




MD5 algorithm in C

Research
G00346996



MD5 Introduction

- The MD5 algorithm was developed by Professor Ronald L. Rivest in 1991
- The MD5 “Message digest algorithm” takes as input a message of arbitrary length and produces as output a 128-bit “fingerprint” or else a “message digest” of the input.
- There are 5 steps to follow in order to compute the message digest of the message
- **Step 1.** Append Padding bits: the input message is “padded” so that its length in bits equals to $448 \bmod 512$. Padding is always performed, even if the length of the message is already $448 \bmod 512$. Padding is performed as follows: a single “1” bit is appended to the message, and then “0” bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. In all, at least one bit and at most 512 bits are appended.
- **Step 2.** Append Length: A 64-bit representation of the length of the message is appended to the result of step 1. If the length of the message is greater than 2^{64} , only the low-order 64 bits will be used. The resulting message (after padding with bits) has a length that is an exact multiple of 512 bits. The input will have a length that is an exact multiple of 16 (32-bit) words.
- **Step 3.** Initialize MD Buffer: A four-word buffer (A, B, C, D) is used to compute the message digest. Each of A, B, C, D is a 32-bit register. These registers are initialized to the following values in hexadecimal

Word A : 01 23 45 67

Word B : 89 ab cd ef

Word C : fe dc ba 98

Word D : 76 54 32 10

- **Step 4 :** Process message in 16 word blocks. Four functions will be defined such that each function takes an input of three 32 bit words and produces a 32-bit words and produces a 32-bit word output.

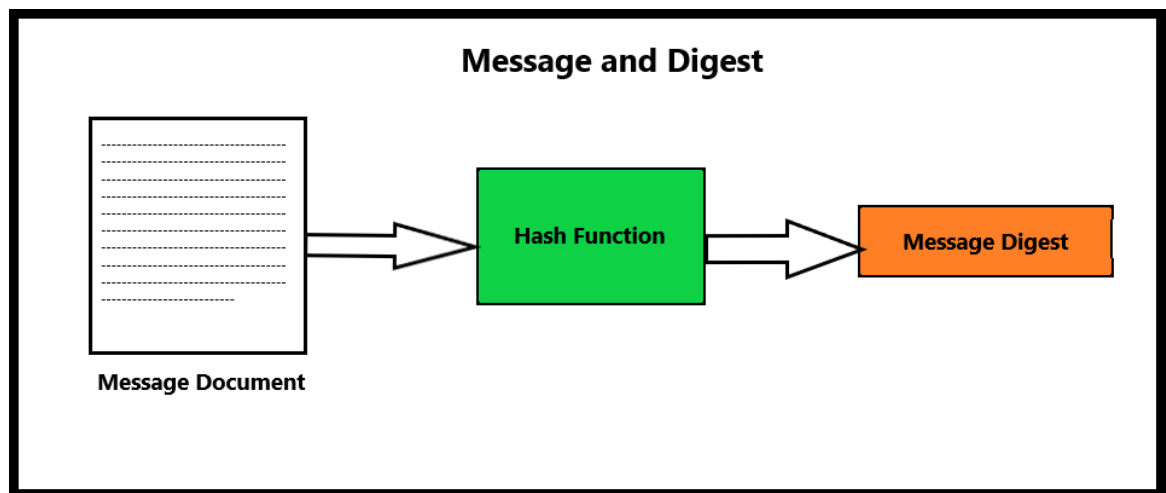
$F(X, Y, Z) == (X \wedge Y) \vee (\sim X \wedge Z)$ Round 1

$F(X, Y, Z) == (X \wedge Z) \vee (Y \wedge \sim Z)$ Round 2

$F(X, Y, Z) == X \oplus Y \oplus Z$ Round 3

$F(X, Y, Z) == Y \oplus (X \wedge \sim Z)$ Round 4

- **Step 5:** Output. The message digest produced as output is A, B C, D. That is, we begin with lower-order byte of A, and end with the high order byte of D



MD5 algorithm possible solutions

```
/*
 * Simple MD5 implementation
 *
 * Compile with: gcc -o md5 -O3 -lm md5.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

// leftrotate function definition
#define LEFTROTATE(x, c) (((x) << (c)) | ((x) >> (32 - (c))))

// These vars will contain the hash
uint32_t h0, h1, h2, h3;

void md5(uint8_t *initial_msg, size_t initial_len) {
    // Message (to prepare)
    uint8_t *msg = NULL;

    // Note: All variables are unsigned 32 bit and wrap modulo 2^32 when calculating
    // r specifies the per-round shift amounts
    uint32_t r[] = {7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22,
5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20,
4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23,
6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21};

    // Use binary integer part of the sines of integers (in radians) as constants// Initialize variables:
    uint32_t k[] = {
0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee,
0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501,
0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be,
0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821,
0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa,
0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8,
0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed,
0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a,
0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c,
0xa4beeaa4, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70,
0x289b7ec6, 0xeeaa127fa, 0xd4ef3085, 0x04881d05,
0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665,
0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039,
0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1,
```

```

0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1,
0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391};
h0 = 0x67452301;
h1 = 0xefcdab89;
h2 = 0x98badcfe;
h3 = 0x10325476;
// Pre-processing: adding a single 1 bit
//append "1" bit to message
/* Notice: the input bytes are considered as bits strings,
where the first bit is the most significant bit of the byte.[37] */
// Pre-processing: padding with zeros
//append "0" bit until message length in bit  $\equiv 448 \pmod{512}$ 
//append length mod (2 pow 64) to message
int new_len = (((initial_len + 8) / 64) + 1) * 64 - 8;
msg = calloc(new_len + 64, 1); // also appends "0" bits
// (we alloc also 64 extra bytes...)
memcpy(msg, initial_msg, initial_len);
msg[initial_len] = 128; // write the "1" bit
uint32_t bits_len = 8*initial_len; // note, we append the len
memcpy(msg + new_len, &bits_len, 4); // in bits at the end of the buffer
// Process the message in successive 512-bit chunks:
//for each 512-bit chunk of message:
int offset;
for(offset=0; offset<new_len; offset += (512/8)) {
// break chunk into sixteen 32-bit words w[j],  $0 \leq j \leq 15$ 
uint32_t *w = (uint32_t *) (msg + offset);
#ifdef DEBUG
printf("offset: %d %x\n", offset, offset);
int j;
for(j=0; j < 64; j++) printf("%x ", ((uint8_t *) w)[j]);
puts("");
#endif
// Initialize hash value for this chunk:
uint32_t a = h0;
uint32_t b = h1;
uint32_t c = h2;
uint32_t d = h3;
// Main loop:
uint32_t i;
for(i = 0; i<64; i++) {
#ifdef ROUNDS
uint8_t *p;
printf("%i: ", i);

```

```

p=(uint8_t *)&a;
printf("%2.2x%2.2x%2.2x%2.2x ", p[0], p[1], p[2], p[3], a);
p=(uint8_t *)&b;
printf("%2.2x%2.2x%2.2x%2.2x ", p[0], p[1], p[2], p[3], b);
p=(uint8_t *)&c;
printf("%2.2x%2.2x%2.2x%2.2x ", p[0], p[1], p[2], p[3], c);
p=(uint8_t *)&d;
printf("%2.2x%2.2x%2.2x%2.2x", p[0], p[1], p[2], p[3], d);
puts("");
#endif
uint32_t f, g;
if (i < 16) {
f = (b & c) | ((~b) & d);
g = i;
} else if (i < 32) {
f = (d & b) | ((~d) & c);
g = (5*i + 1) % 16;
} else if (i < 48) {
f = b ^ c ^ d;
g = (3*i + 5) % 16;
} else {
f = c ^ (b | (~d));
g = (7*i) % 16;
}
#ifdef ROUNDS
printf("f=%x g=%d w[g]=%x\n", f, g, w[g]);
#endif
uint32_t temp = d;
d = c;
c = b;
printf("rotateLeft(%x + %x + %x + %x, %d)\n", a, f, k[i], w[g], r[i]);
b = b + LEFTROTATE((a + f + k[i] + w[g]), r[i]);
a = temp;
}
// Add this chunk's hash to result so far:
h0 += a;
h1 += b;
h2 += c;
h3 += d;
}
// cleanup
free(msg);
}

```

```

int main(int argc, char **argv) {
    if (argc < 2) {
        printf("usage: %s 'string'\n", argv[0]);
        return 1;
    }
    char *msg = argv[1];
    size_t len = strlen(msg);
    // benchmark
    // int i;
    // for (i = 0; i < 1000000; i++) {
        md5(msg, len);
    // }
    //var char digest[16] := h0 append h1 append h2 append h3 //(Output is in little-endian)
    uint8_t *p;
    // display result
    p=(uint8_t *)&h0;
    printf("%2.2x%2.2x%2.2x%2.2x", p[0], p[1], p[2], p[3], h0);
    p=(uint8_t *)&h1;
    printf("%2.2x%2.2x%2.2x%2.2x", p[0], p[1], p[2], p[3], h1);
    p=(uint8_t *)&h2;
    printf("%2.2x%2.2x%2.2x%2.2x", p[0], p[1], p[2], p[3], h2);
    p=(uint8_t *)&h3;
    printf("%2.2x%2.2x%2.2x%2.2x", p[0], p[1], p[2], p[3], h3);
    puts("");
    return 0;
}

```

This was a possible solution that I had seen online, I looked through this solution very carefully to understand what was with the code in the various parts. It indeed followed all the steps that I have listed in the introduction

- It performs the leftrotate of the bits
- It performs the different rounds that's it intended to do
- It carries out the auxiliary functions
- It has specified the shift amounts in a [] of 64
- It uses a buffer that is made up of four **words** that are each 32 bits long (h0, h1, h2, h3)
- It also provides an output of the digest