Hi3861 V100 / Hi3861L V100 Boot Porting

Development Guide

Issue 01

Date 2020-04-30

Trademarks and Permissions

Notice

HiSilicon (Shanghai) Technologies Co., Ltd.

Address:	New R&D Center, 49 Wuhe Road, Bantian, Longgang District, Shenzhen 518129 P. R. China
Website:	https://www.hisilicon.com/en/
Email:	support@hisilicon.com

About This Document

Purpose

This document describes the Hi3861 V100/Hi3861L V100 ROMBoot, LoaderBoot, and FlashBoot processes, providing reference for secondary development of FlashBoot.

Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3861	V100
Hi3861L	V100

Intended Audience

The document is intended for:

- Technical support engineers
- Software development engineers

Symbol Conventions

The following table describes the symbols that may be found in this document.

Symbol	Description
DANGER	Indicates a hazard with a high level of risk which, if not avoided, will result in death or serious injury.
WARNING	Indicates a hazard with a medium level of risk which, if not avoided, could result in death or serious injury.

Symbol	Description
⚠ CAUTION	Indicates a hazard with a low level of risk which, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
📖 NOTE	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

Change History

Issue	Date	Change Description
01	2020-04-30	This issue is the first official release. • In **About This Document**, **Purpose** is updated. • In **1 Introduction to Boot**, the description of boot classification is updated. **Figure 1-1** is updated. • **2.1 Image Downloading and eFUSE Burning** is updated. • **3 LoaderBoot Description** is added. • In **4.2 Boot Directory Structure**, the title is updated. **Table 4-1** is updated. • In **5.3.1 Signing and Encryption Configuration**, **Figure 5-4** is updated. • In **5.3.2 Signature Tool**, **Figure 5-5** is updated. The descriptions of the **-u**, **-f**, and **-z** parameters are added. • In **5.4 Available Boot APIs**, the title is updated.
00B02	2020-02-12	In **Table 4-1**, the names of the chip fixed interface header file directory, link file directory, and driver source file directory are updated. **5.1 Building FlashBoot** and **5.3 Secure Boot Configuration for FlashBoot** are added.
00B01	2020-01-15	This issue is the first draft release.

Contents

1 Introduction to Boot

Hi3861 V100/Hi3861L V100 consists of ROMBoot, FlashBoot, LoaderBoot, and CommonBoot.

ROMBoot provides the following functions:

- Loads LoaderBoot to the RAM, downloads images to the flash memory, and burns the eFUSE.
- Verifies and boots FlashBoot. FlashBoot has two sides: A side and B side. If A side passes the verification, the system directly boots. If A side fails the verification, B side will be verified. If B side passes the verification, the system repairs A side and then boots; otherwise, the system resets and reboots.

FlashBoot provides the following functions:

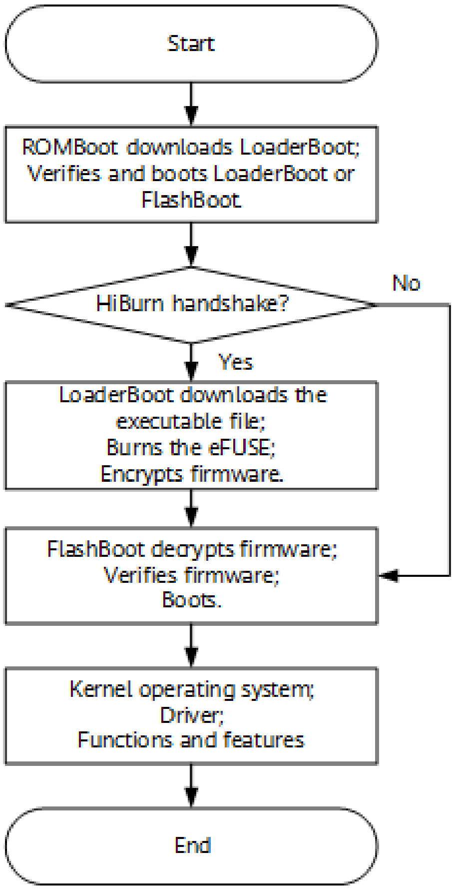
- Upgrades the firmware.
- Verifies and boots the firmware.

LoaderBoot provides the following functions:

- Downloads images to the flash memory.
- Burns the eFUSE (for example, keys related to secure boot and flash encryption).

CommonBoot is a functional module shared by Flashboot and LoaderBoot.

Figure 1-1 Boot process



2 ROMBoot Description

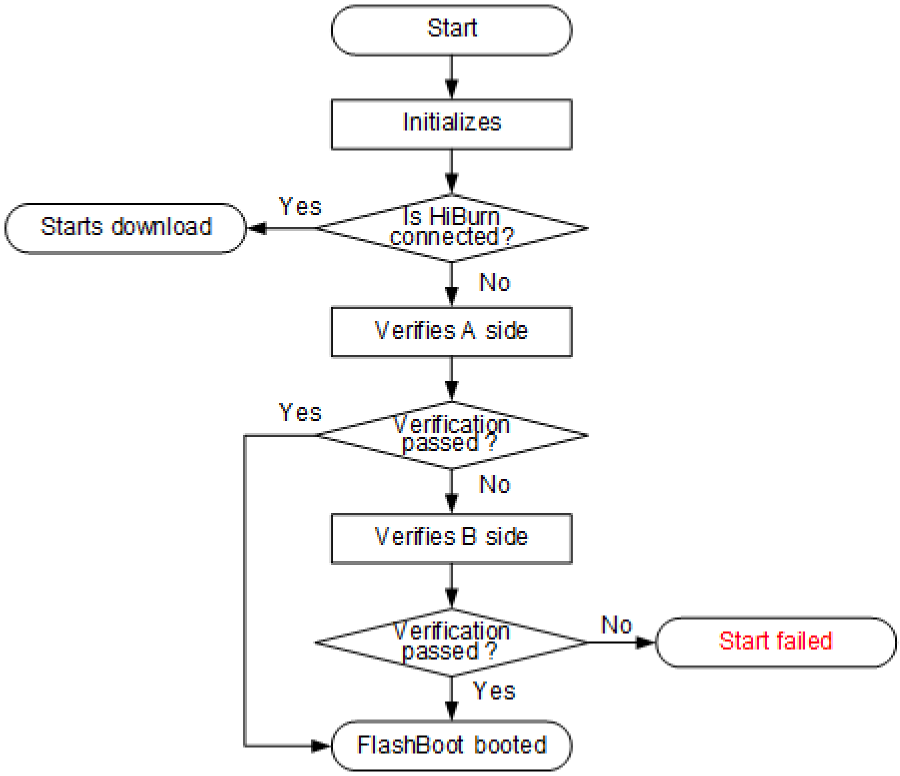
2.1 Image Downloading and eFUSE Burning

For details about how ROMBoot downloads images to the flash memory by using Loaderboot and burns the eFUSE, see the *Hi3861 V100/Hi3861L V100 HiBurn User Guide.*

2.2 Checking and Booting FlashBoot

Figure 2-1 shows the process of verifying and booting FlashBoot.

Figure 2-1 Process of verifying and booting FlashBoot



3 LoaderBoot Description

LoaderBoot is a component that directly interacts with HiBurn. ROMBoot cannot directly implement the burning function. Instead, it needs to load LoaderBoot to the RAM, jump to LoaderBoot, and then burn related content by using LoaderBoot. LoaderBoot can burn the following content:

- FlashBoot
- eFUSE parameter configuration file
- Firmware image (including NV parameters)
- Production test image

NOTE

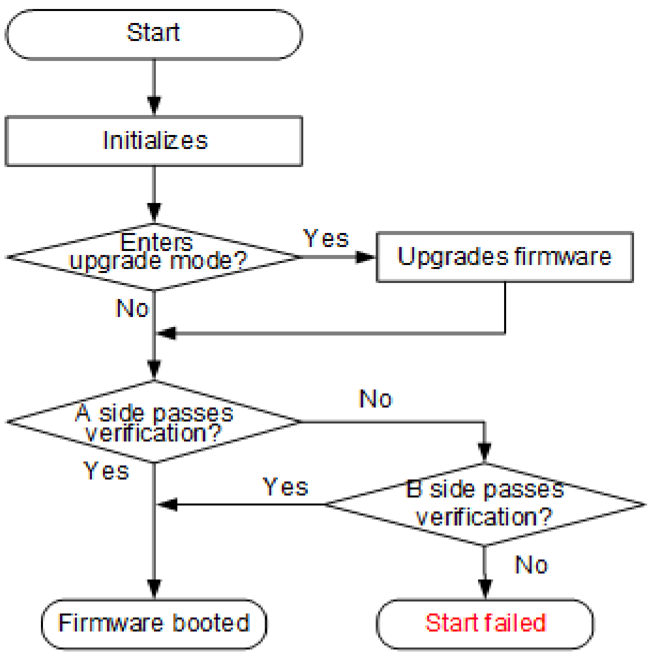
LoaderBoot generally does not involve secondary development. If you need to modify it in some application scenarios, you can directly modify the LoaderBoot source code. The SDK compiles and updates LoaderBoot by default. For details about the directory structure of LoaderBoot, see **4.2 Boot Directory Structure**.

4 FlashBoot Description

4.1 FlashBoot Boot Process

Figure 4-1 shows the process of verifying and booting the firmware.

Figure 4-1 Process of verifying and booting firmware



4.2 Boot Directory Structure

The boot directory of the SDK contains the source code and header files of boot. **Table 4-1** shows the directory structure.

Table 4-1 Boot directory structure

Directory	Path	Description
flashboot	boot\flashboot\include	FlashBoot header files
	boot\flashboot\startup	Boot assembly and main program
	boot\flashboot\drivers	Driver source files, including the flash, ADC, and eFUSE drivers
	boot\flashboot\common	Source files of common components, including NV interfaces and partition table interfaces
	boot\flashboot\upg	Source files of the upgrade function
	boot\flashboot\lzmaram	Compressed FlashBoot files
	boot\flashboot\secure	Encrypted FlashBoot files
	boot\flashboot\lib	Library files
	Makefile	FlashBoot makefile
	module_config.mk	Configuration file of the FlashBoot script
	SConscript	SCons build script
loaderboot	boot\loaderboot\fixed\include	API header files of the chip
	boot\loaderboot\include	LoaderBoot header files
	boot\loaderboot\startup	Boot assembly and main program
	boot\loaderboot\drivers	Driver source files, including the flash, ADC, and eFUSE drivers
	boot\loaderboot\common	Source files of common components, including NV interfaces and partition table interfaces
	boot\loaderboot\secure	Encrypted LoaderBoot files
	Makefile	LoaderBoot makefile
	module_config.mk	Configuration file of the LoaderBoot script

Directory	Path	Description
	SConscript	SCons build script
commonboot	boot\commonboot\crc32	CRC32 driver shared by FlashBoot and LoaderBoot
	boot\commonboot\efuse	eFUSE driver shared by FlashBoot and LoaderBoot
	boot\commonboot\flash	Flash driver shared by FlashBoot and LoaderBoot

5 FlashBoot Secondary Development Guidance

5.1 Building FlashBoot

Run the **sh build.sh** command in the root directory to build the kernel and FlashBoot at the same time. The following files are generated after FlashBoot build:

- **output\bin\Hi3861_boot_signed.bin**: FlashBoot image to be written to the header of the flash memory
- **output\bin\Hi3861_boot_signed_B.bin**: backup FlashBoot image to be written to the tail of the flash memory

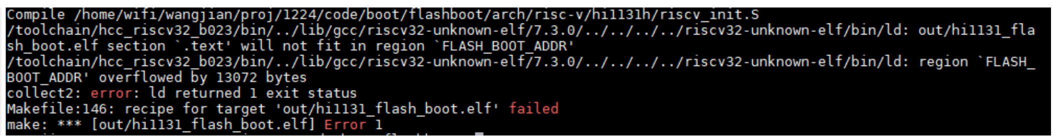
FlashBoot uses the SHA256 signature mode by default and can be directly built. For details about other signing modes, see section **5.3 Secure Boot Configuration for FlashBoot**.

5.2 Adjusting Memory Layout

For details about the memory layout of FlashBoot, see the **flashboot_sha256.lds**, **flashboot_rsa.lds**, and **flashboot_ecc.lds** files in the **build\scripts** directory. The layouts with different suffixes are used for different signing modes.

After code development, a link error may occur due to insufficient space. For example, the error messages shown in **Figure 5-1** might be displayed.

Figure 5-1 Example of a link error due to insufficient space



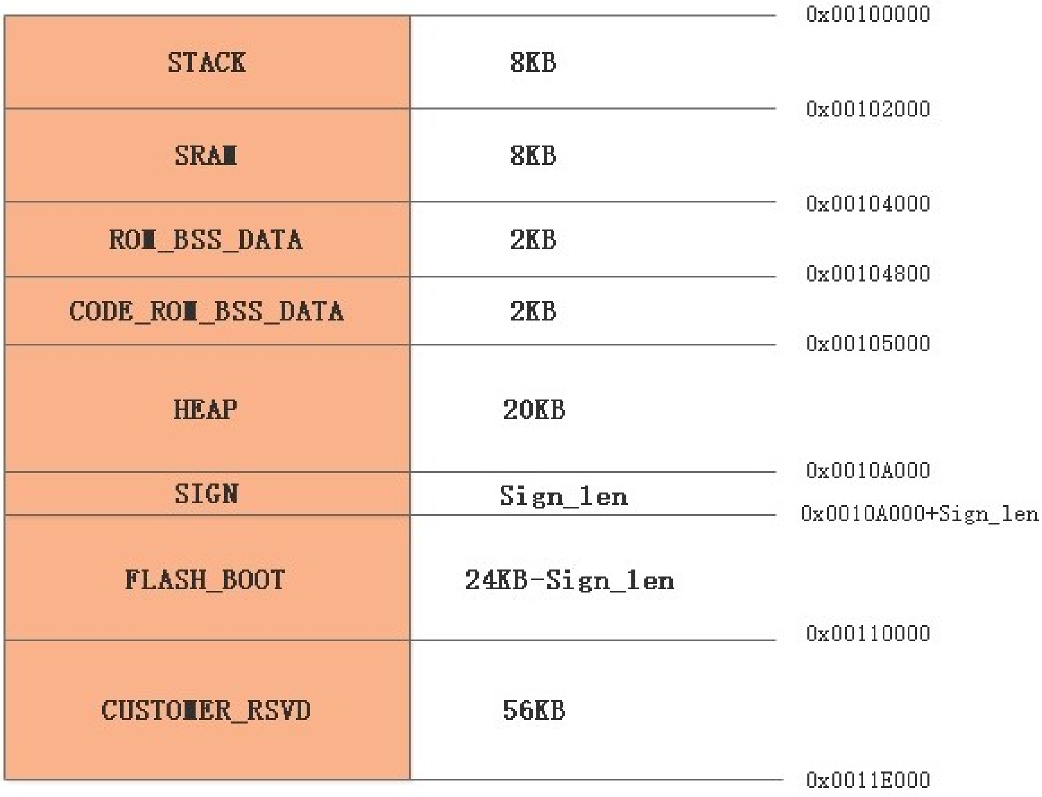
```
Compile /home/wifi/wangjian/proj/1224/code/boot/flashboot/arch/risc-v/hi1131h/riscv_init.S
/toolchain/hcc_riscv32_b023/bin/../lib/gcc/riscv32-unknown-elf/7.3.0/../../../../riscv32-unknown-elf/bin/ld: out/hi1131_fla
sh_boot.elf section `.text' will not fit in region `FLASH_BOOT_ADDR'
/toolchain/hcc_riscv32_b023/bin/../lib/gcc/riscv32-unknown-elf/7.3.0/../../../../riscv32-unknown-elf/bin/ld: region `FLASH_
BOOT_ADDR' overflowed by 13072 bytes
collect2: error: ld returned 1 exit status
Makefile:146: recipe for target 'out/hi1131_flash_boot.elf' failed
make: *** [out/hi1131_flash_boot.elf] Error 1
```

The solution is as follows:

Step 1 Open the link file **build\scripts\flashboot_xxx.lds**.

Step 2 Figure 5-2 shows the current memory usage. You can adjust the size of the error paragraph and the start addresses of adjacent partitions as required. Make sure that the partitions do not overlap.

Figure 5-2 Current memory usage



The partitions are described as follows:

- STACK: stack space configuration during runtime. When a stack overflow problem occurs, the stack space needs to be modified.
- SRAM: data segment private to FlashBoot
- ROM_BSS_DATA: data segment shared by ROMBoot and FlashBoot. The content must not be modified.
- CODE_ROM_BSS_DATA: data segment shared by ROMBoot and FlashBoot. The content must not be modified.
- HEAP: heap space used for dynamic request during runtime
- SIGN: FlashBoot signature. The length of this partition is related to the signing mode. The mapping is as follows:
  - SHA256 signature length: 0x40
  - RSA_V15/RSA_PSS signature length: 0x5A0
  - ECC signature length: 0x150
- FLASH_BOOT: FlashBoot images. A total of 24 KB space is reserved, including the FlashBoot signature header.

- CUSTOMER_RSVD: reserved for the user when the STACK, SRAM, HEAP, or FLASH_BOOT space is insufficient
- FIXED_ROM: code segment shared by ROMBoot and FlashBoot. The content must not be modified.
- CODE_ROM: code segment shared by ROMBoot and FlashBoot. The content must not be modified.

----End

5.3 Secure Boot Configuration for FlashBoot

FlashBoot supports three-level security protection, with the highest security performance at the top level. The SDK uses the lowest security level (SHA256 signature) by default.

- FlashBoot is signed using SHA256. ROMBoot checks the integrity of the FlashBoot image by verifying the SHA256 value and boots FlashBoot.
- FlashBoot is signed using RSA/ECC. ROMBoot verifies the validity of the signature based on the root key hash in the eFUSE and the signature data of FlashBoot, and then boots FashBoot.
- FlashBoot is signed using RSA/ECC, and the code segment is encrypted in AES-CBC mode. ROMBoot generates a key based on the root key salt in the eFUSE, decrypts the FlashBoot code segment based on the initialization vector (IV) in the FlashBoot signature, and then performs RSA/ECC signature verification, and then boots FashBoot.

NOTE

For details about secure boot, see the *Cyber Security Precautions for Hi3861 V100/Hi3861 LV100 Secondary Development.*

5.3.1 Signing and Encryption Configuration

You can configure the FlashBoot signing mode by choosing **Security Settings** in menuconfig. The supported modes are as follows:

- **SHA256**: Only the SHA256 signing of the FlashBoot .bin file is computed.
- **RSA_V15**: 4096-bit root RSA key and 2048-bit level-2 RSA key, using the PKCS1_V15 padding mode.
- **RSA_PSS**: 4096-bit root RSA key and 2048-bit level-2 RSA key, using the PKCS1_PSS padding mode.
- **ECC**: ECDH_BRAIN_POOL_P256R1 key signing.

Figure 5-3 Example for configuring the signing mode

```
(Top) → Security Settings → Signature Algorithm for bootloader and upgrade file

( ) RSA_V15
(X) RSA_PSS
( ) ECC
( ) SHA256
```

You can configure whether to encrypt FlashBoot in menuconfig. When the signing mode is RSA_V15, RSA_PSS, or ECC, the lower-level menu **Enable bootloader**

encryption is displayed in menuconfig. After configuration, the code segment of the signed FlashBoot is encrypted.

Figure 5-4 Example for enabling encryption

```
(Top) → Security Settings

    Signature Algorithm for bootloader and upgrade file (ECC)  --->
[ ] Enable bootloader encryption (NEW)
(0) kernel ver(value form 0 to 48)
(0) boot ver(value form 0 to 16)
[ ] TEE HUKS support
[ ] FLASH ENCRYPT support
```

NOTE

- For details about menuconfig instructions, see the *Hi3861 V100/Hi3861L V100 SDK Development Environment Setup User Guide.*
- The key required for signing and encryption is not provided in the SDK. You need to generate the key by yourself. For details, see **5.3.3 Key File Description**.

5.3.2 Signature Tool

A FlashBoot signing tool is provided in **tools/sign_tool/sign_tool** of the SDK. **Figure 5-5** shows how to obtain the help information of the tool on Linux.

Figure 5-5 Help information of sign_tool

```
sign_tool version 1.2

sign_tool: [options]
           -h help
           -i [input file path]
           -o [output file]
           -r [root key file]          RSA4096 or ECDSA BRAIN_POOL_P256R1
           -s [sub key file]           RSA2048 or ECDSA BRAIN_POOL_P256R1
           -c [sub key category]       sub key category [0: 0xFFFFFFFF]
           -l [sub key id]             sub key id [0: 23]
           -d [die id]                 must be 48 hex nums
           -a [sign alg]               0:RSA PKCS1_V15;  1: RSA PKCS1_PSS;  2: ECC BRAIN_POOL_P256R1
           -v [boot_ver]               range [0, 16], decimal
           -e [aes key file]
           -t generate tail flashboot
           -n Non security bin
           -u [aes key file] encryption upgrade bin
           -f [offset] hexadecimal num,upgrade bin encryption address offset
           -z [length] hexadecimal num,upgrade bin encryption length

  non security example:
  ./sign_tool -i flash_boot.bin -o flash_boot_nos.bin -n

  security example:
  ./sign_tool -i flash_boot.bin -o flash_boot_r.bin -r root_rsa.pem -s sub_rsa.pem -a 1 -e key.txt
```

The arguments are described as follows:

- **-h**: help information
- **-i**: input file, which must contain the path and file name
- **-o**: output FlashBoot_A file, which must contain the path and file name
- **-r**: root key, which must contain the path and file name.
- **-s**: level-2 key, which must contain the path and file name.
- **-e**: encryption key, which must contain the path and file name

- **-t**: output FlashBoot_B file, which must contain the path and file name.
- **-c**: level-2 key type (a 32-bit unsigned integer)
- **-l**: level-2 key ID. The value range is [0, 23].
- **-v**: FlashBoot version. The value range is [0, 16].
- **-n**: SHA256 signature
- **-a**: signature algorithm, either **0** (RSA_V15), **1** (RSA_PSS), or **2** (ECC)
- **-d**: die ID (48-byte hexadecimal). A FlashBoot image signed with this argument can be used only on the chip with the specified die ID.

  You can use this tool to pass required arguments to sign FlashBoot.
- **-u**: encrypted upgrade file
- **-f**: offset address for encryption and upgrade. The value is a hexadecimal number.
- **-z**: length of the encrypted upgrade. The value is a hexadecimal number.

5.3.3 Key File Description

- The SHA256 signature does not require a key.
- The RSA_V15 or RSA_PSS signature requires a 4096-bit RSA root key in the base and a 2048-bit level-2 RSA key. Rename the generated key files **root_rsa.pem** and **sub_rsa.pem** and place them in the **tools\sign_tool** directory. To use the OpenSSL library to generate the keys on Linux, run the following commands:
  ```
  openssl genrsa -out root_rsa.pem 4096
  openssl genrsa -out sub_rsa.pem 2048
  ```
- The ECC signature requires an ECDH_BRAIN_POOL_P256R1 ECC root key and a ECDH_BRAIN_POOL_P256R1 level-2 ECC key. Rename the generated key files **root_ecc.pem** and **sub_ecc.pem** and place them in the **tools\sign_tool** directory. To use the OpenSSL library to generate the keys on Linux, run the following commands:
  ```
  openssl ecparam -genkey -name brainpoolP256r1 -out root_ecc.pem
  openssl ecparam -genkey -name brainpoolP256r1 -out sub_ecc.pem
  ```
- Three values need to be written into the encryption key file:
  - EFUSE_DATA: a 32-byte value the same as the value written to the root_key partition of the eFUSE. It is the required hardware unique key (HUK) generated using the KDF algorithm.
  - CPU_DATA: a 16-byte random number. This 16-byte random number and the other 16-byte random number in ROMBoot are combined to form a 32-byte value for the KDF to generate the key IV using the KDF algorithm.
  - IV_DATA: a 16-byte random number as the IV used for AES-CBC encryption.

  Rename the key file **aes_key.txt** and place it in **tools\sign_tool**. Figure 5-6 shows the format of the key file.

Figure 5-6 aes_key.txt file format

```
[E]:15A146973144B9C52FECD0A15AF7585C2C0A69F5633325F5750260C141FF1047 ;EFUSE_DATA
[C]:970B7913267947EEBD9C9DD396EFE7DD ;CPU_DATA
[I]:E0523B468F261AF46E5B0C7C1338DD17 ;IV_DATA
```

5.4 Available Boot APIs

For details about the boot APIs, see the *Hi3861 V100/Hi3861L V100 Boot API Development Reference.*