment_privkey_hash": "this key in EFUSE SECURE ZONE, encrypted by
upper key",

    "privkey_hash": "1C219D029C32C0744FBDAAE9BC306D0CC57F02049217C3A94D8E
105D5AA264A0"
  },
  "TOOL": {
    "__comment_pubkey_enc": "public key for encryption",

    "pubkey_enc": "B496A6CF209834D1F22C7FEA41172F5888F9540B069874F4700B41
1E77576E03",
    "__comment_pubkey_hash": "public key for hash, only for partition table
and bootloader",

    "pubkey_hash": "B496A6CF209834D1F22C7FEA41172F5888F9540B069874F4700B4
11E77576E03"
  }
}
```

Set the flag CONFIG_EXAMPLE_SECURE_BOOT to 1 in platform_opts.h.

```
/*For secure boot example */  
#define CONFIG_EXAMPLE_SECURE_BOOT 1
```

Change the keys in example_secure_boot.c, make susec_key[] equal to “privkey_enc” and sec_key[] equal to “privkey_hash” in key.json.

```
//these two keys are the same default keys used in SDK  
const uint8_t susec_key[PRIV_KEY_LEN] = {  
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,  
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,  
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,  
    0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x50  
};  
const uint8_t sec_key[PRIV_KEY_LEN] = {  
    0x1C, 0x21, 0x9D, 0x02, 0x9C, 0x32, 0xC0, 0x74,  
    0x4F, 0xBD, 0xAA, 0xE9, 0xBC, 0x30, 0x6D, 0x0C,  
    0xC5, 0x7F, 0x02, 0x04, 0x92, 0x17, 0xC3, 0xA9,  
    0x4D, 0x8E, 0x10, 0x5D, 0x5A, 0xA2, 0x64, 0xA0  
};
```

Enable write SS key function, write S key function, lock SS key function and secure boot function by setting if condition as 1 (details are shown in chapter 5.2.4.2). Build the application and download it to Ameba-ZII to enable secure boot.

If secure boot is enabled successfully, the message will be

```
#  
efuse secure boot: Test Start  
[0]      FF FF FF FF  FF FF FF FF  
[8]      FF FF FF FF  FF FF FF FF  
[16]     FF FF FF FF  FF FF FF FF  
[24]     FF FF FF FF  FF FF FF FF  
  
Write Done.  
  
[0]      00 01 02 03  04 05 06 07  
[8]      08 09 0A 0B  0C 0D 0E 0F  
[16]     10 11 12 13  14 15 16 17  
[24]     18 19 1A 1B  1C 1D 1E 50  
[0]      FF FF FF FF  FF FF FF FF  
[8]      FF FF FF FF  FF FF FF FF  
[16]     FF FF FF FF  FF FF FF FF  
[24]     FF FF FF FF  FF FF FF FF  
  
Write Done.  
  
[0]      1C 21 9D 02  9C 32 C0 74  
[8]      4F BD AA E9  BC 30 6D 0C  
[16]     C5 7F 02 04  92 17 C3 A9  
[24]     4D 8E 10 5D  5A A2 64 A0  
efuse secure boot keys: Test Done  
eFuse Key Locked!!, Super-Secure Key Reading is Inhibited!!  
secure boot is enabled!
```

Enabling secure boot successfully means only encrypted image can boot on this Ameba-ZII board.

To run your own application, you need to encrypt image by setting the “enc” as true in *amebaz2_bootloader.json/amebaz2_firmware_is.json* under *project\realtek_amebaz2_v0_example\EWARM-RELEASE*(details are shown in chapter 5.2.4.3). Also, set “secure_bit=1” in *postbuild_is.bat* under *component\soc\realtek\8710c\misc\iar_utility* (details are shown in chapter 5.2.4.4). Finally, the Ameba-ZII board can run with encrypted image.

5.3 Boot time (TBD)

There is a table to show the time cost of the boot process.

Image Size:374KB	SECURE BOOT			NON-SECURE BOOT		
	ROM	BOOT LOADER	WIFI CONNECT	ROM	BOOT LOADER	WIFI CONNECT
1	96ms	47ms	3507ms	0.0ms	49.4ms	3508.9ms
2	77ms	51ms	3508ms	29.8ms	16.0ms	6046.0ms
3	96ms	46ms	3508ms	0.0ms	47.9ms	3503.1ms
4	96ms	49ms	3502ms	0.0ms	1.1ms	3700.1ms
5	75ms	46ms	3506ms	0.0ms	50.8ms	6509.0ms
6	94ms	44ms	5517ms	0.0ms	49.5ms	3502.2ms
7	80ms	44ms	3504ms	0.0ms	42.5ms	3500.5ms
8	76ms	45ms	3509ms	0.0ms	41.4ms	2998.3ms
9	77ms	44ms	4012ms	0.0ms	48.9ms	3503.5ms
10	95ms	50ms	3506ms	0.0ms	16.3ms	3002.7ms
Average	86.2ms	46.4ms	3757.9ms	2.98ms	36.38ms	3977.43ms

6 Secure JTAG/SWD

For security purpose, many products will disable JTAG/SWD function while mass production. But if one needs to debug the product after mass production, it will need to enable JTAG/SWD function again. AmebaZII supports “Password JTAG/SWD” for this use scenario.

6.1 Functional description (TBD)

6.2 How to enable and use Secure JTAG/SWD (TBD)

7 Over-the-Air (OTA) Firmware Update

Over-the-air programming (OTA) provides a methodology to update device firmware remotely via TCP/IP network connection.

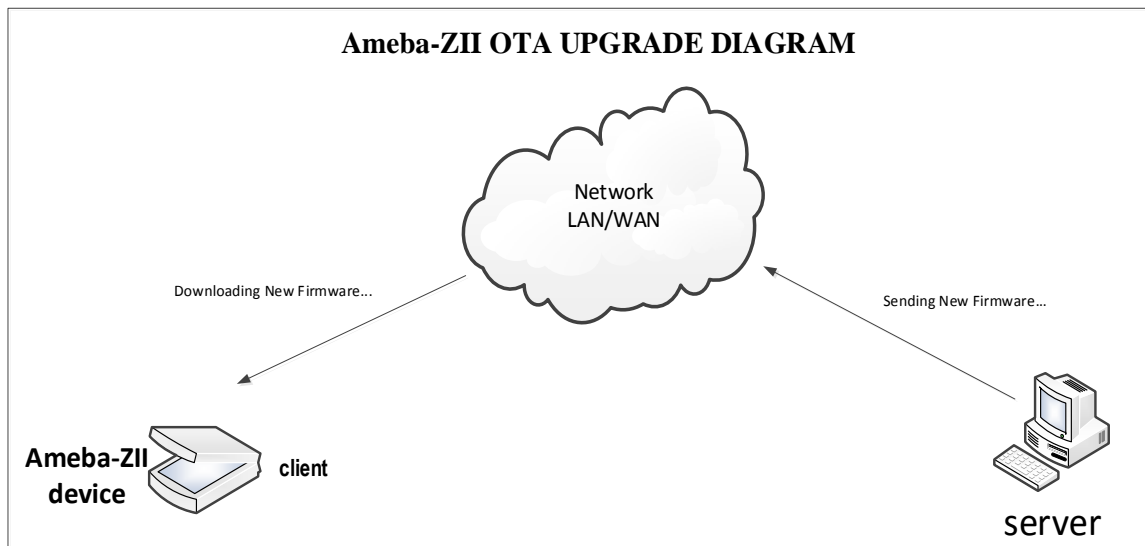


Figure 7-1 Methodology to Update Firmware via OTA

7.1 OTA operation flow

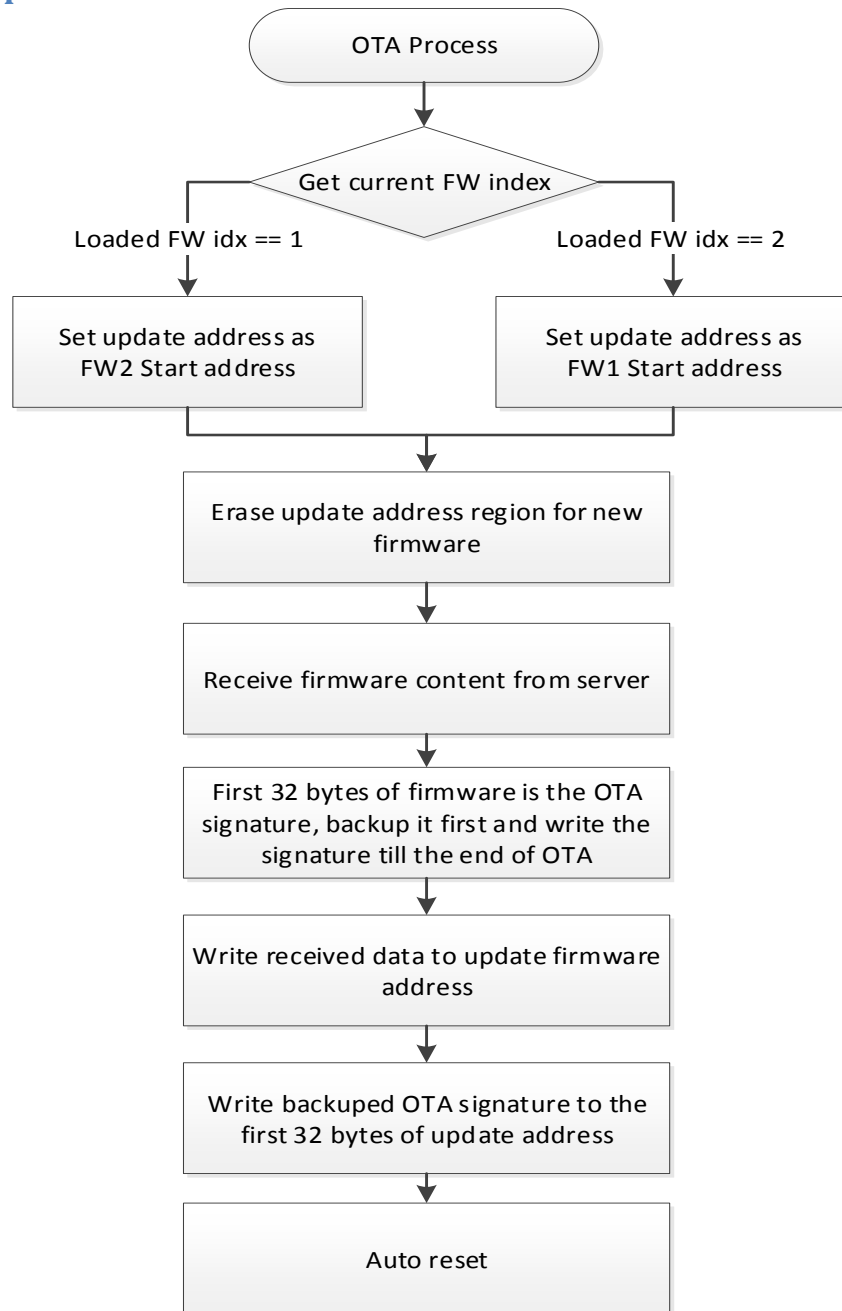



Figure 7-2 OTA Process Flow

 During the step of "Write received data to update firmware address", the 32 bytes OTA signature need set to 0xff, which is invalid signature. The correct OTA signature needs to be appended at the end of OTA process to prevent device booting from incomplete firmware.

7.2 Boot process flow

Boot loader will select latest (based on serial number) updated firmware and load it.

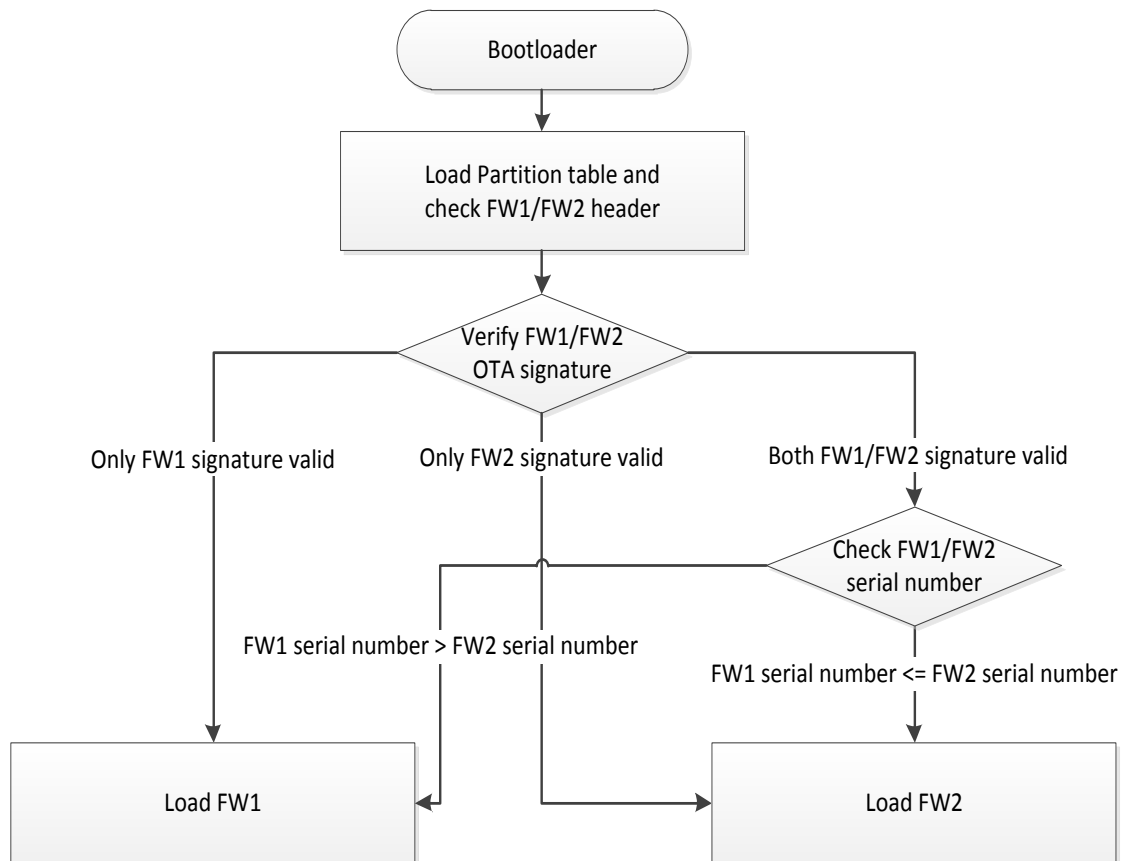


Figure 7-3 Boot Process Flow

7.3 Upgraded partition

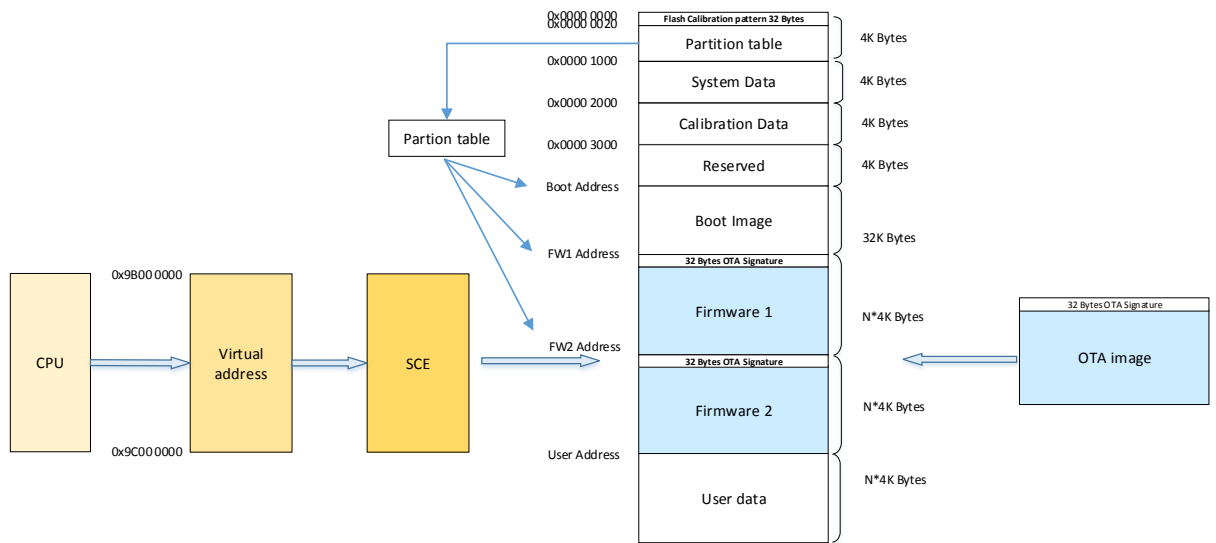


Figure 7-4 OTA update procedure

In Ameba-ZII OTA update procedure, Firmware 1 and Firmware 2 are swapped each other.

The Firmware 1/Firmware 2 addresses are stored in partition records, defined in *partition.json* under *project\realtek_amebaz2_v0_example\EWARM-RELEASE*. Please adjust it according to your firmware size.

```
"fw1":{
  "start_addr": "0x10000",
  "length": "0x80000",
  "type": "FW1",
  "dbg_skip": false,

  "hash_key": "000102030405060708090A0B0C0D0E0F101112131415161718191A1B
1C1D1E5F"
},
"fw2":{
  "start_addr": "0x90000",
  "length": "0x80000",
  "type": "FW2",
  "dbg_skip": false,

  "hash_key": "000102030405060708090A0B0C0D0E0F101112131415161718191A1B
1C1D1E5F"
}
```

7.4 Firmware image output

After building project source files in SDK, it would generate firmware as `firmware_is.bin`, which is the OTA Firmware as mentioned earlier.

7.4.1 OTA firmware swap behavior

When device executes OTA procedure, it would update the other OTA block, rather than the current running OTA block. The OTA firmware swap behavior should be looked like as below figure if the updated firmware keeps using newer serial number value.

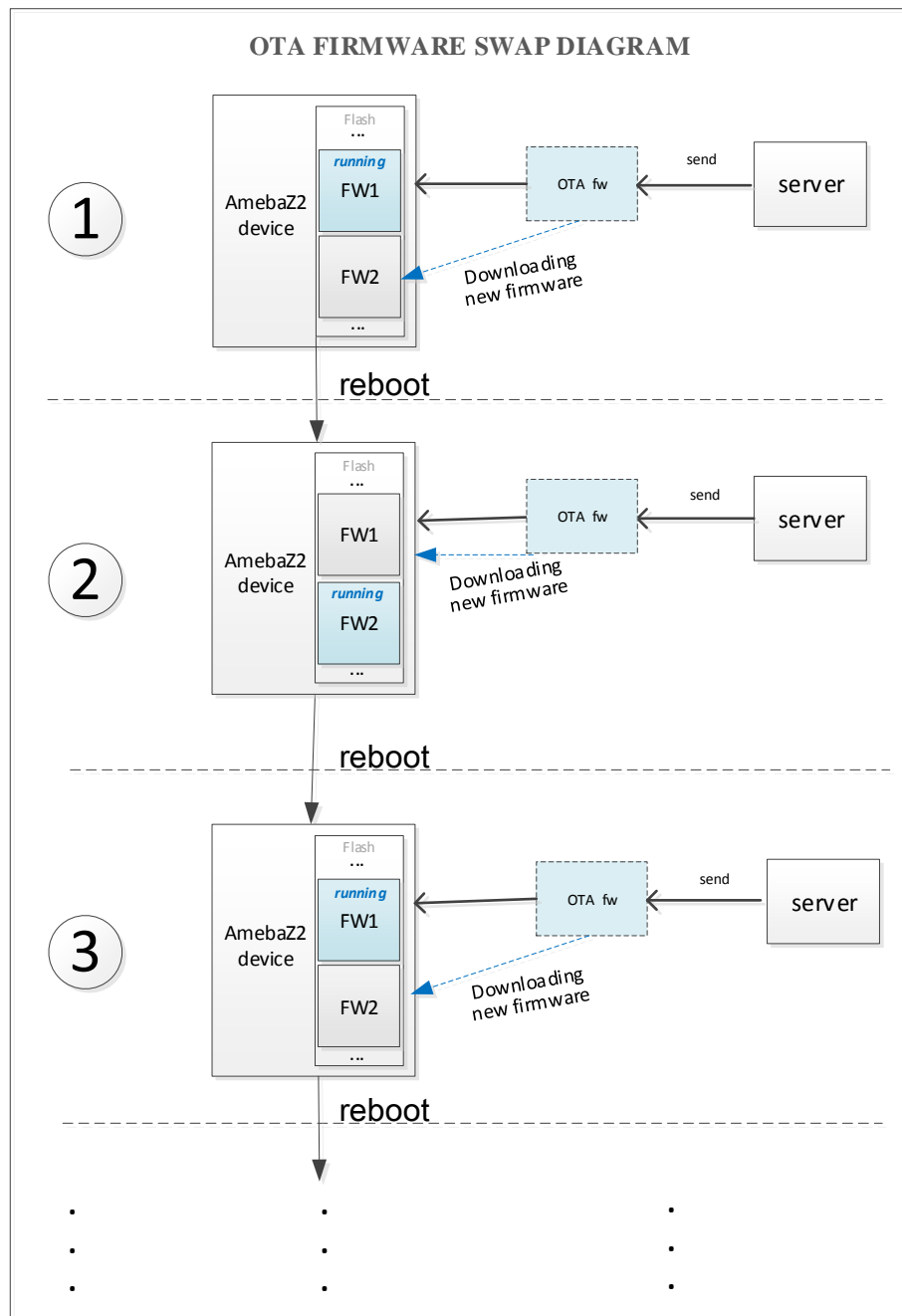


Figure 7-5 OTA Firmware SWAP Procedure

7.4.2 Configuration for building OTA firmware

Before building the project, the bootloader would check the serial number of OTA firmware to determine the boot sequence, the serial number of the OTA firmware need to be configured correctly before project build.

7.4.2.1 Serial number

Ameba-ZII OTA use serial number to decide the boot sequence if the signature of firmware is valid. Hence before building the project, please make sure the serial number is correctly configured. To set the serial number of a firmware, please follow below steps:

Step 1: The serial number setting of a firmware is as same as the serial number of its first image. You can check the images sequence in `project\realtek_amebaz2_v0_example\EWARM-RELEASE\amebaz2_firmware_is.json`.

```
"FIRMWARE":{
  "images":[
    {"img": "FWHS", "offset":"0x00"},
    {"img": "XIP_FLASH_C", "offset":"0x00"},
    {"img": "XIP_FLASH_P", "offset":"0x00"}
  ]
},
```

For this example, the FWHS is located at the top sequence. Hence it is the first image of this firmware.

Step 2: Modify the serial number setting of the first image. Take above figure for example, we need to modify the serial number of “FWHS”:

```
"FWHS": {
  "source":"Debug/Exe/application_is.out",
  "header":{
    "next":null,
    "__comment_type":"Support
Type:PARTAB,BOOT,FWHS_S,FWHS_NS,FWLS,ISP,VOE,WLN,DTCM,ITCM,SRAM,ERA
M,XIP,M0,CPFW",
    "type":"FWHS_S",
    "enc":false,

    "user_key2":"BB0102030405060708090A0B0C0D0E0F10111213141516171
8191A1B1C1D1E1F",
    "__comment_pkey_idx":"assign by program, no need to
configure",
    "serial": 100
  },
},
```


The Serial number is stored as 4 byte digital number and is valid from 1. Please modify it according to your firmware version. Please note that the default number 0 means maximum version number.

Step 3: After building project source files in SDK, it should automatically generate *SDK_folder/project/project_name/EWARM-RELEASE/Debug/Exe/firmware_is.bin*, which is the application of OTA Firmware. The serial information would also be included in this firmware.

7.5 Implement OTA over Wi-Fi

7.5.1 OTA using local download server base on socket

The example shows how device updates image from a local download server. The local download server send image to device based on network socket.

Make sure both device and PC are connecting to the same local network.

7.5.1.1 Build OTA application image

Turn on OTA command

The flag defined in `\project\realtek_amebaz2_v0_example\inc\platform_opts.h`.

```
//on/off relative commands in log service
#define CONFIG_OTA_UPDATE 1
```

Check current loaded firmware index and acquire the upgraded firmware address

```
/* ota_8710c.c */
uint32_t update_ota_prepare_addr(void){
    uint32_t NewFWAddr;
    fw_img_export_info_type_t *pfw_image_info;

    pfw_image_info = get_fw_img_info_tbl();

    printf("\n\r[%s] Get loaded_fw_idx %d\n\r", __FUNCTION__,
pfw_image_info->loaded_fw_idx);
    if(pfw_image_info->loaded_fw_idx == 1)
        NewFWAddr = pfw_image_info->fw2_start_offset;
    else if(pfw_image_info->loaded_fw_idx == 2)
        NewFWAddr = pfw_image_info->fw1_start_offset;
    else {
        printf("\n\r[%s] Unexpected index %d", __FUNCTION__,
pfw_image_info->loaded_fw_idx);
        return -1;
    }
    printf("\n\r[%s]fw1 sn is %d, fw2 sn is %d\n\r", __FUNCTION__,
pfw_image_info->fw1_sn,pfw_image_info->fw2_sn);
    printf("\n\r[%s] NewFWAddr %08X\n\r", __FUNCTION__, NewFWAddr);
    return NewFWAddr;
}
```

Download the firmware to Ameba-ZII board to execute OTA.

7.5.1.2 Setup local download server

Step 1: Build **new** firmware `firmware_is.bin` and place it to `tools\DownloadServer` folder.

Step 2: Edit `start.bat` file: Port = 8082, file = `firmware_is.bin`

```
@echo off
DownloadServer 8082 firmware_is.bin
set /p DUMMY=Press Enter to Continue ...
```

Step 3: Execute `start.bat`.

```
c():checksum 0x202f57d
Listening on port (8082) to send firmware_is.bin (318592 bytes)
waiting for client ...
```

7.5.1.3 Execute OTA procedure

After device connects to AP, enter command: **ATWO=IP[PORT]**. Please note that the device and your PC need under the same AP. The IP in ATWO command is the IP of your PC.

```
# ATWO=192.168.0.103[8082]
[ATWO]: _AT_WLAN_OTA_UPDATE_

[MEM] After do cmd, available heap 92768

#
[update_ota_local_task] Update task start
[update_ota_prepare_addr] fw1 sn is 100, fw2 sn is 0
[update_ota_prepare_addr] NewFWAddr 00090000

[update_ota_local_task] Read info first
[update_ota_local_task] info 12 bytes
[update_ota_local_task] tx file size 0x4dc80
[update_ota_local_task] Current firmware index is 1

[update_ota_erase_upg_region] NewFWLen 318592
[update_ota_erase_upg_region] NewFWBlkSize 78 0x4e
[update_ota_local_task] Start to read data 318592 bytes
.
[update_ota_local_task] sig_backup for 32 bytes from index 0
.....
Read data finished

[update_ota_signature] Append OTA signature
[update_ota_signature] signature:
64 89 F2 09 0A 2A EC 7B 82 3F 1A 15 3C 92 00 66
98 6E 45 94 1E 1D 71 9C E0 E3 15 7A 7F 76 B1 89
[update_ota_local_task] Update task exit
[update_ota_local_task] Ready to reboot
== Rtl8710c IoT Platform ==
```

Local download server success message:

```
c():checksum 0x202f57d  
Listening on port (8082) to send firmware_is.bin (318592 bytes)  
  
waiting for client ...  
Accept client connection from 192.168.0.108  
Send checksum and file size first  
Send checksum byte 12  
Sending file...  
.....  
.....  
.....  
.....  
.....  
Total send 318592 bytes  
Client Disconnected.  
waiting for client ...
```

After finishing downloading image, device will be auto-rebooted, and the bootloader will boot by the firmware with larger serial number.

7.5.2 OTA using local download server based on HTTP

This example shows how device updates image from a local http download server.

The local http download server will send the http response which data part is *firmware_is.bin* after receiving the http request.

Make sure both device and PC are connecting to the same local network.

7.5.2.1 Build OTA application image

Turn on OTA command

The flags defined in `\project\realtek_amebaz2_v0_example\inc\platform_opts.h` and `\component\soc\realtek\8710c\misc\platform\ota_8710c.h`.

```
/* platform_opts.h */
//on/off relative commands in log service
#define CONFIG_OTA_UPDATE 1

#define CONFIG_EXAMPLE_OTA_HTTP 1
```

```
/* ota_8710c.h */
#define HTTP_OTA_UPDATE
```

Define Server IP and PORT in example_ota_http.c file

(In `\component\common\example\ota_http\example_ota_http.c`)

```
#define PORT 8082
#define IP "192.168.0.103"
#define RESOURCE "firmware_is.bin"
```

Check current loaded firmware index and acquire the upgraded firmware address

```
/* ota_8710c.c */
uint32_t update_ota_prepare_addr(void){
    uint32_t NewFWAddr;
    fw_img_export_info_type_t *pfw_image_info;

    pfw_image_info = get_fw_img_info_tbl();

    printf("\n\r[%s] Get loaded_fw_idx %d\n\r", __FUNCTION__,
pfw_image_info->loaded_fw_idx);
    if(pfw_image_info->loaded_fw_idx == 1)
        NewFWAddr = pfw_image_info->fw2_start_offset;
    else if(pfw_image_info->loaded_fw_idx == 2)
        NewFWAddr = pfw_image_info->fw1_start_offset;
    else {
        printf("\n\r[%s] Unexpected index %d", __FUNCTION__,
pfw_image_info->loaded_fw_idx);
        return -1;
    }
    printf("\n\r[%s]fw1 sn is %d, fw2 sn is %d\r\n", __FUNCTION__,
pfw_image_info->fw1_sn,pfw_image_info->fw2_sn);
    printf("\n\r[%s] NewFWAddr %08X\n\r", __FUNCTION__, NewFWAddr);
    return NewFWAddr;
}
```

Download the firmware to Ameba-ZII board to execute OTA.

Communication with Local HTTP download server

1. In *http_update_ota_task()*, after connecting with server, Ameba will send a HTTP request to server : "GET /RESOURCE HTTP/1.1\r\nHost: host\r\n\r\n".
2. The local HTTP download server will send the HTTP response after receiving the request. The response header contains the "Content-Length" which is the length of the *firmware_is.bin*. The response data part is just *firmware_is.bin*.
3. After Ameba receiving the HTTP response, it will parse the http response header to get the content length to judge if the receiving *firmware_is.bin* is completed.

7.5.2.2 Setup local HTTP download server

Step 1: Build **new** firmware firmware_is.bin and place to tools\DownloadServer(HTTP) folder.

Step 2: Edit start.bat file: Port = 8082, file = firmware_is.bin

```
@echo off
DownloadServer 8082 firmware_is.bin
set /p DUMMY=Press Enter to Continue ...
```

Step 3: Execute start.bat.

```
<Local HTTP Download Server>
Listening on port (8082) to send firmware_is.bin (320256 bytes)
waiting for client ...
```

7.5.2.3 Execute OTA procedure

Reboot the device and connect to AP, it should start the OTA update through HTTP protocol after 1 minute.

```
#
#
[update_ota_prepare_addr] fw1 sn is 100, fw2 sn is 0
[update_ota_prepare_addr] NewFWAddr 00090000

[http_update_ota] Download new firmware begin, total size : 320256
[http_update_ota] Current firmware index is 1
[http_update_ota] fw size 320256, NewFWAddr 00090000

[update_ota_erase_upg_region] NewFWLen 320256
[update_ota_erase_upg_region] NewFWBlkSize 79 0x4f.
[http_update_ota] sig_backup for 32 bytes from 0 index
.....
[http_update_ota] Download new firmware 320256 bytes completed

[update_ota_signature] Append OTA signature
[update_ota_signature] signature:
DD E9 FE 19 3B 15 79 99 8A 3C 84 FE 28 FB A2 13
53 0F DE 71 3B 7E 46 48 9F 9D 03 2C DB EB D3 B7
[http_update_ota_task] Update task exit
[http_update_ota_task] Ready to reboot
== Rtl8710c IoT Platform ==
```

Local download server success message:

```
<Local HTTP Download Server>  
Listening on port (8082) to send firmware_is.bin (320256 bytes)  
  
waiting for client ...  
Accept client connection from 192.168.0.108  
Waiting for client's request...  
Receiving GET request, start sending file...  
.....  
.....  
.....  
.....  
.....  
.....  
Total send 320299 bytes  
Client Disconnected.  
waiting for client ...
```

After finishing downloading image, device will be auto-rebooted, and the bootloader will load new firmware if it exists.

7.6 OTA signature

ATSC is an AT command to clear the OTA signature of current running firmware. ATSR is an AT command to recover the OTA signature of the firmware whose signature is cleared. Please note that you need to reset your board after using ATSC & ATSR command. For more details, please refer to AN0025.