GigaDevice MCU

GD32 SBSFU User Guide

Version1.0

Introduction

Secure Boot and Secure Firmware Update ensure the integrity and authenticity of the device during the boot process, while ensuring that only trusted firmware can be loaded and executed. This document describes the implementation of secure boot and secure firmware update functions based on the GD32 MCU, as well as the methods of use.

Table of Contents

Figure

Table

1 Terminology and Abbreviations

The professional terms and abbreviations used in this document are as shown in ***Table 1-1. Common Acronym Comparison Table***.

Table 1-1. Common Acronym Comparison Table

Acronyms	Full Name	Chinese Terms
API	Application programming interface	应用程序接口
IBL	Immutable bootloader	不可变的引导加载程序
IDE	Integrated Development Environment	集成开发环境
ISP	In system programming	在系统可编程
MBL	Main bootloader	主要的引导加载程序
OTP	One time programmable	一次性可编程器件
ROTPK	Root of trust public key	信任根公钥
SBSFU	Secure boot and secure firmware updates	安全启动与安全固件更新

2 SBSFU Introduction

2.1 Introduction

The complexity and variability of the current device operating environment are continuously increasing, which makes it particularly important to prevent and avoid attacks and damage to the equipment. In view of this, the design and implementation of device security are indispensable. Among them, secure boot and secure firmware updates are key components of the device's secure operation, jointly ensuring the integrity, authenticity, and stability of the device during the boot process.

2.2 Features

- Fixed and unchangeable startup entry (the only startup entry);
- Only allow the execution of software code that has passed digital signature verification for integrity and authenticity;
- The safety boot code should be as simple, reliable, and universal as possible;
- Controlled code updates (methods and content of updates);
- Firmware updates are only allowed only when the updates firmware have passed integrity and authenticity checks.

2.3 Secure Boot

Secure Boot is a security mechanism implemented during the device boot process, designed to ensure that the device can only boot software that has been verified for integrity and authenticity through digital signatures:

- SPC or EFUSE ensures a unique boot entry point;
- Use SHA256 and asymmetric encryption to verify integrity and authenticity (currently using ECDSA-Secp256r1-SHA256);
- Random startup time;
- Support for FWDG, tamper detection, etc;
- Inspect the next phase of the code version;
- Delivery of status information;
- Support for the Mbed TLS encryption and decryption library;
- Export API;
- Support for hardware acceleration engines (HAU, CAU, PKCAU, and TRNG);
- SPC or EFUSE prohibits debugging and boot code from SRAM.

2.4 Secure firmware update

The secure firmware update refers to allowing updates only for firmware that has passed integrity and authenticity checks. The characteristics of secure firmware updates are as follows:

- Provide code updates for Ymodem protocol with serial port support;
- Check the integrity and authenticity of the update content, and only the firmware that passes the check can be updated;
- Support for anti-rollback and rollback.

3 SBSFU Project Introduction

3.1 Introduction

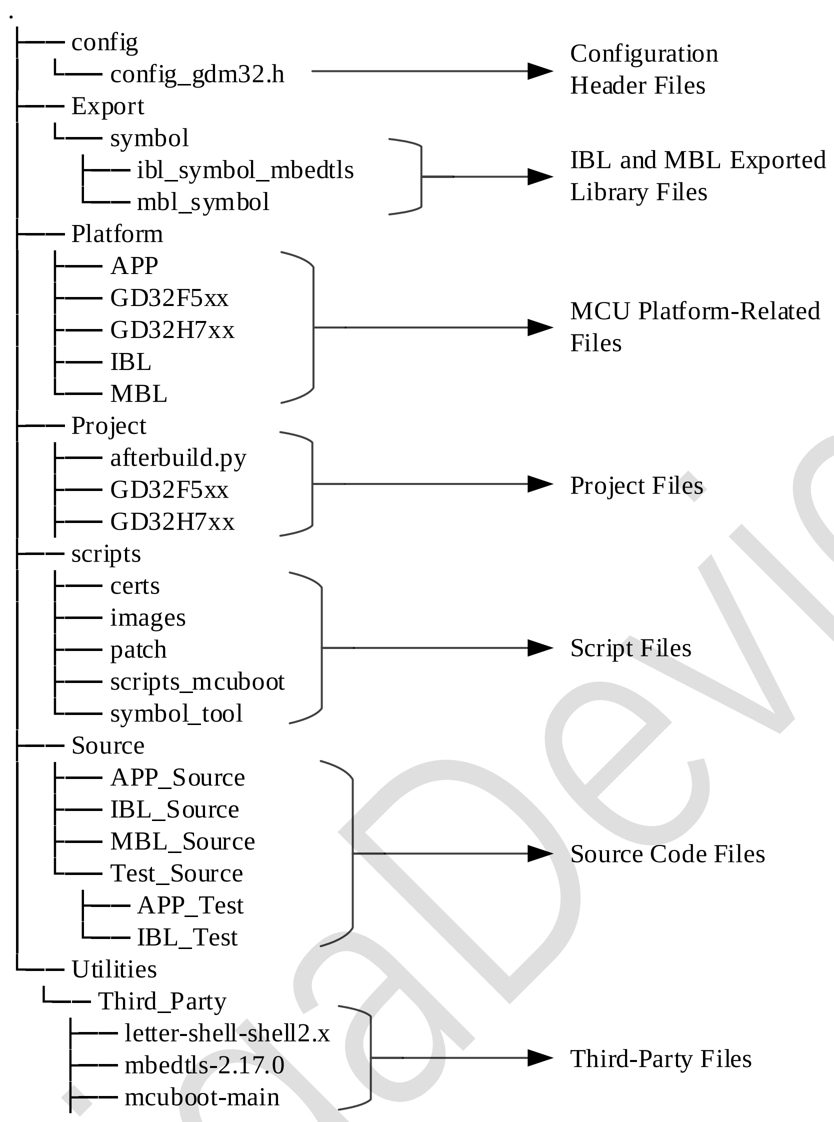
SBSFU is supported by three parts of code: IBL (fixed boot code), MBL (main boot code), and APP (user code). After the device is reset, the IBL code runs first, then the IBL verifies and starts the MBL, and finally, the MBL verifies and starts the user code, completing the boot process.

3.2 Project Structure

The SBSFU project directory structure can refer to ***Figure 3-1. SBFU Project Directory Structure***. The main parts and their functions are as follows:

- Configuration directory: Contains global configuration header files, which will be used in program compilation and script programs;
- Export directory: Contains export library files for IBL and MBL, IBL library files can be called in MBL and APP, MBL export library files can be used in APP;
- Platform directory: contains source files related to the MCU platform;
- Project directory: It contains the project files, currently only supports project management using KEIL IDE;
- Scripts directory: mainly includes the script programs used in the project;
- Source directory: It contains the main implementation code and test code files;
- Utilities directory: The directory where third-party source code is stored, such as mbedtls, shell, and MCUBOOT source code.

Figure 3-1. SBFU Project Directory Structure



```
.
├── config
│   └── config_gdm32.h
├── Export
│   └── symbol
│       ├── ibl_symbol_mbedtls
│       └── mbl_symbol
├── Platform
│   ├── APP
│   ├── GD32F5xx
│   ├── GD32H7xx
│   ├── IBL
│   └── MBL
├── Project
│   ├── afterbuild.py
│   ├── GD32F5xx
│   ├── GD32H7xx
├── scripts
│   ├── certs
│   ├── images
│   ├── patch
│   ├── scripts_mcuboot
│   └── symbol_tool
├── Source
│   ├── APP_Source
│   ├── IBL_Source
│   ├── MBL_Source
│   └── Test_Source
│       ├── APP_Test
│       └── IBL_Test
└── Utilities
    └── Third_Party
        ├── letter-shell-shell2.x
        ├── mbedtls-2.17.0
        ├── mcuboot-main
```

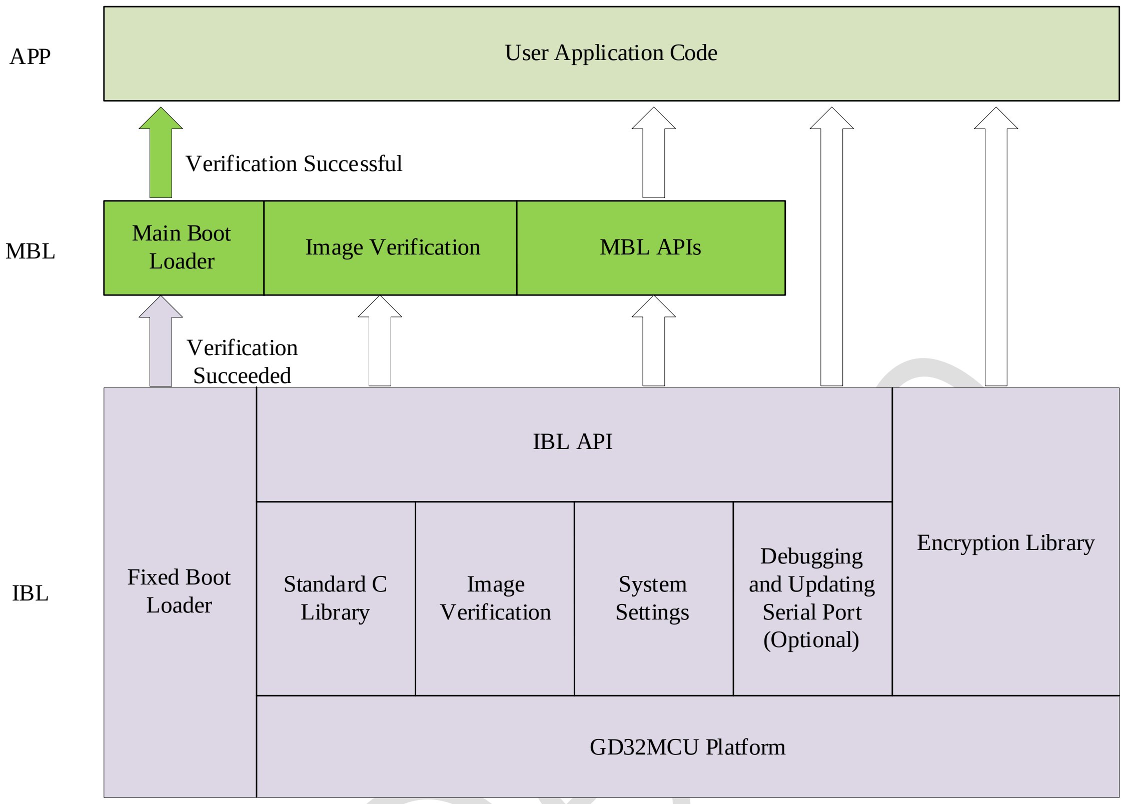
Configuration Header Files
IBL and MBL Exported Library Files
MCU Platform-Related Files
Project Files
Script Files
Source Code Files
Third-Party Files

3.3 Software Framework

The software framework of the SBSFU project, as shown in ***Figure 3-2. Software Framework***, is mainly composed of three parts:

- IBL: Includes the standard C library, encryption and decryption library, IBL's API, Ymodem update code through serial port, and verifies MBL and if verify passed then executes MBL;
- MBL: Contains the main boot-up functions, verified and executed by IBL, MBL also verifies the APP image, and after passing the verification, it executes the APP;
- APP: User code, verified and executed by MBL.

Figure 3-2. **Software Framework**

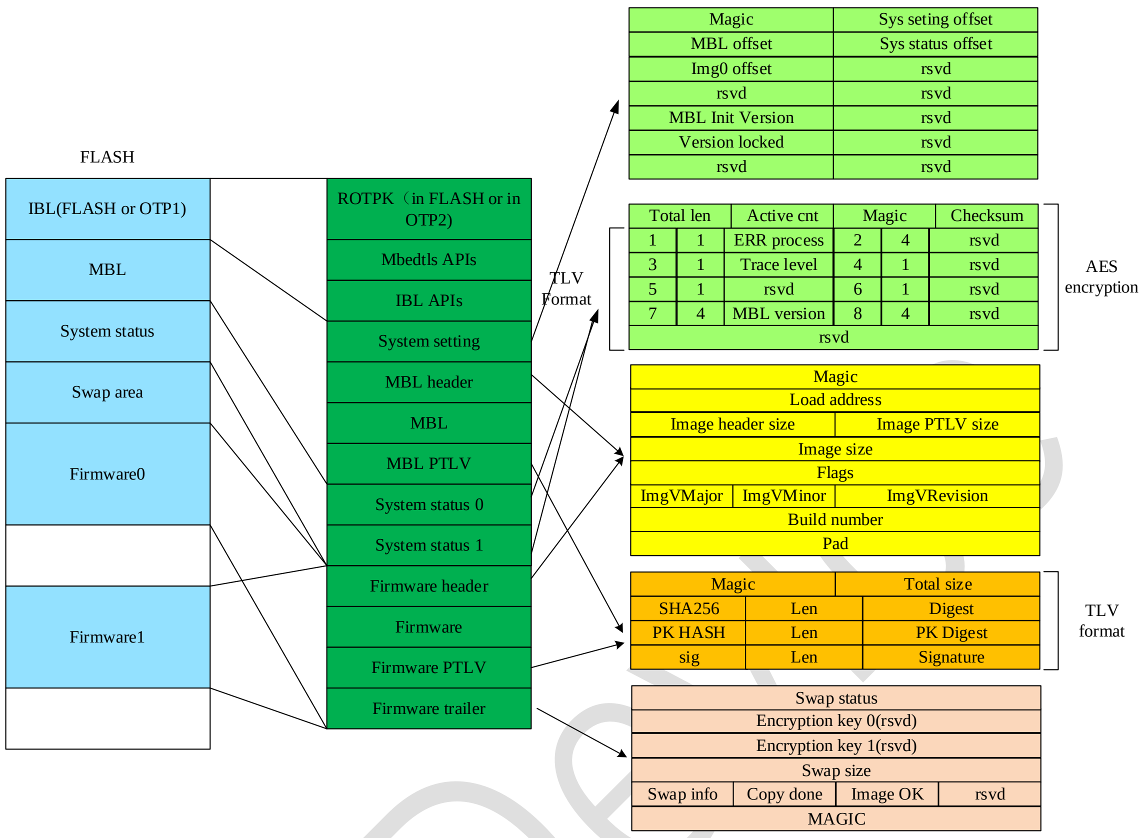


3.4 FLASH Layout

The FLASH layout of the SBSFU project, as shown in ***Figure 3-3. FLASH Layout***.

- IBL can be executed on the chip in the FLASH or OTP1 area (such as GD32F5xx, which supports loading IBL to the OTP1 area for execution), and it is stored in FLASH by default; ROTPK is also stored in FLASH by default, but on GD32F5xx, it supports setting it to the OTP2 area.

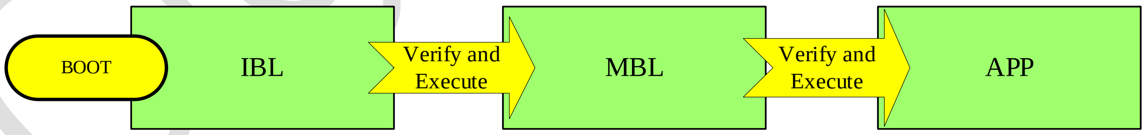
Figure 3-3. **FLASH Layout**



3.5 Trust Chain and Implementation of the Trust Chain

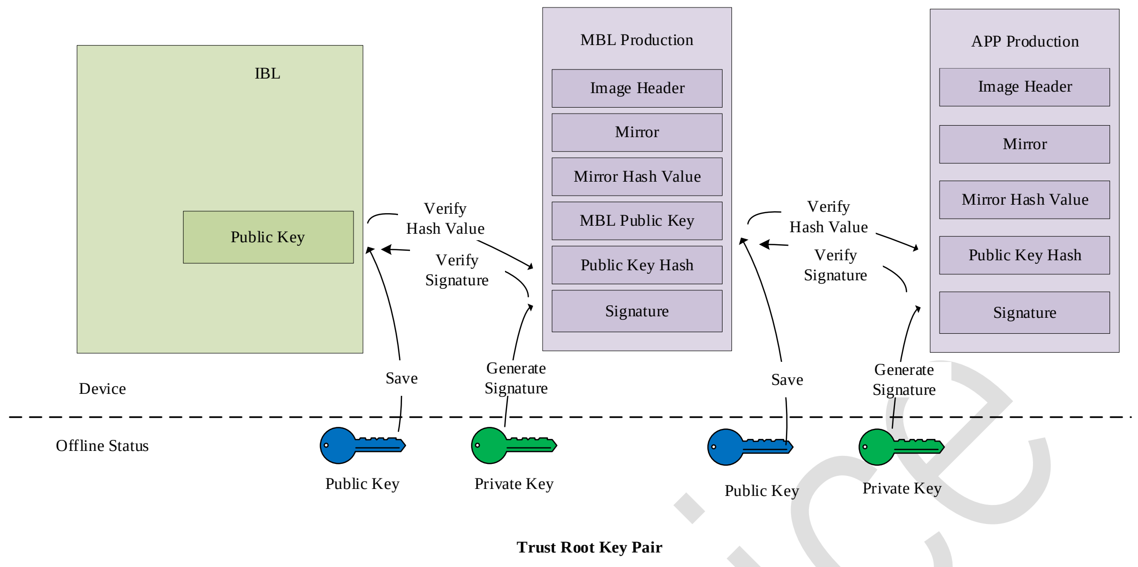
The trust chain of the SBSFU project is shown in *Figure 3-4. Trust chain*. As indicated in the trust chain, after the system starts up, the IBL code will run first. Then, the IBL will verify the MBL code, and after passing the verification, it will execute the MBL code. The MBL code will verify the APP code, and after passing the verification, it will execute the APP code.

Figure 3-4. **Trust chain**



The implementation of the trust chain in the SBSFU project is shown in *Figure 3-5. Implementation of the Trust Chain*. Taking IBL as an example, asymmetric SECP256R1 is used for signature and verification. The public key is stored in FLASH or OTP2. The SHA256 hash value of the public key is appended to the end of the MBL image, along with the SHA256 hash value of the MBL and the signature value of the MBL image signed with the private key. IBL will verify the validity of the public key by checking the hash of the public key, determine the validity of the image through the image's hash, and finally complete the verification of the integrity and authenticity of the image through the signature information. The process of MBL verifying the APP is similar to the verification process of IBL, and currently, the same public key is used for IBL and MBL verification.

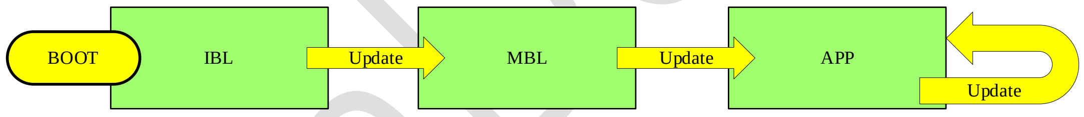
Figure 3-5. Implementation of the Trust Chain



3.6 Implementation of Firmware Update

The SBSFU project supports firmware updates using the serial port YMODEM protocol, and the implementation of firmware updates is shown in ***Figure 3-6. Firmware Update***.

Figure 3-6. Firmware Update



After the system starts up, the IBL (Initial Boot Loader) code will run first. If there is a need to update, the IBL code can update the MBL (Main Boot Loader) firmware area code through the serial port. The APP's firmware is divided into two areas, "Firmware0" and "Firmware1". The "Firmware0" area is the execution area, and the "Firmware1" area is the firmware update area. If there is a need for APP firmware update, MBL can update the APP firmware to the update area through the serial port, and the APP also supports updating the APP firmware to the APP firmware update area. After each firmware update is completed, the system will be reset by software and restarted.

4 IBL Project

The IBL project is one of the trust anchors of the SBSFU project. After the system starts, the IBL project should be executed first, followed by the execution of software code that has passed integrity and authenticity checks by the IBL, to complete the system's boot process. The IBL also integrates basic security code such as the mbedTLS encryption and decryption library, chip drivers, and firmware updates. Its main functions and features are as follows：

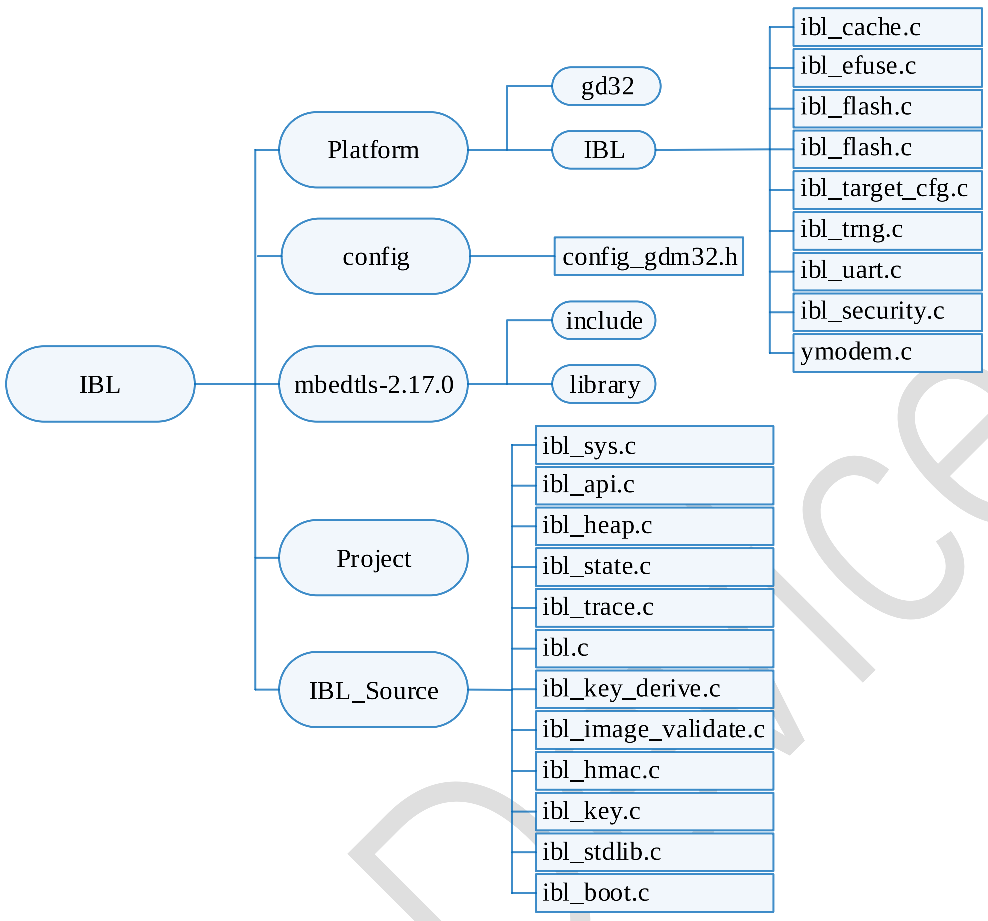
After the reset, the code that is to be executed should start first;

- Support for the mbedTLS encryption and decryption library, and the export of the encryption and decryption library;
- Firmware update implementation of Ymodem protocol over serial port;
- Directly obtain ROTPK;
- Support the verification of the integrity and authenticity of MBL before executing it;
- Support for the management of the MBL version;
- Implement device-level security feature configuration and detection;
- Delivery of exported IBL API and IBL initialization status information.

4.1 IBL Project Code File Structure

The file structure of the IBL project is shown in ***Figure 4-1. IBL Project Code File Structure***. It mainly includes the chip's low-level driver library, the mbedTLS encryption and decryption library, the standard C library, the API code, the image signature verification, MBL image update and jump to excute MBL code.

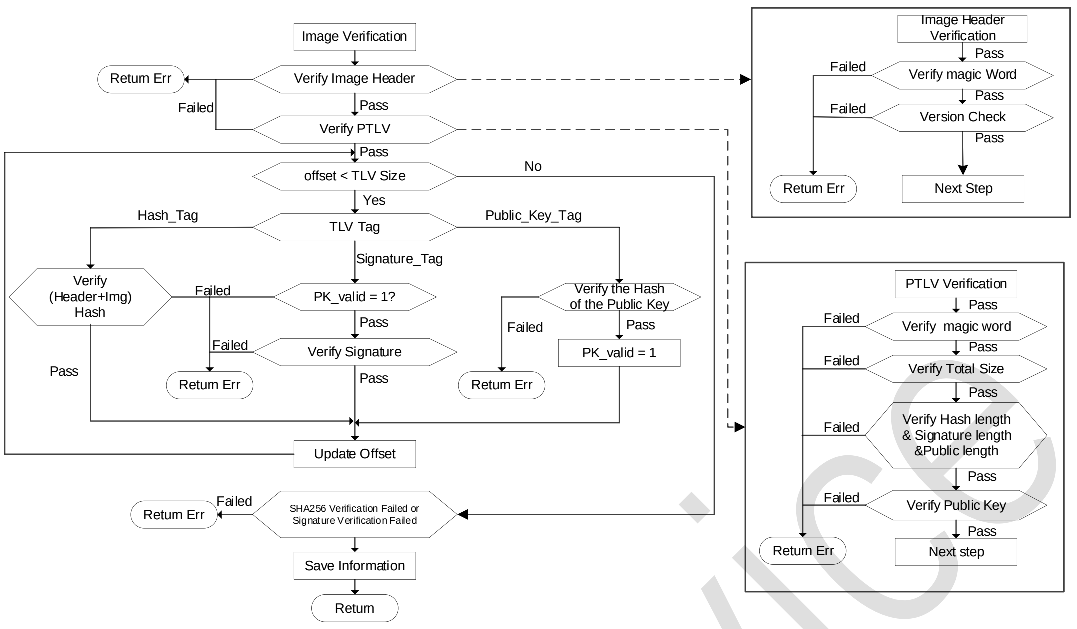
Figure 4-1. IBL Project Code File Structure



4.2 Image Authentication

The firmware signature verification process is shown in ***Figure 4-2. Signature Verification Process***. The verification process begins by verifying the correctness of the image header, checking the Magic word and version number respectively. Then, TLV verification is performed, checking the Magic word, Total Size, hash/signature/public key length, and public key. Based on the offset value set by the system, it is determined whether each part as been verified. Once completed, the information is saved and the next step is proceeded with.

Figure 4-2. **Signature Verification Process**



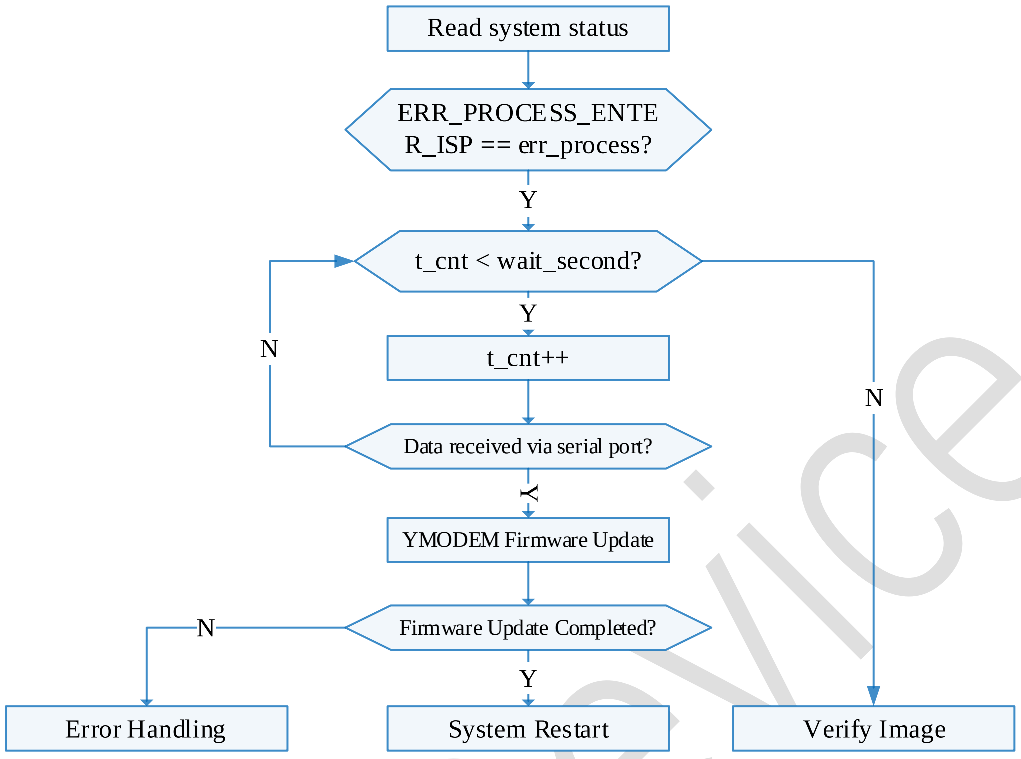
4.3 IBL Updates MBL

In IBL, a YMODEM protocol implemented through a serial port is provided to update the firmware of MBL, and its implementation process is shown in ***Figure 4-3. IBL Updates MBL***.

1. First, it will determine whether the error handling status in the system status is "ERR_PROCESS_ENTER_ISP". If so, it indicates that updating the MBL firmware through the serial port is allowed;
2. In wait_second, check if there is any data input from the serial port; if there is, proceed with the YMODEM protocol firmware upgrade; if not, move on to the step of verifying the image;
3. If the firmware update is successful, the system will restart; if the update fails, the system will enter an error handling state.

Note:
The error handling status can be set and retrieved through the IBL APL: ibl_API ibl_sys_status_set and ibl_sys_status_get.

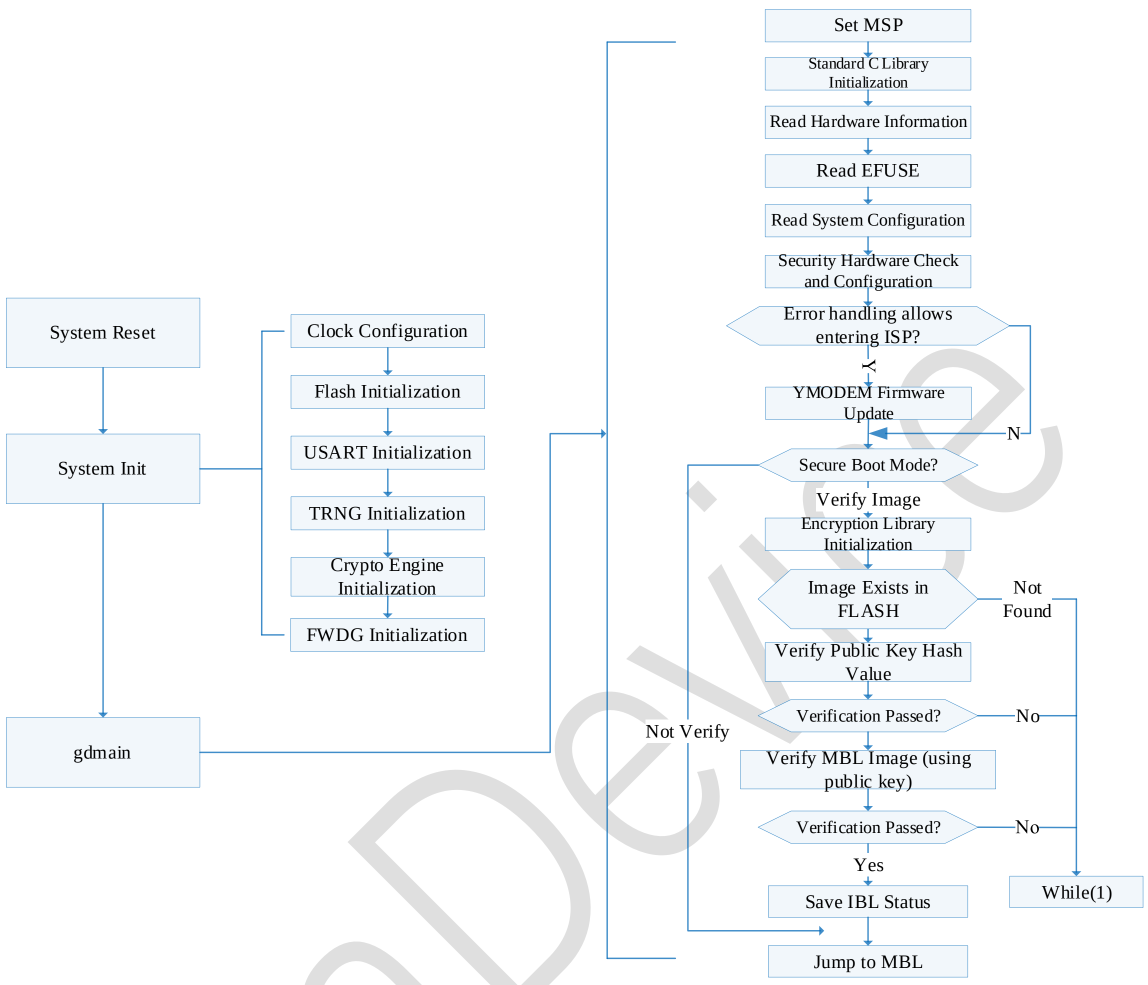
Figure 4-3. IBL Updates MBL



4.4 The Boot Process of IBL

The boot process of IBL, as shown in *Figure 4-4. IBL Boot Process*, configures the corresponding peripherals after system reset for the subsequent signature authentication process. Then it determines whether firmware update is needed. If an update is required, it completes the firmware update and restarts. After a period of delay, it starts to verify the firmware. If the verification passes, it jumps to MBL; otherwise, it enters an infinite loop and waits for the next reset.

Figure 4-4. IBL Boot Process



4.5 Exported Libraries and API Interfaces of IBL

In order to improve the usage rate of code, IBL will export the adapted mbedtls encryption and decryption library and some API interfaces, so that these exported functional libraries can be used in the MBL and APP projects. The exported library of IBL is mainly divided into two parts: the encryption and decryption library of mbedtls and the API interface. The exported library of mbedtls mainly includes common encryption and decryption interfaces, which are not described in detail here. The API interfaces exported by IBL can refer to ***Table 4-1. API Interface***.

Table 4-1. API Interface

Function Name	Description
ibl_log_uart_init	serial port initialization
ibl_printf	formatted serial port printing
ibl_uart_putc	serial port character printing
ibl_trace_ex	serial port trace printing
ibl_rand	random number acquisition
ibl_set_mutex_func	mutex addition

Function Name	Description
ibl_memset	string assignment
ibl_memcmp	string copy
ibl_strlen	string length calculation
ibl_strncmp	string comparison
ibl_uart_rx_to	serial port reception handling
ibl_strtoul	string to decimal conversion
ibl_cal_checksum	checksum calculation
ibl_img_verify_sign	signature verification
ibl_img_verify_hash	hash value verification
ibl_img_verify_hdr	image header verification
ibl_img_verify_pkhash	public key hash verification
ibl_img_validate	image verification
ibl_sys_setting_get	get system configuration
ibl_sys_status_set	set system status
ibl_sys_status_get	set system status
ibl_sys_set_trace_level	set system trace level
ibl_sys_set_err_process	set system error handling
ibl_sys_set_img_flag	set image flag (not enabled)
ibl_sys_reset_img_flag	reset image flag (not enabled)
ibl_sys_set_running_img	set running image (not enabled)
ibl_sys_set_fw_ver	set firmware version (not enabled)
ibl_sys_set_pk_ver	set public key version
ibl_sys_set_trng_seed	set random number seed
ibl_jump_to_img	jump to execute image
ibl_ymodem_download_check	Ymodem update check
ibl_is_valid_flash_offset	FLASH offset legitimacy judgment
ibl_is_valid_flash_addr	FLASH address legitimacy judgment
ibl_flash_total_size	get total size of FLASH
ibl_flash_erase_size	get erase size of FLASH
ibl_flash_init	FLASH Initialization
ibl_flash_read	read FLASH
ibl_flash_write	write FLASH
ibl_flash_write_fast	fast Write FLASH (not enabled)
ibl_flash_erase	erase FLASH
ibl_fmc_unlock	unlock FLASH
ibl_fmc_lock	lock FLASH
ibl_fmc_flag_clear	clear FLASH flags
ibl_fmc_page_erase	erase FLASH page

Function Name	Description
ibl_fmc_mass_erase	erase entire FLASH
ibl_fmc_word_program	flash word programming
ibl_fwdg_reload	FWDG feed dog

4.6 IBL Status information

Before IBL completes the boot process and jumps to MBL, some status information is handed over to MBL through SRAM. This status information is called the initial boot status. In MBL, the initial boot status information can be used to obtain the required boot information, such as the MBL public key. ***Table 4-2. Initial Boot State Information*** shows the content of the initial boot status information.

Table 4-2. Initial Boot State Information

Information Name	Description
reset_flag	reset flag
boot_status	boot status
ibl_ver	IBL version
ibl_opt	IBL configuration information
impl_id	implemented ID (not enabled)
mbl_pk	MBL public key
mbl_info	MBL information (not enabled)

5 MBL Project

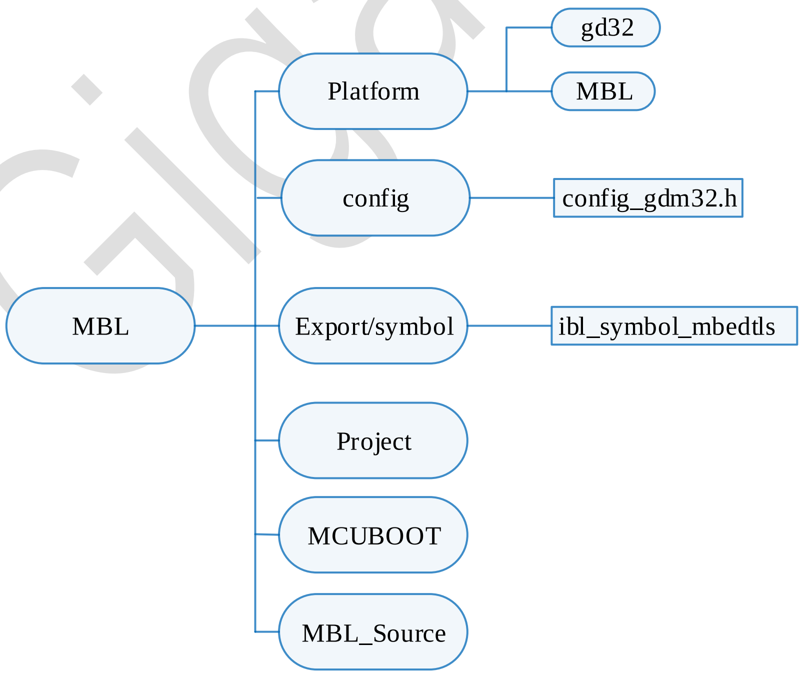
The MBL project is the boot code for the second phase of the SBSFU project. The MBL itself is verified and loaded by the IBL. Its main function is to complete the boot and loading of the application code. To meet the boot and loading functions of the application code, MCUBOOT is adapted in MBL, and the currently supported and verified features are as follows:

- Currently, MBL supports single-image dual-slot APP image management, where slot 0 (primary slot) is the execution area for APP code, and slot 1 (secondary slot, second slot, or scratch slot) is the update area for the APP；
- Support YMODEM protocol via serial port to update the APP code to the update area;
- Support verification of the integrity and authenticity of the APP code before updating and executing the APP code;
- Support the exchange and rollback function of the dual-slot app code, and support power-off recovery during the exchange period;
- Support version management of app code.

5.1 MBL Project File Structure

The file structure of the MBL engineering code is shown in ***Figure 5-1. MBL Engineering Code File Structure***. It mainly includes the MCUBOOT code, IBL API, and the exported library of IBL mbedtls to achieve the management of updating, verification, and execution of the APP firmware.

Figure 5-1. MBL Engineering Code File Structure



5.2 APP Image Exchange Type

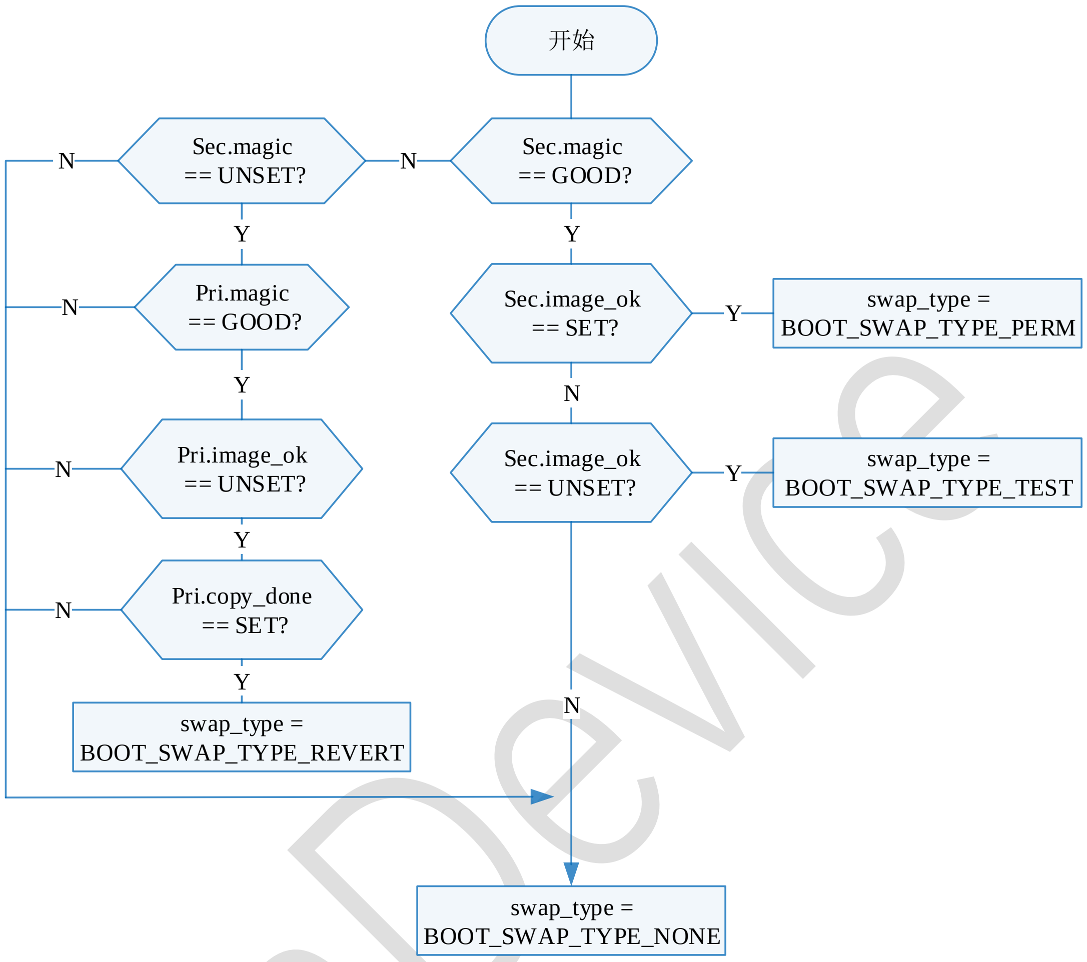
Before MBL exchanges the APP firmware, it first determines the type of firmware exchange based on the trailer information in the APP slot. The content of the trailer information can refer to the layout of the FLASH.

Table 5-1. trailer information and exchange type

Information Name	Value	Description
swap type	BOOT_SWAP_TYPE_NONE	attempt to boot slot0 content
	BOOT_SWAP_TYPE_TEST	switch to the slot1, if not confirmed, restore on next boot.
	BOOT_SWAP_TYPE_PERM	permanently switch to the slot1
	BOOT_SWAP_TYPE_REVERT	switch back to the slot1 after confirmation, change this status to none switch failed
	BOOT_SWAP_TYPE_FAIL	switch failed
	BOOT_SWAP_TYPE_PANIC	encountered an unrecoverable error during switch
copy done	BOOT_FLAG_SET	flag is set
	BOOT_FLAG_BAD	flag is illegal
	BOOT_FLAG_UNSET	flag is not set
	BOOT_FLAG_ANY	flag is irrelevant (used only for control)
image ok	BOOT_FLAG_SET	flag is set
	BOOT_FLAG_BAD	flag is illegal
	BOOT_FLAG_UNSET	flag is not set
	BOOT_FLAG_ANY	flag is irrelevant (used only for control)
magic	BOOT_MAGIC_GOOD	magic is good
	BOOT_MAGIC_BAD	magic is illegal
	BOOT_MAGIC_UNSET	magic is not set
	BOOT_MAGIC_ANY	magic is irrelevant (used only for control)
	BOOT_MAGIC_NOTGOOD	magic is abnormal (used only for control)

The trailer information and the values descriptions of the exchange types can refer to ***Table 5-1. trailer information and exchange type***. The trailer information and exchange types. ***Figure 5-2. Confirmation of Exchange Types*** shows the process of confirming the exchange type. Among them, the "BOOT_SWAP_TYPE_TEST" exchange type, if the exchange is completed without confirmation (setting the image OK information after the exchange to BOOT_FLAG_SET), after the next restart, a "BOOT_SWAP_TYPE_REVERT" exchange will be performed, swapping back to the firmware in the secondary slot.

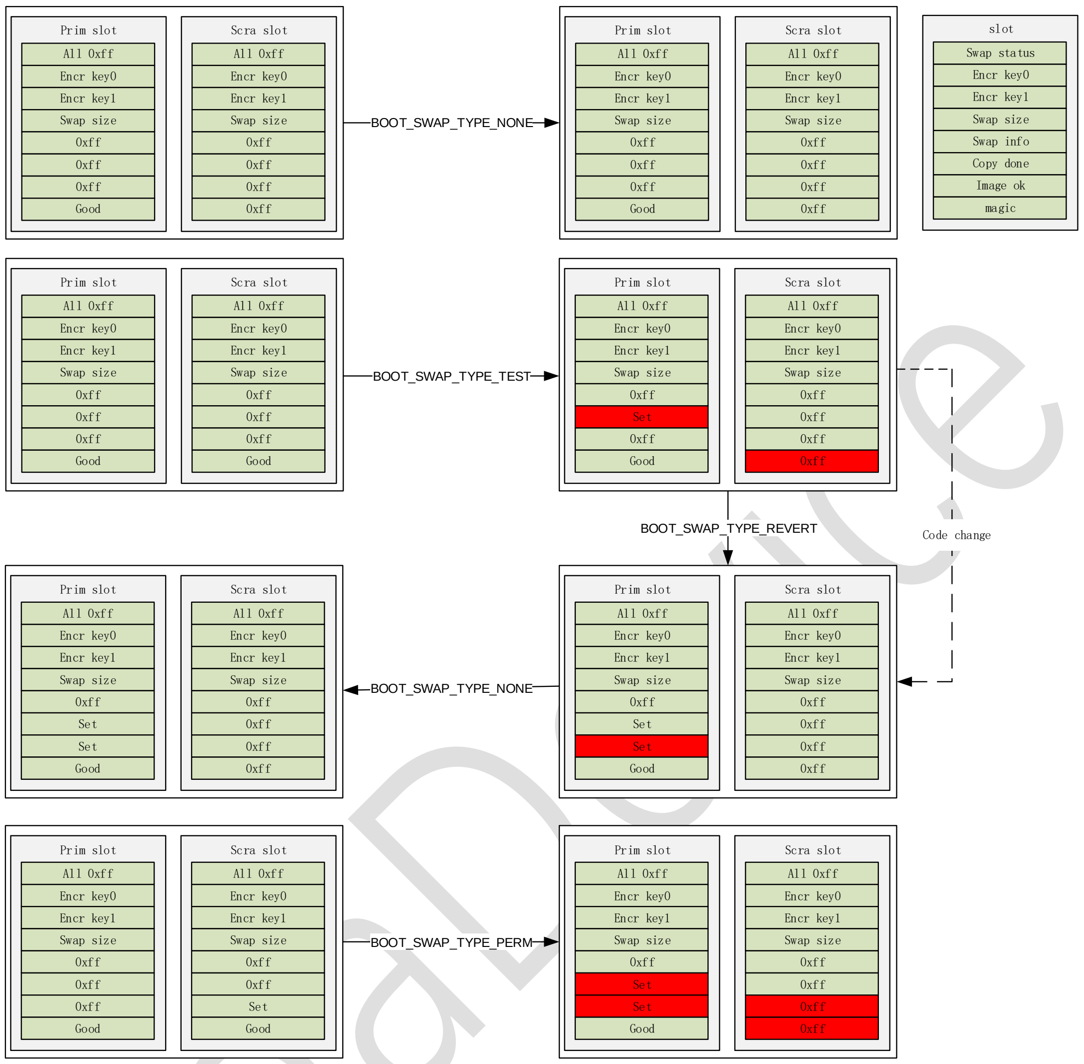
Figure 5-2. Confirmation of Exchange Types



5.3 Change of Trailer Information

Different types of swaps will lead to different changes in the trailer information of the slot. The basic types of swaps and the changes in trailer information can refer to ***Figure 5-3. Modification of trailer information***. The "BOOT_SWAP_TYPE_NONE" swap will not cause any changes in the trailer information. After the "BOOT_SWAP_TYPE_TEST" swap, the image ok can be manually set, so that the next reset will not produce a "BOOT_SWAP_TYPE_REVERT" swap. The "BOOT_SWAP_TYPE_PERM" swap is permanent and will not undergo a rollback swap.

Figure 5-3. Modification of trailer information



5.4 MBL Boot Process

The overall startup process of MBL is shown in ***Initially, after the system*** boots up, since the internal oscillator is no longer utilized, the system clock and certain peripherals (such as the serial port) will be reinitialized;

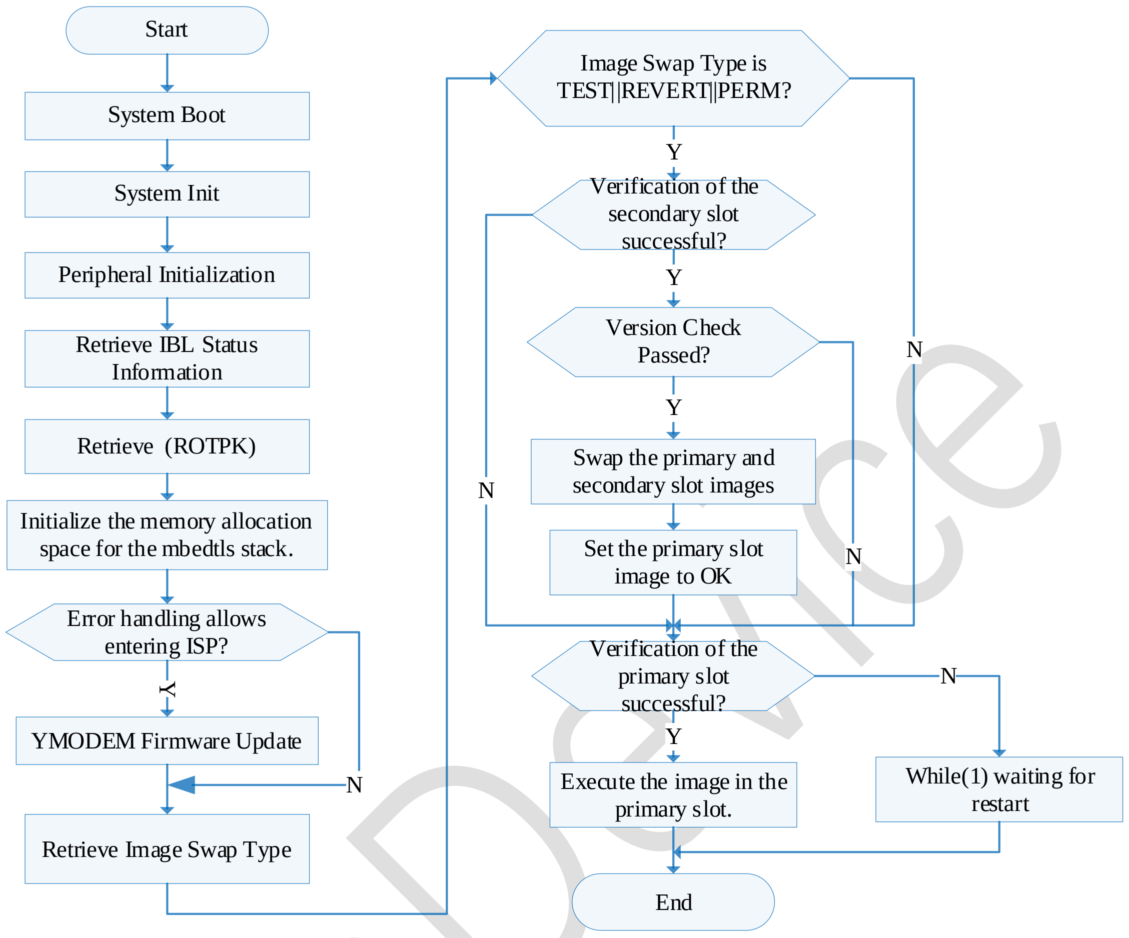
1. Subsequently, the status information of the IBL is obtained, and the ROTPK for firmware signature verification is extracted from the IBL's status information;
2. Then, stack space is allocated for mbedTLS, and once this is done, the mbedTLS library exported by the IBL can be utilized within the MBL project;
3. Following this, it is determined whether a firmware update is necessary. If an update is required, the firmware is updated and the system is rebooted. The firmware update process is similar to how the IBL updates the MBL, but the difference lies in the fact that the MBL updates the APP firmware and places it into the slot1;

4. Next, the type of image swap is retrieved and assessed from the tail information of both the slot0 and slot1;
5. If the swap type indicates that an image swap is necessary, proceed to step 7; otherwise, go to step 9;
6. Verify the slot1 and check the version information; if the verification fails, proceed to step 9;
7. The slot1 is valid, the images of the slot0 and slot1 are swapped, and the valid flag for the image in the slot 0 is set;
8. Verify the image in the primary slot; if the verification is successful, attempt to execute the primary slot. If the verification fails, enter a loop and wait for the system to reboot.

Figure 5-4. MBL Boot Process.

9. Initially, after the system boots up, since the internal oscillator is no longer utilized, the system clock and certain peripherals (such as the serial port) will be reinitialized;
10. Subsequently, the status information of the IBL is obtained, and the ROTPK for firmware signature verification is extracted from the IBL's status information;
11. Then, stack space is allocated for mbedTLS, and once this is done, the mbedTLS library exported by the IBL can be utilized within the MBL project;
12. Following this, it is determined whether a firmware update is necessary. If an update is required, the firmware is updated and the system is rebooted. The firmware update process is similar to how the IBL updates the MBL, but the difference lies in the fact that the MBL updates the APP firmware and places it into the slot1;
13. Next, the type of image swap is retrieved and assessed from the tail information of both the slot0 and slot1;
14. If the swap type indicates that an image swap is necessary, proceed to step 7; otherwise, go to step 9;
15. Verify the slot1 and check the version information; if the verification fails, proceed to step 9;
16. The slot1 is valid, the images of the slot0 and slot1 are swapped, and the valid flag for the image in the slot 0 is set;
17. Verify the image in the primary slot; if the verification is successful, attempt to execute the primary slot. If the verification fails, enter a loop and wait for the system to reboot.

Figure 5-4. MBL Boot Process


Start
System Boot
System Init
Peripheral Initialization
Retrieve IBL Status Information
Retrieve (ROTPK)
Initialize the memory allocation space for the mbedtls stack.
Error handling allows entering ISP?
Y
N
YMODEM Firmware Update
Retrieve Image Swap Type
Image Swap Type is TEST||REVERT||PERM?
Y
N
Verification of the secondary slot successful?
Y
N
Version Check Passed?
Y
N
Swap the primary and secondary slot images
Set the primary slot image to OK
Verification of the primary slot successful?
Y
N
Execute the image in the primary slot.
While(1) waiting for restart
End

5.5 Exported Library of MBL

As previously mentioned, after completing the exchange and firmware update, the APP code will set the magic number of the tail information and the image validity flag. For this reason, MBL will export some interface functions for the APP to call. The exported library interface of MBL can be referred to ***Table 5-2. MBL Export Library Interface***.

Table 5-2. MBL Export Library Interface

Function Name	Description
flash_area_open	open FLASH area
boot_write_magic	write magic number
boot_write_image_ok	write image validity

6 APP Project

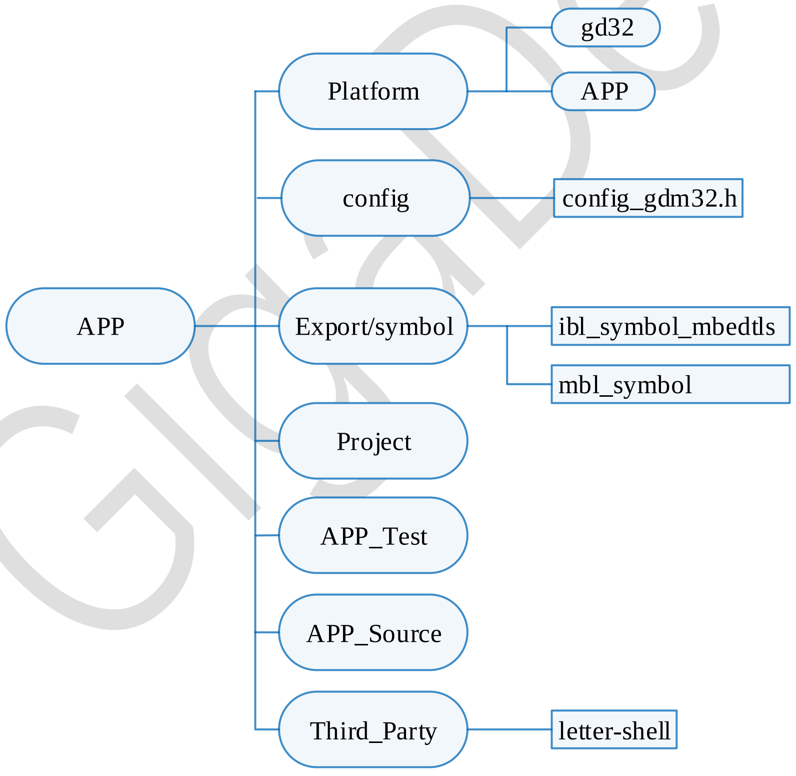
The APP project is the final code in the SBSFU project that completes the boot and execution after passing integrity and authenticity verification, which can be considered as trusted firmware. Currently, the implementation of APP is relatively simple, mainly realizing the following functions:

- Integrated with letter-shell as a serial port interrupt tool;
- Supports updating the firmware to the slot 1 via the APP in slot 0;
- Support API for IBL export and export the mbedTLS library;
- Support partial testing features.

6.1 APP Project Code File Structure

The file structure of the APP project code is shown in ***Figure 6-1. APP project code file structure***. It mainly includes the letter-shell code, test code, IBL API, and the exported library of IBL mbedtls to achieve the firmware update of the APP.

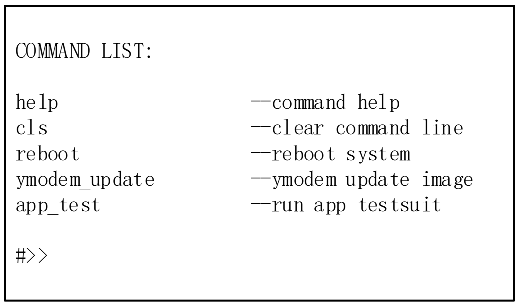
Figure 6-1. APP project code file structure



6.2 APP shell Commands

In the serial terminal, you can list the currently supported commands through the help command. ***Figure 6-2. APP shell command*** shows all the currently supported commands.

Figure 6-2. APP shell command



```
COMMAND LIST:

help                  —command help
cls                   —clear command line
reboot                —reboot system
ymodem_update         —ymodem update image
app_test              —run app testsuit

#>>
```

For an introduction to shell commands, please refer to ***Table 6-1. Shell Commands and Descriptions***.

Table 6-1. Shell Commands and Descriptions

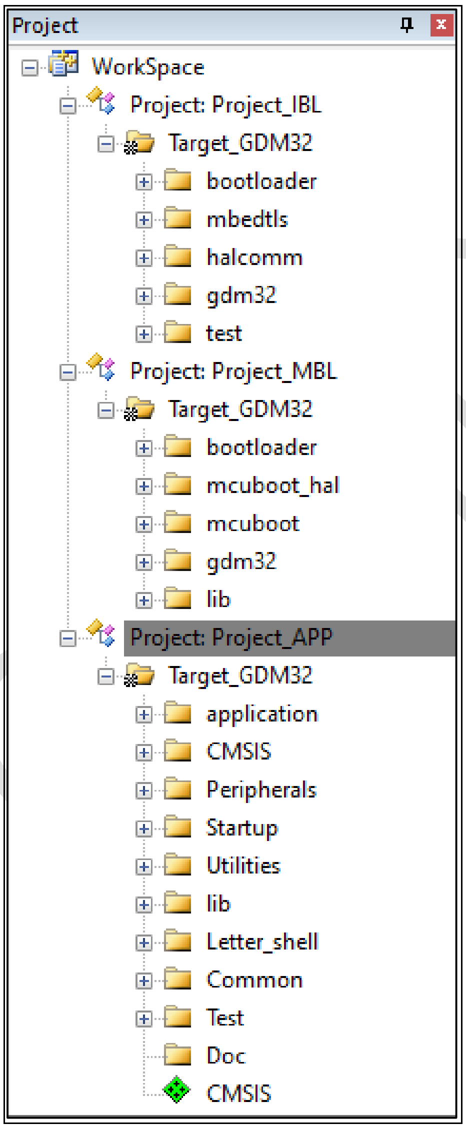
Command Name	Description
help	help command, capable of listing all supported commands
cls	clear screen command, clears the terminal screen
reboot	software reset and reboot
ymodem_update	Ymodem protocol to download APP firmware to the auxiliary slot
app_test	Run test code (currently only used for verifying IBL's mbed TLS exported libraries)

7 Implementation of SBSFU Project on GD32F5XX

7.1 Project Compilation

The SBSFU project for GD32F5xx is managed by the Keil IDE and can be opened by the SBSFU project group through "Project/GD32F5xx/MDK-ARM/SBSFU.uvmpw". As shown in ***Figure 7.1. SBSFU Project Group***, the SBSFU project group contains three independent projects: IBL, MBL, and APP. By using the "Batch Build" button shown in the figure, all projects can be compiled at once. After the compilation is completed, images that can be downloaded to the chip will be generated in the "scripts/images" directory.

Figure 7.1. SBSFU Project Group



```
Project
WorkSpace
  Project: Project_IBL
    Target_GDM32
      bootloader
      mbedtls
      halcomm
      gdm32
      test
  Project: Project_MBL
    Target_GDM32
      bootloader
      mcuboot_hal
      mcuboot
      gdm32
      lib
  Project: Project_APP
    Target_GDM32
      application
      CMSIS
      Peripherals
      Startup
      Utilities
      lib
      Letter_shell
      Common
      Test
      Doc
      CMSIS
```

The generated image can refer to ***Table 7-1. Generated Image***. The generated image. Where mbl-sys.hex will be copied to "MBL/mbl/Objects/mbl.hex", and then can be directly downloaded using the MBL project. The app-sign.hex and app-sign1.hex files will be copied to "APP/app/Objects/app.hex" and "APP/app/Objects/app1.hex" respectively, and then the file can be downloaded using the APP project.

Note:
"mbl-sys.hex" is the file that contains system configuration information, and it must be downloaded with the configuration information included during the initial download.

Table 7-1. Generated Image

Image Name	Description	Usage
ibl.bin	IBL image in bin format	directly usable for downloading
ibl.hex	IBL image in hex format	directly usable for downloading
mbl.bin	MBL image in bin format with image header and signature information added	temporary file
mbl-sign.bin	MBL image in bin format with image header and signature information added	for MBL update
mbl-sys.bin	MBL image in bin format with image header, signature information, and system design information	for the initial download file
mbl-sys.hex	MBL image in hex format with image header, signature information, and system design information added	for the initial download file
app.bin	original APP Image in bin format	temporary file
app-sign.bin	APP image in bin format with image header and signature information added	for App update
app-sign.hex	APP image in hex format with image header and signature information added 1	directly usable for downloading
app-sign1.hex	APP image in hex format with image header and signature information added 2	directly usable for downloading

7.2 Project Download and Verification

- Download the IBL project

The IBL code is the first code to be executed, so there is no need for special treatment, so it can be downloaded and debugged just like a regular project.

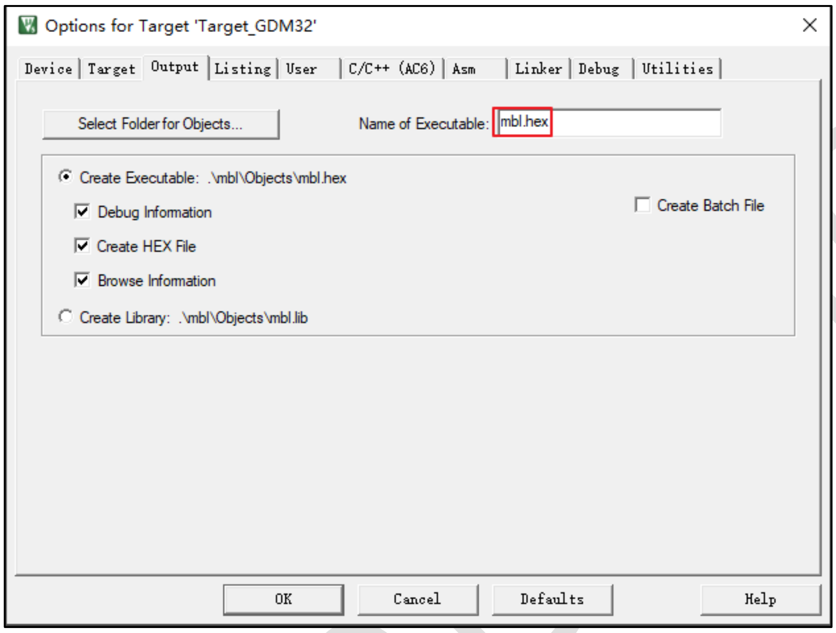
- Download the MBL project

After the MBL project is compiled, a tool program will add the firmware header, firmware signature, and system setting information to the MBL executable code, and then generate "mbl-sys.hex" in the "scripts/images" directory. Subsequently, "mbl-sys.hex" is used to replace the original "mbl.hex" in the project. Both of these files can be downloaded. Here, you can directly use Keil to download the "mbl.hex" file to the chip. For reference, see ***Figure 7-1. MBL project download*** for MBL project downloading, the modified executable file name is changed to "mbl.hex". You don't need to compile the project directly, just download it.

Notes:

- When downloading the code as described above, if you wish to recompile, you need to first modify the executable file name to “mbl”;
- The “mbl-sys.hex” image can be used of the initial download, while the updated firmware should use “mbl-sign.bin”

Figure 7-1. MBL project download



Options for Target 'Target_GDM32'

Device | Target | Output | Listing | User | C/C++ (AC6) | Asm | Linker | Debug | Utilities

Select Folder for Objects... Name of Executable: mbl.hex

Create Executable: .\mbl\Objects\mbl.hex
[✓] Debug Information
[✓] Create HEX File
[✓] Browse Information
Create Library: .\mbl\Objects\mbl.lib
[ ] Create Batch File

OK Cancel Defaults Help

- Download The APP Project

The download of the APP project can refer to the MBL project, where the app.hex can download the image to the main slot, and the app1.hex can download the image to the auxiliary slot.

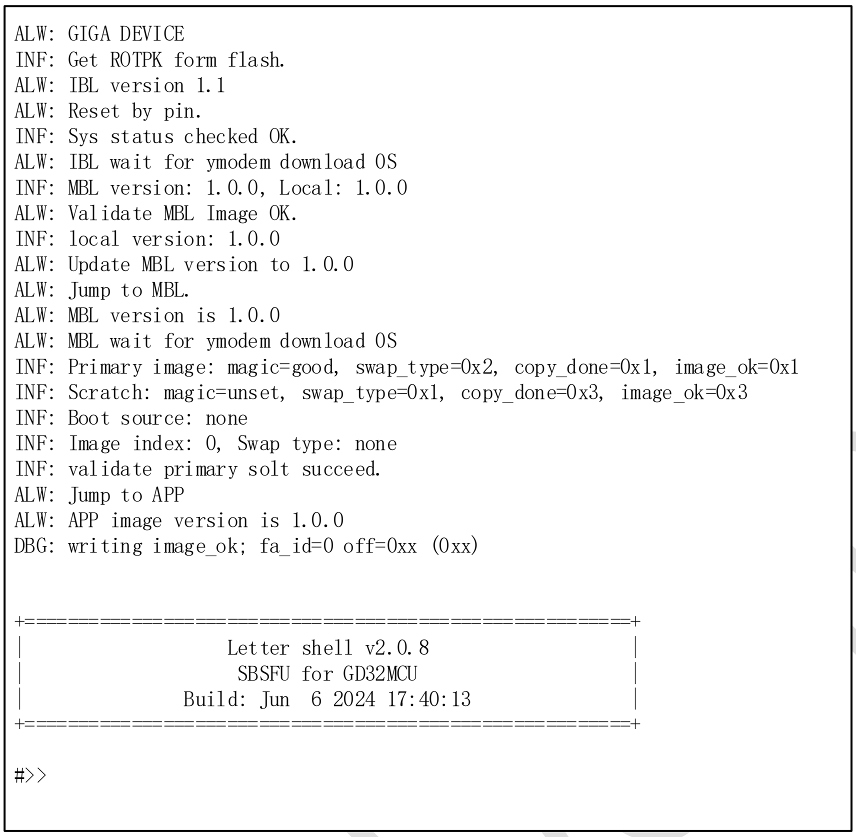
- SBSFU Project Verification

After all projects are correctly downloaded, If the serial port output is as shown in ***Figure 7-2. SBSFU Project Serial Project Output***, it indicates that the project is running correctly.

Note:

The secure boot project defaults to using USART0 with a baud rate of 115200

Figure 7-2. SBSFU Project Serial Project Output



```
ALW: GIGA DEVICE
INF: Get ROTPK form flash.
ALW: IBL version 1.1
ALW: Reset by pin.
INF: Sys status checked OK.
ALW: IBL wait for ymodem download 0S
INF: MBL version: 1.0.0, Local: 1.0.0
ALW: Validate MBL Image OK.
INF: local version: 1.0.0
ALW: Update MBL version to 1.0.0
ALW: Jump to MBL.
ALW: MBL version is 1.0.0
ALW: MBL wait for ymodem download 0S
INF: Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
INF: Scratch: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
INF: Boot source: none
INF: Image index: 0, Swap type: none
INF: validate primary solt succeed.
ALW: Jump to APP
ALW: APP image version is 1.0.0
DBG: writing image_ok; fa_id=0 off=0xx (0xx)


+===========================================+
|              Letter shell v2.0.8          |
|               SBSFU for GD32MCU           |
|           Build: Jun  6 2024 17:40:13     |
+===========================================+

#>>
```

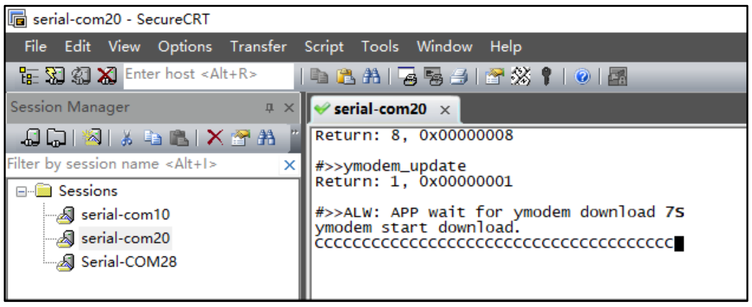
7.3 Firmware Update

The implementation of firmware updates can refer to the implementation of firmware updates. This section will mainly introduce how to update the MBL and APP firmware, and the terminal software uses SecureCRT 7.3.3.

7.4 APP Firmware Update

The firmware update of the APP can follow these steps, as shown in ***Figure 7-3. Terminal sends update command*** .To send an update command, enter the "ymodem_update" command in the terminal. After that, the device will wait for the update confirmation (the user can input any character in the terminal, such as pressing the space key to confirm the update). Before the countdown ends, if the device confirms the need for an update, it will keep outputting 'C' to wait for the update.

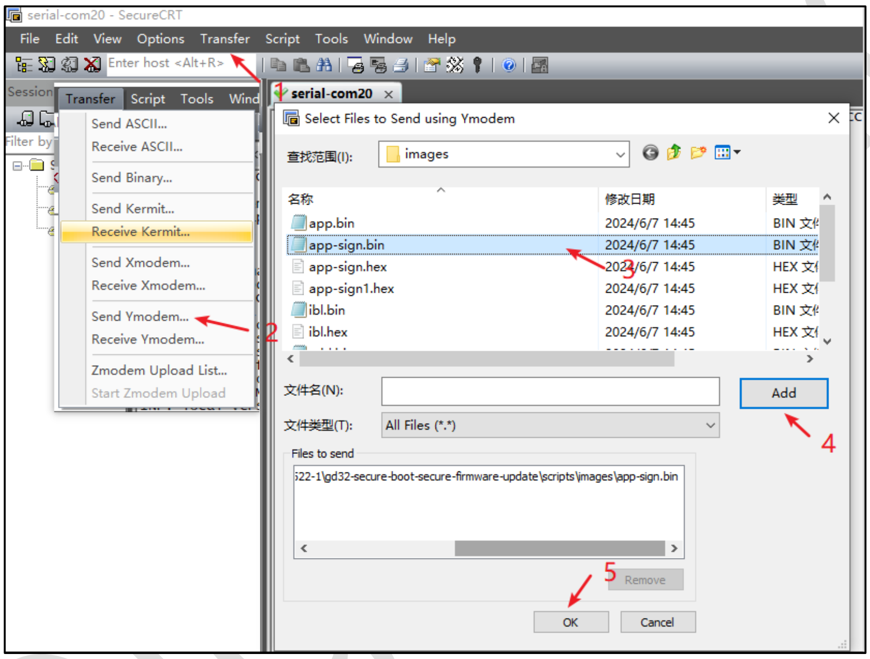
Figure 7-3. Terminal sends update command



```
Return: 8, 0x00000008

#>>ymodem_update
Return: 1, 0x00000001

#>>ALW: APP wait for ymodem download 7s
ymodem start download.
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

After the device enters the waiting for update state, refer to ***Figure 7-4. Terminal sends firmware update***. The terminal sends the firmware update from the terminal to the device. After the update is successful, the device will be reset and restarted by the software.

Figure 7-4. Terminal sends firmware update



The update process and the log printing after the device restart can be referred to as ***Figure 7-5. Terminal Update Log Printing***. From the log, it can be found that after the restart, the MBL program will perform a test exchange, swapping the image in the secondary slot to the primary slot, completing the update of the APP firmware.

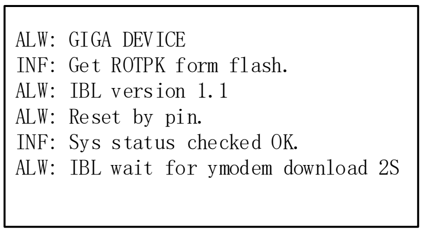
Figure 7-5. Terminal Update Log Printing

```
Starting ymodem transfer.  Press Ctrl+C to cancel.
Transferring app-sign.bin...
  100%       11 KB       11 KB/sec     00:00:01        0 Errors


DBG: writing magic; fa_id=1 off=0xx (0xx)
ALW: Ymodem update success, system reboot.
ALW: GIGA DEVICE
INF: Get ROTPK form flash.
ALW: IBL version 1.1
ALW: Reset by sw.
INF: Sys status checked OK.
ALW: IBL wait for ymodem download OS
INF: MBL version: 1.0.0, Local: 1.0.0
ALW: Validate MBL Image OK.
INF: local version: 1.0.0
ALW: Update MBL version to 1.0.0
ALW: Jump to MBL.
ALW: MBL version is 1.0.0
ALW: MBL wait for ymodem download OS
INF: Primary image: magic=good, swap_type=0x2, copy_done=0x1, image_ok=0x1
INF: Scratch: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
INF: Boot source: none
INF: Image index: 0, Swap type: test
INF: Starting swap using scratch algorithm.
DBG: erasing scratch area
DBG: initializing status; fa_id=2
DBG: writing swap_info; fa_id=2 off=0xx (0xx), swap_type=0x1fd8 image_num=0x83fd8
DBG: writing swap_size; fa_id=2 off=0xx (0xx)
DBG: writing magic; fa_id=2 off=0xx (0xx)
DBG: erasing trailer; fa_id=0
DBG: initializing status; fa_id=0
DBG: writing swap_info; fa_id=0 off=0xx (0xx), swap_type=0xfffd8 image_num=0x19ffd8
DBG: writing swap_size; fa_id=0 off=0xx (0xx)
DBG: writing magic; fa_id=0 off=0xx (0xx)
DBG: erasing trailer; fa_id=1
DBG: erasing scratch area
DBG: writing copy_done; fa_id=0 off=0xx (0xx)
INF: validate primary solt succeed.
ALW: Jump to APP
ALW: APP image version is 1.0.0
DBG: writing image_ok; fa_id=0 off=0xx (0xx)


+===========================================+
|              Letter shell v2.0.8          |
|              SBSFU for GD32MCU            |
|          Build: Jun  7 2024 14:45:22      |
+===========================================+

#>>
```

7.5 MBL Firmware Update

The MBL firmware can be updated through IBL. As shown in the first countdown of the SBSFU project startup in ***Figure 7-6. MBL Update***, the MBL update can be confirmed (the confirmation process is similar to the APP update). After that, the updated firmware can be sent to the device through terminal software to complete the firmware update. After a successful update, the device will also reset and restart.

Figure 7-6. MBL Update



```
ALW: GIGA DEVICE
INF: Get ROTPK form flash.
ALW: IBL version 1.1
ALW: Reset by pin.
INF: Sys status checked OK.
ALW: IBL wait for ymodem download 2S
```

7.5.1 MBL Updates the APP

APP firmware can also be updated through MBL, and the update confirmation (similar to the APP update process) can be made during the second countdown when the SBSFU project is launched, and the firmware update can be completed.

7.6 Implementation of Flash layout

The overall FLASH layout of the SBSFU project can refer to ***Figure 3-3. FLASH Layout*** where the FLASH layouts of IBL, MBL, and APP can be configured through the files "Platform/IBL/ibl_region.h", "Platform/MBL/mbl_region.h", and "Platform/APP/app_region.h" respectively. The main configuration macros used can be referred to in ***Table 7-2. FLASH Layout Configuration Macros.***

Table 7-2. FLASH Layout Configuration Macros

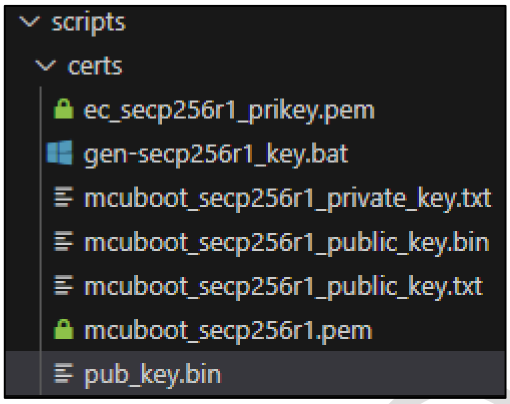
Command Name	Description	Default Value
	IBL	
FLASH_BASE_IBL	base address of IBL	0x08000000
IBL_CODE_START	starting address of IBL Code	0x08000000
IBL_CODE_SIZE	size of IBL code	0x4000
FLASH_BASE_LIB	base address of IBL library	0x08004000
FLASH_LIB_START	starting address of IBL library	0x08004000
FLASH_LIB_SIZE	size of IBL library	0x50000
FLASH_OFFSET_SYS_SETTING	offset address of system setting data	0x60000
	MBL	
MBL_BASE_ADDRESS	base address of MBL	0x08061200
MBL_CODE_START	starting address of MBL code	0x08061200
MBL_CODE_SIZE	size of MBL code	0x1E00
	APP	
APP_CODE_START	starting address of APP code	0x080A0200
APP_CODE_SIZE	size of APP code	0xFFE00

7.7 SBSFU Project Asymmetric Key Configuration

7.7.1 Generation of New Keys

Currently, the SBSFU project uses SECP256R1 for signature and verification. In the "scripts/certs" directory, there is a "gen-secp256r1_key.bat" script provided, which can be used to generate a SECP256R1 key pair. By double-clicking the BAT script, a SECP256R1 key pair will be generated as shown in ***Figure 7-7. Generate a SECP256R1 Key Pair***.

Figure 7-7. Generate a SECP256R1 Key Pair



```
scripts
  certs
    ec_secp256r1_prikey.pem
    gen-secp256r1_key.bat
    mcuboot_secp256r1_private_key.txt
    mcuboot_secp256r1_public_key.bin
    mcuboot_secp256r1_public_key.txt
    mcuboot_secp256r1.pem
    pub_key.bin
```

The files and functions in the "scripts/certs" directory can refer to ***Table 7-3. Description of Key and Certificate Files***.

Table 7-3. Description of Key and Certificate Files

Command Name	Description
ec_secp256r1_prikey.pem	default secp256r1 private key, the script for generating the APP and MBL images will use this file.
pub_key.bin	file saved in bin format of the default secp256r1 public key.
gen-secp256r1_key.bat	script for generating secp256r1 keys, running it will generate the following three files.
mcuboot_secp256r1_public_key.bin	the public key stored in hexadecimal format generated by gen-secp256r1_key.bat, which can be used for downloading to the OTP2 area
mcuboot_secp256r1.pem	the PEM format private key generated by gen -secp256r1_key.bat, which can be used to replace ec_secp256r1_prikey.pem.
mcuboot_secp256r1_private_key.txt	the private key stored in hexadecimal format generate by gen-secp256r1_key.bat, which can be used for signing
mcuboot_secp256r1_public_key.txt	the private key stored in hexadecimal format generate by gen-secp256r1_key.bat, which can be used for verification.

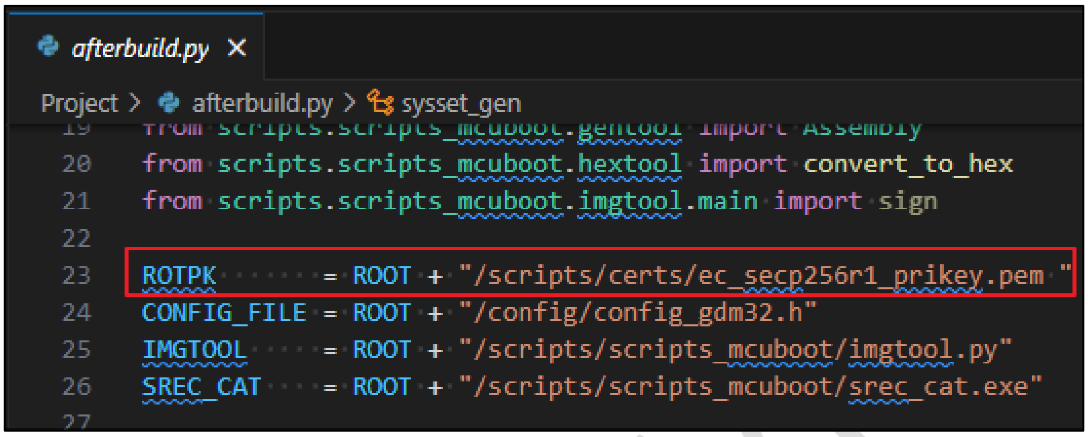
7.7.2 Adaptation of New Keys

If you need to adapt to a new key, you need to make modifications in the following two places:

- Private key adaptation for MBL and APP image generation script.

Generate the private key adaptation for the image generation script, taking MBL as an example, it needs to adapt to the position shown in ***Figure 7-8. Adapt Private Key***. The private key adaptation for the APP image generation script is similar to that of MBL.

Figure 7-8. Adapt Private Key

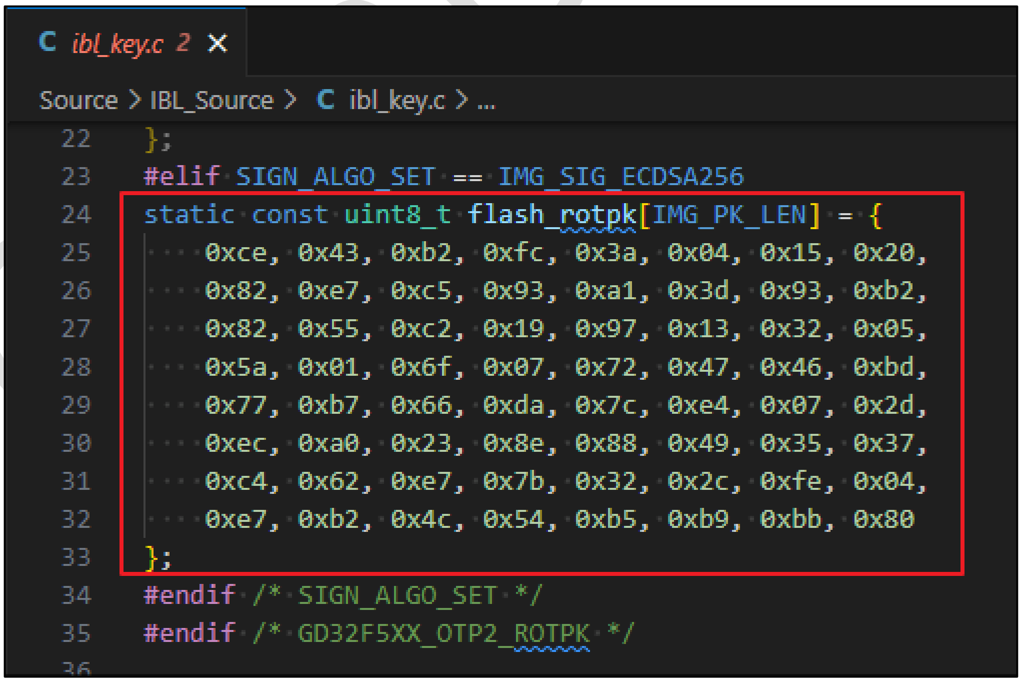


```
afterbuild.py ×
Project > afterbuild.py > sysset_gen
from scripts.scripts_mcuboot.gentool import Assembly
from scripts.scripts_mcuboot.hextool import convert_to_hex
from scripts.scripts_mcuboot.imgtool.main import sign

ROTPK       = ROOT + "/scripts/certs/ec_secp256r1_prikey.pem "
CONFIG_FILE = ROOT + "/config/config_gdm32.h"
IMGTOOL     = ROOT + "/scripts/scripts_mcuboot/imgtool.py"
SREC_CAT    = ROOT + "/scripts/scripts_mcuboot/srec_cat.exe"
```

- Public Key Adaptation in IBL Engineering

When adapting a public key, if you are using a public key stored in FLASH (which is the default storage location), you need to write the hexadecimal data of the public key to the location shown in ***Figure 7-9. Adapting public key*** for public key adaptation.

Figure 7-9. Adapting public key



```
ibl_key.c 2 ×
Source > IBL_Source > ibl_key.c > ...
};
#elif SIGN_ALGO_SET == IMG_SIG_ECDSA256
static const uint8_t flash_rotpk[IMG_PK_LEN] = {
    0xce, 0x43, 0xb2, 0xfc, 0x3a, 0x04, 0x15, 0x20,
    0x82, 0xe7, 0xc5, 0x93, 0xa1, 0x3d, 0x93, 0xb2,
    0x82, 0x55, 0xc2, 0x19, 0x97, 0x13, 0x32, 0x05,
    0x5a, 0x01, 0x6f, 0x07, 0x72, 0x47, 0x46, 0xbd,
    0x77, 0xb7, 0x66, 0xda, 0x7c, 0xe4, 0x07, 0x2d,
    0xec, 0xa0, 0x23, 0x8e, 0x88, 0x49, 0x35, 0x37,
    0xc4, 0x62, 0xe7, 0x7b, 0x32, 0x2c, 0xfe, 0x04,
    0xe7, 0xb2, 0x4c, 0x54, 0xb5, 0xb9, 0xbb, 0x80
};
#endif /* SIGN_ALGO_SET */
#endif /* GD32F5XX_OTP2_ROTPK */
```

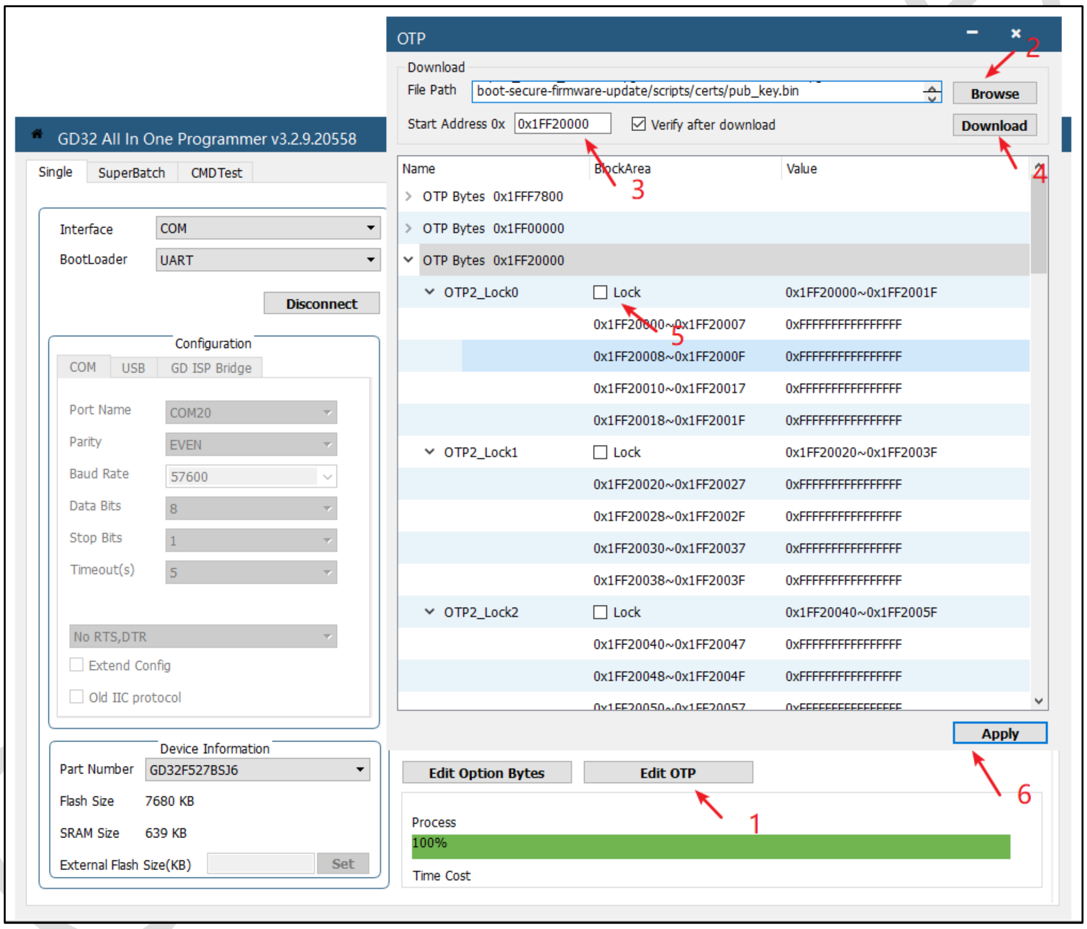
7.8 Adapt the public key to the OTP2 area

The OTP2 area of the GD32F5xx supports read and write locking configurations, making it very suitable for storing public keys with higher security requirements. To write the public key into the OTP2 area and use it normally, the following operations and modifications need to be made:

- Write the public key into the OTP2 area and set appropriate read and write locking.

Write the public key into the OTP2 area, you can use the "GD32AllInOneProgrammer.exe" software, and refer to *Figure 7-10. Write the public key into the OTP2 area*.

Figure 7-10. Write the public key into the OTP2 area



Interface: COM
BootLoader: UART

Port Name: COM20
Parity: EVEN
Baud Rate: 57600
Data Bits: 8
Stop Bits: 1
Timeout(s): 5
No RTS,DTR
☐ Extend Config
☐ Old IIC protocol

Part Number: GD32F527BSJ6
Flash Size: 7680 KB
SRAM Size: 639 KB
External Flash Size(KB):

File Path: boot-secure-firmware-update/scripts/certs/pub_key.bin
Start Address 0x: 0x1FF20000
☑ Verify after download

Name	BlockArea	Value
OTP Bytes 0x1FFF7800		
OTP Bytes 0x1FF00000		
OTP Bytes 0x1FF20000		
OTP2_Lock0	☐ Lock	0x1FF20000~0x1FF2001F
	0x1FF20000~0x1FF20007	0xFFFFFFFFFFFFFFFF
	0x1FF20008~0x1FF2000F	0xFFFFFFFFFFFFFFFF
	0x1FF20010~0x1FF20017	0xFFFFFFFFFFFFFFFF
	0x1FF20018~0x1FF2001F	0xFFFFFFFFFFFFFFFF
OTP2_Lock1	☐ Lock	0x1FF20020~0x1FF2003F
	0x1FF20020~0x1FF20027	0xFFFFFFFFFFFFFFFF
	0x1FF20028~0x1FF2002F	0xFFFFFFFFFFFFFFFF
	0x1FF20030~0x1FF20037	0xFFFFFFFFFFFFFFFF
	0x1FF20038~0x1FF2003F	0xFFFFFFFFFFFFFFFF
OTP2_Lock2	☐ Lock	0x1FF20040~0x1FF2005F
	0x1FF20040~0x1FF20047	0xFFFFFFFFFFFFFFFF
	0x1FF20048~0x1FF2004F	0xFFFFFFFFFFFFFFFF
	0x1FF20050~0x1FF20057	0xFFFFFFFFFFFFFFFF

Process
100%
Time Cost

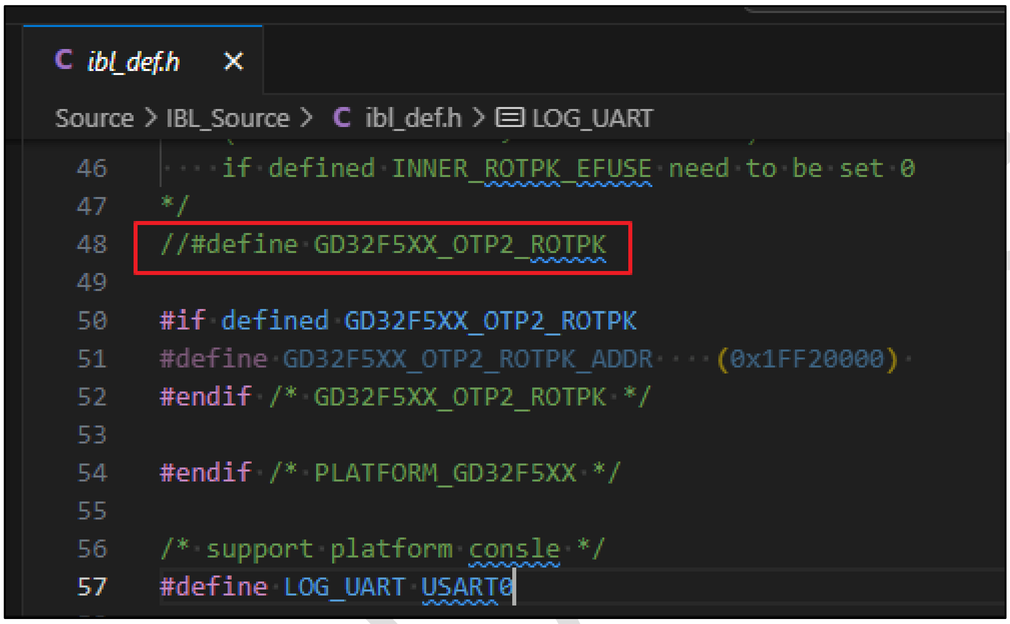
Notes:

- The GD32AllInOneProgrammer.exe Software requires launching from the Bootloader area;
- The current code defaults to using the first 64 bytes of the OTP2 area to store the public key;
- The OTP area only supports one-time programming, and the lock block bytes can only be programmed once from 0xFF to 0x00.

- Public key adaptation in the IBL project

In the IBL project, the macro "GD32F5XX_OTP2_ROTPK" is enabled as shown in ***Figure 7-11. IBL project uses OTP2 to store public keys***. The OTP2 area will be set to read-locked by the code after the IBL reads the key, so that the subsequent code can no longer read the OTP2 area, and MBL can only obtain the public key through the status information of IBL.

Figure 7-11. IBL project uses OTP2 to store public keys



```
    if defined INNER_ROTPK_EFUSE need to be set 0
*/
//#define GD32F5XX_OTP2_ROTPK

#if defined GD32F5XX_OTP2_ROTPK
#define GD32F5XX_OTP2_ROTPK_ADDR    (0x1FF20000)
#endif /* GD32F5XX_OTP2_ROTPK */

#endif /* PLATFORM_GD32F5XX */

/* support platform consle */
#define LOG_UART USART0
```

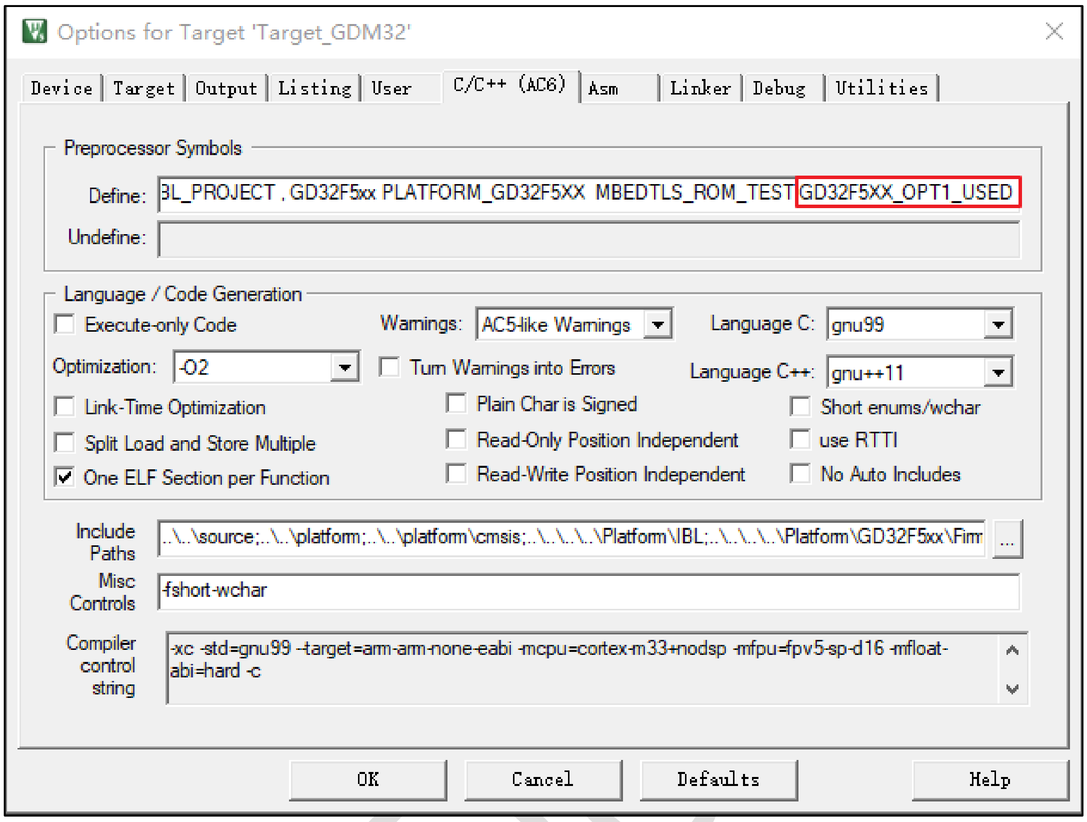
7.9 Adapt IBL to the OTP1 Area

The default placement of the IBL project in SBSFU is in the FLASH. However, on the GD32F5xx chip, there is a 128KB OTP1 area that supports read and write lock configurations, which is suitable for running the IBL project. The SBSFU project also supports configuring the IBL to run in the OTP1 area. The following modifications are needed when using it:

- Modify the IBL project.

In the IBL project, it is necessary to enable the macro "GD32F5XX_OPT1_USED", and there are two places that need to be modified, which can be referred to ***Figure 7-12. IBL is adapted to the OTP1 region***.

Figure 7-12. IBL is adapted to the OTP1 region



Options for Target 'Target_GDM32'

Device | Target | Output | Listing | User | C/C++ (AC6) | Asm | Linker | Debug | Utilities

Preprocessor Symbols

Define: BL_PROJECT , GD32F5xx PLATFORM_GD32F5XX MBEDTLS_ROM_TEST GD32F5XX_OPT1_USED

Undefine:

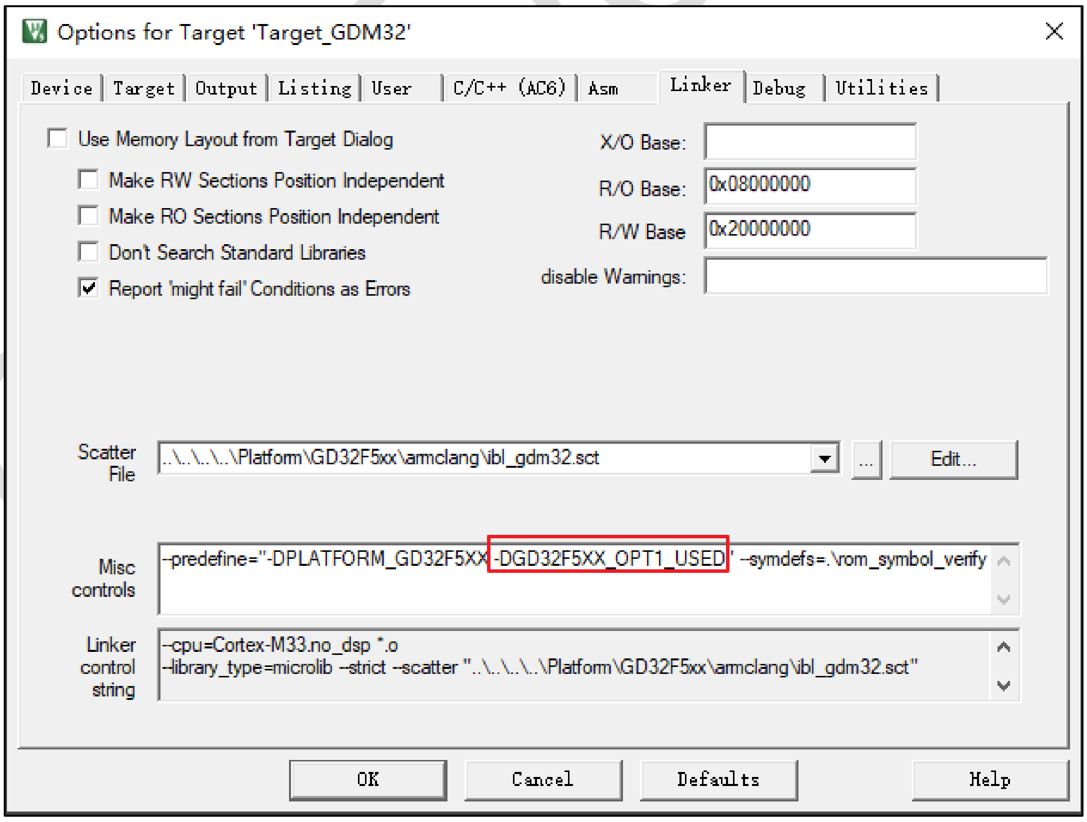
Language / Code Generation

- [ ] Execute-only Code
- Warnings: AC5-like Warnings
- Language C: gnu99
- Optimization: -O2
- [ ] Turn Warnings into Errors
- Language C++: gnu++11
- [ ] Link-Time Optimization
- [ ] Plain Char is Signed
- [ ] Short enums/wchar
- [ ] Split Load and Store Multiple
- [ ] Read-Only Position Independent
- [ ] use RTTI
- [✓] One ELF Section per Function
- [ ] Read-Write Position Independent
- [ ] No Auto Includes

Include Paths: ..\..\source;..\..\platform;..\..\platform\cmsis;..\..\..\..\Platform\IBL;..\..\..\..\Platform\GD32F5xx\Firm

Misc Controls: -fshort-wchar

Compiler control string: -xc -std=gnu99 -target=arm-arm-none-eabi -mcpu=cortex-m33+nodsp -mfpu=fpv5-sp-d16 -mfloat-abi=hard -c



Options for Target 'Target_GDM32'

Device | Target | Output | Listing | User | C/C++ (AC6) | Asm | Linker | Debug | Utilities

- [ ] Use Memory Layout from Target Dialog
- [ ] Make RW Sections Position Independent
- [ ] Make RO Sections Position Independent
- [ ] Don't Search Standard Libraries
- [✓] Report 'might fail' Conditions as Errors

X/O Base:

R/O Base: 0x08000000

R/W Base 0x20000000

disable Warnings:

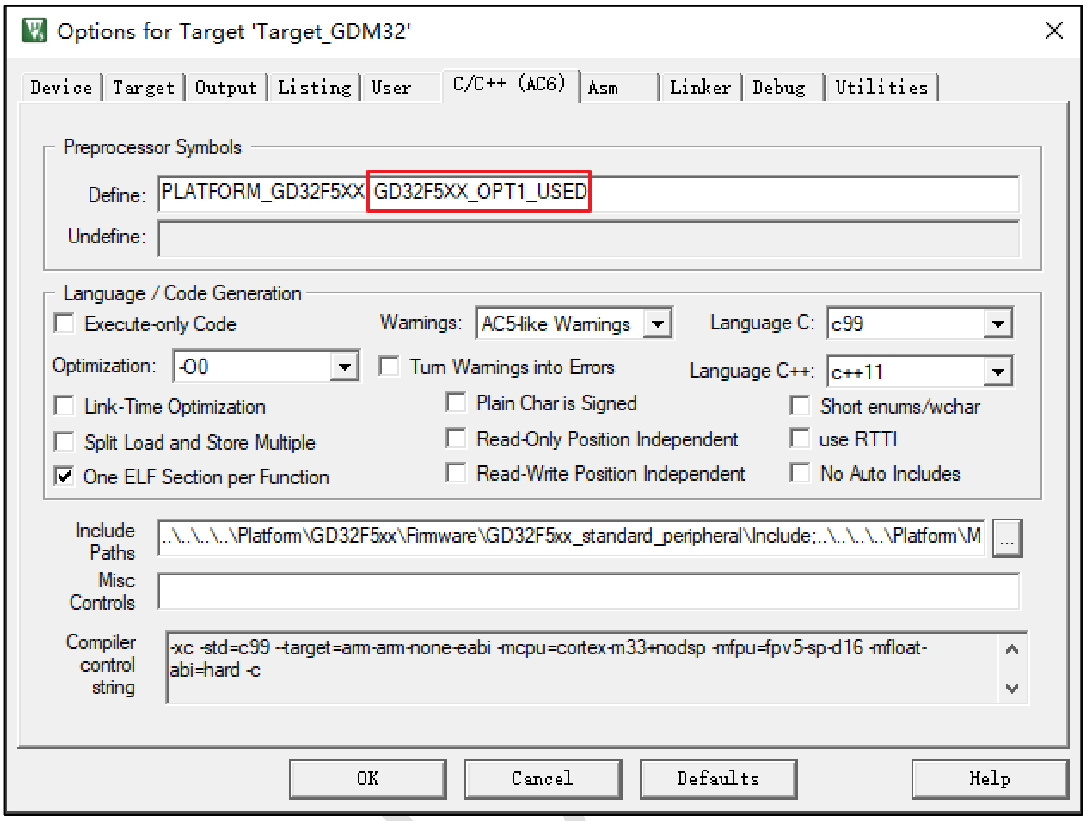
Scatter File: ..\..\..\..\Platform\GD32F5xx\armclang\ibl_gdm32.sct

Misc controls: --predefine="-DPLATFORM_GD32F5XX -DGD32F5XX_OPT1_USED" --symdefs=.\rom_symbol_verify

Linker control string:
--cpu=Cortex-M33.no_dsp *.o
--library_type=microlib --strict --scatter "..\..\..\..\Platform\GD32F5xx\armclang\ibl_gdm32.sct"

− In the MBL and APP projects, the enable macro "GD32F5XX_OPT1_USED" is also required. For modifications in the MBL, you can refer to ***Figure 7-13. MBL modification*** (modifications for the APP project can refer to the MBL project).

Figure 7-13. MBL modification



Preprocessor Symbols
Define: PLATFORM_GD32F5XX GD32F5XX_OPT1_USED
Undefine:

Language / Code Generation
Execute-only Code
Warnings: AC5-like Warnings
Language C: c99
Optimization: -O0
Turn Warnings into Errors
Language C++: c++11
Link-Time Optimization
Plain Char is Signed
Short enums/wchar
Split Load and Store Multiple
Read-Only Position Independent
use RTTI
[✓ One ELF Section per Function]
Read-Write Position Independent
No Auto Includes

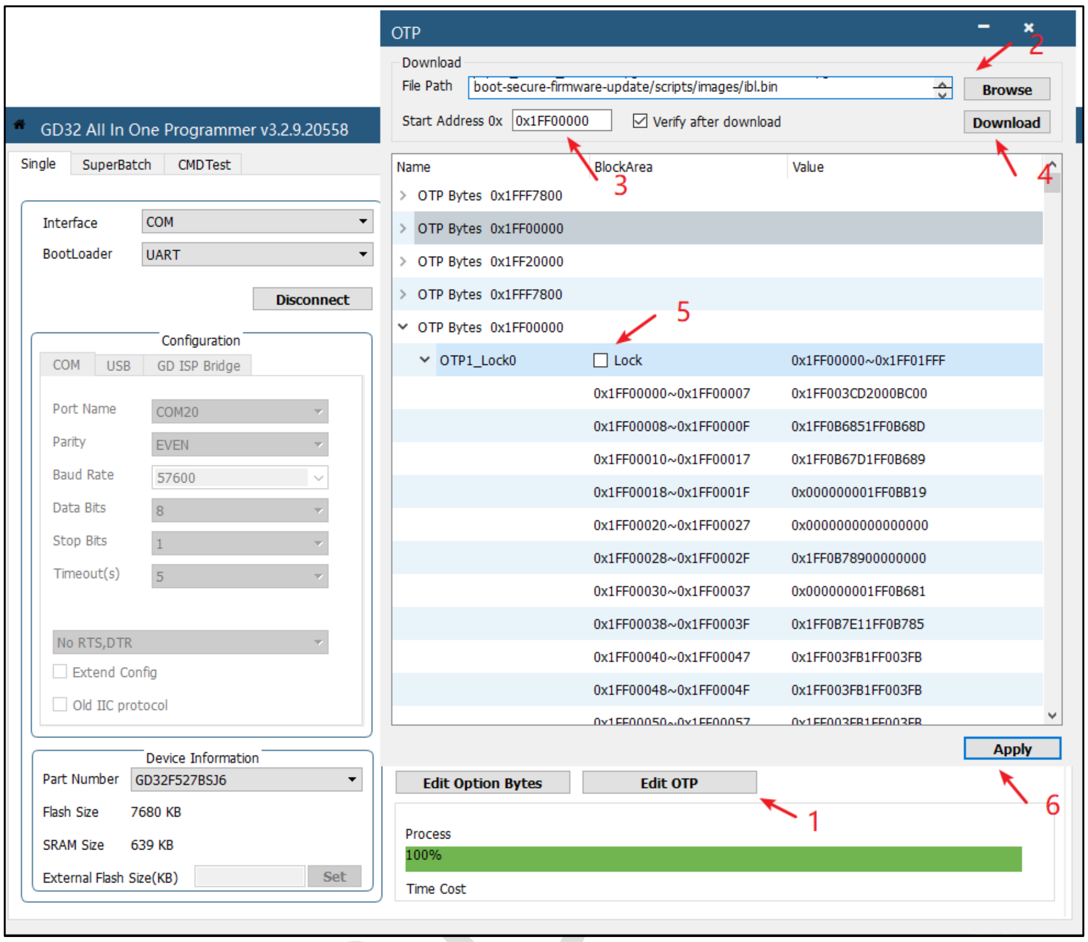
Include Paths: ..\..\..\..\Platform\GD32F5xx\Firmware\GD32F5xx_standard_peripheral\Include;..\..\..\..\Platform\M
Misc Controls:
Compiler control string: -xc -std=c99 --target=arm-arm-none-eabi -mcpu=cortex-m33+nodsp -mfpu=fpv5-sp-d16 -mfloat-abi=hard -c

− Modify the "config/config_gdm32.h" configuration file, and change the macro RE_FLASH_BASE to 0x08000000.

■ Use the GD32AllInOneProgrammer.exe adapter to IBL into the OTP1 area.

After correctly modifying the project and compiling it successfully, an ibl.bin file will be generated in the "scripts/mages" directory. The ibl.bin file is the final executable file for IBL. You can use the GD32AllInOneProgrammer.exe software to download the code and configure OTP1 to be write-locked. For specific steps, you can refer to ***Figure 7-14. Download IBL to the OTP1***.

Figure 7-14. Download IBL to the OTP1



GD32 All In One Programmer v3.2.9.20558

Interface: COM
BootLoader: UART

Configuration

Port Name	COM20
Parity	EVEN
Baud Rate	57600
Data Bits	8
Stop Bits	1
Timeout(s)	5

No RTS,DTR
☐ Extend Config
☐ Old IIC protocol

Device Information

Part Number	GD32F527BSJ6
Flash Size	7680 KB
SRAM Size	639 KB
External Flash Size(KB)	

Download
File Path: boot-secure-firmware-update/scripts/images/ibl.bin
Start Address 0x: 0x1FF00000
☑ Verify after download

Name	BlockArea	Value
OTP Bytes 0x1FFF7800		
OTP Bytes 0x1FF00000		
OTP Bytes 0x1FF20000		
OTP Bytes 0x1FFF7800		
OTP Bytes 0x1FF00000		
OTP1_Lock0	☐ Lock	0x1FF00000~0x1FF01FFF
	0x1FF00000~0x1FF00007	0x1FF003CD2000BC00
	0x1FF00008~0x1FF0000F	0x1FF0B6851FF0B68D
	0x1FF00010~0x1FF00017	0x1FF0B67D1FF0B689
	0x1FF00018~0x1FF0001F	0x000000001FF0BB19
	0x1FF00020~0x1FF00027	0x0000000000000000
	0x1FF00028~0x1FF0002F	0x1FF0B78900000000
	0x1FF00030~0x1FF00037	0x000000001FF0B681
	0x1FF00038~0x1FF0003F	0x1FF0B7E11FF0B785
	0x1FF00040~0x1FF00047	0x1FF003FB1FF003FB
	0x1FF00048~0x1FF0004F	0x1FF003FB1FF003FB
	0x1FF00050~0x1FF00057	0x1FF003FB1FF003FB

Process
100%
Time Cost

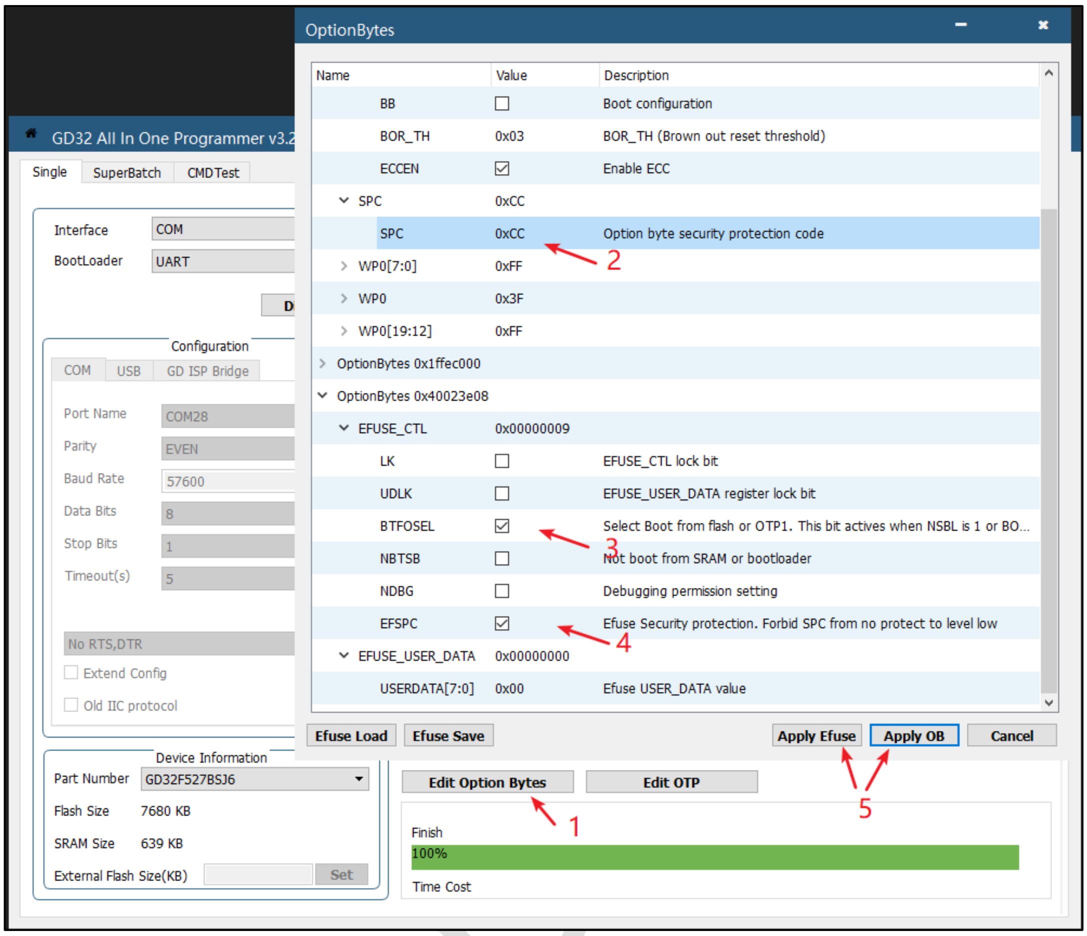
Note:

- Since the OTP1 area is only 128KB, some functions of the IBL may be missing;
- The read lock for OTP1 has not yet been implemented;
- To boot the chip from the OTP1 area, the BTFOSEL bit needs to be set, which is located in the EFUSE, and once enabled, it cannot be undone.

7.10 Locking for Boot Entry

For secure boot, it is necessary to ensure a unique entry point, and after reset, the code of IBL should be run first instead of other code. For the GD32F5xx, it is recommended to configure the chip for high-level security protection and EFUSE settings to achieve this purpose. It is recommended to use the GD32AllInOneProgrammer.exe software for configuration. For the specific steps of configuring the chip for high-level security protection, you can refer to ***Figure 7-15. Configuring high-level security protection***.

Figure 7-15. Configuring high-level security protection



Name	Value	Description
BB	☐	Boot configuration
BOR_TH	0x03	BOR_TH (Brown out reset threshold)
ECCEN	☑	Enable ECC
SPC	0xCC	
SPC	0xCC	Option byte security protection code
WP0[7:0]	0xFF	
WP0	0x3F	
WP0[19:12]	0xFF	
OptionBytes 0x1ffec000		
OptionBytes 0x40023e08		
EFUSE_CTL	0x00000009	
LK	☐	EFUSE_CTL lock bit
UDLK	☐	EFUSE_USER_DATA register lock bit
BTFOSEL	☑	Select Boot from flash or OTP1. This bit actives when NSBL is 1 or BO...
NBTSB	☐	Not boot from SRAM or bootloader
NDBG	☐	Debugging permission setting
EFSPC	☑	Efuse Security protection. Forbid SPC from no protect to level low
EFUSE_USER_DATA	0x00000000	
USERDATA[7:0]	0x00	Efuse USER_DATA value

Interface	COM
BootLoader	UART
Port Name	COM28
Parity	EVEN
Baud Rate	57600
Data Bits	8
Stop Bits	1
Timeout(s)	5
	No RTS,DTR
☐ Extend Config	
☐ Old IIC protocol	
Part Number	GD32F527BSJ6
Flash Size	7680 KB
SRAM Size	639 KB
External Flash Size(KB)	

Finish
100%
Time Cost

Note:

- Once configured to a high security protection level, it cannot be undone, and debugging with a debugger will not be possible;
- After being set to a high security protection level; the option bytes cannot be modified;
- After being set to a high security protection level; it will not be possible to start from RAM and the Bootloader

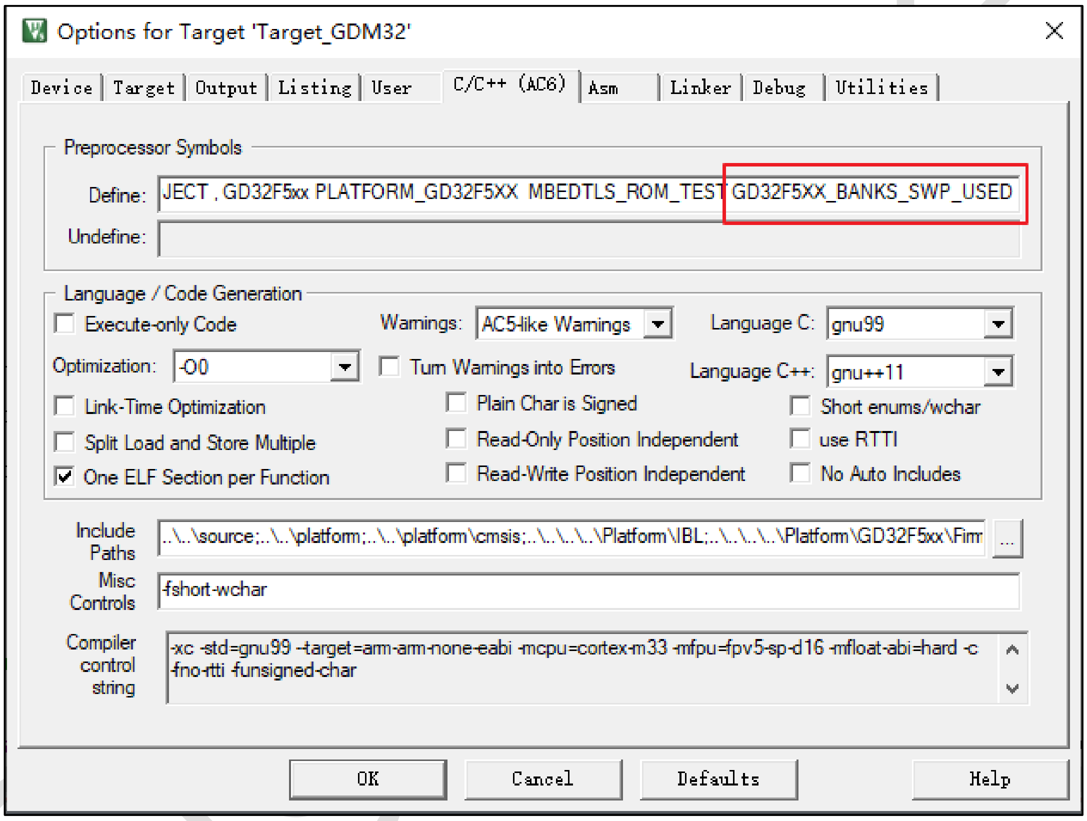
7.11 Dual-bank Switching Feature

The GD32F5xx series of chips supports a dual-bank flash memory structure for some series, and can support the exchange of BANK0 and BANK1 areas, which can be used to achieve the update of the APP image. For more detailed content about dual-bank, please refer to the "GD32F5xx User Manual". The GD32 SBSFU project has been adapted for dual-bank switching on GD32F5xx devices to achieve the update of the APP image (currently only supports 7.5M on-chip flash memory chips).

7.11.1 Enabling Dual-bank

In the SBSFU project, the "config/config_gdm32_otp_bankswp.h" configuration file is provided. When the dual-bank function is needed, the content in the "config/config_gdm32_otp_bankswp.h" file can be used to replace the content in the "config/config_gdm32.h" configuration file. After that, the "GD32F5XX_BANKS_SWP_USED" macro can be added to the IBL, MBL, and APP projects to enable the dual-bank function. Enabling the dual-bank function in the IBL project can refer to ***Figure 7-16. IBL Project enables dual-bank functionality***. The enabling method for MBL and APP is similar to that of IBL.

Figure 7-16. IBL Project enables dual-bank functionality


Options for Target 'Target_GDM32'

Device | Target | Output | Listing | User | C/C++ (AC6) | Asm | Linker | Debug | Utilities

Preprocessor Symbols
Define: JECT , GD32F5xx PLATFORM_GD32F5XX MBEDTLS_ROM_TEST GD32F5XX_BANKS_SWP_USED
Undefine:

Language / Code Generation
- [ ] Execute-only Code
- Warnings: AC5-like Warnings
- Language C: gnu99
- Optimization: -O0
- [ ] Turn Warnings into Errors
- Language C++: gnu++11
- [ ] Link-Time Optimization
- [ ] Plain Char is Signed
- [ ] Short enums/wchar
- [ ] Split Load and Store Multiple
- [ ] Read-Only Position Independent
- [ ] use RTTI
- [✓] One ELF Section per Function
- [ ] Read-Write Position Independent
- [ ] No Auto Includes

Include Paths: ..\..\source;..\..\platform;..\..\platform\cmsis;..\..\..\..\Platform\IBL;..\..\..\..\Platform\GD32F5xx\Firm
Misc Controls: -fshort-wchar
Compiler control string: -xc -std=gnu99 --target=arm-arm-none-eabi -mcpu=cortex-m33 -mfpu=fpv5-sp-d16 -mfloat-abi=hard -c -fno-rtti -funsigned-char

OK | Cancel | Defaults | Help

Note:

- The macro "GD32F5XX_BANKS_SWP_USED", once enabled, requires the simultaneous enabling of "GD32F5XX_BANKS_SWP_USED".

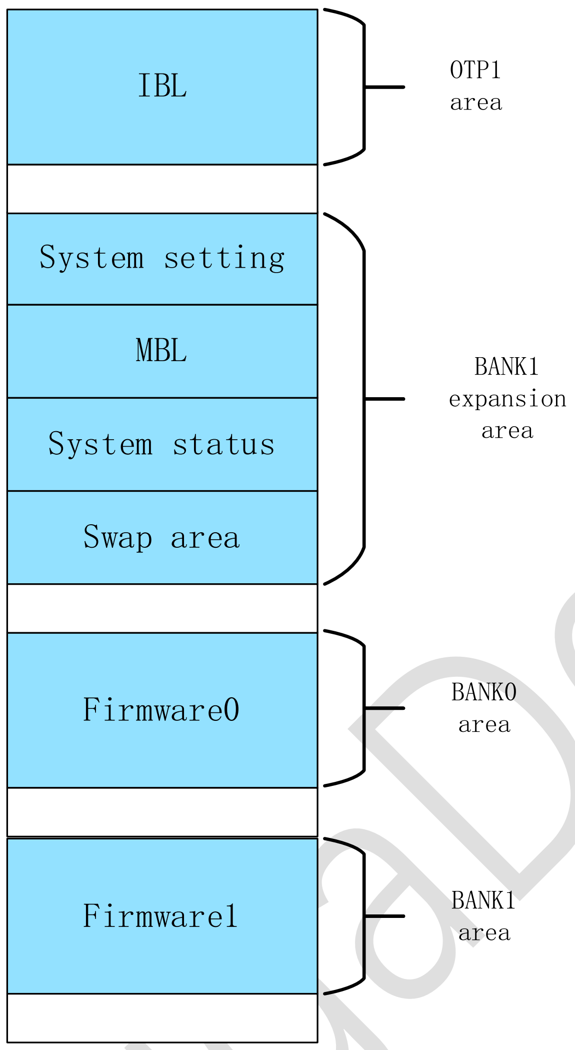
7.11.2 Image Distribution After Enabling Dual-bank

The mirror distribution can be referred to as shown in ***Figure 7-17. Mirror distribution after enabling dual-bank***:

- IBL will run in the OTP1 region;
- System configuration information, system status information, MBL, and the swap area are stored in the extended area of bank1 (the swap area is no longer valid);

- The running firmware (Firmware 0) and the updated firmware (Firmware 1) are stored in Bank 0 and Bank 1, respectively.

Figure 7-17. Mirror distribution after enabling dual-bank



7.12 Update encrypted firmware

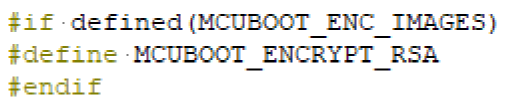
The encrypted firmware header marks APP IMG as ENCRYPTED (0x04). The firmware is encrypted using AES. When a new IMG is created, the key is randomly generated by imgtool. This key should not be reused and is not checked for this, but using TRNG to randomly generate 16-byte blocks makes repetition extremely unlikely. The AES key is asymmetrically encrypted and stored in the TLVs of IMG, and the asymmetrically encrypted private key is stored in the MCU (where the public key is used to encrypt the firmware). Firmware headers and TLVs are still sent as plaintext data. The functionality of hashing and signing is also the same as before, applied to unencrypted data.

The encryption firmware is updated as it is decrypted, as shown in ***Figure 3-3. FLASH Layout***.

The IMG currently running is stored in plain text at Frimware0. The encryption firmware is downloaded to Firmware1 through ymodem. Before firmware update, the encrypted AES key in TLVs is decrypted asymmetrically. After firmware encryption is enabled, the updated firmware is stored in plaintext in Firmware0, and the updated firmware is encrypted in Firmware1 by AES.

- As shown in *Figure 7-18. Enable firmware encryption upgrade*, The related define in mcuboot_config.h in MBL are enabled.

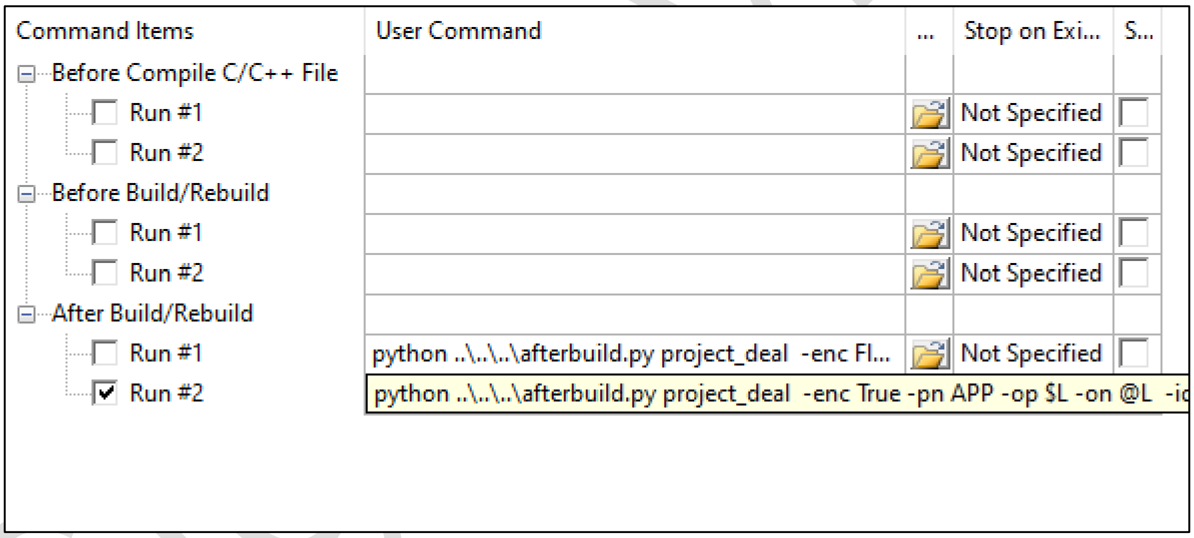
Figure 7-18. Enable firmware encryption upgrade



```
#if defined(MCUBOOT_ENC_IMAGES)
#define MCUBOOT_ENCRYPT_RSA
#endif
```

- As shown in *Figure 7-19. Generate encrypted firmware*. To update the APP firmware through Ymodem, check Run#2. For the firmware that directly loads, select Run#1.

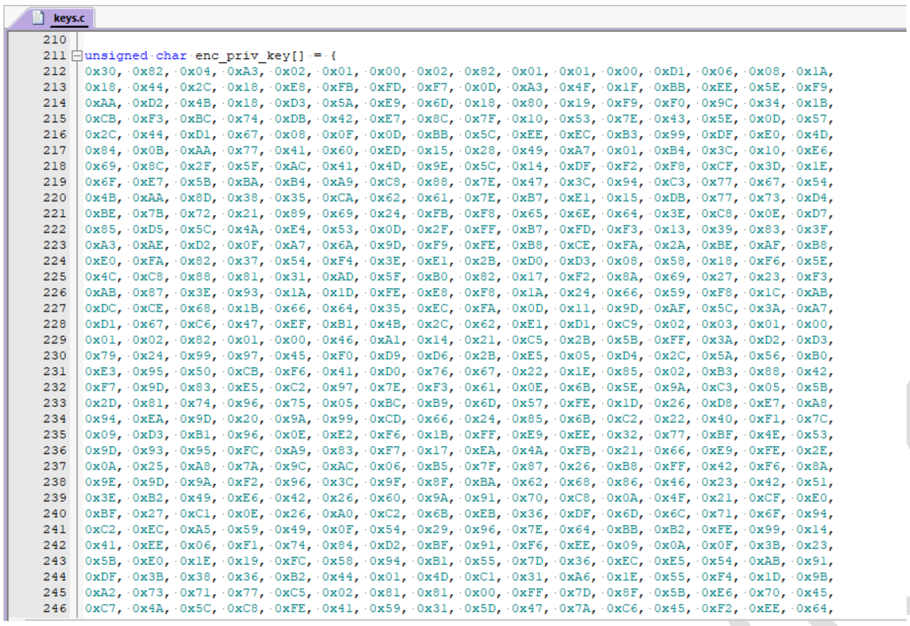
Figure 7-19. Generate encrypted firmware



Command Items	User Command	...	Stop on Exi...	S...
Before Compile C/C++ File				
☐ Run #1			Not Specified	☐
☐ Run #2			Not Specified	☐
Before Build/Rebuild				
☐ Run #1			Not Specified	☐
☐ Run #2			Not Specified	☐
After Build/Rebuild				
☐ Run #1	python ..\..\..\afterbuild.py project_deal -enc Fl...		Not Specified	☐
☑ Run #2	python ..\..\..\afterbuild.py project_deal -enc True -pn APP -op $L -on @L -ic			

- As shown in *Figure 7-20. RSA key*, in the case of RSA, the keys are stored in the keys.c file.

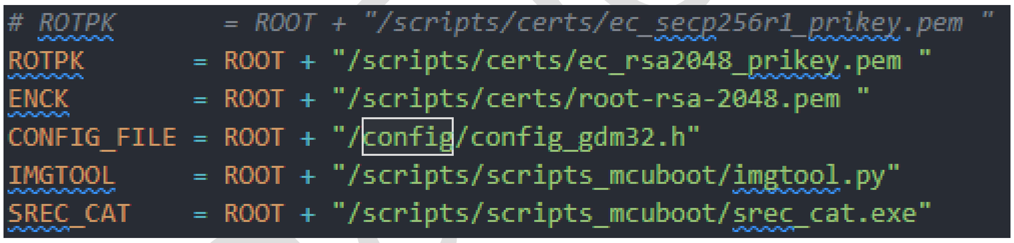
Figure 7-20. RSA key


keys.c

```
unsigned char enc_priv_key[] = {
0x30, 0x82, 0x04, 0xA3, 0x02, 0x01, 0x00, 0x02, 0x82, 0x01, 0x01, 0x00, 0xD1, 0x06, 0x08, 0x1A,
0x18, 0x44, 0x2C, 0x18, 0xE8, 0xFB, 0xFD, 0xF7, 0x0D, 0xA3, 0x4F, 0x1F, 0xBB, 0xEE, 0x5E, 0xF9,
0xAA, 0xD2, 0x4B, 0x18, 0xD3, 0x5A, 0xE9, 0x6D, 0x18, 0x80, 0x19, 0xF9, 0xF0, 0x9C, 0x34, 0x1B,
0xCB, 0xF3, 0xBC, 0x74, 0xDB, 0x42, 0xE7, 0x8C, 0x7F, 0x10, 0x53, 0x7E, 0x43, 0x5E, 0x0D, 0x57,
0x2C, 0x44, 0xD1, 0x67, 0x08, 0x0F, 0x0D, 0xBB, 0x5C, 0xEE, 0xEC, 0xB3, 0x99, 0xDF, 0xE0, 0x4D,
0x84, 0x0B, 0xAA, 0x77, 0x41, 0x60, 0xED, 0x15, 0x28, 0x49, 0xA7, 0x01, 0xB4, 0x3C, 0x10, 0xE6,
0x69, 0x8C, 0x2F, 0x5F, 0xAC, 0x41, 0x4D, 0x9E, 0x5C, 0x14, 0xDF, 0xF2, 0xF8, 0xCF, 0x3D, 0x1E,
0x6F, 0xE7, 0x5B, 0xBA, 0xB4, 0xA9, 0xC8, 0x88, 0x7E, 0x47, 0x3C, 0x94, 0xC3, 0x77, 0x67, 0x54,
0x4B, 0xAA, 0x8D, 0x38, 0x35, 0xCA, 0x62, 0x61, 0x7E, 0xB7, 0xE1, 0x15, 0xDB, 0x77, 0x73, 0xD4,
0xBE, 0x7B, 0x72, 0x21, 0x89, 0x69, 0x24, 0xFB, 0xF8, 0x65, 0x6E, 0x64, 0x3E, 0xC8, 0x0E, 0xD7,
0x85, 0xD5, 0x5C, 0x4A, 0xE4, 0x53, 0x0D, 0x2F, 0xFF, 0xB7, 0xFD, 0xF3, 0x13, 0x39, 0x83, 0x3F,
0xA3, 0xAE, 0xD2, 0x0F, 0xA7, 0x6A, 0x9D, 0xF9, 0xFE, 0xB8, 0xCE, 0xFA, 0x2A, 0xBE, 0xAF, 0xB8,
0xE0, 0xFA, 0x82, 0x37, 0x54, 0xF4, 0x3E, 0xE1, 0x2B, 0xD0, 0xD3, 0x08, 0x58, 0x18, 0xF6, 0x5E,
0x4C, 0xC8, 0x88, 0x81, 0x31, 0xAD, 0x5F, 0xB0, 0x82, 0x17, 0xF2, 0x8A, 0x69, 0x27, 0x23, 0xF3,
0xAB, 0x87, 0x3E, 0x93, 0x1A, 0x1D, 0xFE, 0xE8, 0xF8, 0x1A, 0x24, 0x66, 0x59, 0xF8, 0x1C, 0xAB,
0xDC, 0xCE, 0x68, 0x1B, 0x66, 0x64, 0x35, 0xEC, 0xFA, 0x0D, 0x11, 0x9D, 0xAF, 0x5C, 0x3A, 0xA7,
0xD1, 0x67, 0xC6, 0x47, 0xEF, 0xB1, 0x4B, 0x2C, 0x62, 0xE1, 0xD1, 0xC9, 0x02, 0x03, 0x01, 0x00,
0x01, 0x02, 0x82, 0x01, 0x00, 0x46, 0xA1, 0x14, 0x21, 0xC5, 0x2B, 0x5B, 0xFF, 0x3A, 0xD2, 0xD3,
0x79, 0x24, 0x99, 0x97, 0x45, 0xF0, 0xD9, 0xD6, 0x2B, 0xE5, 0x05, 0xD4, 0x2C, 0x5A, 0x56, 0xB0,
0xE3, 0x95, 0x50, 0xCB, 0xF6, 0x41, 0xD0, 0x76, 0x67, 0x22, 0x1E, 0x85, 0x02, 0xB3, 0x88, 0x42,
0xF7, 0x9D, 0x83, 0xE5, 0xC2, 0x97, 0x7E, 0xF3, 0x61, 0x0E, 0x6B, 0x5E, 0x9A, 0xC3, 0x05, 0x5B,
0x2D, 0x81, 0x74, 0x96, 0x75, 0x05, 0xBC, 0xB9, 0x6D, 0x57, 0xFE, 0x1D, 0x26, 0xD8, 0xE7, 0xA8,
0x94, 0xEA, 0x9D, 0x20, 0x9A, 0x99, 0xCD, 0x66, 0x24, 0x85, 0x6B, 0xC2, 0x22, 0x40, 0xF1, 0x7C,
0x09, 0xD3, 0xB1, 0x96, 0x0E, 0xE2, 0xF6, 0x1B, 0xFF, 0xE9, 0xEE, 0x32, 0x77, 0xBF, 0x4E, 0x53,
0x9D, 0x93, 0x95, 0xFC, 0xA9, 0x83, 0xF7, 0x17, 0xEA, 0x4A, 0xFB, 0x21, 0x66, 0xE9, 0xFE, 0x2E,
0x0A, 0x25, 0xA8, 0x7A, 0x9C, 0xAC, 0x06, 0xB5, 0x7F, 0x87, 0x26, 0xB8, 0xFF, 0x42, 0xF6, 0x8A,
0x9E, 0x9D, 0x9A, 0xF2, 0x96, 0x3C, 0x9F, 0x8F, 0xBA, 0x62, 0x68, 0x86, 0x46, 0x23, 0x42, 0x51,
0x3E, 0xB2, 0x49, 0xE6, 0x42, 0x26, 0x60, 0x9A, 0x91, 0x70, 0xC8, 0x0A, 0x4F, 0x21, 0xCF, 0xE0,
0xBF, 0x27, 0xC1, 0x0E, 0x26, 0xA0, 0xC2, 0x6B, 0xEB, 0x36, 0xDF, 0x6D, 0x6C, 0x71, 0x6F, 0x94,
0xC2, 0xEC, 0xA5, 0x59, 0x49, 0x0F, 0x54, 0x29, 0x96, 0x7E, 0x64, 0xBB, 0xB2, 0xFE, 0x99, 0x14,
0x41, 0xEE, 0x06, 0xF1, 0x74, 0x84, 0xD2, 0xBF, 0x91, 0xF6, 0xEE, 0x09, 0x0A, 0x0F, 0x3B, 0x23,
0x5B, 0xE0, 0x1E, 0x19, 0xFC, 0x58, 0x94, 0xB1, 0x55, 0x7D, 0x36, 0xEC, 0xE5, 0x54, 0xAB, 0x91,
0xDF, 0x3B, 0x38, 0x36, 0xB2, 0x44, 0x01, 0x4D, 0xC1, 0x31, 0xA6, 0x1E, 0x55, 0xF4, 0x1D, 0x9B,
0xA2, 0x73, 0x71, 0x77, 0xC5, 0x02, 0x81, 0x81, 0x00, 0xFF, 0x7D, 0x8F, 0x5B, 0xE6, 0x70, 0x45,
0xC7, 0x4A, 0x5C, 0xC8, 0xFE, 0x41, 0x59, 0x31, 0x5D, 0x47, 0x7A, 0xC6, 0x45, 0xF2, 0xEE, 0x64,
```

■ As shown in ***Figure 7-21. Keys configuration in the afterbuild.py***, ROTPK corresponds to the asymmetric key pair signed in the firmware, and ENCK corresponds to the asymmetric key pair given to the encrypted AES key.

Figure 7-21. Keys configuration in the afterbuild.py



```
# ROTPK        = ROOT + "/scripts/certs/ec_secp256r1_prikey.pem "
ROTPK          = ROOT + "/scripts/certs/ec_rsa2048_prikey.pem "
ENCK           = ROOT + "/scripts/certs/root-rsa-2048.pem "
CONFIG_FILE = ROOT + "/config/config_gdm32.h"
IMGTOOL        = ROOT + "/scripts/scripts_mcuboot/imgtool.py"
SREC_CAT       = ROOT + "/scripts/scripts_mcuboot/srec_cat.exe"
```

8 Implementation of SBSFU Project on GD32H7xx

The implementation of SBSFU project on GD32H7xx follows a similar architecture and usage process to that on GD32F5xx. The differences that need attention are:

- The project group for GD32H7xx is located in "Project/GD32H7xx/MDK-ARM/SBSFU.uvmpw";
- If compiling the IBL project with the O2 optimization level, the macro "MEM_CMP_NOT_INV_ICACHE" needs to be enabled.

9 Implementation of SBSFU Project on GD32G5x3

The implementation of SBSFU project on GD32G5x3 follows a similar architecture and usage process to that on GD32F5xx. The differences that need attention are:

- The project group for GD32G5x3 is located in "Project/GD32G553/MDK-ARM/SBSFU.uvmpw"; The Project group whose bank exchange is enabled is located in "Project/GD32G553_bankswap/MDK-ARM/SBSFU.uvmpw";
- Because the APP code cannot get the exact address of the IBL interface function after dual bank switching is enabled, G5X3 APP does not have the function of updating firmware.

10 Appendix A: Description of Functional Macros

Table 10-1. Functional Macro Description

Name	Description	Value
LOG_UART	define the used serial port	USART0：PA9、PA10
		USART1：PB15、PA8
		USART2：PB10、PB11
PLATFORM_GD32F5XX	use the GD32F5XX series chip	
INNER_ROTPK_EFUSE	whether POTPK is stored in EFUSE	0：not stored in EFUSE
		1：stored in EFUSE
INNER_HUK_EFUSE	whether HUK is stored in EFUSE	0：not stored in EFUSE
		1：stored in EFUSE
GD32F5XX_OTP2_ROTPK	whether GD32F5XX ROTPK is stored in OTP2	/
IMG_VERIFY_OPT	verify the type of image	IBL_VERIFY_CERT_IMG：verify certificate (not implemented)
		IBL_VERIFY_IMG_ONLY：verify image only
		IBL_VERIFY_NONE：no verification, jump directly
SPC_PROTECT_ENABLE	enable security protection	/
FWDG_PROTECT_ENABLE	enable watchdog protection	/
IBL_TEST_SUIT	enable IBL testing	/
SYS_STATUS_ENCRPTED	encrypt storage of system state	0：not encrypted (not verified)
		1：encrypted
SIGN_ALGO_SET	signature verification algorithm	IMG_SIG_ED25519：ED25519
		IMG_SIG_ECDSA256：ECDSA256
GD32F5XX_OPT1_USED	GD32F5XX series stores IBL in OPT1	/
RE_FLASH_BASE	starting address of IBL shared date	/
RE_SRAM_BASE	referenced RAM starting address	/
RE_SHARED_DATA_START	starting data area starting address	If you want to use the IBL's encryption library, the subsequent code RAM address needs to be utilized, the space following the starting address of the shared data.
RE_MBL_DATA_START	MBL data area starting address	
RE_APP_DATA_START	APP data area starting	

Name	Description	Value
	address	
RE_VTOR_ALIGNMENT	alignment address of interrupt vectors	default is 0x200
RE_SYS_SET_OFFSET	the offset address of the system configuration data, relative to the offset from RE_FLASH_BASE	default is 0
RE_MBL_OFFSET	offset address of system configuration data, offset relative to RE_FLASH_BASE	default is 0x1000 (system configuration data is 4KB, the smallest page unit)
RE_SYS_STATUS_OFFSET	offset address of system state	default is 0x20000
RE_IMG_0_APP_OFFSET	offset address of APP0	
RE_IMG_1_APP_OFFSET	offset address of APP1	
RE_MBL_VERSION	version of MBL	currently does not support automatic version generation, manual addition is required when calling the script
RE_APP_VERSION	version of APP	currently does not support automatic version generation, manual addition is required when calling the script
FLASH_BASE_IBL	base address of IBL code segment	
IBL_CODE_START	starting address of IBL code segment	default is FLASH_BASE_IBL
IBL_CODE_SIZE	size of IBL code segment	
FLASH_BASE_LIB	base address of IBL library	
FLASH_LIB_START	starting address of IBL library	
FLASH_LIB_SIZE	size of IBL library	
FLASH_OFFSET_SYS_SETTING	offset address of system configuration data, offset relative to the base address of Flash	
FLASH_API_ARRAY_BASE	starting address information of Flash API	default is FLASH_LIB_START
FLASH_API_ARRAY_RSVD	size information of Flash API	default is 0x800
IBL_DATA_START	starting address of IBL data segment	
IBL_DATA_SIZE	size of IBL data segment	
IBL_HEAP_SIZE	size of IBL HEAP segment	

Name	Description	Value
IBL_MSP_STACK_SIZE	size of IBL STACK segment	
IBL_SHARED_DATA_START	starting address of IBL shared data	
IBL_SHARED_DATA_SIZE	size of IBL shared data	
MBL_BASE_ADDRESS	base address of MBL code segment	
MBL_CODE_START	starting address of MBL code segment	
MBL_CODE_SIZE	size of MBL code segment	
MBL_SHARED_DATA_START	starting address of MBL shared data	
MBL_SHARED_DATA_SIZE	size of MBL shared data	
MBL_DATA_START	starting address of MBL data	
MBL_BUF_SIZE	size of data allocated to mbedtls by MBL	
MBL_MSP_STACK_SIZE	size of MBL stack	
BOOTUTIL_HW_DOWNGRADE_PREVENTION	prevent firmware downgrade	
MCUBOOT_DOWNGRADE_PREVENTION_SECURITY_COUNTER	use secure counter	1: use a security counter (not implemented)
		0: use version number for comparison (verified)
MCUBOOT_USE_MBED_TLS	select the encryption library as mebedtls	
MCUBOOT_IMAGE_NUMBER	number of supported images	currently, only the case where the image is 1 has been implemented
MCUBOOT_HAVE_LOGGING	enable log	
MCUBOOT_LOG_LEVEL	Log print level	MCUBOOT_LOG_LEVEL_OFF (0)
		MCUBOOT_LOG_LEVEL_ERROR (1)
		MCUBOOT_LOG_LEVEL_WARNING (2)
		MCUBOOT_LOG_LEVEL_INFO (3)
		MCUBOOT_LOG_LEVEL_DEBUG (4)
		MCUBOOT_LOG_LEVEL_SIM (5)
MCUBOOT_SIGN_EC256	the signature verification algorithm used in ESDSA256	
MCUBOOT_VALIDATE_PRIMARY_SLOT	verify the main slot at each startup	
MCUBOOT_USE_IBL_VERIFY_SIG	use IBL’s API for signature	

Name	Description	Value
N	verification	
GD32F5XX_BANKS_SWP_USED	enable the dual-bank switching function of GD32F5XX	

11 Version History

Version Number	Description	Date
1.0	1. initial Release Version	June 2024
1.0	1. add description of GD32F5xx dual-bank switching	July 2024

Important Notice