# 4.1 安全散列算法加速器

一、实验目的

本节课主要学习 K210 芯片中的安全散列算法加速器的功能。

二、实验准备

1.实验元件

K210 芯片中的核心 0 和核心 1

2.元件特性

SHA256 加速器是用来计算 SHA-256 的计算单元,SHA256 是 SHA-2 下细分出的一种算法。SHA-2，名称来自于安全散列算法 2（英语：Secure Hash Algorithm 2）的缩写，一种密码散列函数算法标准，由美国国家安全局研发，属于 SHA 算法之一，是 SHA-1 的后继者。SHA-2 下又可再分为六个不同的算法标准，包括了：SHA-224、SHA-256、SHA-384、SHA-512、SHA-512/224、SHA-512/256。简单来说，SHA-256 是一个哈希函数，也就是散列算法，它可以从任何一种数据中创建小的数字“指纹”，然后把消息或者数据打乱混合并且压缩成摘要，使数据量变小，固定数据的格式，重新创建一个叫做散列值（或哈希值）的指纹，散列值通常是一个短的随机字母和数字组成的字符串来表示。经过 SHA256 算法处理过后都会产生一个 256bit 长度的哈希值，称为消息摘要。这个摘要相当于是个长度为 32 个字节的数组，通常由一个长度为 64 的十六进制字符串来表示，其中 1 个字节=8 位，一个十六进制的字符的长度为 4 位。

SHA-2 下的不同算法标准只是生成摘要的长度、循环运行的次数等微小差异，

基本结构算法是一致的。

3.SDK 中对应 API 功能

对应的头文件 sha256.h

支持 SHA-256 的计算。

为用户提供以下接口：

- sha256_init：初始化 SHA256 加速器外设。
- sha256_update：传入一个数据块参与 SHA256 Hash 计算。
- sha256_final：结束对数据的 SHA256 Hash 计算。
- sha256_hard_calculate：一次性对连续的数据计算它的 SHA256 Hash。

三、实验原理

1.常量初始化

SHA256 算法中用到了 8 个哈希初值以及 64 个哈希常量。

其中，SHA256 算法的 8 个哈希初值如下：

h0 := 0x6a09e667

h1 := 0xbb67ae85

h2 := 0x3c6ef372

h3 := 0xa54ff53a

h4 := 0x510e527f

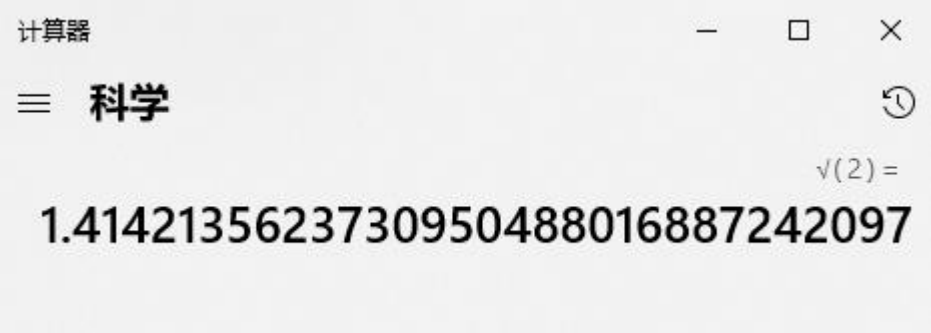
h5 := 0x9b05688c

h6 := 0x1f83d9ab

h7 := 0x5be0cd19

这些初值是对自然数中前 8 个质数（2,3,5,7,11,13,17,19）的平方根的小数部分取前 32bit 而来。以 H0 的由来为例，根号 2（$\sqrt{2}$）的小数点部分约为 0.

$1.41421356237309504880168872420 97 \approx 6 * 16^{-1} + a * 16^{-2} + 0 * 16^{-3} \ldots$




所以，质数 2 的平方根的小数部分取前 32bit 就对应是 0x6a09e667。

在 SHA256 算法中，用到的 64 个哈希常量如下：

428a2f98 71374491 b5c0fbcf e9b5dba5

3956c25b 59f111f1 923f82a4 ab1c5ed5

d807aa98 12835b01 243185be 550c7dc3

72be5d74 80deb1fe 9bdc06a7 c19bf174

e49b69c1 efbe4786 0fc19dc6 240ca1cc

2de92c6f 4a7484aa 5cb0a9dc 76f988da

983e5152 a831c66d b00327c8 bf597fc7

c6e00bf3 d5a79147 06ca6351 14292967

27b70a85 2e1b2138 4d2c6dfc 53380d13

650a7354 766a0abb 81c2c92e 92722c85

a2bfe8a1 a81a664b c24b8b70 c76c51a3

d192e819 d6990624 f40e3585 106aa070

19a4c116 1e376c08 2748774c 34b0bcb5

391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3

748f82ee 78a5636f 84c87814 8cc70208

90befffa a4506ceb bef9a3f7 c67178f2

和 8 个哈希初值类似，这些常量是对自然数中前 64 个质数(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97…)的立方根的小数部分取前 32bit 而来。

2.信息预处理(pre-processing)

SHA256 算法中的预处理就是在想要 Hash 的消息后面补充需要的信息，使整个消息满足指定的结构。

信息的预处理分为两个步骤：附加填充比特和附加长度。

步骤一：附加填充比特

在报文末尾进行填充，使报文长度在对 512 取模以后的余数是 448

填充是这样进行的：先补第一个比特为 1，其他后面都补 0，直到长度满足对 512 取模后余数是 448。

需要注意的是，信息必须进行填充，也就是说，即使长度已经满足对 512 取模后余数是 448，补位也必须要进行，这时要填充 512 个比特。

因此，填充是至少补一位，最多补 512 位。

例：以信息“abc”为例显示补位的过程。

a,b,c 对应的 ASCII 码分别是 97,98,99

于是原始信息的二进制编码为：01100001 01100010 01100011

补位第一步，首先补一个“1” ： 01100001011000100 01100011 1

补位第二步，补 423 个“0”：01100001 01100010 01100011 10000000 00000000 … 00000000

补位完成后的数据如下（为了简介用 16 进制表示）：

61626380 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000

为什么是 448?

因为在第一步的预处理后，第二步会再附加上一个 64bit 的数据，用来表示原始报文的长度信息。而 448+64=512，正好拼成了一个完整的结构。

步骤二：附加长度值

附加长度值就是将原始数据（第一步填充前的消息）的长度信息补到已经进行了填充操作的消息后面。

SHA256 用一个 64 位的数据来表示原始消息的长度。

因此，通过 SHA256 计算的消息长度必须要小于 2^64 ，当然绝大多数情况这足够大了。

长度信息的编码方式为 64-bit big-endian integer

关于 Big endian 的含义，文末给出了补充

回到刚刚的例子，消息“abc”，3 个字符，占用 24 个 bit

因此，在进行了补长度的操作以后，整个消息就变成下面这样了（16 进制格式）

61626380 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000018

3. 逻辑运算

SHA256 散列函数中涉及的操作全部是逻辑的位运算，包括如下的逻辑函数：

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$

$$Ma(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\Sigma_0(x) = S^2(x) \oplus S^{13}(x) \oplus S^{22}(x)$$

$$\Sigma_1(x) = S^6(x) \oplus S^{11}(x) \oplus S^{25}(x)$$

$$\sigma_0(x) = S^7(x) \oplus S^{18}(x) \oplus R^3(x)$$

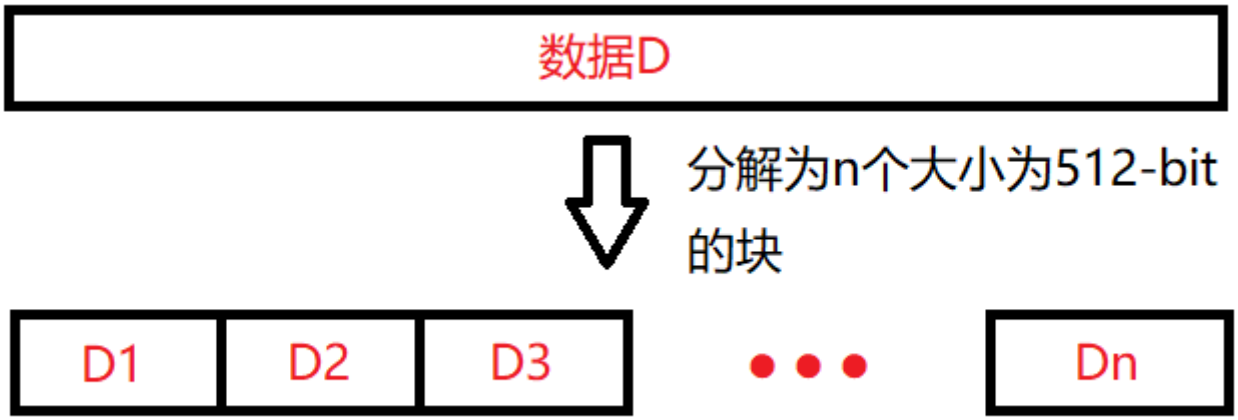
$$\sigma_1(x) = S^{17}(x) \oplus S^{19}(x) \oplus R^{10}(x)$$

| 逻辑运算 | 含义 |
| --- | --- |
| $\wedge$ | 按位“与” |
| $\neg$ | 按位“补” |
| $\oplus$ | 按位“异或” |
| $S^n$ | 循环右移 n 个 bit |
| $R^n$ | 右移 n 个 bit |

4. 计算消息摘要

以下 SHA256 算法的主体部分，即消息摘要，是如何计算的。

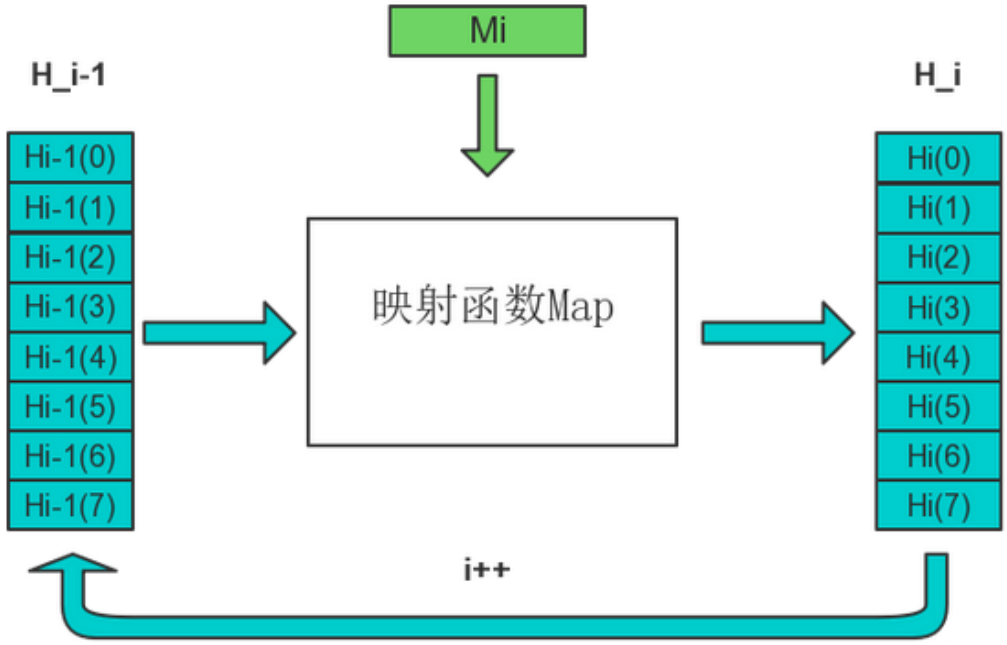
首先：将消息分解成 512-bit 大小的块




假设数据 D 可以被分解为 n 个块，于是整个算法需要做的就是完成 n 次迭代，n 次迭代的结果就是最终的哈希值，即 256bit 的数字摘要。

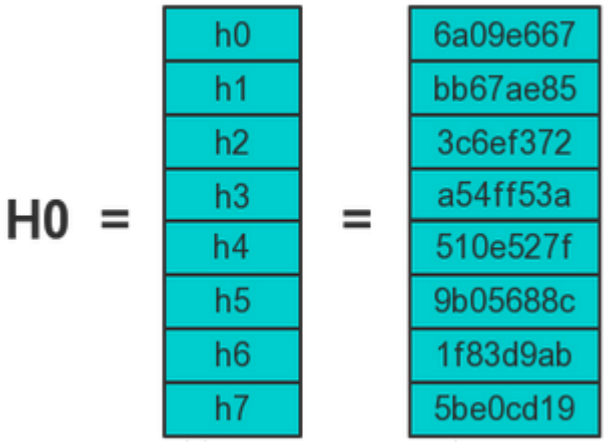
一个 256-bit 的摘要的初始值 H0，经过第一个数据块进行运算，得到 H1，即完成了第一次迭代，H1 经过第二个数据块得到 H2，……，依次处理，最后得到 Hn，Hn 即为最终的 256-bit 消息摘要。

将每次迭代进行的映射用$ Map(H_{i-1}) = H_{i} $表示，于是迭代可以更形象的展示为：



图中 256-bit 的 Hi 被描述 8 个小块，这是因为 SHA256 算法中的最小运算单元称为“字”（Word），一个字是 32 位。

此外，第一次迭代中，映射的初值设置为前面介绍的 8 个哈希初值，如下图所示：




下面开始介绍每一次迭代的内容，即映射$ Map(H_{i-1}) = H_{i} $的具体算法。

STEP1：构造 64 个字（word）

对于每一块，将块分解为 16 个 32-bit 的 big-endian 的字，记为 w[0], …, w[15]

也就是说，前 16 个字直接由消息的第 i 个块分解得到

其余的字由如下迭代公式得到：

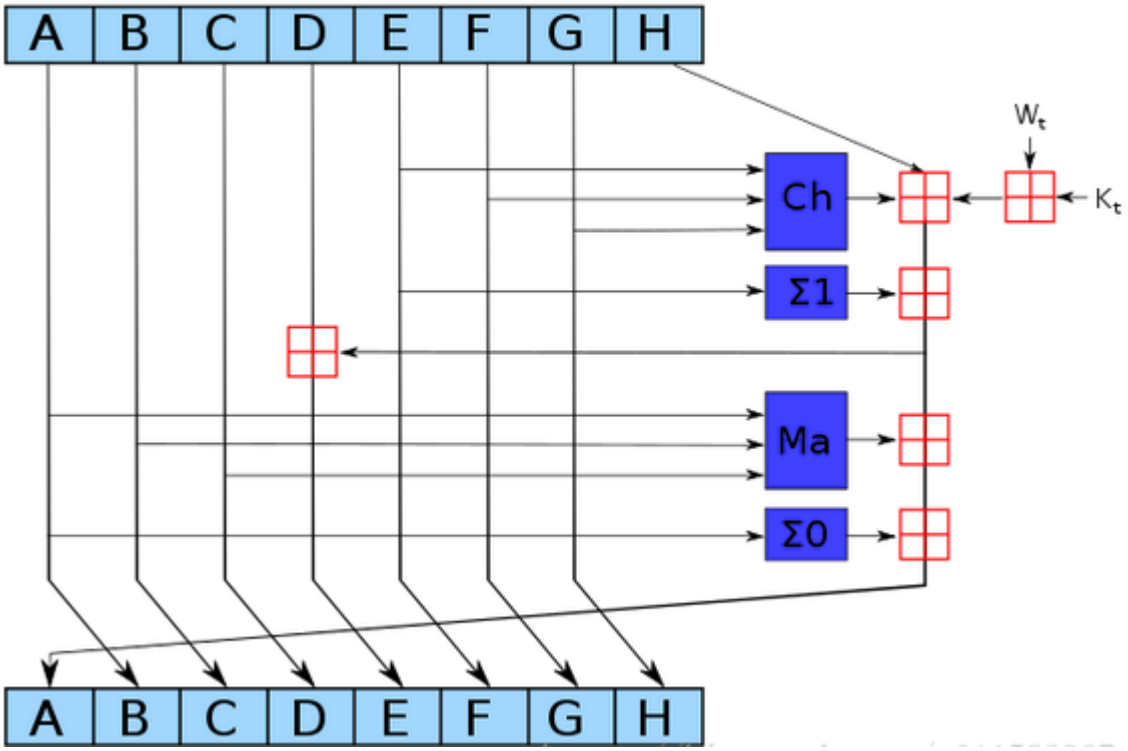
$$W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$$

STEP2：进行 64 次循环

映射 $ Map(H_{i-1}) = H_{i} $ 包含了 64 次加密循环

即进行 64 次加密循环即可完成一次迭代

每次加密循环可以由下图描述：




图中，ABCDEFGH 这 8 个字（word）在按照一定的规则进行更新，其中深蓝色方块是事先定义好的非线性逻辑函数。红色田字方块代表 mod $ 2^{32} $ addition，即将两个数字加在一起，如果结果大于$ 2^{32} ，你必须除以 2^{32} $ 并找到余数。ABCDEFGH 一开始的初始值分别为 $ H_{i-1}(0),H_{i-1}(1),\cdots,H_{i-1}(7) $。

Kt 是第 t 个密钥，对应我们上文提到的 64 个常量。

Wt 是本区块产生第 t 个 word。原消息被切成固定长度 512-bit 的区块，对每一个区块，产生 64 个 word，通过重复运行循环 n 次对 ABCDEFGH 这八个字循环加密。

最后一次循环所产生的八个字合起来即是第 i 个块对应到的散列字符串 $ H_{i} $。

## 四、实验过程

1. 计算字符串‘abc’的哈希值，并且打印出来。

```
uint32_t i;
printf("\n");
cycle = read_cycle();
sha256_hard_calculate((uint8_t *)"abc", 3, hash);
for (i = 0; i < SHA256_HASH_LEN;)
{
    if (hash[i] != compare1[i])
        total_check_tag = 1;
    printf("%02x", hash[i++]);
    if (!(i % 4))
        printf(" ");
}
printf("\n");
```

2. 计算字符串‘abcdefghabcdefghabcdefghabcdefghabcdefghabcdefghabcdefghabcdefgha’ 的哈希值，并且打印出来。

```
sha256_hard_calculate((uint8_t *)"abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij", 60, hash);
for (i = 0; i < SHA256_HASH_LEN;)
{
    if (hash[i] != compare2[i])
        total_check_tag = 1;
    printf("%02x", hash[i++]);
    if (!(i % 4))
        printf(" ");
}
printf("\n");
```

3. 计算字符串‘abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij’的哈希值，并且打印出来。

```
sha256_hard_calculate((uint8_t *)"abcdefghabcdefghabcdefghabcdefghabcdefghabcdefghabcdefghabcdefgha", 65, hash);
for (i = 0; i < SHA256_HASH_LEN;)
{
    if (hash[i] != compare3[i])
        total_check_tag = 1;
    printf("%02x", hash[i++]);
    if (!(i % 4))
        printf(" ");
}
printf("\n");
```

4. 计算多个字符‘a’的哈希值，并且打印出来。

```
memset(data_buf, 'a', sizeof(data_buf));
sha256_hard_calculate(data_buf, sizeof(data_buf), hash);
for (i = 0; i < SHA256_HASH_LEN;)
{
    if (hash[i] != compare4[i])
        total_check_tag = 1;
    printf("%02x", hash[i++]);
    if (!(i % 4))
        printf(" ");
}
printf("\n");
```

5.分块计算多个字符‘a’的哈希值，并打印出来。

```
sha256_context_t context;
sha256_init(&context, sizeof(data_buf));
sha256_update(&context, data_buf, 1111);
sha256_update(&context, data_buf + 1111, sizeof(data_buf) - 1111);
sha256_final(&context, hash);
for (i = 0; i < SHA256_HASH_LEN;)
{
    if (hash[i] != compare4[i])
        total_check_tag = 1;
    printf("%02x", hash[i++]);
    if (!(i % 4))
        printf(" ");
}
printf("\n");
```

6.编译调试，烧录运行

把本课程资料中的 sha256 复制到 SDK 中的 src 目录下，然后进入 build 目录，运行以下命令编译。

cmake .. -DPROJ=sha256 -G "MinGW Makefiles"

make

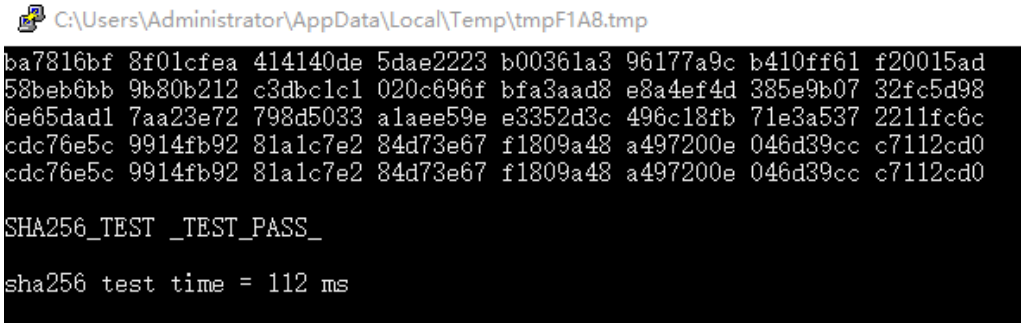
```
[ 97%] Building C object CMakeFiles/sha256.dir/src/sha256/main.c.obj
[100%] Linking C executable sha256
Generating .bin file ...
[100%] Built target sha256
PS C:\K210\SDK\kendryte-standalone-sdk-develop\build>
```

编译完成后，在 build 文件夹下会生成 sha256.bin 文件。

使用 type-C 数据线连接电脑与 K210 开发板，打开 kflash，选择对应的设备，再将程序固件烧录到 K210 开发板上。

五、实验现象

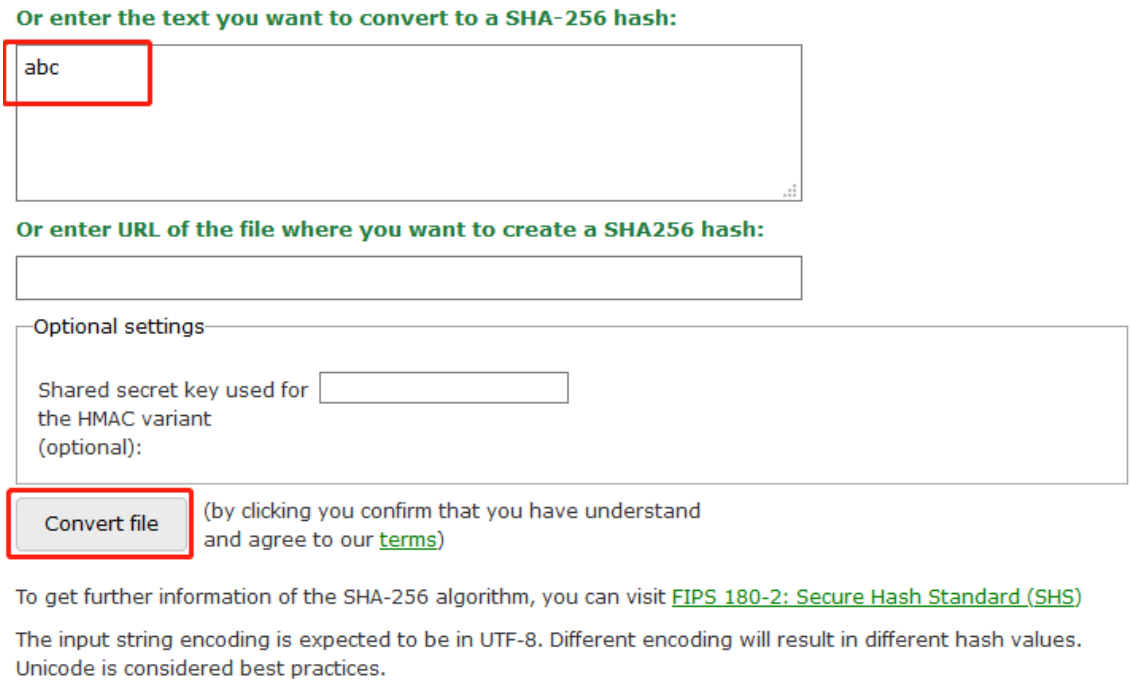
烧录固件完成后，系统会自动弹出一个终端窗口，并且打印每一步计算出的哈希值。




以下是一个在线的 SHA256 验证网站，可以到里面输入验证：

https://hash.online-convert.com/sha256-generator

在以下红色方框中输入想要转化的字符串，然后点击 Convert file。可能存在网络比较慢的问题，需要耐心等待完成。



等待转化成功后，就可以看到摘要的哈希值，与上面程序中第一行的哈希值对比，是完全一致的。


六、实验总结

1.SHA256 是 SHA-2 加密系统中的一员，并且所有的 SHA-2 成员的基础加密算法都是一致的，只是生成摘要的长度和循环次数不同。

2.SHA256 每次生成的摘要为 256bit。

3.SHA256 是目前安全散列算法中应用比较广的一种。

附：API

对应的头文件 sha256.h

### sha256_init

**描述**

初始化 SHA256 加速器外设.

**函数原型**

```
void sha256_init(sha256_context_t *context, size_t input_len)
```

**参数**

| 参数名称 | 描述 | 输入输出 |
| --- | --- | --- |
| context | SHA256 的上下文对象 | 输入 |
| input_len | 待计算 SHA256 hash 的消息的长度 | 输入 |

### 返回值

无。

### 举例

```
sha256_context_t context;
sha256_init(&context, 128U);
```

## sha256_update

### 描述

传入一个数据块参与 SHA256 Hash 计算

### 函数原型

```
void sha256_update(sha256_context_t *context, const void *input, size_t
input_len)
```

### 参数

| 参数名称 | 描述 | 输入输出 |
| --- | --- | --- |
| context | SHA256 的上下文对象 | 输入 |
| input | 待加入计算的 SHA256 计算的数据块 | 输入 |
| buf_len | 待加入计算的 SHA256 计算数据块的长度 | 输入 |

### 返回值

无。

## sha256_final

### 描述

结束对数据的 SHA256 Hash 计算

### 函数原型

```
void sha256_final(sha256_context_t *context, uint8_t *output)
```

### 参数

| 参数名称 | 描述 | 输入输出 |
| --- | --- | --- |
| context | SHA256 的上下文对象 | 输入 |
| output | 存放 SHA256 计算的结果，需保证传入这个 buffer 的大小为 32Bytes 以上 | 输出 |

### 返回值

无。

## sha256_hard_calculate

### 描述

一次性对连续的数据计算它的 SHA256 Hash

### 函数原型

```
void sha256_hard_calculate(const uint8_t *input, size_t input_len, uint8_t *output)
```

### 参数

| 参数名称 | 描述 | 输入输出 |
| --- | --- | --- |
| input | 待 SHA256 计算的数据 | 输入 |
| input_len | 待 SHA256 计算数据的长度 | 输入 |
| output | 存放 SHA256 计算的结果，需保证传入这个 buffer 的大小为 32Bytes 以上 | 输出 |

### 返回值

无。

## 举例

```
uint8_t hash[32];
sha256_hard_calculate((uint8_t *)"abc", 3, hash);
```

# 例程

## 进行一次计算

```
sha256_context_t context;
sha256_init(&context, input_len);
sha256_update(&context, input, input_len);
sha256_final(&context, output);
```

或者可以直接调用 sha256_hard_calculate 函数

```
sha256_hard_calculate(input, input_len, output);
```

## 进行分块计算

```
sha256_context_t context;
sha256_init(&context, input_piece1_len + input_piece2_len);
sha256_update(&context, input_piece1, input_piece1_len);
sha256_update(&context, input_piece2, input_piece2_len);
sha256_final(&context, output);
```