LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-434

THE MD4 MESSAGE DIGEST ALGORITHM

Ronald L. Rivest

October 1990

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

The MD4 Message Digest Algorithm

Ronald L. Rivest*
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

The MD4 message digest algorithm takes an input message of arbitrary length and produces an output 128-bit "fingerprint" or "message digest", in such a way that it is (hopefully) computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest. The MD4 algorithm is thus ideal for digital signature applications: a large file can be securely "compressed" with MD4 before being signed with (say) the RSA public-key cryptosystem.

The MD4 algorithm is designed to be quite fast on 32-bit machines. For example, on a SUN Sparc station, MD4 runs at 1,450,000 bytes/second (11.6 Mbit/sec). In addition, the MD4 algorithm does not require any large substitution tables; the algorithm can be coded quite compactly.

The MD4 algorithm is being placed in the public domain for review and possible adoption as a standard.

1 Introduction

One-way functions were first described in the mid-1970's, as a means to avoid having to store passwords in a time-shared computer system. (This idea is due to Needham (see Wilkes [10, page 91]) and Evans, Kantrowitz, and Weiss [4].)

In 1976, Diffie and Hellman [3] began the exploration of the relationship between one-way functions and other kinds of cryptographic operations, a theme that continues to the present day. They also propose the operation of exponentiating modulo a prime as a candidate one-way function, attributing this idea to John Gill.

A major application of one-way functions was given by Davies and Price in 1980 [2]. They introduce the idea of signing a message M by signing $h(M)$ with a public-key cryptosystem, where h is a one-way function. This procedure has the advantage of permitting considerable improvement in efficiency if M is long, since computing

*Supported by RSA Data Security (Redwood City, California, 94065). email address: rivest@theory.lcs.mit.edu

$h(M)$ and signing the result can take much less time than directly signing all of M—note that h can be fast to compute and $h(M)$ can be short (say 128 bits). The value $h(M)$ is often called the "message digest" of the message M. This application is the motivation for the development of the MD4 algorithm.

The theoretical importance of one-way functions as a foundation for cryptography is becoming increasingly clear. Although there has been much work in this area, I'll cite only a couple of references, since the present paper is oriented more towards the practical utilization of one-way functions than towards their theoretical importance. First, Impagliazzo, Levin, and Luby [5] have recently shown that the existence of one-way functions is necessary and sufficient for the existence of secure pseudo-random generators. Using known results, this implies that secure private-key encryption and secure zero-knowledge protocols can be based on any one-way function. Second, Rompel [9] has recently shown (building on the work of Naor and Yung [8]) that the existence of one-way functions is necessary and sufficient for the existence of secure digital signature schemes.

Turning to the practical utilization of one-way functions, it is clear that high speed is a major design criterion. It is desirable to be able to sign very large files (e.g., megabytes in length) very quickly, and to verify these signatures very quickly as well. In one application, for example, it is desired to check the signature on an executable code module before loading and executing it, in order to verify the authenticity of the code module.

There have been many concrete proposals for efficient one-way functions; I don't attempt to survey these proposals here, but merely to give a couple of examples. One approach is to base the one-way function on an efficient conventional encryption scheme. For example, Merkle [7] shows how to construct a secure one-way function if DES is a good random block cipher. However, even his fastest method only hashes 18 bits of message per application of DES. As a second example, Damgård [1] studies the design of one-way functions, proves some theorems regarding the security of one-way functions that work in a "block by block" manner, and proposes a fast one-way function based on the knapsack problem. The design of MD4 was influenced by Damgård's work.

2 Overview and Design Goals

The first design goal is, of course, *security*: it should be computationally infeasible to find two messages M_1 and M_2 that have the same message digest. Here we define a task to be "computationally infeasible" if it requires more than 2^{64} operations.

The goal of security is to be achieved *directly*, without assumptions. That is, we do not wish to design a hash function that is secure if, say, factoring is difficult. While much of cryptography is based on such reductions, "the buck stops" at one-way functions—these need to be designed to be difficult to invert without any further assumptions.

The second design goal is *speed*: the algorithm should be as fast as possible. Here

we are interested in speed in *software*; designs that require special-purpose hardware are of limited interest. In particular, we are interested in algorithms that are fast on 32-bit architectures, since that is becoming the dominant standard processor architecture. Thus, the algorithm should be based on a simple set of primitive operations on 32-bit words.

The third design goal is *simplicity and compactness*: the algorithm should be simple to describe and simple to program, without requiring large programs or substitution tables. This is not only desirable from an engineering standpoint, but also desirable from a security viewpoint, since a simple algorithm is more likely to receive the necessary critical review.

A fourth design goal was to *favor little-endian architectures.* Some processor architectures (such as the Intel 80xxx line) are "little endian": they store the least-significant byte of a word in the low-address byte position. Others (such as a SUN Sparcstation) are "big-endian": the most-significant byte of a word goes in the low-address byte position. This distinction is significant when treating a message as a sequence of 32-bit words, since one architecture or the other will have to byte-reverse each word before processing. Since the big-endian processors are generally faster (it seems), it was decided to let them do the reversing. This incurs a performance penalty of about 25%.

3 Terminology and Notation

In this note a *word* is a 32-bit quantity and a byte is an 8-bit quantity. A sequence of bits can be interpreted in a natural manner as a sequence of bytes, where each consecutive group of 8 bits is interpreted as a byte with the high-order (most significant) bit of each byte listed first. Similarly, a sequence of bytes can be interpreted as a sequence of 32-bit words, where each consecutive group of 4 bytes is interpreted as a word with the low-order (least significant) byte given first.

Let the symbol "+" denote addition of words (i.e., modulo-2^{32} addition). Let $(X <<< s)$ denote the 32-bit value obtained by circularly shifting (rotating) X left by s bit positions. Let $\neg X$ denote the bit-wise complement of X, and let $X \vee Y$ denote the bit-wise OR of X and Y. Let $X \oplus Y$ denote the bit-wise XOR of X and Y, and let XY denote the bit-wise AND of X and Y.

4 MD4 Algorithm Description

We begin by supposing that we have a b-bit message as input, and that we wish to find its message digest. Here b is an arbitrary nonnegative integer; b may be zero, it need not be a multiple of 8, and it may be arbitrarily large. We imagine the bits of the message written down as follows:

$$m_0 m_1 ... m_{b-1}.$$

The following five steps are performed to compute the message digest of the message.

Step 1. Append padding bits

The message is padded (extended) so that its length (in bits) is congruent to 448, modulo 512. That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long. Padding is always performed, even if the length of the message is already congruent to 448, modulo 512 (in which case 512 bits of padding are added).

Padding is performed as follows: a single "1" bit is appended to the message, and then enough zero bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512. (This padding operation is invertible, so that different inputs yield different outputs—this would not be true if we merely padded with 0's.)

Step 2. Append length

A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step. These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions. In the unlikely event that b is greater than 2^{64}, then only the low-order 64 bits of b are used.

At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words. Let $M[0 \ldots N-1]$ denote the words of the resulting message, where N is a multiple of 16.

Step 3. Initialize MD buffer

A 4-word buffer (A, B, C, D) is used to compute the message digest. Here each of A, B, C, D is a 32-bit register. These registers are initialized to the following values (in hexadecimal, low-order bytes first):

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

We first define three auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

$$\begin{aligned} f(X,Y,Z) &= XY \vee (\neg X)Z \\ g(X,Y,Z) &= XY \vee XZ \vee YZ \\ h(X,Y,Z) &= X \oplus Y \oplus Z \end{aligned}$$

In each bit position f acts as a conditional: **if** x **then** y **else** z. In each bit position g acts as a majority function: if at least two of x, y, z are one, then g has a one in that position. The function h is the bit-wise *xor* or *parity* function. It is interesting

to note that if the bits of X, Y, and Z are independent and unbiased, the each bit of $f(X,Y,Z)$ is independent and unbiased, and similarly for $g(X,Y,Z)$ and $h(X,Y,Z)$.

MD4 utilizes two "magic constants" in rounds two and three. The round two constant is $\sqrt{2}$ and the round 3 constant is $\sqrt{3}$. (See Knuth [6, page 660].) Here are their values in octal and hex (with high-order digits given first).

	Octal	Hex
Round 2 constant ($\sqrt{2}$):	013240474631	5A827999
Round 3 constant ($\sqrt{3}$):	015666365641	6ED9EBA1

Do the following:

```
For i = 0 to N/16-1 do        /* process each 16-word block */
        Set X[j] to M[i*16+j], for j = 0, 1, ..., 15.
        Save A as AA, B as BB, C as CC, and D as DD.

        [Round 1]
          Let [A B C D i s] denote the operation
                A = (A + f(B,C,D) + X[i]) <<< s  .
          Do the following 16 operations:
                [A B C D 0  3]
                [D A B C 1  7]
                [C D A B 2  11]
                [B C D A 3  19]
                [A B C D 4  3]
                [D A B C 5  7]
                [C D A B 6  11]
                [B C D A 7  19]
                [A B C D 8  3]
                [D A B C 9  7]
                [C D A B 10 11]
                [B C D A 11 19]
                [A B C D 12 3]
                [D A B C 13 7]
                [C D A B 14 11]
                [B C D A 15 19]

        [Round 2]
          Let [A B C D i s] denote the operation
                A = (A + g(B,C,D) + X[i] + 5A827999) <<< s .
          Do the following 16 operations:
                [A B C D 0  3]
                [D A B C 4  5]
                [C D A B 8  9]
                [B C D A 12 13]
                [A B C D 1  3]
```

```
            [D A B C 5  5]
            [C D A B 9  9]
            [B C D A 13 13]
            [A B C D 2  3]
            [D A B C 6  5]
            [C D A B 10 9]
            [B C D A 14 13]
            [A B C D 3  3]
            [D A B C 7  5]
            [C D A B 11 9]
            [B C D A 15 13]

      [Round 3]
        Let [A B C D i s] denote the operation
              A = (A + h(B,C,D) + X[i] + 6ED9EBA1) <<< s .
        Do the following 16 operations:
            [A B C D 0  3]
            [D A B C 8  9]
            [C D A B 4  11]
            [B C D A 12 15]
            [A B C D 2  3]
            [D A B C 10 9]
            [C D A B 6  11]
            [B C D A 14 15]
            [A B C D 1  3]
            [D A B C 9  9]
            [C D A B 5  11]
            [B C D A 13 15]
            [A B C D 3  3]
            [D A B C 11 9]
            [C D A B 7  11]
            [B C D A 15 15]

Then perform the following additions:
            A = A + AA
            B = B + BB
            C = C + CC
            D = D + DD
(That is, each of the four registers is incremented by the value
it had before this block was started.)

end /* of loop on i */
```

Step 5. Output

The message digest produced as output is A, B, C, D. That is, we begin with the low-order byte of A, and end with the high-order byte of D.

This completes the description of MD4.

5 Extensions

If more than 128 bits of output are required, then the following procedure is recommended to obtain a 256-bit output. No provision is made for obtaining more than 256 bits.

Two copies of MD4 are run in parallel over the input. The first copy is standard as described above. The second copy is modified as follows.

The initial state of the second copy is:

word A: `00 11 22 33`
word B: `44 55 66 77`
word C: `88 99 aa bb`
word D: `cc dd ee ff`

The magic constants in rounds 2 and 3 for the second copy of MD4 are changed from $\sqrt{2}$ and $\sqrt{3}$ to $\sqrt[3]{2}$ and $\sqrt[3]{3}$:

	Octal	Hex
Round 2 constant ($\sqrt[3]{2}$):	`012050505746`	`50a28be6`
Round 3 constant ($\sqrt[3]{3}$):	`013423350444`	`5c4dd124`

Finally, after every 16-word block is processed (including the last block), the values of the A registers in the two copies are exchanged.

The final message digest is obtaining by appending the result of the second copy of MD4 to the end of the result of the first copy of MD4.

6 Implementation

The MD4 algorithm has been implemented in C and run on many different workstations. Here is a sampling of the running times achieved. (These running times include all processing, including byte-reversing the input words, but does not include file I/O time.) Assembly language implementations would of course be even faster.

Processor	*Speed (Bytes/second)*
SUN Sparc station	1,450,000
DEC MicroVax II	70,000
20MHz 80286	32,000

A reference implementation of MD4 in C is available as file `md4.doc` by anonymous ftp from `theory.lcs.mit.edu` and `rsa.com`, or from *RSA Data Security, 10 Twin Dolphin Drive, Redwood City, California 94065, phone: 1-800-PUBLIKEY.*

If you implement MD4 based on the description in this paper rather than the reference implementation, you may wish to check the following input-output values for MD4.

Input	Output
`""`	`31d6cfe0d16ae931b73c59d7e0c089c0`
`"a"`	`bde52cb31de33e46245e05fbdbd6fb24`
`"abc"`	`a448017aaf21d8525fc10ae87aa6729d`
`"message digest"`	`d9130a8164549fe818874806e1c7014b`
`"abcdefghijklmnopqrstuvwxyz"`	`d79e1c308aa5bbcdeea8ed63df412da9`

7 Summary

The MD4 message digest algorithm is simple to implement, and provides a "fingerprint" or message digest of a message of arbitrary length.

It is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of 2^{64} operations, and that the difficulty of coming up with any message having a given message digest is on the order of 2^{128} operations. The MD4 algorithm has been carefully scrutinized for weaknesses. It is, however, a relatively new algorithm and further security analysis is of course justified, as is the case with any new proposal of this sort.[1] The level of security provided by MD4 should be sufficient for implementing very high security hybrid digital signature schemes based on MD4 and the RSA public-key cryptosystem.

8 Acknowledgments

I'd like to thank Don Coppersmith, Burt Kaliski, Ralph Merkle, and Noam Nisan for numerous helpful comments and suggestions.

References

[1] Ivan Bjerre Damgård. A design principle for hash functions. In G. Brassard, editor, *Proceedings CRYPTO 89*, pages 416–427. Springer, 1990. Lecture Notes in Computer Science No. 435.

[2] D. W. Davies and W. L. Price. The application of digital signatures based on public-key cryptosystems. In *Proc. Fifth Intl. Computer Communications Conference*, pages 525–530, October 1980.

[1] Ralph Merkle has recently shown how to find two messages that collide in a modified form of MD4 that omits the third round. This technique does not seem to generalize to handle the full MD4 method, or even one in which the first or second round is omitted instead of the third.

[3] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT-22:644–654, November 1976.

[4] A. Evans, W. Kantrowitz, and E. Weiss. A user authentication scheme not requiring secrecy in the computer. *CACM*, 17:437–442, August 1974.

[5] Russell Impagliazzo, Leonid A. Levin, and Michael Luby. Pseudo-random generation from one-way functions. In *Proc. 21th ACM Symposium on Theory of Computing*, pages 12–24, Seattle, 1989. ACM.

[6] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, 1969. Second edition, 1981.

[7] Ralph C. Merkle. One way hash functions and DES. In G. Brassard, editor, *Proceedings CRYPTO 89*, pages 428–446. Springer, 1990. Lecture Notes in Computer Science No. 435.

[8] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proc. 21th ACM Symposium on Theory of Computing*, pages 33–43, Seattle, 1989. ACM.

[9] John Rompel. One-way functions are necessary and sufficient for secure signatures. In *Proc. 22nd ACM Symposium on Theory of Computing*, pages 387–394, Baltimore, Maryland, 1990. ACM.

[10] M. V. Wilkes. *Time-sharing computer systems.* Elsevier, 1975. Third edition.

```
/*
** ********************************************************************
** md4.h -- Header file for implementation of MD4 Message Digest Algorithm **
** Updated: 2/13/90 by Ronald L. Rivest                                **
** (C) 1990 RSA Data Security, Inc.                                    **
** ********************************************************************
*/

/* MDstruct is the data structure for a message digest computation.
*/
typedef struct {
  unsigned int buffer[4];     /* Holds 4-word result of MD computation */
  unsigned char count[8];     /* Number of bits processed so far */
  unsigned int done;          /* Nonzero means MD computation finished */
} MDstruct, *MDptr;

/* MDbegin(MD)
** Input: MD -- an MDptr
** Initialize the MDstruct prepatory to doing a message digest computation.
*/
extern void MDbegin();

/* MDupdate(MD,X,count)
** Input: MD -- an MDptr
**        X -- a pointer to an array of unsigned characters.
**        count -- the number of bits of X to use (an unsigned int).
** Updates MD using the first ''count'' bits of X.
** The array pointed to by X is not modified.
** If count is not a multiple of 8, MDupdate uses high bits of last byte.
** This is the basic input routine for a user.
** The routine terminates the MD computation when count < 512, so
** every MD computation should end with one call to MDupdate with a
** count less than 512.  Zero is OK for a count.
*/
extern void MDupdate();

/* MDprint(MD)
** Input: MD -- an MDptr
** Prints message digest buffer MD as 32 hexadecimal digits.
** Order is from low-order byte of buffer[0] to high-order byte of buffer[3].
** Each byte is printed with high-order hexadecimal digit first.
*/
extern void MDprint();

/*
** End of md4.h
*/
```

```
/*
** ********************************************************************
** md4.c -- Implementation of MD4 Message Digest Algorithm            **
** Updated: 2/16/90 by Ronald L. Rivest                               **
** (C) 1990 RSA Data Security, Inc.                                   **
** ********************************************************************
*/

/*
** To use MD4:
**   -- Include md4.h in your program
**   -- Declare an MDstruct MD to hold the state of the digest computation.
**   -- Initialize MD using MDbegin(&MD)
**   -- For each full block (64 bytes) X you wish to process, call
**          MDupdate(&MD,X,512)
**      (512 is the number of bits in a full block.)
**   -- For the last block (less than 64 bytes) you wish to process,
**          MDupdate(&MD,X,n)
**      where n is the number of bits in the partial block. A partial
**      block terminates the computation, so every MD computation should
**      terminate by processing a partial block, even if it has n = 0.
**   -- The message digest is available in MD.buffer[0] ... MD.buffer[3].
**      (Least-significant byte of each word should be output first.)
**   -- You can print out the digest using MDprint(&MD)
*/

/* Implementation notes:
** This implementation assumes that ints are 32-bit quantities.
** If the machine stores the least-significant byte of an int in the
** least-addressed byte (eg., VAX and 8086), then LOWBYTEFIRST should be
** set to TRUE.  Otherwise (eg., SUNS), LOWBYTEFIRST should be set to
** FALSE.  Note that on machines with LOWBYTEFIRST FALSE the routine
** MDupdate modifies has a side-effect on its input array (the order of bytes
** in each word are reversed).  If this is undesired a call to MDreverse(X) can
** reverse the bytes of X back into order after each call to MDupdate.
*/
#define TRUE  1
#define FALSE 0
#define LOWBYTEFIRST FALSE

/* Compile-time includes
*/
#include <stdio.h>
#include "md4.h"

/* Compile-time declarations of MD4 ``magic constants''.
*/
#define I0  0x67452301       /* Initial values for MD buffer */
```

```
#define I1  0xefcdab89
#define I2  0x98badcfe
#define I3  0x10325476
#define C2  013240474631     /* round 2 constant = sqrt(2) in octal */
#define C3  015666365641     /* round 3 constant = sqrt(3) in octal */
/* C2 and C3 are from Knuth, The Art of Programming, Volume 2
** (Seminumerical Algorithms), Second Edition (1981), Addison-Wesley.
** Table 2, page 660.
*/
#define fs1  3               /* round 1 shift amounts */
#define fs2  7
#define fs3 11
#define fs4 19
#define gs1  3               /* round 2 shift amounts */
#define gs2  5
#define gs3  9
#define gs4 13
#define hs1  3               /* round 3 shift amounts */
#define hs2  9
#define hs3 11
#define hs4 15


/* Compile-time macro declarations for MD4.
** Note: The ''rot'' operator uses the variable ''tmp''.
** It assumes tmp is declared as unsigned int, so that the >>
** operator will shift in zeros rather than extending the sign bit.
*/
#define f(X,Y,Z)             ((X&Y) | ((~X)&Z))
#define g(X,Y,Z)             ((X&Y) | (X&Z) | (Y&Z))
#define h(X,Y,Z)             (X^Y^Z)
#define rot(X,S)             (tmp=X,(tmp<<S) | (tmp>>(32-S)))
#define ff(A,B,C,D,i,s)      A = rot((A + f(B,C,D) + X[i]),s)
#define gg(A,B,C,D,i,s)      A = rot((A + g(B,C,D) + X[i] + C2),s)
#define hh(A,B,C,D,i,s)      A = rot((A + h(B,C,D) + X[i] + C3),s)

/* MDprint(MDp)
** Print message digest buffer MDp as 32 hexadecimal digits.
** Order is from low-order byte of buffer[0] to high-order byte of buffer[3].
** Each byte is printed with high-order hexadecimal digit first.
** This is a user-callable routine.
*/
void
MDprint(MDp)
MDptr MDp;
{ int i,j;
  for (i=0;i<4;i++)
    for (j=0;j<32;j=j+8)
```

```
      printf("%02x",(MDp->buffer[i]>>j) & 0xFF);
}

/* MDbegin(MDp)
** Initialize message digest buffer MDp.
** This is a user-callable routine.
*/
void
MDbegin(MDp)
MDptr MDp;
{ int i;
  MDp->buffer[0] = I0;
  MDp->buffer[1] = I1;
  MDp->buffer[2] = I2;
  MDp->buffer[3] = I3;
  for (i=0;i<8;i++) MDp->count[i] = 0;
  MDp->done = 0;
}

/* MDreverse(X)
** Reverse the byte-ordering of every int in X.
** Assumes X is an array of 16 ints.
** The macro revx reverses the byte-ordering of the next word of X.
*/
#define revx { t = (*X << 16) | (*X >> 16); \
               *X++ = ((t & 0xFF00FF00) >> 8) | ((t & 0x00FF00FF) << 8); }
MDreverse(X)
unsigned int *X;
{ register unsigned int t;
  revx; revx; revx; revx; revx; revx; revx; revx;
  revx; revx; revx; revx; revx; revx; revx; revx;
}

/* MDblock(MDp,X)
** Update message digest buffer MDp->buffer using 16-word data block X.
** Assumes all 16 words of X are full of data.
** Does not update MDp->count.
** This routine is not user-callable.
*/
static void
MDblock(MDp,X)
MDptr MDp;
unsigned int *X;
{
  register unsigned int tmp, A, B, C, D;
#if LOWBYTEFIRST == FALSE
  MDreverse(X);
#endif
```

```
A = MDp->buffer[0];
B = MDp->buffer[1];
C = MDp->buffer[2];
D = MDp->buffer[3];
/* Update the message digest buffer */
ff(A , B , C , D ,  0 , fs1); /* Round 1 */
ff(D , A , B , C ,  1 , fs2);
ff(C , D , A , B ,  2 , fs3);
ff(B , C , D , A ,  3 , fs4);
ff(A , B , C , D ,  4 , fs1);
ff(D , A , B , C ,  5 , fs2);
ff(C , D , A , B ,  6 , fs3);
ff(B , C , D , A ,  7 , fs4);
ff(A , B , C , D ,  8 , fs1);
ff(D , A , B , C ,  9 , fs2);
ff(C , D , A , B , 10 , fs3);
ff(B , C , D , A , 11 , fs4);
ff(A , B , C , D , 12 , fs1);
ff(D , A , B , C , 13 , fs2);
ff(C , D , A , B , 14 , fs3);
ff(B , C , D , A , 15 , fs4);
gg(A , B , C , D ,  0 , gs1); /* Round 2 */
gg(D , A , B , C ,  4 , gs2);
gg(C , D , A , B ,  8 , gs3);
gg(B , C , D , A , 12 , gs4);
gg(A , B , C , D ,  1 , gs1);
gg(D , A , B , C ,  5 , gs2);
gg(C , D , A , B ,  9 , gs3);
gg(B , C , D , A , 13 , gs4);
gg(A , B , C , D ,  2 , gs1);
gg(D , A , B , C ,  6 , gs2);
gg(C , D , A , B , 10 , gs3);
gg(B , C , D , A , 14 , gs4);
gg(A , B , C , D ,  3 , gs1);
gg(D , A , B , C ,  7 , gs2);
gg(C , D , A , B , 11 , gs3);
gg(B , C , D , A , 15 , gs4);
hh(A , B , C , D ,  0 , hs1); /* Round 3 */
hh(D , A , B , C ,  8 , hs2);
hh(C , D , A , B ,  4 , hs3);
hh(B , C , D , A , 12 , hs4);
hh(A , B , C , D ,  2 , hs1);
hh(D , A , B , C , 10 , hs2);
hh(C , D , A , B ,  6 , hs3);
hh(B , C , D , A , 14 , hs4);
hh(A , B , C , D ,  1 , hs1);
hh(D , A , B , C ,  9 , hs2);
hh(C , D , A , B ,  5 , hs3);
```

```
  hh(B , C , D , A , 13 , hs4);
  hh(A , B , C , D ,  3 , hs1);
  hh(D , A , B , C , 11 , hs2);
  hh(C , D , A , B ,  7 , hs3);
  hh(B , C , D , A , 15 , hs4);
  MDp->buffer[0] += A;
  MDp->buffer[1] += B;
  MDp->buffer[2] += C;
  MDp->buffer[3] += D;
}

/* MDupdate(MDp,X,count)
** Input: MDp -- an MDptr
**        X -- a pointer to an array of unsigned characters.
**        count -- the number of bits of X to use.
**                 (if not a multiple of 8, uses high bits of last byte.)
** Update MDp using the number of bits of X given by count.
** This is the basic input routine for an MD4 user.
** The routine completes the MD computation when count < 512, so
** every MD computation should end with one call to MDupdate with a
** count less than 512.  A call with count 0 will be ignored if the
** MD has already been terminated (done != 0), so an extra call with count
** 0 can be given as a ``courtesy close'' to force termination if desired.
*/
void
MDupdate(MDp,X,count)
MDptr MDp;
unsigned char *X;
unsigned int count;
{ unsigned int i, tmp, bit, byte, mask;
  unsigned char XX[64];
  unsigned char *p;
  /* return with no error if this is a courtesy close with count
  ** zero and MDp->done is true.
  */
  if (count == 0 && MDp->done) return;
  /* check to see if MD is already done and report error */
  if (MDp->done) { printf("\nError: MDupdate MD already done."); return; }
  /* Add count to MDp->count */
  tmp = count;
  p = MDp->count;
  while (tmp)
    { tmp += *p;
      *p++ = tmp;
      tmp = tmp >> 8;
    }
  /* Process data */
  if (count == 512)
```

```
    { /* Full block of data to handle */
      MDblock(MDp,(unsigned int *)X);
    }
  else if (count > 512) /* Check for count too large */
    { printf("\nError: MDupdate called with illegal count value %d.",count);
      return;
    }
  else /* partial block -- must be last block so finish up */
    { /* Find out how many bytes and residual bits there are */
      byte = count >> 3;
      bit =  count & 7;
      /* Copy X into XX since we need to modify it */
      for (i=0;i<=byte;i++)   XX[i] = X[i];
      for (i=byte+1;i<64;i++) XX[i] = 0;
      /* Add padding '1' bit and low-order zeros in last byte */
      mask = 1 << (7 - bit);
      XX[byte] = (XX[byte] | mask) & ~( mask - 1);
      /* If room for bit count, finish up with this block */
      if (byte <= 55)
        { for (i=0;i<8;i++) XX[56+i] = MDp->count[i];
          MDblock(MDp,(unsigned int *)XX);
        }
      else /* need to do two blocks to finish up */
        { MDblock(MDp,(unsigned int *)XX);
          for (i=0;i<56;i++) XX[i] = 0;
          for (i=0;i<8;i++)  XX[56+i] = MDp->count[i];
          MDblock(MDp,(unsigned int *)XX);
        }
      /* Set flag saying we're done with MD computation */
      MDp->done = 1;
    }
}

/*
** End of md4.c
*/
```

```
/*
** ********************************************************************
** md4driver.c -- sample routines to test MD4 message digest algorithm.    **
** Updated: 2/16/90 by Ronald L. Rivest                                    **
** (C) 1990 RSA Data Security, Inc.                                        **
** ********************************************************************
*/

#include <stdio.h>
#include "md4.h"

/* MDtimetrial()
** A time trial routine, to measure the speed of MD4.
** Measures speed for 1M blocks = 64M bytes.
*/
MDtimetrial()
{ unsigned int X[16];
  MDstruct MD;
  int i;
  double t;
  for (i=0;i<16;i++) X[i] = 0x01234567 + i;
  printf("MD4 time trial. Processing 1 million 64-character blocks...\n");
  clock();
  MDbegin(&MD);
  for (i=0;i<1000000;i++) MDupdate(&MD,X,512);
  MDupdate(&MD,X,0);
  t = (double) clock(); /* in microseconds */
  MDprint(&MD); printf(" is digest of 64M byte test input.\n");
  printf("Seconds to process test input:   %g\n",t/1e6);
  printf("Characters processed per second: %ld.\n",(int)(64e12/t));
}

/* MDstring(s)
** Computes the message digest for string s.
** Prints out message digest, a space, the string (in quotes) and a carriage
** return.
*/
MDstring(s)
unsigned char *s;
{ unsigned int i, len = strlen(s);
  MDstruct MD;
  MDbegin(&MD);
  for (i=0;i+64<=len;i=i+64) MDupdate(&MD,s+i,512);
  MDupdate(&MD,s+i,(len-i)*8);
  MDprint(&MD);
  printf(" \"%s\"\n",s);
}
```

```
/* MDfile(filename)
** Computes the message digest for a specified file.
** Prints out message digest, a space, the file name, and a carriage return.
*/
MDfile(filename)
char *filename;
{ FILE *f = fopen(filename,"rb");
  unsigned char X[64];
  MDstruct MD;
  int b;
  if (f == NULL) { printf("%s can't be opened.\n",filename); return; }
  MDbegin(&MD);
  while ((b=fread(X,1,64,f))!=0) MDupdate(&MD,X,b*8);
  MDupdate(&MD,X,0);
  MDprint(&MD);
  printf(" %s\n",filename);
  fclose(f);
}

/* MDfilter()
** Writes the message digest of the data from stdin onto stdout, followed
** by a carriage return.
*/
MDfilter()
{ unsigned char X[64];
  MDstruct MD;
  int b;
  MDbegin(&MD);
  while ((b=fread(X,1,64,stdin))!=0) MDupdate(&MD,X,b*8);
  MDupdate(&MD,X,0);
  MDprint(&MD);
  printf("\n");
}

/* MDtestsuite()
** Run a standard suite of test data.
*/
MDtestsuite()
{
  printf("MD4 test suite results:\n");
  MDstring("");
  MDstring("a");
  MDstring("abc");
  MDstring("message digest");
  MDstring("abcdefghijklmnopqrstuvwxyz");
  MDstring("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789");
  MDfile("foo"); /* Contents of file foo are "abc" */
}
```

```
main(argc,argv)
int argc;
char *argv[];
{ int i;
  /* For each command line argument in turn:
  ** filename             -- prints message digest and name of file
  ** -sstring             -- prints message digest and contents of string
  ** -t                   -- prints time trial statistics for 64M bytes
  ** -x                   -- execute a standard suite of test data
  ** (no args)            -- writes messages digest of stdin onto stdout
  */
  if (argc==1) MDfilter();
  else
    for (i=1;i<argc;i++)
      if (argv[i][0]=='-' && argv[i][1]=='s') MDstring(argv[i]+2);
      else if (strcmp(argv[i],"-t")==0)       MDtimetrial();
      else if (strcmp(argv[i],"-x")==0)       MDtestsuite();
      else                                    MDfile(argv[i]);
}

/*
** end of md4driver.c
*/
```

```
>--------------------------------------------------------------------------
---- Sample session.  Compiling and using MD4 on SUN Sparcstation --------
--------------------------------------------------------------------------
>ls
total 66
-rw-rw-r--  1 rivest          3 Feb 14 17:40 abcfile
-rwxrwxr-x  1 rivest      24576 Feb 17 12:28 md4
-rw-rw-r--  1 rivest       9347 Feb 17 00:37 md4.c
-rw-rw-r--  1 rivest      25150 Feb 17 12:25 md4.doc
-rw-rw-r--  1 rivest       1844 Feb 16 21:21 md4.h
-rw-rw-r--  1 rivest       3497 Feb 17 12:27 md4driver.c
>
>cc -o md4 -04 md4.c md4driver.c
md4.c:
md4driver.c:
Linking:
>
>md4 -x
MD4 test suite results:
31d6cfe0d16ae931b73c59d7e0c089c0 ""
bde52cb31de33e46245e05fbdbd6fb24 "a"
a448017aaf21d8525fc10ae87aa6729d "abc"
d9130a8164549fe818874806e1c7014b "message digest"
d79e1c308aa5bbcdeea8ed63df412da9 "abcdefghijklmnopqrstuvwxyz"
043f8582f241db351ce627e153e7f0e4
          "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"
a448017aaf21d8525fc10ae87aa6729d abcfile
>
>md4 -sabc -shi
a448017aaf21d8525fc10ae87aa6729d "abc"
cfaee2512bd25eb033236f0cd054e308 "hi"
>
>md4 *
a448017aaf21d8525fc10ae87aa6729d abcfile
d316f994da0e951cf9502928a1f73300 md4
379adb39eada0dfdbbdfdcd0d9def8c4 md4.c
9a3f73327c65954198b1f45a3aa12665 md4.doc
37fe165ac177b461ff78b86d10e4ff33 md4.h
7dcba2e2dc4d8f1408d08beb17dabb2a md4.o
08790161bfddc6f5788b4353875cb1c3 md4driver.c
1f84a7f690b0545d2d0480d5d3c26eea md4driver.o
>
>cat abcfile | md4
a448017aaf21d8525fc10ae87aa6729d
>
>md4 -t
MD4 time trial. Processing 1 million 64-character blocks...
6325bf77e5891c7c0d8104b64cc6e9ef is digest of 64M byte test input.
```

```
Seconds to process test input:   44.0982
Characters processed per second: 1451305.
>
>
------------------------ end of sample session --------------------------
```