GigaDevice Semiconductor Inc.

Proper use of UID for firmware protection

Application Note
AN073

Table of Contents

List of Figures

List of Tables

1. Introduction

This application note aims to protect software code by increasing the complexity of software code, and prevent software code from being ported to other platforms or chips.

This application note is divided into two parts. The first part introduces the application scenarios and the capability of software code; the second part introduces how to protect software code.

This application note is theoretically applicable to the whole series of GD32 MCU.

2. Application scenario and capability analysis

2.1. Application scenario

The GD32 MCU provide hardware read and write protection to prevent illegal reading of flash, and this function can protect the intellectual property of firmware. However, in actual product development or mass production, there are many application scenarios that do not require read and write protection, such as:

- Cooperative development of product, multiple developers provide their respective code in flash for subsequent development or mass production.
- The algorithm provider solidifies the algorithm code in flash for the customer to use.
- Users need to set parameters according to the usage environment and save the parameters to flash.
- When the product is delivered, the software functions are still unstable or incomplete, and it is necessary to update and iterate the software functions according to user's feedback.

In the above cases, the MCU usually cannot enable read and write protection, which will cause the firmware in the flash to be easily read, ported or reversed. Therefore, in order to prevent the firmware from being stolen or ported to other platforms, other methods are needed to protect the firmware.

2.2. Capability required for code

In the case of not enabling hardware read and write protection, from the perspective of code design, the code needs to have the following two capabilities:

- If the firmware is obtained from the MCU, but the firmware cannot run on other chips.
- The code need to design complexly, and third parties can not analyze it easily.

3. Process methods

3.1. Overview

The 96-bit unique device ID is unique for each MCU. When the code is bound to the UID, the code can only run on the specified chip. Meanwhile, improving the design method, increasing the redundant design, increasing the verification hardware and using specific peripherals can be used to increase the complexity of the code.

The following sections describe five methods to increase code complexity. They are not mutually exclusive, and users can use them in combination with each other.

3.2. Use special peripherals

This section describes how to increase the complexity of code by using special peripherals. For example, when need to use UID, it is not recommend to read directly through the register, but it is recommend to read through DMA. This method can increase the complexity of code analysis. Similarly, the MPU and privilege levels can be used to increase the complexity of code.

In addition, the hardware security protection can be enable, then firmware can be protected from being read and debugged.

3.3. Improve design method

This section describes how to binds the code with MCU UID.

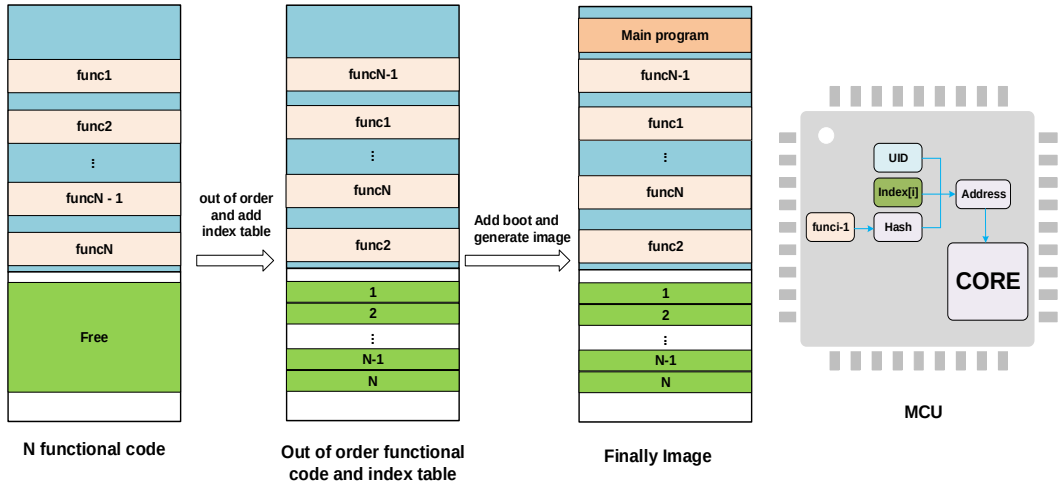
This method is shown as ***The calculation methods of code integrity*** include but not limited to hash, CRC, checksum, etc. User need to define scramble method. When the first segment of code is processed, the integrity result of all the code and the UID are used as parameters to calculate its disorder address.

Figure 3-1. Improved Design Method Schematic. The N segments of functional code can be load to specified address by scatter loading file, and each functional code size is equal. Computer application read BIN or HEX file, use UID and the result of integrity calculation for previous functional code as two parameters, use specific computing method to generate address, and disorder N segments of functional code. Scatter loading functional code can refer to ***AN075 "Introduction of library invocation scheme based on MDK implementation"***.

The calculation methods of code integrity include but not limited to hash, CRC, checksum, etc. User need to define scramble method. When the first segment of code is processed, the

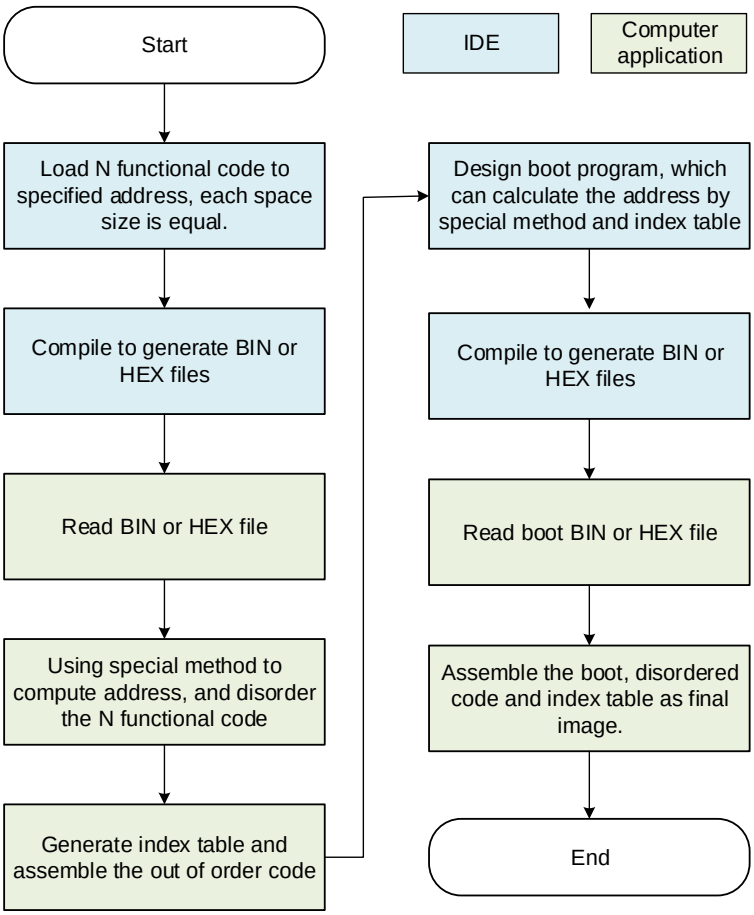
integrity result of all the code and the UID are used as parameters to calculate its disorder address.

Figure 3-1. Improved Design Method Schematic



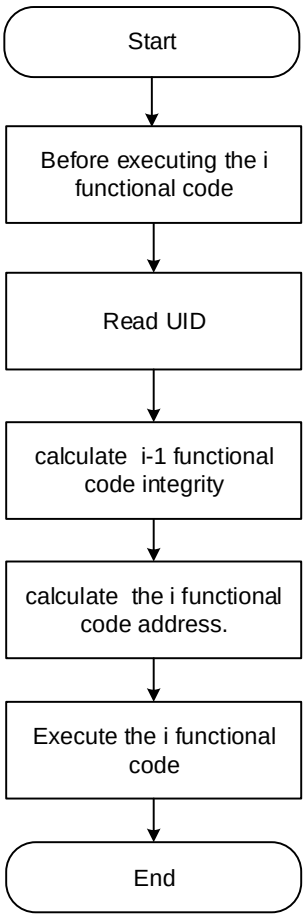
In order to ensure the availability of the calculated result, position encoding calculation needs to be performed based on the calculation result, and the encoding result is stored in the index table. The final Image includes the index table, Finally, design a boot program for calculating functi-onal code addresses, the image assemble the boot, disordered functional code and index table as final image. The process of image generate is shown as ***Figure 3-2. Image file generation flow chart*** .

Figure 3-2. Image file generation flow chart



When a functional code need to be executed, according to the UID, the integrity result of the previous code and the index table, use the same calculation method as the computer application, calculate the correct address of the functional code. For example, the process of executing the i-th functional code is shown as ***Figure 3-3. Program running flow chart***, if UID or previous code have changed, then calculate wrong address, and cause the program incorrectly.

Figure 3-3. Program running flow chart



If functional code execute sequentially, the example logic is easy to understand, and it is easy to be plagiarized. The example is shown as ***Table 3-1. The example of executing functional code sequentially***.

Table 3-1. The example of executing functional code sequentially

```
/* execute functional code */
func1();
func2();
func3();
func4();
func5();
```

When disorder the N functional code, the program logic is difficult to understand. The example is shown as ***Table 3-2. The example of executing out-of-order functional code***.

Table 3-2. The example of executing out-of-order functional code

```
/* define function pointer */
void (*func)(void);
/* define index address */
uint32 * index = (uint32 *)(0x0801F000);
/* calculate full code's completeness when index = 0 */
```

```
uint32 * previous_ res = calculate_code_ completeness (address_full_code,full_size);
/* calculate next functional code's address */
uint32 * address = calculate_address(getuid(),index[0], previous_ res);
func = address;
/* execute functional code */
func();
for(i = 1 ; i < N; i ++ )
{
    /* calculate previous code's completeness */
    previous_ res = calculate_code_ completeness (address,0x1000);
    /* calculate next functional code's address */
    address = calculate_address(getuid(),index[i], previous_ res);
    func = address;
    /* execute functional code */
    func();
}
```

3.4. Add redundant code

This section describes how to increase the complexity of the code by adding redundant code. The redundant code need to bind with the UID, the result of redundant code execution will affect the main code. If the result is abnormal, the main code will run abnormally. This section will describes two method to achieve it.

3.4.1. Reference Implementation 1

Add special function code in software programming. The example is shown as ***Table 3-3. The example of special function code***.

Table 3-3. The example of special function code

function	return	Implementation
fun0	0	Read flash address 0 and do operation with UID
fun1	1	Read flash address 1 and do operation with UID
fun2	2	Read flash address 2 and do operation with UID
fun3	3	Read flash address 3 and do operation with UID
….	…	…
funn	n	Read flash address n and do operation with UID

These functions return different value, when the value need to be called during program design, it is obtained indirectly by calling these functions. The flowchart is shown as ***Figure 3-4. Implementation flowchart of redundant code***.

In the case of no special function code, the normal code logic is simple and easy to understand. The example is shown as ***Table 3-4. The example of common code***.

Table 3-4. The example of common code

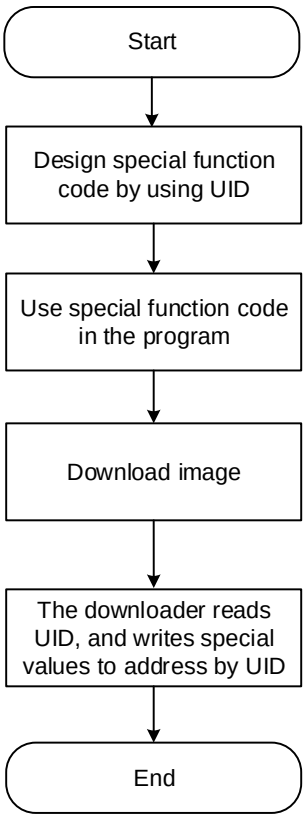
```
/* execute test code */
uint8_t *buf = NULL;
for ( i = 0; i < 10; i++ ){
    printf("%d\r\n", i );
}
buf = (uint8_t *)malloc(200);
```

After adding special function, the program is bound to the chip UID. The example is shown as ***Table 3-5. The example of adding special function***.

Table 3-5. The example of adding special function

```
/* execute test code */
uint8_t *buf = NULL;
for ( i = 0; i < 10 * func1() ; i++ ){
    printf("%d\r\n", i );
}
buf = (uint8_t *)malloc(100*func2( ));
```

Figure 3-4. Implementation flowchart of redundant code

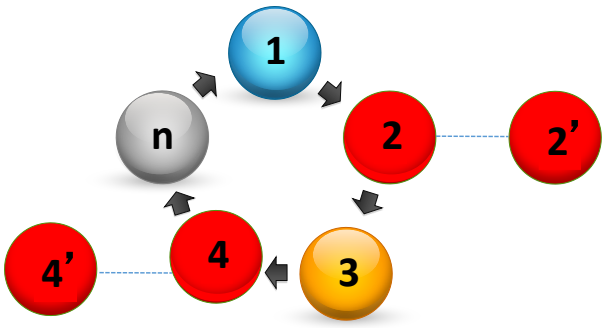


3.4.2. Reference Implementation 2

This method increase the complexity of the code by adding pairs of bug and patching bug

code. After the bug code is executed, the patched bug code must be executed, otherwise the program will be abnormal. Run the bug code and patching bug code according to the chip UID and random number. This method is shown as *Figure 3-5. Bug and patching bug diagram*.

Figure 3-5. Bug and patching bug diagram



The example of bug code is shown as *Table 3-6. The example of bug code* .

Table 3-6. The example of bug code

```
/* bug code */
int flag1 = 0;
void fun1(void)
{
    if (1 == flag1){
        flag1 = 0;
        *((uint32_t *)0) = 3;
        return;
    }
    if (0 == flag1){
        flag1 = 1;
    }
    else if (2 == flag1){
        flag1 = 0;
    }
}
```

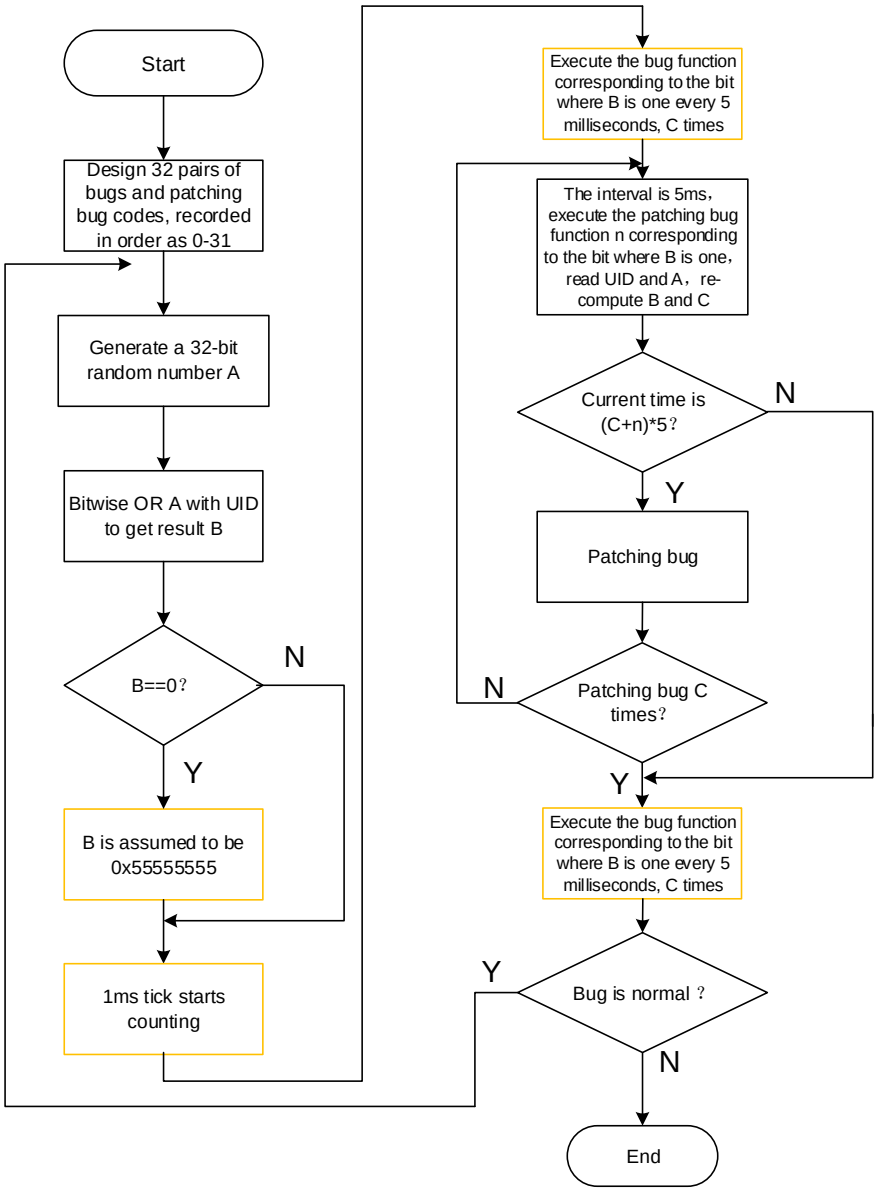
The example of patching bug code is shown as *Table 3-7. The example of patching bug code example*.

Table 3-7. The example of patching bug code example

```
/* fixs bug code */
void fun11(void)
{
    /* time check right */
    flag1 = 2;
}
```

The flowchart of adding bug and patching bug is shown as ***Figure 3-6. The flowchart of adding bug and patching bug***.

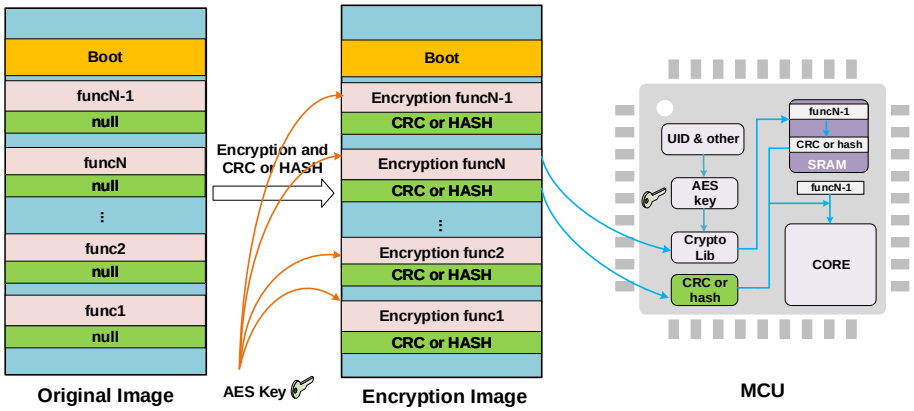
Figure 3-6. The flowchart of adding bug and patching bug



3.5. Software encryption

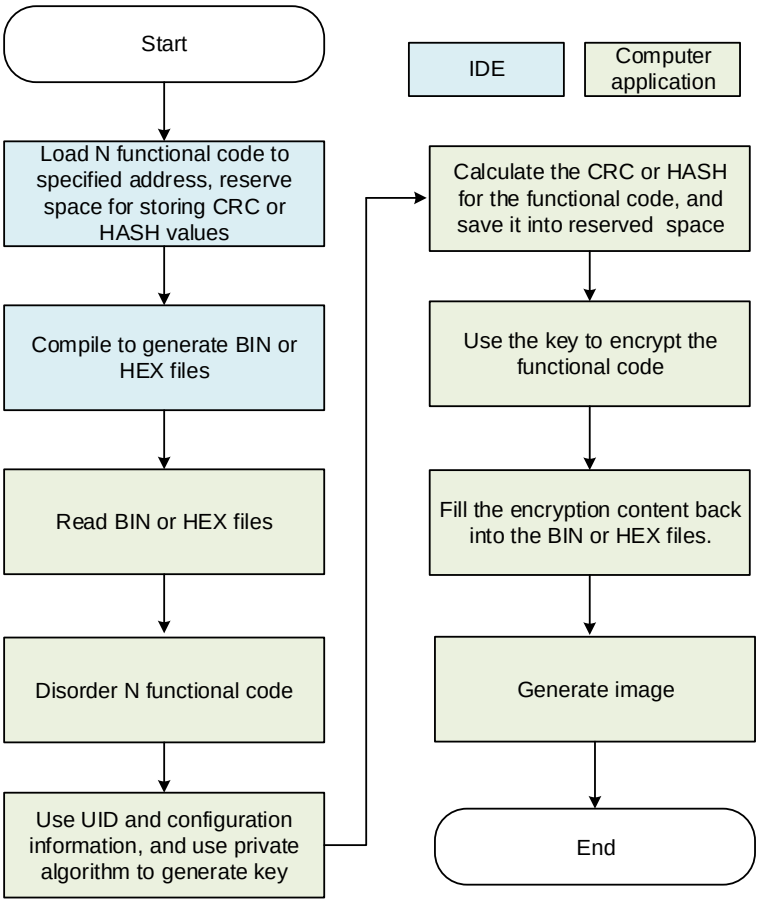
This section describes how to increase the complexity of code by using software encryption and decryption algorithm. The algorithm can be used to encrypt and decrypt functional code, generate the key by using UID and user-defined configuration information. The encryption and decryption algorithm can use open source library, such as the mbedTLS encryption algorithm library for small embedded device. This method is shown as ***Figure 3-7. Schematic of software encryption and decryption***.

Figure 3-7. Schematic of software encryption and decryption



The computer application calculate the CRC or HASH value of each functional code, and store the result in the reserved space. Then, each functional code is encrypted with an AES key which haved generated by using UID and user-defined configuration information, and fill back into BIN or HEX file. Finally, download the encrypted image to the chip. The image generate is shown as ***Figure 3-8. Encrypted image generation flow chart***.

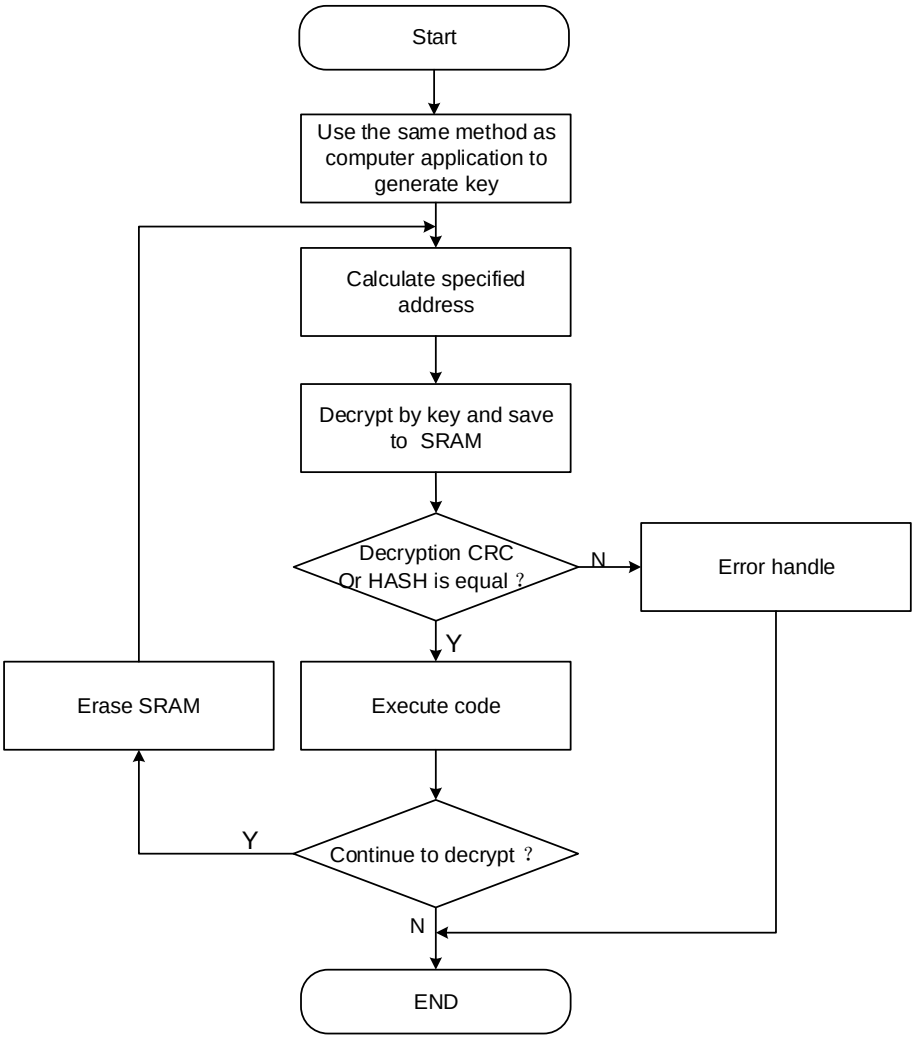
Figure 3-8. Encrypted image generation flow chart



When execute boot code，it generate AES key by using the same method as computer application, and decrypt the encrypted image, calculate the decrypted image's CRC or HASH value. The flowchart is shown as ***Figure 3-9．Decrypt Image flow chart***.

Although the boot is not encrypted, it will increase the difficulty of disassembly parsing due to the algorithmic calculation. It should be noted that the AES key can be transimit by DMA or interrupt.

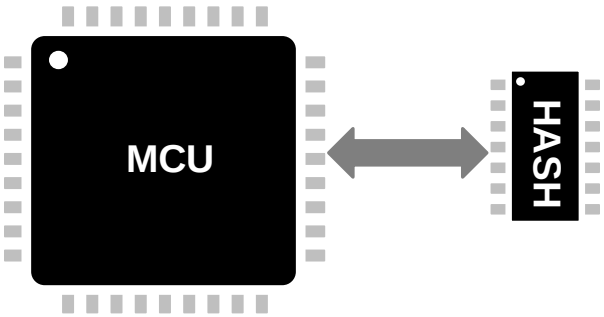
Figure 3-9. Decrypt Image flow chart



3.6. Encryption chip

This section describes how to protect code by using encryption chip. MCU connect to hash chip, which utilize the storage security of the hash chip.Hash chip can identify the legitimacy of the request. This schematic is shown as ***Figure 3-10. Schematic diagram of HASH chip connected to MCU***.

Figure 3-10. Schematic diagram of HASH chip connected to MCU

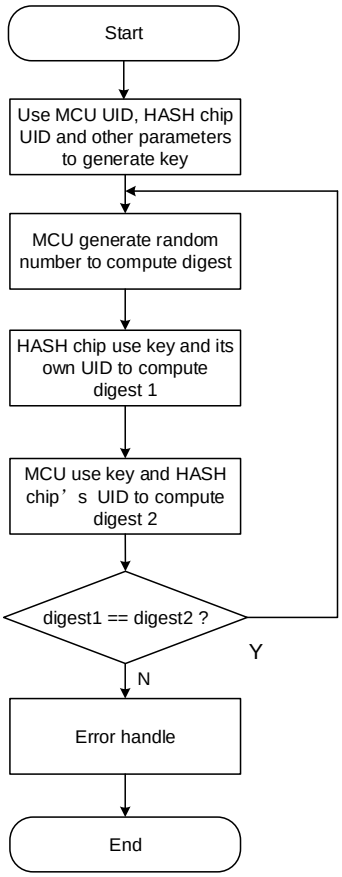


The certification process is shown as

Figure 3-11. Certification flow **chart**.

Firstly, the computer application will use the UID of MCU, the UID of HASH chip and the custom parameters to derive the secret key for calculating the digest, and write digest into the HASH chip. When the MCU program runs, it will generate a random number and send it to the HASH chip, the HASH chip uses the random number, the derived key and its own UID to calculate the digest 1. Meanwhile, MCU uses the random number, the derived key and the UID of the HASH chip to calculate the digest 2. MCU will compares the two digests, If it is different, it will enter the exception handling code, which enter an infinite loop or other operations. Since HASH has the feature that it cannot be changed after writing, it can achieve the effect of binding a HASH chip to a device.

Figure 3-11. Certification flow chart



4. Revision history

Table 4-1. Revision history

Revision No.	Description	Date
1.0	Initial Release	Oct.28 2022

Important Notice

This document is the property of GigaDevice Semiconductor Inc. and its subsidiaries (the "Company"). This document, including any product of the Company described in this document (the “Product”), is owned by the Company under the intellectual property laws and treaties of the People’s Republic of China and other jurisdictions worldwide. The Company reserves all rights under such laws and treaties and does not grant any license under its patents, copyrights, trademarks, or other intellectual property rights. The names and brands of third party referred thereto (if any) are the property of their respective owner and referred to for identification purposes only.

The Company makes no warranty of any kind, express or implied, with regard to this document or any Product, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Company does not assume any liability arising out of the application or use of any Product described in this document. Any information provided in this document is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Except for customized products which has been expressly identified in the applicable agreement, the Products are designed, developed, and/or manufactured for ordinary business, industrial, personal, and/or household applications only. The Products are not designed, intended, or authorized for use as components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, atomic energy control instruments, combustion control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or Product could cause personal injury, death, property or environmental damage ("Unintended Uses"). Customers shall take any and all actions to ensure using and selling the Products in accordance with the applicable laws and regulations. The Company is not liable, in whole or in part, and customers shall and hereby do release the Company as well as it’s suppliers and/or distributors from any claim, damage, or other liability arising from or related to all Unintended Uses of the Products. Customers shall indemnify and hold the Company as well as it’s suppliers and/or distributors harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of the Products.

Information in this document is provided solely in connection with the Products. The Company reserves the right to make changes, corrections, modifications or improvements to this document and Products and services described herein at any time, without notice.