Network Working Group
Request for Comments: 4634
Updates: 3174
Category: Informational

D. Eastlake 3rd
Motorola Labs
T. Hansen
AT&T Labs

July 2006

US Secure Hash Algorithms (SHA and HMAC-SHA)

Status of This Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2006).

Abstract

The United States of America has adopted a suite of Secure Hash Algorithms (SHAs), including four beyond SHA-1, as part of a Federal Information Processing Standard (FIPS), specifically SHA-224 (RFC 3874), SHA-256, SHA-384, and SHA-512. The purpose of this document is to make source code performing these hash functions conveniently available to the Internet community. The sample code supports input strings of arbitrary bit length. SHA-1's sample code from RFC 3174 has also been updated to handle input strings of arbitrary bit length. Most of the text herein was adapted by the authors from FIPS 180-2.

Code to perform SHA-based HMACs, with arbitrary bit length text, is also included.

Table of Contents

1. Overview of Contents 3
1.1. License 4
2. Notation for Bit Strings and Integers 4
3. Operations on Words 5
4. Message Padding and Parsing 6
4.1. SHA-224 and SHA-256 7
4.2. SHA-384 and SHA-512 8
5. Functions and Constants Used 9
5.1. SHA-224 and SHA-256 9
5.2. SHA-384 and SHA-512 10
6. Computing the Message Digest 11
6.1. SHA-224 and SHA-256 Initialization 11
6.2. SHA-224 and SHA-256 Processing 11
6.3. SHA-384 and SHA-512 Initialization 13
6.4. SHA-384 and SHA-512 Processing 14
7. SHA-Based HMACs 15
8. C Code for SHAs 15
8.1. The .h File 18
8.2. The SHA Code 24
8.2.1. sha1.c 24
8.2.2. sha224-256.c 33
8.2.3. sha384-512.c 45
8.2.4. usha.c 67
8.2.5. sha-private.h 72
8.3. The HMAC Code 73
8.4. The Test Driver 78
9. Security Considerations 106
10. Normative References 106
11. Informative References 106

1. Overview of Contents

NOTE: Much of the text below is taken from [FIPS180-2] and assertions therein of the security of the algorithms described are made by the US Government, the author of [FIPS180-2], and not by the authors of this document.

The text below specifies Secure Hash Algorithms, SHA-224 [RFC3874], SHA-256, SHA-384, and SHA-512, for computing a condensed representation of a message or a data file. (SHA-1 is specified in [RFC3174].) When a message of any length < 2^64 bits (for SHA-224 and SHA-256) or < 2^128 bits (for SHA-384 and SHA-512) is input to one of these algorithms, the result is an output called a message digest. The message digests range in length from 224 to 512 bits, depending on the algorithm. Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed hash authentication codes, or in the generation of random numbers [RFC4086].

The four algorithms specified in this document are called secure because it is computationally infeasible to (1) find a message that corresponds to a given message digest, or (2) find two different messages that produce the same message digest. Any change to a message in transit will, with very high probability, result in a different message digest. This will result in a verification failure when the secure hash algorithm is used with a digital signature algorithm or a keyed-hash message authentication algorithm.

The code provided herein supports input strings of arbitrary bit length. SHA-1's sample code from [RFC3174] has also been updated to handle input strings of arbitrary bit length. See Section 1.1 for license information for this code.

Section 2 below defines the terminology and functions used as building blocks to form these algorithms. Section 3 describes the fundamental operations on words from which these algorithms are built. Section 4 describes how messages are padded up to an integral multiple of the required block size and then parsed into blocks. Section 5 defines the constants and the composite functions used to specify these algorithms. Section 6 gives the actual specification for the SHA-224, SHA-256, SHA-384, and SHA-512 functions. Section 7 provides pointers to the specification of HMAC keyed message authentication codes based on the SHA algorithms. Section 8 gives sample code for the SHA algorithms and Section 9 code for SHA-based HMACs. The SHA-based HMACs will accept arbitrary bit length text.

1.1. License

Permission is granted for all uses, commercial and non-commercial, of the sample code found in Section 8. Royalty free license to use, copy, modify and distribute the software found in Section 8 is granted, provided that this document is identified in all material mentioning or referencing this software, and provided that redistributed derivative works do not contain misleading author or version information.

The authors make no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind.

2. Notation for Bit Strings and Integers

The following terminology related to bit strings and integers will be used:

a. A hex digit is an element of the set {0, 1, ... , 9, A, ... , F}. A hex digit is the representation of a 4-bit string. Examples: 7 = 0111, A = 1010.

b. A word equals a 32-bit or 64-bit string, which may be represented as a sequence of 8 or 16 hex digits, respectively. To convert a word to hex digits, each 4-bit string is converted to its hex equivalent as described in (a) above. Example:

   1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23.

   Throughout this document, the "big-endian" convention is used when expressing both 32-bit and 64-bit words, so that within each word the most significant bit is shown in the left-most bit position.

c. An integer may be represented as a word or pair of words.

   An integer between 0 and 2^32 - 1 inclusive may be represented as a 32-bit word. The least significant four bits of the integer are represented by the right-most hex digit of the word representation. Example: the integer 291 = 2^8+2^5+2^1+2^0 = 256+32+2+1 is represented by the hex word 00000123.

   The same holds true for an integer between 0 and 2^64-1 inclusive, which may be represented as a 64-bit word.

If Z is an integer, 0 <= z < 2^64, then z = (2^32)x + y where 0 <= x < 2^32 and 0 <= y < 2^32. Since x and y can be represented as words X and Y, respectively, z can be represented as the pair of words (X,Y).

d. block = 512-bit or 1024-bit string. A block (e.g., B) may be represented as a sequence of 32-bit or 64-bit words.

3. Operations on Words

The following logical operators will be applied to words in all four hash operations specified herein. SHA-224 and SHA-256 operate on 32-bit words, while SHA-384 and SHA-512 operate on 64-bit words.

In the operations below, x<<n is obtained as follows: discard the left-most n bits of x and then pad the result with n zeroed bits on the right (the result will still be the same number of bits).

a. Bitwise logical word operations

X AND Y = bitwise logical "and" of X and Y.

X OR Y = bitwise logical "inclusive-or" of X and Y.

X XOR Y = bitwise logical "exclusive-or" of X and Y.

NOT X = bitwise logical "complement" of X.

Example:

```
         01101100101110011101001001111011
   XOR   01100101110000010110100110110111
         --------------------------------
     =   00001001011110001011101111001100
```

b. The operation X + Y is defined as follows: words X and Y represent w-bit integers x and y, where 0 <= x < 2^w and 0 <= y < 2^w. For positive integers n and m, let

n mod m

be the remainder upon dividing n by m. Compute

z = (x + y) mod 2^w.

Then 0 <= z < 2^w. Convert z to a word, Z, and define Z = X + Y.

c. The right shift operation SHR^n(x), where x is a w-bit word and n is an integer with 0 <= n < w, is defined by

```
SHR^n(x) = x>>n
```

d. The rotate right (circular right shift) operation ROTR^n(x), where x is a w-bit word and n is an integer with 0 <= n < w, is defined by

```
ROTR^n(x) = (x>>n) OR (x<<(w-n))
```

e. The rotate left (circular left shift) operation ROTL^n(x), where x is a w-bit word and n is an integer with 0 <= n < w, is defined by

```
ROTL^n(X)  =  (x<<n) OR (x>>w-n)
```

Note the following equivalence relationships, where w is fixed in each relationship:

```
ROTL^n(x) = ROTR^(w-x)(x)

ROTR^n(x) = ROTL^(w-n)(x)
```

4. Message Padding and Parsing

The hash functions specified herein are used to compute a message digest for a message or data file that is provided as input. The message or data file should be considered to be a bit string. The length of the message is the number of bits in the message (the empty message has length 0). If the number of bits in a message is a multiple of 8, for compactness we can represent the message in hex. The purpose of message padding is to make the total length of a padded message a multiple of 512 for SHA-224 and SHA-256 or a multiple of 1024 for SHA-384 and SHA-512.

The following specifies how this padding shall be performed. As a summary, a "1" followed by a number of "0"s followed by a 64-bit or 128-bit integer are appended to the end of the message to produce a padded message of length 512*n or 1024*n. The minimum number of "0"s necessary to meet this criterion is used. The appended integer is the length of the original message. The padded message is then processed by the hash function as n 512-bit or 1024-bit blocks.

4.1. SHA-224 and SHA-256

Suppose a message has length L < 2^64. Before it is input to the hash function, the message is padded on the right as follows:

a. "1" is appended. Example: if the original message is "01010000", this is padded to "010100001".

b. K "0"s are appended where K is the smallest, non-negative solution to the equation

```
L + 1 + K = 448 (mod 512)
```

c. Then append the 64-bit block that is L in binary representation. After appending this block, the length of the message will be a multiple of 512 bits.

Example: Suppose the original message is the bit string

```
01100001 01100010 01100011 01100100 01100101
```

After step (a), this gives

```
01100001 01100010 01100011 01100100 01100101 1
```

Since L = 40, the number of bits in the above is 41 and K = 407 "0"s are appended, making the total now 448. This gives the following in hex:

```
61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000
```

The 64-bit representation of L = 40 is hex 00000000 00000028. Hence the final padded message is the following hex:

```
61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000028
```

4.2. SHA-384 and SHA-512

Suppose a message has length L < 2^128. Before it is input to the hash function, the message is padded on the right as follows:

a. "1" is appended. Example: if the original message is "01010000", this is padded to "010100001".

b. K "0"s are appended where K is the smallest, non-negative solution to the equation

```
L + 1 + K = 896 (mod 1024)
```

c. Then append the 128-bit block that is L in binary representation. After appending this block, the length of the message will be a multiple of 1024 bits.

Example: Suppose the original message is the bit string

```
01100001 01100010 01100011 01100100 01100101
```

After step (a) this gives

```
01100001 01100010 01100011 01100100 01100101 1
```

Since L = 40, the number of bits in the above is 41 and K = 855 "0"s are appended, making the total now 896. This gives the following in hex:

```
61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

The 128-bit representation of L = 40 is hex 00000000 00000000 00000000 00000028. Hence the final padded message is the following hex:

```
61626364 65800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000028
```

5. Functions and Constants Used

The following subsections give the six logical functions and the table of constants used in each of the hash functions.

5.1. SHA-224 and SHA-256

SHA-224 and SHA-256 use six logical functions, where each function operates on 32-bit words, which are represented as x, y, and z. The result of each function is a new 32-bit word.

```
CH( x, y, z) = (x AND y) XOR ( (NOT x) AND z)

MAJ( x, y, z) = (x AND y) XOR (x AND z) XOR (y AND z)

BSIG0(x) = ROTR^2(x) XOR ROTR^13(x) XOR ROTR^22(x)

BSIG1(x) = ROTR^6(x) XOR ROTR^11(x) XOR ROTR^25(x)

SSIG0(x) = ROTR^7(x) XOR ROTR^18(x) XOR SHR^3(x)

SSIG1(x) = ROTR^17(x) XOR ROTR^19(x) XOR SHR^10(x)
```

SHA-224 and SHA-256 use the same sequence of sixty-four constant 32-bit words, K0, K1, ..., K63. These words represent the first thirty-two bits of the fractional parts of the cube roots of the first sixty-four prime numbers. In hex, these constant words are as follows (from left to right):

```
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
```

5.2. SHA-384 and SHA-512

SHA-384 and SHA-512 each use six logical functions, where each function operates on 64-bit words, which are represented as x, y, and z. The result of each function is a new 64-bit word.

```
CH( x, y, z) = (x AND y) XOR ( (NOT x) AND z)

MAJ( x, y, z) = (x AND y) XOR (x AND z) XOR (y AND z)

BSIG0(x) = ROTR^28(x) XOR ROTR^34(x) XOR ROTR^39(x)

BSIG1(x) = ROTR^14(x) XOR ROTR^18(x) XOR ROTR^41(x)

SSIG0(x) = ROTR^1(x) XOR ROTR^8(x) XOR SHR^7(x)

SSIG1(x) = ROTR^19(x) XOR ROTR^61(x) XOR SHR^6(x)
```

SHA-384 and SHA-512 use the same sequence of eighty constant 64-bit words, K0, K1, ... K79. These words represent the first sixty-four bits of the fractional parts of the cube roots of the first eighty prime numbers. In hex, these constant words are as follows (from left to right):

```
428a2f98d728ae22 7137449123ef65cd b5c0fbcfec4d3b2f e9b5dba58189dbbc
3956c25bf348b538 59f111f1b605d019 923f82a4af194f9b ab1c5ed5da6d8118
d807aa98a3030242 12835b0145706fbe 243185be4ee4b28c 550c7dc3d5ffb4e2
72be5d74f27b896f 80deb1fe3b1696b1 9bdc06a725c71235 c19bf174cf692694
e49b69c19ef14ad2 efbe4786384f25e3 0fc19dc68b8cd5b5 240ca1cc77ac9c65
2de92c6f592b0275 4a7484aa6ea6e483 5cb0a9dcbd41fbd4 76f988da831153b5
983e5152ee66dfab a831c66d2db43210 b00327c898fb213f bf597fc7beef0ee4
c6e00bf33da88fc2 d5a79147930aa725 06ca6351e003826f 142929670a0e6e70
27b70a8546d22ffc 2e1b21385c26c926 4d2c6dfc5ac42aed 53380d139d95b3df
650a73548baf63de 766a0abb3c77b2a8 81c2c92e47edaee6 92722c851482353b
a2bfe8a14cf10364 a81a664bbc423001 c24b8b70d0f89791 c76c51a30654be30
d192e819d6ef5218 d69906245565a910 f40e35855771202a 106aa07032bbd1b8
19a4c116b8d2d0c8 1e376c085141ab53 2748774cdf8eeb99 34b0bcb5e19b48a8
391c0cb3c5c95a63 4ed8aa4ae3418acb 5b9cca4f7763e373 682e6ff3d6b2b8a3
748f82ee5defb2fc 78a5636f43172f60 84c87814a1f0ab72 8cc702081a6439ec
90befffa23631e28 a4506cebde82bde9 bef9a3f7b2c67915 c67178f2e372532b
ca273eceea26619c d186b8c721c0c207 eada7dd6cde0eb1e f57d4f7fee6ed178
06f067aa72176fba 0a637dc5a2c898a6 113f9804bef90dae 1b710b35131c471b
28db77f523047d84 32caab7b40c72493 3c9ebe0a15c9bebc 431d67c49c100d4c
4cc5d4becb3e42b6 597f299cfc657e2a 5fcb6fab3ad6faec 6c44198c4a475817
```

6. Computing the Message Digest

The output of each of the secure hash functions, after being applied to a message of N blocks, is the hash quantity H(N). For SHA-224 and SHA-256, H(i) can be considered to be eight 32-bit words, H(i)0, H(i)1, ... H(i)7. For SHA-384 and SHA-512, it can be considered to be eight 64-bit words, H(i)0, H(i)1, ..., H(i)7.

As described below, the hash words are initialized, modified as each message block is processed, and finally concatenated after processing the last block to yield the output. For SHA-256 and SHA-512, all of the H(N) variables are concatenated while the SHA-224 and SHA-384 hashes are produced by omitting some from the final concatenation.

6.1. SHA-224 and SHA-256 Initialization

For SHA-224, the initial hash value, H(0), consists of the following 32-bit words in hex:

```
H(0)0 = c1059ed8
H(0)1 = 367cd507
H(0)2 = 3070dd17
H(0)3 = f70e5939
H(0)4 = ffc00b31
H(0)5 = 68581511
H(0)6 = 64f98fa7
H(0)7 = befa4fa4
```

For SHA-256, the initial hash value, H(0), consists of the following eight 32-bit words, in hex. These words were obtained by taking the first thirty-two bits of the fractional parts of the square roots of the first eight prime numbers.

```
H(0)0 = 6a09e667
H(0)1 = bb67ae85
H(0)2 = 3c6ef372
H(0)3 = a54ff53a
H(0)4 = 510e527f
H(0)5 = 9b05688c
H(0)6 = 1f83d9ab
H(0)7 = 5be0cd19
```

6.2. SHA-224 and SHA-256 Processing

SHA-224 and SHA-256 perform identical processing on messages blocks and differ only in how H(0) is initialized and how they produce their final output. They may be used to hash a message, M, having a length of L bits, where 0 <= L < 2^64. The algorithm uses (1) a message

schedule of sixty-four 32-bit words, (2) eight working variables of 32 bits each, and (3) a hash value of eight 32-bit words.

The words of the message schedule are labeled W0, W1, ..., W63. The eight working variables are labeled a, b, c, d, e, f, g, and h. The words of the hash value are labeled H(i)0, H(i)1, ..., H(i)7, which will hold the initial hash value, H(0), replaced by each successive intermediate hash value (after each message block is processed), H(i), and ending with the final hash value, H(N), after all N blocks are processed. They also use two temporary words, T1 and T2.

The input message is padded as described in Section 4.1 above then parsed into 512-bit blocks, which are considered to be composed of 16 32-bit words M(i)0, M(i)1, ..., M(i)15. The following computations are then performed for each of the N message blocks. All addition is performed modulo 2^32.

```
For i = 1 to N

   1. Prepare the message schedule W:
      For t = 0 to 15
         Wt = M(i)t
      For t = 16 to 63
         Wt = SSIG1(W(t-2)) + W(t-7) + SSIG0(t-15) + W(t-16)

   2. Initialize the working variables:
      a = H(i-1)0
      b = H(i-1)1
      c = H(i-1)2
      d = H(i-1)3
      e = H(i-1)4
      f = H(i-1)5
      g = H(i-1)6
      h = H(i-1)7

   3. Perform the main hash computation:
      For t = 0 to 63
         T1 = h + BSIG1(e) + CH(e,f,g) + Kt + Wt
         T2 = BSIG0(a) + MAJ(a,b,c)
         h = g
         g = f
         f = e
         e = d + T1
         d = c
         c = b
         b = a
         a = T1 + T2
```

4. Compute the intermediate hash value H(i):
   H(i)0 = a + H(i-1)0
   H(i)1 = b + H(i-1)1
   H(i)2 = c + H(i-1)2
   H(i)3 = d + H(i-1)3
   H(i)4 = e + H(i-1)4
   H(i)5 = f + H(i-1)5
   H(i)6 = g + H(i-1)6
   H(i)7 = h + H(i-1)7

After the above computations have been sequentially performed for all of the blocks in the message, the final output is calculated. For SHA-256, this is the concatenation of all of H(N)0, H(N)1, through H(N)7. For SHA-224, this is the concatenation of H(N)0, H(N)1, through H(N)6.

6.3. SHA-384 and SHA-512 Initialization

For SHA-384, the initial hash value, H(0), consists of the following eight 64-bit words, in hex. These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the ninth through sixteenth prime numbers.

```
H(0)0 = cbbb9d5dc1059ed8
H(0)1 = 629a292a367cd507
H(0)2 = 9159015a3070dd17
H(0)3 = 152fecd8f70e5939
H(0)4 = 67332667ffc00b31
H(0)5 = 8eb44a8768581511
H(0)6 = db0c2e0d64f98fa7
H(0)7 = 47b5481dbefa4fa4
```

For SHA-512, the initial hash value, H(0), consists of the following eight 64-bit words, in hex. These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers.

```
H(0)0 = 6a09e667f3bcc908
H(0)1 = bb67ae8584caa73b
H(0)2 = 3c6ef372fe94f82b
H(0)3 = a54ff53a5f1d36f1
H(0)4 = 510e527fade682d1
H(0)5 = 9b05688c2b3e6c1f
H(0)6 = 1f83d9abfb41bd6b
H(0)7 = 5be0cd19137e2179
```

6.4. SHA-384 and SHA-512 Processing

SHA-384 and SHA-512 perform identical processing on message blocks and differ only in how H(0) is initialized and how they produce their final output. They may be used to hash a message, M, having a length of L bits, where 0 <= L < 2^128. The algorithm uses (1) a message schedule of eighty 64-bit words, (2) eight working variables of 64 bits each, and (3) a hash value of eight 64-bit words.

The words of the message schedule are labeled W0, W1, ..., W79. The eight working variables are labeled a, b, c, d, e, f, g, and h. The words of the hash value are labeled H(i)0, H(i)1, ..., H(i)7, which will hold the initial hash value, H(0), replaced by each successive intermediate hash value (after each message block is processed), H(i), and ending with the final hash value, H(N) after all N blocks are processed.

The input message is padded as described in Section 4.2 above, then parsed into 1024-bit blocks, which are considered to be composed of 16 64-bit words M(i)0, M(i)1, ..., M(i)15. The following computations are then performed for each of the N message blocks. All addition is performed modulo 2^64.

For i = 1 to N

1. Prepare the message schedule W:

```
For t = 0 to 15
   Wt = M(i)t
For t = 16 to 79
   Wt = SSIG1(W(t-2)) + W(t-7) + SSIG0(t-15) + W(t-16)
```

2. Initialize the working variables:

```
a = H(i-1)0
b = H(i-1)1
c = H(i-1)2
d = H(i-1)3
e = H(i-1)4
f = H(i-1)5
g = H(i-1)6
h = H(i-1)7
```

3. Perform the main hash computation:

```
For t = 0 to 79
   T1 = h + BSIG1(e) + CH(e,f,g) + Kt + Wt
   T2 = BSIG0(a) + MAJ(a,b,c)
   h = g
   g = f
   f = e
```

```
        e = d + T1
        d = c
        c = b
        b = a
        a = T1 + T2
```

4. Compute the intermediate hash value H(i):

```
   H(i)0 = a + H(i-1)0
   H(i)1 = b + H(i-1)1
   H(i)2 = c + H(i-1)2
   H(i)3 = d + H(i-1)3
   H(i)4 = e + H(i-1)4
   H(i)5 = f + H(i-1)5
   H(i)6 = g + H(i-1)6
   H(i)7 = h + H(i-1)7
```

After the above computations have been sequentially performed for all of the blocks in the message, the final output is calculated. For SHA-512, this is the concatenation of all of H(N)0, H(N)1, through H(N)7. For SHA-384, this is the concatenation of H(N)0, H(N)1, through H(N)5.

7. SHA-Based HMACs

HMAC is a method for computing a keyed MAC (message authentication code) using a hash function as described in [RFC2104]. It uses a key to mix in with the input text to produce the final hash.

Sample code is also provided, in Section 8.3 below, to perform HMAC based on any of the SHA algorithms described herein. The sample code found in [RFC2104] was written in terms of a specified text size. Since SHA is defined in terms of an arbitrary number of bits, the sample HMAC code has been written to allow the text input to HMAC to have an arbitrary number of octets and bits. A fixed-length interface is also provided.

8. C Code for SHAs

Below is a demonstration implementation of these secure hash functions in C. Section 8.1 contains the header file sha.h, which declares all constants, structures, and functions used by the sha and hmac functions. Section 8.2 contains the C code for sha1.c, sha224-256.c, sha384-512.c, and usha.c along with sha-private.h, which provides some declarations common to all the sha functions. Section 8.3 contains the C code for the hmac functions. Section 8.4 contains a test driver to exercise the code.

For each of the digest length $$$, there is the following set of constants, a structure, and functions:

```
Constants:
   SHA$$$HashSize      number of octets in the hash
   SHA$$$HashSizeBits  number of bits in the hash
   SHA$$$_Message_Block_Size
                       number of octets used in the intermediate
                       message blocks
   shaSuccess = 0      constant returned by each function on success
   shaNull = 1         constant returned by each function when
                       presented with a null pointer parameter
   shaInputTooLong = 2  constant returned by each function when the
                       input data is too long
   shaStateError       constant returned by each function when
                       SHA$$$Input is called after SHA$$$FinalBits or
                       SHA$$$Result.

Structure:
   typedef SHA$$$Context
                       an opaque structure holding the complete state
                       for producing the hash

Functions:
               int SHA$$$Reset(SHA$$$Context *);
         Reset the hash context state
   int SHA$$$Input(SHA$$$Context *, const uint8_t *octets,
               unsigned int bytecount);
         Incorporate bytecount octets into the hash.
   int SHA$$$FinalBits(SHA$$$Context *, const uint8_t octet,
               unsigned int bitcount);
         Incorporate bitcount bits into the hash.  The bits are in
         the upper portion of the octet.  SHA$$$Input() cannot be
         called after this.
   int SHA$$$Result(SHA$$$Context *,
               uint8_t Message_Digest[SHA$$$HashSize]);
         Do the final calculations on the hash and copy the value
         into Message_Digest.
```

In addition, functions with the prefix USHA are provided that take a SHAversion value (SHA$$$) to select the SHA function suite. They add the following constants, structure, and functions:

```
Constants:
   shaBadParam         constant returned by USHA functions when
                       presented with a bad SHAversion (SHA$$$)
                       parameter
```

```
      SHA$$$              SHAversion enumeration values, used by usha
                          and hmac functions to select the SHA function
                          suite

   Structure:
      typedef USHAContext
                          an opaque structure holding the complete state
                          for producing the hash

   Functions:
      int USHAReset(USHAContext *, SHAversion whichSha);
            Reset the hash context state.
      int USHAInput(USHAContext *,
                  const uint8_t *bytes, unsigned int bytecount);
            Incorporate bytecount octets into the hash.
      int USHAFinalBits(USHAContext *,
                  const uint8_t bits, unsigned int bitcount);
                  Incorporate bitcount bits into the hash.
      int USHAResult(USHAContext *,
                  uint8_t Message_Digest[USHAMaxHashSize]);
            Do the final calculations on the hash and copy the value
            into Message_Digest.  Octets in Message_Digest beyond
      USHAHashSize(whichSha) are left untouched.
                  int USHAHashSize(enum SHAversion whichSha);
            The number of octets in the given hash.
      int USHAHashSizeBits(enum SHAversion whichSha);
            The number of bits in the given hash.
      int USHABlockSize(enum SHAversion whichSha);
            The internal block size for the given hash.

   The hmac functions follow the same pattern to allow any length of
   text input to be used.

   Structure:
      typedef HMACContext an opaque structure holding the complete state
                          for producing the hash

   Functions:
      int hmacReset(HMACContext *ctx, enum SHAversion whichSha,
                  const unsigned char *key, int key_len);
            Reset the hash context state.
      int hmacInput(HMACContext *ctx, const unsigned char *text,
                  int text_len);
            Incorporate text_len octets into the hash.
      int hmacFinalBits(HMACContext *ctx, const uint8_t bits,
                  unsigned int bitcount);
            Incorporate bitcount bits into the hash.
```

```
      int hmacResult(HMACContext *ctx,
                  uint8_t Message_Digest[USHAMaxHashSize]);
            Do the final calculations on the hash and copy the value
            into Message_Digest.  Octets in Message_Digest beyond
            USHAHashSize(whichSha) are left untouched.
```

In addition, a combined interface is provided, similar to that shown in RFC 2104, that allows a fixed-length text input to be used.

```
      int hmac(SHAversion whichSha,
                  const unsigned char *text, int text_len,
                  const unsigned char *key, int key_len,
                  uint8_t Message_Digest[USHAMaxHashSize]);
            Calculate the given digest for the given text and key, and
            return the resulting hash.  Octets in Message_Digest beyond
            USHAHashSize(whichSha) are left untouched.
```

8.1. The .h File

```
/*************************** sha.h ***************************/
/****************** See RFC 4634 for details ******************/
#ifndef _SHA_H_
#define _SHA_H_

/*
 *  Description:
 *      This file implements the Secure Hash Signature Standard
 *      algorithms as defined in the National Institute of Standards
 *      and Technology Federal Information Processing Standards
 *      Publication (FIPS PUB) 180-1 published on April 17, 1995, 180-2
 *      published on August 1, 2002, and the FIPS PUB 180-2 Change
 *      Notice published on February 28, 2004.
 *
 *      A combined document showing all algorithms is available at
 *              http://csrc.nist.gov/publications/fips/
 *              fips180-2/fips180-2withchangenotice.pdf
 *
 *      The five hashes are defined in these sizes:
 *              SHA-1           20 byte / 160 bit
 *              SHA-224         28 byte / 224 bit
 *              SHA-256         32 byte / 256 bit
 *              SHA-384         48 byte / 384 bit
 *              SHA-512         64 byte / 512 bit
 */

#include <stdint.h>
/*
 * If you do not have the ISO standard stdint.h header file, then you
```

```
 * must typedef the following:
 *    name             meaning
 *  uint64_t         unsigned 64 bit integer
 *  uint32_t         unsigned 32 bit integer
 *  uint8_t          unsigned 8 bit integer (i.e., unsigned char)
 *  int_least16_t    integer of >= 16 bits
 *
 */

#ifndef _SHA_enum_
#define _SHA_enum_
/*
 *  All SHA functions return one of these values.
 */
enum {
    shaSuccess = 0,
    shaNull,            /* Null pointer parameter */
    shaInputTooLong,    /* input data too long */
    shaStateError,      /* called Input after FinalBits or Result */
    shaBadParam         /* passed a bad parameter */
};
#endif /* _SHA_enum_ */

/*
 *  These constants hold size information for each of the SHA
 *  hashing operations
 */
enum {
    SHA1_Message_Block_Size = 64, SHA224_Message_Block_Size = 64,
    SHA256_Message_Block_Size = 64, SHA384_Message_Block_Size = 128,
    SHA512_Message_Block_Size = 128,
    USHA_Max_Message_Block_Size = SHA512_Message_Block_Size,

    SHA1HashSize = 20, SHA224HashSize = 28, SHA256HashSize = 32,
    SHA384HashSize = 48, SHA512HashSize = 64,
    USHAMaxHashSize = SHA512HashSize,

    SHA1HashSizeBits = 160, SHA224HashSizeBits = 224,
    SHA256HashSizeBits = 256, SHA384HashSizeBits = 384,
    SHA512HashSizeBits = 512, USHAMaxHashSizeBits = SHA512HashSizeBits
};

/*
 *  These constants are used in the USHA (unified sha) functions.
 */
typedef enum SHAversion {
    SHA1, SHA224, SHA256, SHA384, SHA512
} SHAversion;
```

```
/*
 *  This structure will hold context information for the SHA-1
 *  hashing operation.
 */
typedef struct SHA1Context {
    uint32_t Intermediate_Hash[SHA1HashSize/4]; /* Message Digest */

    uint32_t Length_Low;                /* Message length in bits */
    uint32_t Length_High;               /* Message length in bits */

    int_least16_t Message_Block_Index;  /* Message_Block array index */
                                        /* 512-bit message blocks */
    uint8_t Message_Block[SHA1_Message_Block_Size];

    int Computed;                       /* Is the digest computed? */
    int Corrupted;                      /* Is the digest corrupted? */
} SHA1Context;

/*
 *  This structure will hold context information for the SHA-256
 *  hashing operation.
 */
typedef struct SHA256Context {
    uint32_t Intermediate_Hash[SHA256HashSize/4]; /* Message Digest */

    uint32_t Length_Low;                /* Message length in bits */
    uint32_t Length_High;               /* Message length in bits */

    int_least16_t Message_Block_Index;  /* Message_Block array index */
                                        /* 512-bit message blocks */
    uint8_t Message_Block[SHA256_Message_Block_Size];

    int Computed;                       /* Is the digest computed? */
    int Corrupted;                      /* Is the digest corrupted? */
} SHA256Context;

/*
 *  This structure will hold context information for the SHA-512
 *  hashing operation.
 */
typedef struct SHA512Context {
#ifdef USE_32BIT_ONLY
    uint32_t Intermediate_Hash[SHA512HashSize/4]; /* Message Digest  */
    uint32_t Length[4];                 /* Message length in bits */
#else /* !USE_32BIT_ONLY */
    uint64_t Intermediate_Hash[SHA512HashSize/8]; /* Message Digest */
    uint64_t Length_Low, Length_High;   /* Message length in bits */
#endif /* USE_32BIT_ONLY */
```

```
    int_least16_t Message_Block_Index;  /* Message_Block array index */
                                        /* 1024-bit message blocks */
    uint8_t Message_Block[SHA512_Message_Block_Size];

    int Computed;                       /* Is the digest computed?*/
    int Corrupted;                      /* Is the digest corrupted? */
} SHA512Context;

/*
 *  This structure will hold context information for the SHA-224
 *  hashing operation. It uses the SHA-256 structure for computation.
 */
typedef struct SHA256Context SHA224Context;

/*
 *  This structure will hold context information for the SHA-384
 *  hashing operation. It uses the SHA-512 structure for computation.
 */
typedef struct SHA512Context SHA384Context;

/*
 *  This structure holds context information for all SHA
 *  hashing operations.
 */
typedef struct USHAContext {
    int whichSha;               /* which SHA is being used */
    union {
      SHA1Context sha1Context;
      SHA224Context sha224Context; SHA256Context sha256Context;
      SHA384Context sha384Context; SHA512Context sha512Context;
    } ctx;
} USHAContext;

/*
 *  This structure will hold context information for the HMAC
 *  keyed hashing operation.
 */
typedef struct HMACContext {
    int whichSha;               /* which SHA is being used */
    int hashSize;               /* hash size of SHA being used */
    int blockSize;              /* block size of SHA being used */
    USHAContext shaContext;     /* SHA context */
    unsigned char k_opad[USHA_Max_Message_Block_Size];
                        /* outer padding - key XORd with opad */
} HMACContext;
```

```
/*
 *  Function Prototypes
 */

/* SHA-1 */
extern int SHA1Reset(SHA1Context *);
extern int SHA1Input(SHA1Context *, const uint8_t *bytes,
                     unsigned int bytecount);
extern int SHA1FinalBits(SHA1Context *, const uint8_t bits,
                         unsigned int bitcount);
extern int SHA1Result(SHA1Context *,
                      uint8_t Message_Digest[SHA1HashSize]);

/* SHA-224 */
extern int SHA224Reset(SHA224Context *);
extern int SHA224Input(SHA224Context *, const uint8_t *bytes,
                       unsigned int bytecount);
extern int SHA224FinalBits(SHA224Context *, const uint8_t bits,
                           unsigned int bitcount);
extern int SHA224Result(SHA224Context *,
                        uint8_t Message_Digest[SHA224HashSize]);

/* SHA-256 */
extern int SHA256Reset(SHA256Context *);
extern int SHA256Input(SHA256Context *, const uint8_t *bytes,
                       unsigned int bytecount);
extern int SHA256FinalBits(SHA256Context *, const uint8_t bits,
                           unsigned int bitcount);
extern int SHA256Result(SHA256Context *,
                        uint8_t Message_Digest[SHA256HashSize]);

/* SHA-384 */
extern int SHA384Reset(SHA384Context *);
extern int SHA384Input(SHA384Context *, const uint8_t *bytes,
                       unsigned int bytecount);
extern int SHA384FinalBits(SHA384Context *, const uint8_t bits,
                           unsigned int bitcount);
extern int SHA384Result(SHA384Context *,
                        uint8_t Message_Digest[SHA384HashSize]);

/* SHA-512 */
extern int SHA512Reset(SHA512Context *);
extern int SHA512Input(SHA512Context *, const uint8_t *bytes,
                       unsigned int bytecount);
extern int SHA512FinalBits(SHA512Context *, const uint8_t bits,
                           unsigned int bitcount);
extern int SHA512Result(SHA512Context *,
                        uint8_t Message_Digest[SHA512HashSize]);
```

```
/* Unified SHA functions, chosen by whichSha */
extern int USHAReset(USHAContext *, SHAversion whichSha);
extern int USHAInput(USHAContext *,
                     const uint8_t *bytes, unsigned int bytecount);
extern int USHAFinalBits(USHAContext *,
                         const uint8_t bits, unsigned int bitcount);
extern int USHAResult(USHAContext *,
                      uint8_t Message_Digest[USHAMaxHashSize]);
extern int USHABlockSize(enum SHAversion whichSha);
extern int USHAHashSize(enum SHAversion whichSha);
extern int USHAHashSizeBits(enum SHAversion whichSha);

/*
 * HMAC Keyed-Hashing for Message Authentication, RFC2104,
 * for all SHAs.
 * This interface allows a fixed-length text input to be used.
 */
extern int hmac(SHAversion whichSha, /* which SHA algorithm to use */
    const unsigned char *text,     /* pointer to data stream */
    int text_len,                  /* length of data stream */
    const unsigned char *key,      /* pointer to authentication key */
    int key_len,                   /* length of authentication key */
    uint8_t digest[USHAMaxHashSize]); /* caller digest to fill in */

/*
 * HMAC Keyed-Hashing for Message Authentication, RFC2104,
 * for all SHAs.
 * This interface allows any length of text input to be used.
 */
extern int hmacReset(HMACContext *ctx, enum SHAversion whichSha,
                     const unsigned char *key, int key_len);
extern int hmacInput(HMACContext *ctx, const unsigned char *text,
                     int text_len);

extern int hmacFinalBits(HMACContext *ctx, const uint8_t bits,
                         unsigned int bitcount);
extern int hmacResult(HMACContext *ctx,
                      uint8_t digest[USHAMaxHashSize]);

#endif /* _SHA_H_ */
```

8.2. The SHA Code

This code is primarily intended as expository and could be optimized further. For example, the assignment rotations through the variables a, b, ..., h could be treated as a cycle and the loop unrolled, rather than doing the explicit copying.

Note that there are alternative representations of the Ch() and Maj() functions controlled by an ifdef.

8.2.1. sha1.c

```
/*************************** sha1.c ***************************/
/******************** See RFC 4634 for details ******************/
/*
 *  Description:
 *      This file implements the Secure Hash Signature Standard
 *      algorithms as defined in the National Institute of Standards
 *      and Technology Federal Information Processing Standards
 *      Publication (FIPS PUB) 180-1 published on April 17, 1995, 180-2
 *      published on August 1, 2002, and the FIPS PUB 180-2 Change
 *      Notice published on February 28, 2004.
 *
 *      A combined document showing all algorithms is available at
 *              http://csrc.nist.gov/publications/fips/
 *              fips180-2/fips180-2withchangenotice.pdf
 *
 *      The SHA-1 algorithm produces a 160-bit message digest for a
 *      given data stream.  It should take about 2**n steps to find a
 *      message with the same digest as a given message and
 *      2**(n/2) to find any two messages with the same digest,
 *      when n is the digest size in bits.  Therefore, this
 *      algorithm can serve as a means of providing a
 *      "fingerprint" for a message.
 *
 *  Portability Issues:
 *      SHA-1 is defined in terms of 32-bit "words".  This code
 *      uses <stdint.h> (included via "sha.h") to define 32 and 8
 *      bit unsigned integer types.  If your C compiler does not
 *      support 32 bit unsigned integers, this code is not
 *      appropriate.
 *
 *  Caveats:
 *      SHA-1 is designed to work with messages less than 2^64 bits
 *      long. This implementation uses SHA1Input() to hash the bits
 *      that are a multiple of the size of an 8-bit character, and then
 *      uses SHA1FinalBits() to hash the final few bits of the input.
 */
```

```
#include "sha.h"
#include "sha-private.h"

/*
 *  Define the SHA1 circular left shift macro
 */
#define SHA1_ROTL(bits,word) \
                (((word) << (bits)) | ((word) >> (32-(bits))))

/*
 * add "length" to the length
 */
static uint32_t addTemp;
#define SHA1AddLength(context, length)                    \
    (addTemp = (context)->Length_Low,                     \
     (context)->Corrupted =                               \
        (((context)->Length_Low += (length)) < addTemp) && \
        (++(context)->Length_High == 0) ? 1 : 0)

/* Local Function Prototypes */
static void SHA1Finalize(SHA1Context *context, uint8_t Pad_Byte);
static void SHA1PadMessage(SHA1Context *, uint8_t Pad_Byte);
static void SHA1ProcessMessageBlock(SHA1Context *);

/*
 *  SHA1Reset
 *
 *  Description:
 *      This function will initialize the SHA1Context in preparation
 *      for computing a new SHA1 message digest.
 *
 *  Parameters:
 *      context: [in/out]
 *          The context to reset.
 *
 *  Returns:
 *      sha Error Code.
 *
 */
int SHA1Reset(SHA1Context *context)
{
    if (!context)
        return shaNull;

    context->Length_Low             = 0;
    context->Length_High            = 0;
    context->Message_Block_Index    = 0;
```

```
    /* Initial Hash Values: FIPS-180-2 section 5.3.1 */
    context->Intermediate_Hash[0]   = 0x67452301;
    context->Intermediate_Hash[1]   = 0xEFCDAB89;
    context->Intermediate_Hash[2]   = 0x98BADCFE;
    context->Intermediate_Hash[3]   = 0x10325476;
    context->Intermediate_Hash[4]   = 0xC3D2E1F0;

    context->Computed   = 0;
    context->Corrupted  = 0;

    return shaSuccess;
}

/*
 *  SHA1Input
 *
 *  Description:
 *      This function accepts an array of octets as the next portion
 *      of the message.
 *
 *  Parameters:
 *      context: [in/out]
 *          The SHA context to update
 *      message_array: [in]
 *          An array of characters representing the next portion of
 *          the message.
 *      length: [in]
 *          The length of the message in message_array
 *
 *  Returns:
 *      sha Error Code.
 *
 */
int SHA1Input(SHA1Context *context,
    const uint8_t *message_array, unsigned length)
{
  if (!length)
    return shaSuccess;

  if (!context || !message_array)
    return shaNull;

  if (context->Computed) {
    context->Corrupted = shaStateError;
    return shaStateError;
  }

  if (context->Corrupted)
```

```
     return context->Corrupted;

  while (length-- && !context->Corrupted) {
    context->Message_Block[context->Message_Block_Index++] =
      (*message_array & 0xFF);

    if (!SHA1AddLength(context, 8) &&
      (context->Message_Block_Index == SHA1_Message_Block_Size))
      SHA1ProcessMessageBlock(context);

    message_array++;
  }

  return shaSuccess;
}

/*
 * SHA1FinalBits
 *
 * Description:
 *   This function will add in any final bits of the message.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   message_bits: [in]
 *     The final bits of the message, in the upper portion of the
 *     byte. (Use 0b###00000 instead of 0b00000### to input the
 *     three bits ###.)
 *   length: [in]
 *     The number of bits in message_bits, between 1 and 7.
 *
 * Returns:
 *   sha Error Code.
 */
int SHA1FinalBits(SHA1Context *context, const uint8_t message_bits,
    unsigned int length)
{
  uint8_t masks[8] = {
      /* 0 0b00000000 */ 0x00, /* 1 0b10000000 */ 0x80,
      /* 2 0b11000000 */ 0xC0, /* 3 0b11100000 */ 0xE0,
      /* 4 0b11110000 */ 0xF0, /* 5 0b11111000 */ 0xF8,
      /* 6 0b11111100 */ 0xFC, /* 7 0b11111110 */ 0xFE
  };
  uint8_t markbit[8] = {
      /* 0 0b10000000 */ 0x80, /* 1 0b01000000 */ 0x40,
      /* 2 0b00100000 */ 0x20, /* 3 0b00010000 */ 0x10,
      /* 4 0b00001000 */ 0x08, /* 5 0b00000100 */ 0x04,
```

```
      /* 6 0b00000010 */ 0x02, /* 7 0b00000001 */ 0x01
  };

  if (!length)
    return shaSuccess;

  if (!context)
    return shaNull;

  if (context->Computed || (length >= 8) || (length == 0)) {
    context->Corrupted = shaStateError;
    return shaStateError;
  }

  if (context->Corrupted)
     return context->Corrupted;

  SHA1AddLength(context, length);
  SHA1Finalize(context,
    (uint8_t) ((message_bits & masks[length]) | markbit[length]));

  return shaSuccess;
}

/*
 * SHA1Result
 *
 * Description:
 *   This function will return the 160-bit message digest into the
 *   Message_Digest array provided by the caller.
 *   NOTE: The first octet of hash is stored in the 0th element,
 *      the last octet of hash in the 19th element.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to use to calculate the SHA-1 hash.
 *   Message_Digest: [out]
 *     Where the digest is returned.
 *
 * Returns:
 *   sha Error Code.
 *
 */
int SHA1Result(SHA1Context *context,
    uint8_t Message_Digest[SHA1HashSize])
{
  int i;
```

```
  if (!context || !Message_Digest)
    return shaNull;

  if (context->Corrupted)
    return context->Corrupted;

  if (!context->Computed)
    SHA1Finalize(context, 0x80);

  for (i = 0; i < SHA1HashSize; ++i)
    Message_Digest[i] = (uint8_t) (context->Intermediate_Hash[i>>2]
              >> 8 * ( 3 - ( i & 0x03 ) ));

  return shaSuccess;
}

/*
 * SHA1Finalize
 *
 * Description:
 *   This helper function finishes off the digest calculations.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   Pad_Byte: [in]
 *     The last byte to add to the digest before the 0-padding
 *     and length. This will contain the last bits of the message
 *     followed by another single bit. If the message was an
 *     exact multiple of 8-bits long, Pad_Byte will be 0x80.
 *
 * Returns:
 *   sha Error Code.
 *
 */
static void SHA1Finalize(SHA1Context *context, uint8_t Pad_Byte)
{
  int i;
  SHA1PadMessage(context, Pad_Byte);
  /* message may be sensitive, clear it out */
  for (i = 0; i < SHA1_Message_Block_Size; ++i)
    context->Message_Block[i] = 0;
  context->Length_Low = 0;  /* and clear length */
  context->Length_High = 0;
  context->Computed = 1;
}

/*
```

```
 * SHA1PadMessage
 *
 * Description:
 *   According to the standard, the message must be padded to an
 *   even 512 bits. The first padding bit must be a '1'. The last
 *   64 bits represent the length of the original message. All bits
 *   in between should be 0. This helper function will pad the
 *   message according to those rules by filling the Message_Block
 *   array accordingly. When it returns, it can be assumed that the
 *   message digest has been computed.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to pad
 *   Pad_Byte: [in]
 *     The last byte to add to the digest before the 0-padding
 *     and length. This will contain the last bits of the message
 *     followed by another single bit. If the message was an
 *     exact multiple of 8-bits long, Pad_Byte will be 0x80.
 *
 * Returns:
 *   Nothing.
 */
static void SHA1PadMessage(SHA1Context *context, uint8_t Pad_Byte)
{
  /*
   * Check to see if the current message block is too small to hold
   * the initial padding bits and length. If so, we will pad the
   * block, process it, and then continue padding into a second
   * block.
   */
  if (context->Message_Block_Index >= (SHA1_Message_Block_Size - 8)) {
    context->Message_Block[context->Message_Block_Index++] = Pad_Byte;
    while (context->Message_Block_Index < SHA1_Message_Block_Size)
      context->Message_Block[context->Message_Block_Index++] = 0;

    SHA1ProcessMessageBlock(context);
  } else
    context->Message_Block[context->Message_Block_Index++] = Pad_Byte;

  while (context->Message_Block_Index < (SHA1_Message_Block_Size - 8))
    context->Message_Block[context->Message_Block_Index++] = 0;

  /*
   * Store the message length as the last 8 octets
   */
  context->Message_Block[56] = (uint8_t) (context->Length_High >> 24);
  context->Message_Block[57] = (uint8_t) (context->Length_High >> 16);
```

```
  context->Message_Block[58] = (uint8_t) (context->Length_High >> 8);
  context->Message_Block[59] = (uint8_t) (context->Length_High);
  context->Message_Block[60] = (uint8_t) (context->Length_Low >> 24);
  context->Message_Block[61] = (uint8_t) (context->Length_Low >> 16);
  context->Message_Block[62] = (uint8_t) (context->Length_Low >> 8);
  context->Message_Block[63] = (uint8_t) (context->Length_Low);

  SHA1ProcessMessageBlock(context);
}

/*
 * SHA1ProcessMessageBlock
 *
 * Description:
 *   This helper function will process the next 512 bits of the
 *   message stored in the Message_Block array.
 *
 * Parameters:
 *   None.
 *
 * Returns:
 *   Nothing.
 *
 * Comments:
 *   Many of the variable names in this code, especially the
 *   single character names, were used because those were the
 *   names used in the publication.
 */
static void SHA1ProcessMessageBlock(SHA1Context *context)
{
  /* Constants defined in FIPS-180-2, section 4.2.1 */
  const uint32_t K[4] = {
      0x5A827999, 0x6ED9EBA1, 0x8F1BBCDC, 0xCA62C1D6
  };
  int        t;               /* Loop counter */
  uint32_t   temp;            /* Temporary word value */
  uint32_t   W[80];           /* Word sequence */
  uint32_t   A, B, C, D, E;   /* Word buffers */

  /*
   * Initialize the first 16 words in the array W
   */
  for (t = 0; t < 16; t++) {
    W[t]  = ((uint32_t)context->Message_Block[t * 4]) << 24;
    W[t] |= ((uint32_t)context->Message_Block[t * 4 + 1]) << 16;
    W[t] |= ((uint32_t)context->Message_Block[t * 4 + 2]) << 8;
    W[t] |= ((uint32_t)context->Message_Block[t * 4 + 3]);
  }
```

```
  for (t = 16; t < 80; t++)
    W[t] = SHA1_ROTL(1, W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]);

  A = context->Intermediate_Hash[0];
  B = context->Intermediate_Hash[1];
  C = context->Intermediate_Hash[2];
  D = context->Intermediate_Hash[3];
  E = context->Intermediate_Hash[4];

  for (t = 0; t < 20; t++) {
    temp = SHA1_ROTL(5,A) + SHA_Ch(B, C, D) + E + W[t] + K[0];
    E = D;
    D = C;
    C = SHA1_ROTL(30,B);
    B = A;
    A = temp;
  }

  for (t = 20; t < 40; t++) {
    temp = SHA1_ROTL(5,A) + SHA_Parity(B, C, D) + E + W[t] + K[1];
    E = D;
    D = C;
    C = SHA1_ROTL(30,B);
    B = A;
    A = temp;
  }

  for (t = 40; t < 60; t++) {
    temp = SHA1_ROTL(5,A) + SHA_Maj(B, C, D) + E + W[t] + K[2];
    E = D;
    D = C;
    C = SHA1_ROTL(30,B);
    B = A;
    A = temp;
  }

  for (t = 60; t < 80; t++) {
    temp = SHA1_ROTL(5,A) + SHA_Parity(B, C, D) + E + W[t] + K[3];
    E = D;
    D = C;
    C = SHA1_ROTL(30,B);
    B = A;
    A = temp;
  }

  context->Intermediate_Hash[0] += A;
  context->Intermediate_Hash[1] += B;
  context->Intermediate_Hash[2] += C;
```

```
  context->Intermediate_Hash[3] += D;
  context->Intermediate_Hash[4] += E;

  context->Message_Block_Index = 0;
}
```

8.2.2. sha224-256.c

```
/*************************** sha224-256.c ***************************/
/********************** See RFC 4634 for details **********************/
/*
 * Description:
 *   This file implements the Secure Hash Signature Standard
 *   algorithms as defined in the National Institute of Standards
 *   and Technology Federal Information Processing Standards
 *   Publication (FIPS PUB) 180-1 published on April 17, 1995, 180-2
 *   published on August 1, 2002, and the FIPS PUB 180-2 Change
 *   Notice published on February 28, 2004.
 *
 *   A combined document showing all algorithms is available at
 *       http://csrc.nist.gov/publications/fips/
 *       fips180-2/fips180-2withchangenotice.pdf
 *
 *   The SHA-224 and SHA-256 algorithms produce 224-bit and 256-bit
 *   message digests for a given data stream. It should take about
 *   2**n steps to find a message with the same digest as a given
 *   message and 2**(n/2) to find any two messages with the same
 *   digest, when n is the digest size in bits. Therefore, this
 *   algorithm can serve as a means of providing a
 *   "fingerprint" for a message.
 *
 * Portability Issues:
 *   SHA-224 and SHA-256 are defined in terms of 32-bit "words".
 *   This code uses <stdint.h> (included via "sha.h") to define 32
 *   and 8 bit unsigned integer types. If your C compiler does not
 *   support 32 bit unsigned integers, this code is not
 *   appropriate.
 *
 * Caveats:
 *   SHA-224 and SHA-256 are designed to work with messages less
 *   than 2^64 bits long. This implementation uses SHA224/256Input()
 *   to hash the bits that are a multiple of the size of an 8-bit
 *   character, and then uses SHA224/256FinalBits() to hash the
 *   final few bits of the input.
 */

#include "sha.h"
#include "sha-private.h"
```

```
/* Define the SHA shift, rotate left and rotate right macro */
#define SHA256_SHR(bits,word)      ((word) >> (bits))
#define SHA256_ROTL(bits,word)                          \
  (((word) << (bits)) | ((word) >> (32-(bits))))
#define SHA256_ROTR(bits,word)                          \
  (((word) >> (bits)) | ((word) << (32-(bits))))

/* Define the SHA SIGMA and sigma macros */
#define SHA256_SIGMA0(word)   \
  (SHA256_ROTR( 2,word) ^ SHA256_ROTR(13,word) ^ SHA256_ROTR(22,word))
#define SHA256_SIGMA1(word)   \
  (SHA256_ROTR( 6,word) ^ SHA256_ROTR(11,word) ^ SHA256_ROTR(25,word))
#define SHA256_sigma0(word)   \
  (SHA256_ROTR( 7,word) ^ SHA256_ROTR(18,word) ^ SHA256_SHR( 3,word))
#define SHA256_sigma1(word)   \
  (SHA256_ROTR(17,word) ^ SHA256_ROTR(19,word) ^ SHA256_SHR(10,word))

/*
 * add "length" to the length
 */
static uint32_t addTemp;
#define SHA224_256AddLength(context, length)               \
  (addTemp = (context)->Length_Low, (context)->Corrupted = \
    (((context)->Length_Low += (length)) < addTemp) &&     \
    (++(context)->Length_High == 0) ? 1 : 0)

/* Local Function Prototypes */
static void SHA224_256Finalize(SHA256Context *context,
  uint8_t Pad_Byte);
static void SHA224_256PadMessage(SHA256Context *context,
  uint8_t Pad_Byte);
static void SHA224_256ProcessMessageBlock(SHA256Context *context);
static int SHA224_256Reset(SHA256Context *context, uint32_t *H0);
static int SHA224_256ResultN(SHA256Context *context,
  uint8_t Message_Digest[], int HashSize);

/* Initial Hash Values: FIPS-180-2 Change Notice 1 */
static uint32_t SHA224_H0[SHA256HashSize/4] = {
    0xC1059ED8, 0x367CD507, 0x3070DD17, 0xF70E5939,
    0xFFC00B31, 0x68581511, 0x64F98FA7, 0xBEFA4FA4
};

/* Initial Hash Values: FIPS-180-2 section 5.3.2 */
static uint32_t SHA256_H0[SHA256HashSize/4] = {
  0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,
  0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19
};
```

```
/*
 * SHA224Reset
 *
 * Description:
 *   This function will initialize the SHA384Context in preparation
 *   for computing a new SHA224 message digest.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to reset.
 *
 * Returns:
 *   sha Error Code.
 */
int SHA224Reset(SHA224Context *context)
{
  return SHA224_256Reset(context, SHA224_H0);
}

/*
 * SHA224Input
 *
 * Description:
 *   This function accepts an array of octets as the next portion
 *   of the message.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   message_array: [in]
 *     An array of characters representing the next portion of
 *     the message.
 *   length: [in]
 *     The length of the message in message_array
 *
 * Returns:
 *   sha Error Code.
 *
 */
int SHA224Input(SHA224Context *context, const uint8_t *message_array,
    unsigned int length)
{
  return SHA256Input(context, message_array, length);
}

/*
 * SHA224FinalBits
 *
```

```
 * Description:
 *   This function will add in any final bits of the message.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   message_bits: [in]
 *     The final bits of the message, in the upper portion of the
 *     byte. (Use 0b###00000 instead of 0b00000### to input the
 *     three bits ###.)
 *   length: [in]
 *     The number of bits in message_bits, between 1 and 7.
 *
 * Returns:
 *   sha Error Code.
 */
int SHA224FinalBits( SHA224Context *context,
    const uint8_t message_bits, unsigned int length)
{
  return SHA256FinalBits(context, message_bits, length);
}

/*
 * SHA224Result
 *
 * Description:
 *   This function will return the 224-bit message
 *   digest into the Message_Digest array provided by the caller.
 *   NOTE: The first octet of hash is stored in the 0th element,
 *      the last octet of hash in the 28th element.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to use to calculate the SHA hash.
 *   Message_Digest: [out]
 *     Where the digest is returned.
 *
 * Returns:
 *   sha Error Code.
 */
int SHA224Result(SHA224Context *context,
    uint8_t Message_Digest[SHA224HashSize])
{
  return SHA224_256ResultN(context, Message_Digest, SHA224HashSize);
}

/*
 * SHA256Reset
```

```
 *
 * Description:
 *   This function will initialize the SHA256Context in preparation
 *   for computing a new SHA256 message digest.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to reset.
 *
 * Returns:
 *   sha Error Code.
 */
int SHA256Reset(SHA256Context *context)
{
  return SHA224_256Reset(context, SHA256_H0);
}

/*
 * SHA256Input
 *
 * Description:
 *   This function accepts an array of octets as the next portion
 *   of the message.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   message_array: [in]
 *     An array of characters representing the next portion of
 *     the message.
 *   length: [in]
 *     The length of the message in message_array
 *
 * Returns:
 *   sha Error Code.
 */
int SHA256Input(SHA256Context *context, const uint8_t *message_array,
    unsigned int length)
{
  if (!length)
    return shaSuccess;

  if (!context || !message_array)
    return shaNull;

  if (context->Computed) {
    context->Corrupted = shaStateError;
    return shaStateError;
```

```
  }

  if (context->Corrupted)
     return context->Corrupted;

  while (length-- && !context->Corrupted) {
    context->Message_Block[context->Message_Block_Index++] =
            (*message_array & 0xFF);

    if (!SHA224_256AddLength(context, 8) &&
      (context->Message_Block_Index == SHA256_Message_Block_Size))
      SHA224_256ProcessMessageBlock(context);

    message_array++;
  }

  return shaSuccess;

}

/*
 * SHA256FinalBits
 *
 * Description:
 *   This function will add in any final bits of the message.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   message_bits: [in]
 *     The final bits of the message, in the upper portion of the
 *     byte. (Use 0b###00000 instead of 0b00000### to input the
 *     three bits ###.)
 *   length: [in]
 *     The number of bits in message_bits, between 1 and 7.
 *
 * Returns:
 *   sha Error Code.
 */
int SHA256FinalBits(SHA256Context *context,
    const uint8_t message_bits, unsigned int length)
{
  uint8_t masks[8] = {
      /* 0 0b00000000 */ 0x00, /* 1 0b10000000 */ 0x80,
      /* 2 0b11000000 */ 0xC0, /* 3 0b11100000 */ 0xE0,
      /* 4 0b11110000 */ 0xF0, /* 5 0b11111000 */ 0xF8,
      /* 6 0b11111100 */ 0xFC, /* 7 0b11111110 */ 0xFE
  };
```

```
  uint8_t markbit[8] = {
      /* 0 0b10000000 */ 0x80, /* 1 0b01000000 */ 0x40,
      /* 2 0b00100000 */ 0x20, /* 3 0b00010000 */ 0x10,
      /* 4 0b00001000 */ 0x08, /* 5 0b00000100 */ 0x04,
      /* 6 0b00000010 */ 0x02, /* 7 0b00000001 */ 0x01
  };

  if (!length)
    return shaSuccess;

  if (!context)
    return shaNull;

  if ((context->Computed) || (length >= 8) || (length == 0)) {
    context->Corrupted = shaStateError;
    return shaStateError;
  }

  if (context->Corrupted)
    return context->Corrupted;

  SHA224_256AddLength(context, length);
  SHA224_256Finalize(context, (uint8_t)
    ((message_bits & masks[length]) | markbit[length]));

  return shaSuccess;
}

/*
 * SHA256Result
 *
 * Description:
 *   This function will return the 256-bit message
 *   digest into the Message_Digest array provided by the caller.
 *   NOTE: The first octet of hash is stored in the 0th element,
 *      the last octet of hash in the 32nd element.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to use to calculate the SHA hash.
 *   Message_Digest: [out]
 *     Where the digest is returned.
 *
 * Returns:
 *   sha Error Code.
 */
int SHA256Result(SHA256Context *context, uint8_t Message_Digest[])
{
```

```
  return SHA224_256ResultN(context, Message_Digest, SHA256HashSize);
}

/*
 * SHA224_256Finalize
 *
 * Description:
 *   This helper function finishes off the digest calculations.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   Pad_Byte: [in]
 *     The last byte to add to the digest before the 0-padding
 *     and length. This will contain the last bits of the message
 *     followed by another single bit. If the message was an
 *     exact multiple of 8-bits long, Pad_Byte will be 0x80.
 *
 * Returns:
 *   sha Error Code.
 */
static void SHA224_256Finalize(SHA256Context *context,
    uint8_t Pad_Byte)
{
  int i;
  SHA224_256PadMessage(context, Pad_Byte);
  /* message may be sensitive, so clear it out */
  for (i = 0; i < SHA256_Message_Block_Size; ++i)
    context->Message_Block[i] = 0;
  context->Length_Low = 0;  /* and clear length */
  context->Length_High = 0;
  context->Computed = 1;
}

/*
 * SHA224_256PadMessage
 *
 * Description:
 *   According to the standard, the message must be padded to an
 *   even 512 bits. The first padding bit must be a '1'. The
 *   last 64 bits represent the length of the original message.
 *   All bits in between should be 0. This helper function will pad
 *   the message according to those rules by filling the
 *   Message_Block array accordingly. When it returns, it can be
 *   assumed that the message digest has been computed.
 *
 * Parameters:
 *   context: [in/out]
```

```
 *     The context to pad
 *   Pad_Byte: [in]
 *     The last byte to add to the digest before the 0-padding
 *     and length. This will contain the last bits of the message
 *     followed by another single bit. If the message was an
 *     exact multiple of 8-bits long, Pad_Byte will be 0x80.
 *
 * Returns:
 *   Nothing.
 */
static void SHA224_256PadMessage(SHA256Context *context,
    uint8_t Pad_Byte)
{
  /*
   * Check to see if the current message block is too small to hold
   * the initial padding bits and length. If so, we will pad the
   * block, process it, and then continue padding into a second
   * block.
   */
  if (context->Message_Block_Index >= (SHA256_Message_Block_Size-8)) {
    context->Message_Block[context->Message_Block_Index++] = Pad_Byte;
    while (context->Message_Block_Index < SHA256_Message_Block_Size)
      context->Message_Block[context->Message_Block_Index++] = 0;
    SHA224_256ProcessMessageBlock(context);
  } else
    context->Message_Block[context->Message_Block_Index++] = Pad_Byte;

  while (context->Message_Block_Index < (SHA256_Message_Block_Size-8))
    context->Message_Block[context->Message_Block_Index++] = 0;

  /*
   * Store the message length as the last 8 octets
   */
  context->Message_Block[56] = (uint8_t)(context->Length_High >> 24);
  context->Message_Block[57] = (uint8_t)(context->Length_High >> 16);
  context->Message_Block[58] = (uint8_t)(context->Length_High >> 8);
  context->Message_Block[59] = (uint8_t)(context->Length_High);
  context->Message_Block[60] = (uint8_t)(context->Length_Low >> 24);
  context->Message_Block[61] = (uint8_t)(context->Length_Low >> 16);
  context->Message_Block[62] = (uint8_t)(context->Length_Low >> 8);
  context->Message_Block[63] = (uint8_t)(context->Length_Low);

  SHA224_256ProcessMessageBlock(context);
}

/*
 * SHA224_256ProcessMessageBlock
 *
```

```
 * Description:
 *   This function will process the next 512 bits of the message
 *   stored in the Message_Block array.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *
 * Returns:
 *   Nothing.
 *
 * Comments:
 *   Many of the variable names in this code, especially the
 *   single character names, were used because those were the
 *   names used in the publication.
 */
static void SHA224_256ProcessMessageBlock(SHA256Context *context)
{
  /* Constants defined in FIPS-180-2, section 4.2.2 */
  static const uint32_t K[64] = {
      0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b,
      0x59f111f1, 0x923f82a4, 0xab1c5ed5, 0xd807aa98, 0x12835b01,
      0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7,
      0xc19bf174, 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
      0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da, 0x983e5152,
      0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
      0x06ca6351, 0x14292967, 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc,
      0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
      0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819,
      0xd6990624, 0xf40e3585, 0x106aa070, 0x19a4c116, 0x1e376c08,
      0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f,
      0x682e6ff3, 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
      0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
  };
  int        t, t4;                  /* Loop counter */
  uint32_t   temp1, temp2;           /* Temporary word value */
  uint32_t   W[64];                  /* Word sequence */
  uint32_t   A, B, C, D, E, F, G, H;  /* Word buffers */

  /*
   * Initialize the first 16 words in the array W
   */
  for (t = t4 = 0; t < 16; t++, t4 += 4)
    W[t] = (((uint32_t)context->Message_Block[t4]) << 24) |
           (((uint32_t)context->Message_Block[t4 + 1]) << 16) |
           (((uint32_t)context->Message_Block[t4 + 2]) << 8) |
           (((uint32_t)context->Message_Block[t4 + 3]));
```

```
  for (t = 16; t < 64; t++)
    W[t] = SHA256_sigma1(W[t-2]) + W[t-7] +
        SHA256_sigma0(W[t-15]) + W[t-16];

  A = context->Intermediate_Hash[0];
  B = context->Intermediate_Hash[1];
  C = context->Intermediate_Hash[2];
  D = context->Intermediate_Hash[3];
  E = context->Intermediate_Hash[4];
  F = context->Intermediate_Hash[5];
  G = context->Intermediate_Hash[6];
  H = context->Intermediate_Hash[7];

  for (t = 0; t < 64; t++) {
    temp1 = H + SHA256_SIGMA1(E) + SHA_Ch(E,F,G) + K[t] + W[t];
    temp2 = SHA256_SIGMA0(A) + SHA_Maj(A,B,C);
    H = G;
    G = F;
    F = E;
    E = D + temp1;
    D = C;
    C = B;
    B = A;
    A = temp1 + temp2;
  }

  context->Intermediate_Hash[0] += A;
  context->Intermediate_Hash[1] += B;
  context->Intermediate_Hash[2] += C;
  context->Intermediate_Hash[3] += D;
  context->Intermediate_Hash[4] += E;
  context->Intermediate_Hash[5] += F;
  context->Intermediate_Hash[6] += G;
  context->Intermediate_Hash[7] += H;

  context->Message_Block_Index = 0;
}

/*
 * SHA224_256Reset
 *
 * Description:
 *   This helper function will initialize the SHA256Context in
 *   preparation for computing a new SHA256 message digest.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to reset.
```

```
 *   H0
 *     The initial hash value to use.
 *
 * Returns:
 *   sha Error Code.
 */
static int SHA224_256Reset(SHA256Context *context, uint32_t *H0)
{
  if (!context)
    return shaNull;

  context->Length_Low           = 0;
  context->Length_High          = 0;
  context->Message_Block_Index  = 0;

  context->Intermediate_Hash[0] = H0[0];
  context->Intermediate_Hash[1] = H0[1];
  context->Intermediate_Hash[2] = H0[2];
  context->Intermediate_Hash[3] = H0[3];
  context->Intermediate_Hash[4] = H0[4];
  context->Intermediate_Hash[5] = H0[5];
  context->Intermediate_Hash[6] = H0[6];
  context->Intermediate_Hash[7] = H0[7];

  context->Computed  = 0;
  context->Corrupted = 0;

  return shaSuccess;
}

/*
 * SHA224_256ResultN
 *
 * Description:
 *   This helper function will return the 224-bit or 256-bit message
 *   digest into the Message_Digest array provided by the caller.
 *   NOTE: The first octet of hash is stored in the 0th element,
 *      the last octet of hash in the 28th/32nd element.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to use to calculate the SHA hash.
 *   Message_Digest: [out]
 *     Where the digest is returned.
 *   HashSize: [in]
 *     The size of the hash, either 28 or 32.
 *
 * Returns:
```

```
 *   sha Error Code.
 */
static int SHA224_256ResultN(SHA256Context *context,
    uint8_t Message_Digest[], int HashSize)
{
  int i;

  if (!context || !Message_Digest)
    return shaNull;

  if (context->Corrupted)
    return context->Corrupted;

  if (!context->Computed)
    SHA224_256Finalize(context, 0x80);

  for (i = 0; i < HashSize; ++i)
    Message_Digest[i] = (uint8_t)
      (context->Intermediate_Hash[i>>2] >> 8 * ( 3 - ( i & 0x03 ) ));

  return shaSuccess;
}
```

8.2.3. sha384-512.c

```
/************************** sha384-512.c ***************************/
/********************** See RFC 4634 for details ********************/
/*
 * Description:
 *   This file implements the Secure Hash Signature Standard
 *   algorithms as defined in the National Institute of Standards
 *   and Technology Federal Information Processing Standards
 *   Publication (FIPS PUB) 180-1 published on April 17, 1995, 180-2
 *   published on August 1, 2002, and the FIPS PUB 180-2 Change
 *   Notice published on February 28, 2004.
 *
 *   A combined document showing all algorithms is available at
 *       http://csrc.nist.gov/publications/fips/
 *       fips180-2/fips180-2withchangenotice.pdf
 *
 *   The SHA-384 and SHA-512 algorithms produce 384-bit and 512-bit
 *   message digests for a given data stream. It should take about
 *   2**n steps to find a message with the same digest as a given
 *   message and 2**(n/2) to find any two messages with the same
 *   digest, when n is the digest size in bits. Therefore, this
 *   algorithm can serve as a means of providing a
 *   "fingerprint" for a message.
 *
```

```
 * Portability Issues:
 *   SHA-384 and SHA-512 are defined in terms of 64-bit "words",
 *   but if USE_32BIT_ONLY is #defined, this code is implemented in
 *   terms of 32-bit "words". This code uses <stdint.h> (included
 *   via "sha.h") to define the 64, 32 and 8 bit unsigned integer
 *   types. If your C compiler does not support 64 bit unsigned
 *   integers, and you do not #define USE_32BIT_ONLY, this code is
 *   not appropriate.
 *
 * Caveats:
 *   SHA-384 and SHA-512 are designed to work with messages less
 *   than 2^128 bits long. This implementation uses
 *   SHA384/512Input() to hash the bits that are a multiple of the
 *   size of an 8-bit character, and then uses SHA384/256FinalBits()
 *   to hash the final few bits of the input.
 *
 */

#include "sha.h"
#include "sha-private.h"

#ifdef USE_32BIT_ONLY
/*
 * Define 64-bit arithmetic in terms of 32-bit arithmetic.
 * Each 64-bit number is represented in a 2-word array.
 * All macros are defined such that the result is the last parameter.
 */

/*
 * Define shift, rotate left and rotate right functions
 */
#define SHA512_SHR(bits, word, ret) (                          \
    /* (((uint64_t)((word))) >> (bits)) */                     \
    (ret)[0] = (((bits) < 32) && ((bits) >= 0)) ?              \
      ((word)[0] >> (bits)) : 0,                               \
    (ret)[1] = ((bits) > 32) ? ((word)[0] >> ((bits) - 32)) :  \
      ((bits) == 32) ? (word)[0] :                             \
      ((bits) >= 0) ?                                          \
        (((word)[0] << (32 - (bits))) |                        \
        ((word)[1] >> (bits))) : 0 )

#define SHA512_SHL(bits, word, ret) (                          \
    /* (((uint64_t)(word)) << (bits)) */                       \
    (ret)[0] = ((bits) > 32) ? ((word)[1] << ((bits) - 32)) :  \
         ((bits) == 32) ? (word)[1] :                          \
         ((bits) >= 0) ?                                       \
           (((word)[0] << (bits)) |                            \
           ((word)[1] >> (32 - (bits)))) :                     \
```

```
         0,                                                     \
    (ret)[1] = (((bits) < 32) && ((bits) >= 0)) ?               \
        ((word)[1] << (bits)) : 0 )

/*
 * Define 64-bit OR
 */
#define SHA512_OR(word1, word2, ret) (                          \
    (ret)[0] = (word1)[0] | (word2)[0],                         \
    (ret)[1] = (word1)[1] | (word2)[1] )

/*
 * Define 64-bit XOR
 */
#define SHA512_XOR(word1, word2, ret) (                         \
    (ret)[0] = (word1)[0] ^ (word2)[0],                         \
    (ret)[1] = (word1)[1] ^ (word2)[1] )

/*
 * Define 64-bit AND
 */
#define SHA512_AND(word1, word2, ret) (                         \
    (ret)[0] = (word1)[0] & (word2)[0],                         \
    (ret)[1] = (word1)[1] & (word2)[1] )

/*
 * Define 64-bit TILDA
 */
#define SHA512_TILDA(word, ret)                                 \
  ( (ret)[0] = ~(word)[0], (ret)[1] = ~(word)[1] )

/*
 * Define 64-bit ADD
 */
#define SHA512_ADD(word1, word2, ret) (                         \
    (ret)[1] = (word1)[1], (ret)[1] += (word2)[1],              \
    (ret)[0] = (word1)[0] + (word2)[0] + ((ret)[1] < (word1)[1]) )

/*
 * Add the 4word value in word2 to word1.
 */
static uint32_t ADDTO4_temp, ADDTO4_temp2;
#define SHA512_ADDTO4(word1, word2) (                           \
    ADDTO4_temp = (word1)[3],                                   \
    (word1)[3] += (word2)[3],                                   \
    ADDTO4_temp2 = (word1)[2],                                  \
    (word1)[2] += (word2)[2] + ((word1)[3] < ADDTO4_temp),      \
    ADDTO4_temp = (word1)[1],                                   \
```

```
    (word1)[1] += (word2)[1] + ((word1)[2] < ADDTO4_temp2),    \
    (word1)[0] += (word2)[0] + ((word1)[1] < ADDTO4_temp) )

/*
 * Add the 2word value in word2 to word1.
 */
static uint32_t ADDTO2_temp;
#define SHA512_ADDTO2(word1, word2) (                          \
    ADDTO2_temp = (word1)[1],                                  \
    (word1)[1] += (word2)[1],                                  \
    (word1)[0] += (word2)[0] + ((word1)[1] < ADDTO2_temp) )

/*
 * SHA rotate   ((word >> bits) | (word << (64-bits)))
 */
static uint32_t ROTR_temp1[2], ROTR_temp2[2];
#define SHA512_ROTR(bits, word, ret) (                         \
    SHA512_SHR((bits), (word), ROTR_temp1),                    \
    SHA512_SHL(64-(bits), (word), ROTR_temp2),                 \
    SHA512_OR(ROTR_temp1, ROTR_temp2, (ret)) )

/*
 * Define the SHA SIGMA and sigma macros
 *  SHA512_ROTR(28,word) ^ SHA512_ROTR(34,word) ^ SHA512_ROTR(39,word)
 */
static uint32_t SIGMA0_temp1[2], SIGMA0_temp2[2],
  SIGMA0_temp3[2], SIGMA0_temp4[2];
#define SHA512_SIGMA0(word, ret) (                             \
    SHA512_ROTR(28, (word), SIGMA0_temp1),                     \
    SHA512_ROTR(34, (word), SIGMA0_temp2),                     \
    SHA512_ROTR(39, (word), SIGMA0_temp3),                     \
    SHA512_XOR(SIGMA0_temp2, SIGMA0_temp3, SIGMA0_temp4),      \
    SHA512_XOR(SIGMA0_temp1, SIGMA0_temp4, (ret)) )

/*
 * SHA512_ROTR(14,word) ^ SHA512_ROTR(18,word) ^ SHA512_ROTR(41,word)
 */
static uint32_t SIGMA1_temp1[2], SIGMA1_temp2[2],
  SIGMA1_temp3[2], SIGMA1_temp4[2];
#define SHA512_SIGMA1(word, ret) (                             \
    SHA512_ROTR(14, (word), SIGMA1_temp1),                     \
    SHA512_ROTR(18, (word), SIGMA1_temp2),                     \
    SHA512_ROTR(41, (word), SIGMA1_temp3),                     \
    SHA512_XOR(SIGMA1_temp2, SIGMA1_temp3, SIGMA1_temp4),      \
    SHA512_XOR(SIGMA1_temp1, SIGMA1_temp4, (ret)) )

/*
 * (SHA512_ROTR( 1,word) ^ SHA512_ROTR( 8,word) ^ SHA512_SHR( 7,word))
```

```
 */
static uint32_t sigma0_temp1[2], sigma0_temp2[2],
  sigma0_temp3[2], sigma0_temp4[2];
#define SHA512_sigma0(word, ret) (                              \
    SHA512_ROTR( 1, (word), sigma0_temp1),                      \
    SHA512_ROTR( 8, (word), sigma0_temp2),                      \
    SHA512_SHR( 7, (word), sigma0_temp3),                       \
    SHA512_XOR(sigma0_temp2, sigma0_temp3, sigma0_temp4),       \
    SHA512_XOR(sigma0_temp1, sigma0_temp4, (ret)) )

/*
 * (SHA512_ROTR(19,word) ^ SHA512_ROTR(61,word) ^ SHA512_SHR( 6,word))
 */
static uint32_t sigma1_temp1[2], sigma1_temp2[2],
  sigma1_temp3[2], sigma1_temp4[2];
#define SHA512_sigma1(word, ret) (                              \
    SHA512_ROTR(19, (word), sigma1_temp1),                      \
    SHA512_ROTR(61, (word), sigma1_temp2),                      \
    SHA512_SHR( 6, (word), sigma1_temp3),                       \
    SHA512_XOR(sigma1_temp2, sigma1_temp3, sigma1_temp4),       \
    SHA512_XOR(sigma1_temp1, sigma1_temp4, (ret)) )

#undef SHA_Ch
#undef SHA_Maj

#ifndef USE_MODIFIED_MACROS
/*
 * These definitions are the ones used in FIPS-180-2, section 4.1.3
 *  Ch(x,y,z)   ((x & y) ^ (~x & z))
 */
static uint32_t Ch_temp1[2], Ch_temp2[2], Ch_temp3[2];
#define SHA_Ch(x, y, z, ret) (                                  \
    SHA512_AND(x, y, Ch_temp1),                                 \
    SHA512_TILDA(x, Ch_temp2),                                  \
    SHA512_AND(Ch_temp2, z, Ch_temp3),                          \
    SHA512_XOR(Ch_temp1, Ch_temp3, (ret)) )
/*
 *  Maj(x,y,z)  (((x)&(y)) ^ ((x)&(z)) ^ ((y)&(z)))
 */
static uint32_t Maj_temp1[2], Maj_temp2[2],
  Maj_temp3[2], Maj_temp4[2];
#define SHA_Maj(x, y, z, ret) (                                 \
    SHA512_AND(x, y, Maj_temp1),                                \
    SHA512_AND(x, z, Maj_temp2),                                \
    SHA512_AND(y, z, Maj_temp3),                                \
    SHA512_XOR(Maj_temp2, Maj_temp3, Maj_temp4),                \
    SHA512_XOR(Maj_temp1, Maj_temp4, (ret)) )
```

```
#else /* !USE_32BIT_ONLY */
/*
 * These definitions are potentially faster equivalents for the ones
 * used in FIPS-180-2, section 4.1.3.
 *   ((x & y) ^ (~x & z)) becomes
 *   ((x & (y ^ z)) ^ z)
 */
#define SHA_Ch(x, y, z, ret) (                                  \
   (ret)[0] = (((x)[0] & ((y)[0] ^ (z)[0])) ^ (z)[0]),          \
   (ret)[1] = (((x)[1] & ((y)[1] ^ (z)[1])) ^ (z)[1]) )

/*
 *   ((x & y) ^ (x & z) ^ (y & z)) becomes
 *   ((x & (y | z)) | (y & z))
 */
#define SHA_Maj(x, y, z, ret) (                                  \
   ret[0] = (((x)[0] & ((y)[0] | (z)[0])) | ((y)[0] & (z)[0])), \
   ret[1] = (((x)[1] & ((y)[1] | (z)[1])) | ((y)[1] & (z)[1])) )
#endif /* USE_MODIFIED_MACROS */

/*
 * add "length" to the length
 */
static uint32_t addTemp[4] = { 0, 0, 0, 0 };
#define SHA384_512AddLength(context, length) (                        \
    addTemp[3] = (length), SHA512_ADDTO4((context)->Length, addTemp), \
    (context)->Corrupted = (((context)->Length[3] == 0) &&            \
       ((context)->Length[2] == 0) && ((context)->Length[1] == 0) &&  \
       ((context)->Length[0] < 8)) ? 1 : 0 )

/* Local Function Prototypes */
static void SHA384_512Finalize(SHA512Context *context,
  uint8_t Pad_Byte);
static void SHA384_512PadMessage(SHA512Context *context,
  uint8_t Pad_Byte);
static void SHA384_512ProcessMessageBlock(SHA512Context *context);
static int SHA384_512Reset(SHA512Context *context, uint32_t H0[]);
static int SHA384_512ResultN( SHA512Context *context,
  uint8_t Message_Digest[], int HashSize);

/* Initial Hash Values: FIPS-180-2 sections 5.3.3 and 5.3.4 */
static uint32_t SHA384_H0[SHA512HashSize/4] = {
    0xCBBB9D5D, 0xC1059ED8, 0x629A292A, 0x367CD507, 0x9159015A,
    0x3070DD17, 0x152FECD8, 0xF70E5939, 0x67332667, 0xFFC00B31,
    0x8EB44A87, 0x68581511, 0xDB0C2E0D, 0x64F98FA7, 0x47B5481D,
    0xBEFA4FA4
};
```

```
static uint32_t SHA512_H0[SHA512HashSize/4] = {
    0x6A09E667, 0xF3BCC908, 0xBB67AE85, 0x84CAA73B, 0x3C6EF372,
    0xFE94F82B, 0xA54FF53A, 0x5F1D36F1, 0x510E527F, 0xADE682D1,
    0x9B05688C, 0x2B3E6C1F, 0x1F83D9AB, 0xFB41BD6B, 0x5BE0CD19,
    0x137E2179
};

#else /* !USE_32BIT_ONLY */

/* Define the SHA shift, rotate left and rotate right macro */
#define SHA512_SHR(bits,word)  (((uint64_t)(word)) >> (bits))
#define SHA512_ROTR(bits,word) ((((uint64_t)(word)) >> (bits)) | \
                               (((uint64_t)(word)) << (64-(bits))))

/* Define the SHA SIGMA and sigma macros */
#define SHA512_SIGMA0(word)   \
 (SHA512_ROTR(28,word) ^ SHA512_ROTR(34,word) ^ SHA512_ROTR(39,word))
#define SHA512_SIGMA1(word)   \
 (SHA512_ROTR(14,word) ^ SHA512_ROTR(18,word) ^ SHA512_ROTR(41,word))
#define SHA512_sigma0(word)   \
 (SHA512_ROTR( 1,word) ^ SHA512_ROTR( 8,word) ^ SHA512_SHR( 7,word))
#define SHA512_sigma1(word)   \
 (SHA512_ROTR(19,word) ^ SHA512_ROTR(61,word) ^ SHA512_SHR( 6,word))

/*
 * add "length" to the length
 */
static uint64_t addTemp;
#define SHA384_512AddLength(context, length)                    \
   (addTemp = context->Length_Low, context->Corrupted =         \
    ((context->Length_Low += length) < addTemp) &&              \
    (++context->Length_High == 0) ? 1 : 0)

/* Local Function Prototypes */
static void SHA384_512Finalize(SHA512Context *context,
  uint8_t Pad_Byte);
static void SHA384_512PadMessage(SHA512Context *context,
  uint8_t Pad_Byte);
static void SHA384_512ProcessMessageBlock(SHA512Context *context);
static int SHA384_512Reset(SHA512Context *context, uint64_t H0[]);
static int SHA384_512ResultN(SHA512Context *context,
  uint8_t Message_Digest[], int HashSize);

/* Initial Hash Values: FIPS-180-2 sections 5.3.3 and 5.3.4 */
static uint64_t SHA384_H0[] = {
    0xCBBB9D5DC1059ED8ll, 0x629A292A367CD507ll, 0x9159015A3070DD17ll,
    0x152FECD8F70E5939ll, 0x67332667FFC00B31ll, 0x8EB44A8768581511ll,
    0xDB0C2E0D64F98FA7ll, 0x47B5481DBEFA4FA4ll
```

```
};
static uint64_t SHA512_H0[] = {
    0x6A09E667F3BCC908ll, 0xBB67AE8584CAA73Bll, 0x3C6EF372FE94F82Bll,
    0xA54FF53A5F1D36F1ll, 0x510E527FADE682D1ll, 0x9B05688C2B3E6C1Fll,
    0x1F83D9ABFB41BD6Bll, 0x5BE0CD19137E2179ll
};

#endif /* USE_32BIT_ONLY */

/*
 * SHA384Reset
 *
 * Description:
 *   This function will initialize the SHA384Context in preparation
 *   for computing a new SHA384 message digest.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to reset.
 *
 * Returns:
 *   sha Error Code.
 *
 */
int SHA384Reset(SHA384Context *context)
{
  return SHA384_512Reset(context, SHA384_H0);
}

/*
 * SHA384Input
 *
 * Description:
 *   This function accepts an array of octets as the next portion
 *   of the message.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   message_array: [in]
 *     An array of characters representing the next portion of
 *     the message.
 *   length: [in]
 *     The length of the message in message_array
 *
 * Returns:
 *   sha Error Code.
 *
```

```
 */
int SHA384Input(SHA384Context *context,
    const uint8_t *message_array, unsigned int length)
{
  return SHA512Input(context, message_array, length);
}

/*
 * SHA384FinalBits
 *
 * Description:
 *   This function will add in any final bits of the message.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   message_bits: [in]
 *     The final bits of the message, in the upper portion of the
 *     byte. (Use 0b###00000 instead of 0b00000### to input the
 *     three bits ###.)
 *   length: [in]
 *     The number of bits in message_bits, between 1 and 7.
 *
 * Returns:
 *   sha Error Code.
 *
 */
int SHA384FinalBits(SHA384Context *context,
    const uint8_t message_bits, unsigned int length)
{
  return SHA512FinalBits(context, message_bits, length);
}

/*
 * SHA384Result
 *
 * Description:
 *   This function will return the 384-bit message
 *   digest into the Message_Digest array provided by the caller.
 *   NOTE: The first octet of hash is stored in the 0th element,
 *      the last octet of hash in the 48th element.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to use to calculate the SHA hash.
 *   Message_Digest: [out]
 *     Where the digest is returned.
 *
```

```
 * Returns:
 *   sha Error Code.
 *
 */
int SHA384Result(SHA384Context *context,
    uint8_t Message_Digest[SHA384HashSize])
{
  return SHA384_512ResultN(context, Message_Digest, SHA384HashSize);
}

/*
 * SHA512Reset
 *
 * Description:
 *   This function will initialize the SHA512Context in preparation
 *   for computing a new SHA512 message digest.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to reset.
 *
 * Returns:
 *   sha Error Code.
 *
 */
int SHA512Reset(SHA512Context *context)
{
  return SHA384_512Reset(context, SHA512_H0);
}

/*
 * SHA512Input
 *
 * Description:
 *   This function accepts an array of octets as the next portion
 *   of the message.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   message_array: [in]
 *     An array of characters representing the next portion of
 *     the message.
 *   length: [in]
 *     The length of the message in message_array
 *
 * Returns:
 *   sha Error Code.
```

```
 *
 */
int SHA512Input(SHA512Context *context,
        const uint8_t *message_array,
        unsigned int length)
{
  if (!length)
    return shaSuccess;

  if (!context || !message_array)
    return shaNull;

  if (context->Computed) {
    context->Corrupted = shaStateError;
    return shaStateError;
  }

  if (context->Corrupted)
     return context->Corrupted;

  while (length-- && !context->Corrupted) {
    context->Message_Block[context->Message_Block_Index++] =
            (*message_array & 0xFF);

    if (!SHA384_512AddLength(context, 8) &&
      (context->Message_Block_Index == SHA512_Message_Block_Size))
      SHA384_512ProcessMessageBlock(context);

    message_array++;
  }

  return shaSuccess;
}

/*
 * SHA512FinalBits
 *
 * Description:
 *   This function will add in any final bits of the message.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   message_bits: [in]
 *     The final bits of the message, in the upper portion of the
 *     byte. (Use 0b###00000 instead of 0b00000### to input the
 *     three bits ###.)
 *   length: [in]
```

```
 *     The number of bits in message_bits, between 1 and 7.
 *
 * Returns:
 *   sha Error Code.
 *
 */
int SHA512FinalBits(SHA512Context *context,
    const uint8_t message_bits, unsigned int length)
{
  uint8_t masks[8] = {
      /* 0 0b00000000 */ 0x00, /* 1 0b10000000 */ 0x80,
      /* 2 0b11000000 */ 0xC0, /* 3 0b11100000 */ 0xE0,
      /* 4 0b11110000 */ 0xF0, /* 5 0b11111000 */ 0xF8,
      /* 6 0b11111100 */ 0xFC, /* 7 0b11111110 */ 0xFE
  };
  uint8_t markbit[8] = {
      /* 0 0b10000000 */ 0x80, /* 1 0b01000000 */ 0x40,
      /* 2 0b00100000 */ 0x20, /* 3 0b00010000 */ 0x10,
      /* 4 0b00001000 */ 0x08, /* 5 0b00000100 */ 0x04,
      /* 6 0b00000010 */ 0x02, /* 7 0b00000001 */ 0x01
  };

  if (!length)
    return shaSuccess;

  if (!context)
    return shaNull;

  if ((context->Computed) || (length >= 8) || (length == 0)) {
    context->Corrupted = shaStateError;
    return shaStateError;
  }

  if (context->Corrupted)
     return context->Corrupted;

  SHA384_512AddLength(context, length);
  SHA384_512Finalize(context, (uint8_t)
    ((message_bits & masks[length]) | markbit[length]));

  return shaSuccess;
}

/*
 * SHA384_512Finalize
 *
 * Description:
 *   This helper function finishes off the digest calculations.
```

```
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   Pad_Byte: [in]
 *     The last byte to add to the digest before the 0-padding
 *     and length. This will contain the last bits of the message
 *     followed by another single bit. If the message was an
 *     exact multiple of 8-bits long, Pad_Byte will be 0x80.
 *
 * Returns:
 *   sha Error Code.
 *
 */
static void SHA384_512Finalize(SHA512Context *context,
    uint8_t Pad_Byte)
{
  int_least16_t i;
  SHA384_512PadMessage(context, Pad_Byte);
  /* message may be sensitive, clear it out */
  for (i = 0; i < SHA512_Message_Block_Size; ++i)
    context->Message_Block[i] = 0;
#ifdef USE_32BIT_ONLY    /* and clear length */
  context->Length[0] = context->Length[1] = 0;
  context->Length[2] = context->Length[3] = 0;
#else /* !USE_32BIT_ONLY */
  context->Length_Low = 0;
  context->Length_High = 0;
#endif /* USE_32BIT_ONLY */
  context->Computed = 1;
}

/*
 * SHA512Result
 *
 * Description:
 *   This function will return the 512-bit message
 *   digest into the Message_Digest array provided by the caller.
 *   NOTE: The first octet of hash is stored in the 0th element,
 *      the last octet of hash in the 64th element.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to use to calculate the SHA hash.
 *   Message_Digest: [out]
 *     Where the digest is returned.
 *
 * Returns:
```

```
 *   sha Error Code.
 *
 */
int SHA512Result(SHA512Context *context,
    uint8_t Message_Digest[SHA512HashSize])
{
  return SHA384_512ResultN(context, Message_Digest, SHA512HashSize);
}

/*
 * SHA384_512PadMessage
 *
 * Description:
 *   According to the standard, the message must be padded to an
 *   even 1024 bits. The first padding bit must be a '1'. The
 *   last 128 bits represent the length of the original message.
 *   All bits in between should be 0. This helper function will
 *   pad the message according to those rules by filling the
 *   Message_Block array accordingly. When it returns, it can be
 *   assumed that the message digest has been computed.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to pad
 *   Pad_Byte: [in]
 *     The last byte to add to the digest before the 0-padding
 *     and length. This will contain the last bits of the message
 *     followed by another single bit. If the message was an
 *     exact multiple of 8-bits long, Pad_Byte will be 0x80.
 *
 * Returns:
 *   Nothing.
 *
 */
static void SHA384_512PadMessage(SHA512Context *context,
    uint8_t Pad_Byte)
{
  /*
   * Check to see if the current message block is too small to hold
   * the initial padding bits and length. If so, we will pad the
   * block, process it, and then continue padding into a second
   * block.
   */
  if (context->Message_Block_Index >= (SHA512_Message_Block_Size-16)) {
    context->Message_Block[context->Message_Block_Index++] = Pad_Byte;
    while (context->Message_Block_Index < SHA512_Message_Block_Size)
      context->Message_Block[context->Message_Block_Index++] = 0;
```

```
    SHA384_512ProcessMessageBlock(context);
  } else
    context->Message_Block[context->Message_Block_Index++] = Pad_Byte;

  while (context->Message_Block_Index < (SHA512_Message_Block_Size-16))
    context->Message_Block[context->Message_Block_Index++] = 0;

  /*
   * Store the message length as the last 16 octets
   */
#ifdef USE_32BIT_ONLY
  context->Message_Block[112] = (uint8_t)(context->Length[0] >> 24);
  context->Message_Block[113] = (uint8_t)(context->Length[0] >> 16);
  context->Message_Block[114] = (uint8_t)(context->Length[0] >> 8);
  context->Message_Block[115] = (uint8_t)(context->Length[0]);
  context->Message_Block[116] = (uint8_t)(context->Length[1] >> 24);
  context->Message_Block[117] = (uint8_t)(context->Length[1] >> 16);
  context->Message_Block[118] = (uint8_t)(context->Length[1] >> 8);
  context->Message_Block[119] = (uint8_t)(context->Length[1]);

  context->Message_Block[120] = (uint8_t)(context->Length[2] >> 24);
  context->Message_Block[121] = (uint8_t)(context->Length[2] >> 16);
  context->Message_Block[122] = (uint8_t)(context->Length[2] >> 8);
  context->Message_Block[123] = (uint8_t)(context->Length[2]);
  context->Message_Block[124] = (uint8_t)(context->Length[3] >> 24);
  context->Message_Block[125] = (uint8_t)(context->Length[3] >> 16);
  context->Message_Block[126] = (uint8_t)(context->Length[3] >> 8);
  context->Message_Block[127] = (uint8_t)(context->Length[3]);
#else /* !USE_32BIT_ONLY */
  context->Message_Block[112] = (uint8_t)(context->Length_High >> 56);
  context->Message_Block[113] = (uint8_t)(context->Length_High >> 48);
  context->Message_Block[114] = (uint8_t)(context->Length_High >> 40);
  context->Message_Block[115] = (uint8_t)(context->Length_High >> 32);
  context->Message_Block[116] = (uint8_t)(context->Length_High >> 24);
  context->Message_Block[117] = (uint8_t)(context->Length_High >> 16);
  context->Message_Block[118] = (uint8_t)(context->Length_High >> 8);
  context->Message_Block[119] = (uint8_t)(context->Length_High);

  context->Message_Block[120] = (uint8_t)(context->Length_Low >> 56);
  context->Message_Block[121] = (uint8_t)(context->Length_Low >> 48);
  context->Message_Block[122] = (uint8_t)(context->Length_Low >> 40);
  context->Message_Block[123] = (uint8_t)(context->Length_Low >> 32);
  context->Message_Block[124] = (uint8_t)(context->Length_Low >> 24);
  context->Message_Block[125] = (uint8_t)(context->Length_Low >> 16);
  context->Message_Block[126] = (uint8_t)(context->Length_Low >> 8);
  context->Message_Block[127] = (uint8_t)(context->Length_Low);
#endif /* USE_32BIT_ONLY */
```

```
  SHA384_512ProcessMessageBlock(context);
}

/*
 * SHA384_512ProcessMessageBlock
 *
 * Description:
 *   This helper function will process the next 1024 bits of the
 *   message stored in the Message_Block array.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *
 * Returns:
 *   Nothing.
 *
 * Comments:
 *   Many of the variable names in this code, especially the
 *   single character names, were used because those were the
 *   names used in the publication.
 *
 *
 */
static void SHA384_512ProcessMessageBlock(SHA512Context *context)
{
  /* Constants defined in FIPS-180-2, section 4.2.3 */
#ifdef USE_32BIT_ONLY
  static const uint32_t K[80*2] = {
      0x428A2F98, 0xD728AE22, 0x71374491, 0x23EF65CD, 0xB5C0FBCF,
      0xEC4D3B2F, 0xE9B5DBA5, 0x8189DBBC, 0x3956C25B, 0xF348B538,
      0x59F111F1, 0xB605D019, 0x923F82A4, 0xAF194F9B, 0xAB1C5ED5,
      0xDA6D8118, 0xD807AA98, 0xA3030242, 0x12835B01, 0x45706FBE,
      0x243185BE, 0x4EE4B28C, 0x550C7DC3, 0xD5FFB4E2, 0x72BE5D74,
      0xF27B896F, 0x80DEB1FE, 0x3B1696B1, 0x9BDC06A7, 0x25C71235,
      0xC19BF174, 0xCF692694, 0xE49B69C1, 0x9EF14AD2, 0xEFBE4786,
      0x384F25E3, 0x0FC19DC6, 0x8B8CD5B5, 0x240CA1CC, 0x77AC9C65,
      0x2DE92C6F, 0x592B0275, 0x4A7484AA, 0x6EA6E483, 0x5CB0A9DC,
      0xBD41FBD4, 0x76F988DA, 0x831153B5, 0x983E5152, 0xEE66DFAB,
      0xA831C66D, 0x2DB43210, 0xB00327C8, 0x98FB213F, 0xBF597FC7,
      0xBEEF0EE4, 0xC6E00BF3, 0x3DA88FC2, 0xD5A79147, 0x930AA725,
      0x06CA6351, 0xE003826F, 0x14292967, 0x0A0E6E70, 0x27B70A85,
      0x46D22FFC, 0x2E1B2138, 0x5C26C926, 0x4D2C6DFC, 0x5AC42AED,
      0x53380D13, 0x9D95B3DF, 0x650A7354, 0x8BAF63DE, 0x766A0ABB,
      0x3C77B2A8, 0x81C2C92E, 0x47EDAEE6, 0x92722C85, 0x1482353B,
      0xA2BFE8A1, 0x4CF10364, 0xA81A664B, 0xBC423001, 0xC24B8B70,
      0xD0F89791, 0xC76C51A3, 0x0654BE30, 0xD192E819, 0xD6EF5218,
      0xD6990624, 0x5565A910, 0xF40E3585, 0x5771202A, 0x106AA070,
```

```
    0x32BBD1B8, 0x19A4C116, 0xB8D2D0C8, 0x1E376C08, 0x5141AB53,
    0x2748774C, 0xDF8EEB99, 0x34B0BCB5, 0xE19B48A8, 0x391C0CB3,
    0xC5C95A63, 0x4ED8AA4A, 0xE3418ACB, 0x5B9CCA4F, 0x7763E373,
    0x682E6FF3, 0xD6B2B8A3, 0x748F82EE, 0x5DEFB2FC, 0x78A5636F,
    0x43172F60, 0x84C87814, 0xA1F0AB72, 0x8CC70208, 0x1A6439EC,
    0x90BEFFFA, 0x23631E28, 0xA4506CEB, 0xDE82BDE9, 0xBEF9A3F7,
    0xB2C67915, 0xC67178F2, 0xE372532B, 0xCA273ECE, 0xEA26619C,
    0xD186B8C7, 0x21C0C207, 0xEADA7DD6, 0xCDE0EB1E, 0xF57D4F7F,
    0xEE6ED178, 0x06F067AA, 0x72176FBA, 0x0A637DC5, 0xA2C898A6,
    0x113F9804, 0xBEF90DAE, 0x1B710B35, 0x131C471B, 0x28DB77F5,
    0x23047D84, 0x32CAAB7B, 0x40C72493, 0x3C9EBE0A, 0x15C9BEBC,
    0x431D67C4, 0x9C100D4C, 0x4CC5D4BE, 0xCB3E42B6, 0x597F299C,
    0xFC657E2A, 0x5FCB6FAB, 0x3AD6FAEC, 0x6C44198C, 0x4A475817
};
int     t, t2, t8;                  /* Loop counter */
uint32_t  temp1[2], temp2[2],       /* Temporary word values */
      temp3[2], temp4[2], temp5[2];
uint32_t  W[2*80];                  /* Word sequence */
uint32_t  A[2], B[2], C[2], D[2],   /* Word buffers */
      E[2], F[2], G[2], H[2];

/* Initialize the first 16 words in the array W */
for (t = t2 = t8 = 0; t < 16; t++, t8 += 8) {
  W[t2++] = ((((uint32_t)context->Message_Block[t8    ])) << 24) |
            ((((uint32_t)context->Message_Block[t8 + 1])) << 16) |
            ((((uint32_t)context->Message_Block[t8 + 2])) << 8) |
            ((((uint32_t)context->Message_Block[t8 + 3])));
  W[t2++] = ((((uint32_t)context->Message_Block[t8 + 4])) << 24) |
            ((((uint32_t)context->Message_Block[t8 + 5])) << 16) |
            ((((uint32_t)context->Message_Block[t8 + 6])) << 8) |
            ((((uint32_t)context->Message_Block[t8 + 7])));
}

for (t = 16; t < 80; t++, t2 += 2) {
  /* W[t] = SHA512_sigma1(W[t-2]) + W[t-7] +
    SHA512_sigma0(W[t-15]) + W[t-16]; */
  uint32_t *Wt2 = &W[t2-2*2];
  uint32_t *Wt7 = &W[t2-7*2];
  uint32_t *Wt15 = &W[t2-15*2];
  uint32_t *Wt16 = &W[t2-16*2];
  SHA512_sigma1(Wt2, temp1);
  SHA512_ADD(temp1, Wt7, temp2);
  SHA512_sigma0(Wt15, temp1);
  SHA512_ADD(temp1, Wt16, temp3);
  SHA512_ADD(temp2, temp3, &W[t2]);
}

A[0] = context->Intermediate_Hash[0];
```

```
  A[1] = context->Intermediate_Hash[1];
  B[0] = context->Intermediate_Hash[2];
  B[1] = context->Intermediate_Hash[3];
  C[0] = context->Intermediate_Hash[4];
  C[1] = context->Intermediate_Hash[5];
  D[0] = context->Intermediate_Hash[6];
  D[1] = context->Intermediate_Hash[7];
  E[0] = context->Intermediate_Hash[8];
  E[1] = context->Intermediate_Hash[9];
  F[0] = context->Intermediate_Hash[10];
  F[1] = context->Intermediate_Hash[11];
  G[0] = context->Intermediate_Hash[12];
  G[1] = context->Intermediate_Hash[13];
  H[0] = context->Intermediate_Hash[14];
  H[1] = context->Intermediate_Hash[15];

  for (t = t2 = 0; t < 80; t++, t2 += 2) {
    /*
     * temp1 = H + SHA512_SIGMA1(E) + SHA_Ch(E,F,G) + K[t] + W[t];
     */
    SHA512_SIGMA1(E,temp1);
    SHA512_ADD(H, temp1, temp2);
    SHA_Ch(E,F,G,temp3);
    SHA512_ADD(temp2, temp3, temp4);
    SHA512_ADD(&K[t2], &W[t2], temp5);
    SHA512_ADD(temp4, temp5, temp1);
    /*
     * temp2 = SHA512_SIGMA0(A) + SHA_Maj(A,B,C);
     */
    SHA512_SIGMA0(A,temp3);
    SHA_Maj(A,B,C,temp4);
    SHA512_ADD(temp3, temp4, temp2);
    H[0] = G[0]; H[1] = G[1];
    G[0] = F[0]; G[1] = F[1];
    F[0] = E[0]; F[1] = E[1];
    SHA512_ADD(D, temp1, E);
    D[0] = C[0]; D[1] = C[1];
    C[0] = B[0]; C[1] = B[1];
    B[0] = A[0]; B[1] = A[1];
    SHA512_ADD(temp1, temp2, A);
  }

  SHA512_ADDTO2(&context->Intermediate_Hash[0], A);
  SHA512_ADDTO2(&context->Intermediate_Hash[2], B);
  SHA512_ADDTO2(&context->Intermediate_Hash[4], C);
  SHA512_ADDTO2(&context->Intermediate_Hash[6], D);
  SHA512_ADDTO2(&context->Intermediate_Hash[8], E);
  SHA512_ADDTO2(&context->Intermediate_Hash[10], F);
```

```
  SHA512_ADDTO2(&context->Intermediate_Hash[12], G);
  SHA512_ADDTO2(&context->Intermediate_Hash[14], H);

#else /* !USE_32BIT_ONLY */
  static const uint64_t K[80] = {
      0x428A2F98D728AE22ll, 0x7137449123EF65CDll, 0xB5C0FBCFEC4D3B2Fll,
      0xE9B5DBA58189DBBCll, 0x3956C25BF348B538ll, 0x59F111F1B605D019ll,
      0x923F82A4AF194F9Bll, 0xAB1C5ED5DA6D8118ll, 0xD807AA98A3030242ll,
      0x12835B0145706FBEll, 0x243185BE4EE4B28Cll, 0x550C7DC3D5FFB4E2ll,
      0x72BE5D74F27B896Fll, 0x80DEB1FE3B1696B1ll, 0x9BDC06A725C71235ll,
      0xC19BF174CF692694ll, 0xE49B69C19EF14AD2ll, 0xEFBE4786384F25E3ll,
      0x0FC19DC68B8CD5B5ll, 0x240CA1CC77AC9C65ll, 0x2DE92C6F592B0275ll,
      0x4A7484AA6EA6E483ll, 0x5CB0A9DCBD41FBD4ll, 0x76F988DA831153B5ll,
      0x983E5152EE66DFABll, 0xA831C66D2DB43210ll, 0xB00327C898FB213Fll,
      0xBF597FC7BEEF0EE4ll, 0xC6E00BF33DA88FC2ll, 0xD5A79147930AA725ll,
      0x06CA6351E003826Fll, 0x142929670A0E6E70ll, 0x27B70A8546D22FFCll,
      0x2E1B21385C26C926ll, 0x4D2C6DFC5AC42AEDll, 0x53380D139D95B3DFll,
      0x650A73548BAF63DEll, 0x766A0ABB3C77B2A8ll, 0x81C2C92E47EDAEE6ll,
      0x92722C851482353Bll, 0xA2BFE8A14CF10364ll, 0xA81A664BBC423001ll,
      0xC24B8B70D0F89791ll, 0xC76C51A30654BE30ll, 0xD192E819D6EF5218ll,
      0xD69906245565A910ll, 0xF40E35855771202All, 0x106AA07032BBD1B8ll,
      0x19A4C116B8D2D0C8ll, 0x1E376C085141AB53ll, 0x2748774CDF8EEB99ll,
      0x34B0BCB5E19B48A8ll, 0x391C0CB3C5C95A63ll, 0x4ED8AA4AE3418ACBll,
      0x5B9CCA4F7763E373ll, 0x682E6FF3D6B2B8A3ll, 0x748F82EE5DEFB2FCll,
      0x78A5636F43172F60ll, 0x84C87814A1F0AB72ll, 0x8CC702081A6439ECll,
      0x90BEFFFA23631E28ll, 0xA4506CEBDE82BDE9ll, 0xBEF9A3F7B2C67915ll,
      0xC67178F2E372532Bll, 0xCA273ECEEA26619Cll, 0xD186B8C721C0C207ll,
      0xEADA7DD6CDE0EB1Ell, 0xF57D4F7FEE6ED178ll, 0x06F067AA72176FBAll,
      0x0A637DC5A2C898A6ll, 0x113F9804BEF90DAEll, 0x1B710B35131C471Bll,
      0x28DB77F523047D84ll, 0x32CAAB7B40C72493ll, 0x3C9EBE0A15C9BEBCll,
      0x431D67C49C100D4Cll, 0x4CC5D4BECB3E42B6ll, 0x597F299CFC657E2All,
      0x5FCB6FAB3AD6FAECll, 0x6C44198C4A475817ll
  };
  int        t, t8;                  /* Loop counter */
  uint64_t   temp1, temp2;           /* Temporary word value */
  uint64_t   W[80];                  /* Word sequence */
  uint64_t   A, B, C, D, E, F, G, H;  /* Word buffers */

  /*
   * Initialize the first 16 words in the array W
   */
  for (t = t8 = 0; t < 16; t++, t8 += 8)
    W[t] = ((uint64_t)(context->Message_Block[t8  ]) << 56) |
           ((uint64_t)(context->Message_Block[t8 + 1]) << 48) |
           ((uint64_t)(context->Message_Block[t8 + 2]) << 40) |
           ((uint64_t)(context->Message_Block[t8 + 3]) << 32) |
           ((uint64_t)(context->Message_Block[t8 + 4]) << 24) |
           ((uint64_t)(context->Message_Block[t8 + 5]) << 16) |
```

```
            ((uint64_t)(context->Message_Block[t8 + 6]) << 8) |
            ((uint64_t)(context->Message_Block[t8 + 7]));

  for (t = 16; t < 80; t++)
    W[t] = SHA512_sigma1(W[t-2]) + W[t-7] +
        SHA512_sigma0(W[t-15]) + W[t-16];

  A = context->Intermediate_Hash[0];
  B = context->Intermediate_Hash[1];
  C = context->Intermediate_Hash[2];
  D = context->Intermediate_Hash[3];
  E = context->Intermediate_Hash[4];
  F = context->Intermediate_Hash[5];
  G = context->Intermediate_Hash[6];
  H = context->Intermediate_Hash[7];

  for (t = 0; t < 80; t++) {
    temp1 = H + SHA512_SIGMA1(E) + SHA_Ch(E,F,G) + K[t] + W[t];
    temp2 = SHA512_SIGMA0(A) + SHA_Maj(A,B,C);
    H = G;
    G = F;
    F = E;
    E = D + temp1;
    D = C;
    C = B;
    B = A;
    A = temp1 + temp2;
  }

  context->Intermediate_Hash[0] += A;
  context->Intermediate_Hash[1] += B;
  context->Intermediate_Hash[2] += C;
  context->Intermediate_Hash[3] += D;
  context->Intermediate_Hash[4] += E;
  context->Intermediate_Hash[5] += F;
  context->Intermediate_Hash[6] += G;
  context->Intermediate_Hash[7] += H;
#endif /* USE_32BIT_ONLY */

  context->Message_Block_Index = 0;
}

/*
 * SHA384_512Reset
 *
 * Description:
 *   This helper function will initialize the SHA512Context in
 *   preparation for computing a new SHA384 or SHA512 message
```

```
 *   digest.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to reset.
 *   H0
 *     The initial hash value to use.
 *
 * Returns:
 *   sha Error Code.
 *
 */
#ifdef USE_32BIT_ONLY
static int SHA384_512Reset(SHA512Context *context, uint32_t H0[])
#else /* !USE_32BIT_ONLY */
static int SHA384_512Reset(SHA512Context *context, uint64_t H0[])
#endif /* USE_32BIT_ONLY */
{
  int i;
  if (!context)
    return shaNull;

  context->Message_Block_Index = 0;

#ifdef USE_32BIT_ONLY
  context->Length[0] = context->Length[1] = 0;
  context->Length[2] = context->Length[3] = 0;

  for (i = 0; i < SHA512HashSize/4; i++)
    context->Intermediate_Hash[i] = H0[i];
#else /* !USE_32BIT_ONLY */
  context->Length_High = context->Length_Low = 0;

  for (i = 0; i < SHA512HashSize/8; i++)
    context->Intermediate_Hash[i] = H0[i];
#endif /* USE_32BIT_ONLY */

  context->Computed = 0;
  context->Corrupted = 0;

  return shaSuccess;
}

/*
 * SHA384_512ResultN
 *
 * Description:
 *   This helper function will return the 384-bit or 512-bit message
```

```
 *   digest into the Message_Digest array provided by the caller.
 *   NOTE: The first octet of hash is stored in the 0th element,
 *      the last octet of hash in the 48th/64th element.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to use to calculate the SHA hash.
 *   Message_Digest: [out]
 *     Where the digest is returned.
 *   HashSize: [in]
 *     The size of the hash, either 48 or 64.
 *
 * Returns:
 *   sha Error Code.
 *
 */
static int SHA384_512ResultN(SHA512Context *context,
    uint8_t Message_Digest[], int HashSize)
{
  int i;

#ifdef USE_32BIT_ONLY
  int i2;
#endif /* USE_32BIT_ONLY */

  if (!context || !Message_Digest)
    return shaNull;

  if (context->Corrupted)
    return context->Corrupted;

  if (!context->Computed)
    SHA384_512Finalize(context, 0x80);

#ifdef USE_32BIT_ONLY
  for (i = i2 = 0; i < HashSize; ) {
    Message_Digest[i++]=(uint8_t)(context->Intermediate_Hash[i2]>>24);
    Message_Digest[i++]=(uint8_t)(context->Intermediate_Hash[i2]>>16);
    Message_Digest[i++]=(uint8_t)(context->Intermediate_Hash[i2]>>8);
    Message_Digest[i++]=(uint8_t)(context->Intermediate_Hash[i2++]);
    Message_Digest[i++]=(uint8_t)(context->Intermediate_Hash[i2]>>24);
    Message_Digest[i++]=(uint8_t)(context->Intermediate_Hash[i2]>>16);
    Message_Digest[i++]=(uint8_t)(context->Intermediate_Hash[i2]>>8);
    Message_Digest[i++]=(uint8_t)(context->Intermediate_Hash[i2++]);
  }
#else /* !USE_32BIT_ONLY */
  for (i = 0; i < HashSize; ++i)
    Message_Digest[i] = (uint8_t)
```

```
      (context->Intermediate_Hash[i>>3] >> 8 * ( 7 - ( i % 8 ) ));
#endif /* USE_32BIT_ONLY */

  return shaSuccess;
}
```

8.2.4. usha.c

```
/*************************** usha.c ***************************/
/******************** See RFC 4634 for details *****************/
/*
 *  Description:
 *     This file implements a unified interface to the SHA algorithms.
 */

#include "sha.h"

/*
 *  USHAReset
 *
 *  Description:
 *      This function will initialize the SHA Context in preparation
 *      for computing a new SHA message digest.
 *
 *  Parameters:
 *      context: [in/out]
 *          The context to reset.
 *      whichSha: [in]
 *          Selects which SHA reset to call
 *
 *  Returns:
 *      sha Error Code.
 *
 */
int USHAReset(USHAContext *ctx, enum SHAversion whichSha)
{
  if (ctx) {
    ctx->whichSha = whichSha;
    switch (whichSha) {
      case SHA1:   return SHA1Reset((SHA1Context*)&ctx->ctx);
      case SHA224: return SHA224Reset((SHA224Context*)&ctx->ctx);
      case SHA256: return SHA256Reset((SHA256Context*)&ctx->ctx);
      case SHA384: return SHA384Reset((SHA384Context*)&ctx->ctx);
      case SHA512: return SHA512Reset((SHA512Context*)&ctx->ctx);
      default: return shaBadParam;
    }
  } else {
    return shaNull;
```

```
  }
}

/*
 *  USHAInput
 *
 *  Description:
 *      This function accepts an array of octets as the next portion
 *      of the message.
 *
 *  Parameters:
 *      context: [in/out]
 *          The SHA context to update
 *      message_array: [in]
 *          An array of characters representing the next portion of
 *          the message.
 *      length: [in]
 *          The length of the message in message_array
 *
 *  Returns:
 *      sha Error Code.
 *
 */
int USHAInput(USHAContext *ctx,
              const uint8_t *bytes, unsigned int bytecount)
{
  if (ctx) {
    switch (ctx->whichSha) {
      case SHA1:
        return SHA1Input((SHA1Context*)&ctx->ctx, bytes, bytecount);
      case SHA224:
        return SHA224Input((SHA224Context*)&ctx->ctx, bytes,
            bytecount);
      case SHA256:
        return SHA256Input((SHA256Context*)&ctx->ctx, bytes,
            bytecount);
      case SHA384:
        return SHA384Input((SHA384Context*)&ctx->ctx, bytes,
            bytecount);
      case SHA512:
        return SHA512Input((SHA512Context*)&ctx->ctx, bytes,
            bytecount);
      default: return shaBadParam;
    }
  } else {
    return shaNull;
  }
}
```

```
/*
 * USHAFinalBits
 *
 * Description:
 *   This function will add in any final bits of the message.
 *
 * Parameters:
 *   context: [in/out]
 *     The SHA context to update
 *   message_bits: [in]
 *     The final bits of the message, in the upper portion of the
 *     byte. (Use 0b###00000 instead of 0b00000### to input the
 *     three bits ###.)
 *   length: [in]
 *     The number of bits in message_bits, between 1 and 7.
 *
 * Returns:
 *   sha Error Code.
 */
int USHAFinalBits(USHAContext *ctx,
                  const uint8_t bits, unsigned int bitcount)
{
  if (ctx) {
    switch (ctx->whichSha) {
      case SHA1:
        return SHA1FinalBits((SHA1Context*)&ctx->ctx, bits, bitcount);
      case SHA224:
        return SHA224FinalBits((SHA224Context*)&ctx->ctx, bits,
            bitcount);
      case SHA256:
        return SHA256FinalBits((SHA256Context*)&ctx->ctx, bits,
            bitcount);
      case SHA384:
        return SHA384FinalBits((SHA384Context*)&ctx->ctx, bits,
            bitcount);
      case SHA512:
        return SHA512FinalBits((SHA512Context*)&ctx->ctx, bits,
            bitcount);
      default: return shaBadParam;
    }
  } else {
    return shaNull;
  }
}

/*
 * USHAResult
 *
```

```
 * Description:
 *   This function will return the 160-bit message digest into the
 *   Message_Digest array provided by the caller.
 *   NOTE: The first octet of hash is stored in the 0th element,
 *      the last octet of hash in the 19th element.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to use to calculate the SHA-1 hash.
 *   Message_Digest: [out]
 *     Where the digest is returned.
 *
 * Returns:
 *   sha Error Code.
 *
 */
int USHAResult(USHAContext *ctx,
               uint8_t Message_Digest[USHAMaxHashSize])
{
  if (ctx) {
    switch (ctx->whichSha) {
      case SHA1:
        return SHA1Result((SHA1Context*)&ctx->ctx, Message_Digest);
      case SHA224:
        return SHA224Result((SHA224Context*)&ctx->ctx, Message_Digest);
      case SHA256:
        return SHA256Result((SHA256Context*)&ctx->ctx, Message_Digest);
      case SHA384:
        return SHA384Result((SHA384Context*)&ctx->ctx, Message_Digest);
      case SHA512:
        return SHA512Result((SHA512Context*)&ctx->ctx, Message_Digest);
      default: return shaBadParam;
    }
  } else {
    return shaNull;
  }
}

/*
 * USHABlockSize
 *
 * Description:
 *   This function will return the blocksize for the given SHA
 *   algorithm.
 *
 * Parameters:
 *   whichSha:
 *     which SHA algorithm to query
```

```
 *
 * Returns:
 *   block size
 *
 */
int USHABlockSize(enum SHAversion whichSha)
{
  switch (whichSha) {
    case SHA1:   return SHA1_Message_Block_Size;
    case SHA224: return SHA224_Message_Block_Size;
    case SHA256: return SHA256_Message_Block_Size;
    case SHA384: return SHA384_Message_Block_Size;
    default:
    case SHA512: return SHA512_Message_Block_Size;
  }
}

/*
 * USHAHashSize
 *
 * Description:
 *   This function will return the hashsize for the given SHA
 *   algorithm.
 *
 * Parameters:
 *   whichSha:
 *     which SHA algorithm to query
 *
 * Returns:
 *   hash size
 *
 */
int USHAHashSize(enum SHAversion whichSha)
{
  switch (whichSha) {
    case SHA1:   return SHA1HashSize;
    case SHA224: return SHA224HashSize;
    case SHA256: return SHA256HashSize;
    case SHA384: return SHA384HashSize;
    default:
    case SHA512: return SHA512HashSize;
  }
}

/*
 * USHAHashSizeBits
 *
 * Description:
```

```
 *   This function will return the hashsize for the given SHA
 *   algorithm, expressed in bits.
 *
 * Parameters:
 *   whichSha:
 *     which SHA algorithm to query
 *
 * Returns:
 *   hash size in bits
 *
 */
int USHAHashSizeBits(enum SHAversion whichSha)
{
  switch (whichSha) {
    case SHA1:   return SHA1HashSizeBits;
    case SHA224: return SHA224HashSizeBits;
    case SHA256: return SHA256HashSizeBits;
    case SHA384: return SHA384HashSizeBits;
    default:
    case SHA512: return SHA512HashSizeBits;
  }
}
```

8.2.5. sha-private.h

```
/************************** sha-private.h **************************/
/*********************** See RFC 4634 for details **********************/
#ifndef _SHA_PRIVATE__H
#define _SHA_PRIVATE__H
/*
 * These definitions are defined in FIPS-180-2, section 4.1.
 * Ch() and Maj() are defined identically in sections 4.1.1,
 * 4.1.2 and 4.1.3.
 *
 * The definitions used in FIPS-180-2 are as follows:
 */

#ifndef USE_MODIFIED_MACROS
#define SHA_Ch(x,y,z)        (((x) & (y)) ^ ((~(x)) & (z)))
#define SHA_Maj(x,y,z)       (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))

#else /* USE_MODIFIED_MACROS */
/*
 * The following definitions are equivalent and potentially faster.
 */

#define SHA_Ch(x, y, z)      (((x) & ((y) ^ (z))) ^ (z))
#define SHA_Maj(x, y, z)     (((x) & ((y) | (z))) | ((y) & (z)))
```

```
#endif /* USE_MODIFIED_MACROS */

#define SHA_Parity(x, y, z)  ((x) ^ (y) ^ (z))

#endif /* _SHA_PRIVATE__H */
```

8.3 The HMAC Code

```
/*************************** hmac.c ***************************/
/******************** See RFC 4634 for details ******************/
/*
 *  Description:
 *      This file implements the HMAC algorithm (Keyed-Hashing for
 *      Message Authentication, RFC2104), expressed in terms of the
 *      various SHA algorithms.
 */

#include "sha.h"

/*
 *  hmac
 *
 *  Description:
 *      This function will compute an HMAC message digest.
 *
 *  Parameters:
 *      whichSha: [in]
 *          One of SHA1, SHA224, SHA256, SHA384, SHA512
 *      key: [in]
 *          The secret shared key.
 *      key_len: [in]
 *          The length of the secret shared key.
 *      message_array: [in]
 *          An array of characters representing the message.
 *      length: [in]
 *          The length of the message in message_array
 *      digest: [out]
 *          Where the digest is returned.
 *          NOTE: The length of the digest is determined by
 *              the value of whichSha.
 *
 *  Returns:
 *      sha Error Code.
 *
 */
int hmac(SHAversion whichSha, const unsigned char *text, int text_len,
    const unsigned char *key, int key_len,
    uint8_t digest[USHAMaxHashSize])
```

```
{
  HMACContext ctx;
  return hmacReset(&ctx, whichSha, key, key_len) ||
         hmacInput(&ctx, text, text_len) ||
         hmacResult(&ctx, digest);
}

/*
 *  hmacReset
 *
 *  Description:
 *      This function will initialize the hmacContext in preparation
 *      for computing a new HMAC message digest.
 *
 *  Parameters:
 *      context: [in/out]
 *          The context to reset.
 *      whichSha: [in]
 *          One of SHA1, SHA224, SHA256, SHA384, SHA512
 *      key: [in]
 *          The secret shared key.
 *      key_len: [in]
 *          The length of the secret shared key.
 *
 *  Returns:
 *      sha Error Code.
 *
 */
int hmacReset(HMACContext *ctx, enum SHAversion whichSha,
    const unsigned char *key, int key_len)
{
  int i, blocksize, hashsize;

  /* inner padding - key XORd with ipad */
  unsigned char k_ipad[USHA_Max_Message_Block_Size];

  /* temporary buffer when keylen > blocksize */
  unsigned char tempkey[USHAMaxHashSize];

  if (!ctx) return shaNull;

  blocksize = ctx->blockSize = USHABlockSize(whichSha);
  hashsize = ctx->hashSize = USHAHashSize(whichSha);

  ctx->whichSha = whichSha;

  /*
   * If key is longer than the hash blocksize,
```

```
   * reset it to key = HASH(key).
   */
  if (key_len > blocksize) {
    USHAContext tctx;
    int err = USHAReset(&tctx, whichSha) ||
              USHAInput(&tctx, key, key_len) ||
              USHAResult(&tctx, tempkey);
    if (err != shaSuccess) return err;

    key = tempkey;
    key_len = hashsize;
  }

  /*
   * The HMAC transform looks like:
   *
   * SHA(K XOR opad, SHA(K XOR ipad, text))
   *
   * where K is an n byte key.
   * ipad is the byte 0x36 repeated blocksize times
   * opad is the byte 0x5c repeated blocksize times
   * and text is the data being protected.
   */

  /* store key into the pads, XOR'd with ipad and opad values */
  for (i = 0; i < key_len; i++) {
    k_ipad[i] = key[i] ^ 0x36;
    ctx->k_opad[i] = key[i] ^ 0x5c;
  }
  /* remaining pad bytes are '\0' XOR'd with ipad and opad values */
  for ( ; i < blocksize; i++) {
    k_ipad[i] = 0x36;
    ctx->k_opad[i] = 0x5c;
  }

  /* perform inner hash */
  /* init context for 1st pass */
  return USHAReset(&ctx->shaContext, whichSha) ||
         /* and start with inner pad */
         USHAInput(&ctx->shaContext, k_ipad, blocksize);
}

/*
 *  hmacInput
 *
 *  Description:
 *      This function accepts an array of octets as the next portion
 *      of the message.
```

```
 *
 *  Parameters:
 *      context: [in/out]
 *          The HMAC context to update
 *      message_array: [in]
 *          An array of characters representing the next portion of
 *          the message.
 *      length: [in]
 *          The length of the message in message_array
 *
 *  Returns:
 *      sha Error Code.
 *
 */
int hmacInput(HMACContext *ctx, const unsigned char *text,
    int text_len)
{
  if (!ctx) return shaNull;
  /* then text of datagram */
  return USHAInput(&ctx->shaContext, text, text_len);
}

/*
 * HMACFinalBits
 *
 * Description:
 *   This function will add in any final bits of the message.
 *
 * Parameters:
 *   context: [in/out]
 *     The HMAC context to update
 *   message_bits: [in]
 *     The final bits of the message, in the upper portion of the
 *     byte. (Use 0b###00000 instead of 0b00000### to input the
 *     three bits ###.)
 *   length: [in]
 *     The number of bits in message_bits, between 1 and 7.
 *
 * Returns:
 *   sha Error Code.
 */
int hmacFinalBits(HMACContext *ctx,
    const uint8_t bits,
    unsigned int bitcount)
{
  if (!ctx) return shaNull;
  /* then final bits of datagram */
  return USHAFinalBits(&ctx->shaContext, bits, bitcount);
```

```
}

/*
 * HMACResult
 *
 * Description:
 *   This function will return the N-byte message digest into the
 *   Message_Digest array provided by the caller.
 *   NOTE: The first octet of hash is stored in the 0th element,
 *      the last octet of hash in the Nth element.
 *
 * Parameters:
 *   context: [in/out]
 *     The context to use to calculate the HMAC hash.
 *   digest: [out]
 *     Where the digest is returned.
 *   NOTE 2: The length of the hash is determined by the value of
 *      whichSha that was passed to hmacReset().
 *
 * Returns:
 *   sha Error Code.
 *
 */
int hmacResult(HMACContext *ctx, uint8_t *digest)
{
  if (!ctx) return shaNull;

  /* finish up 1st pass */
  /* (Use digest here as a temporary buffer.) */
  return USHAResult(&ctx->shaContext, digest) ||

         /* perform outer SHA */
         /* init context for 2nd pass */
         USHAReset(&ctx->shaContext, ctx->whichSha) ||

         /* start with outer pad */
         USHAInput(&ctx->shaContext, ctx->k_opad, ctx->blockSize) ||

         /* then results of 1st hash */
         USHAInput(&ctx->shaContext, digest, ctx->hashSize) ||

         /* finish up 2nd pass */
         USHAResult(&ctx->shaContext, digest);
}
```

8.4. The Test Driver

The following code is a main program test driver to exercise the code in sha1.c, sha224-256.c, and sha384-512.c. The test driver can also be used as a stand-alone program for generating the hashes.

See also [RFC2202], [RFC4231], and [SHAVS].

```
/************************** shatest.c ***************************/
/********************* See RFC 4634 for details ******************/
/*
 *  Description:
 *    This file will exercise the SHA code performing
 *      the three tests documented in FIPS PUB 180-2
 *        (http://csrc.nist.gov/publications/fips/
 *         fips180-2/fips180-2withchangenotice.pdf)
 *      one that calls SHAInput with an exact multiple of 512 bits
 *      the seven tests documented for each algorithm in
 *        "The Secure Hash Algorithm Validation System (SHAVS)",
 *        three of which are bit-level tests
 *        (http://csrc.nist.gov/cryptval/shs/SHAVS.pdf)
 *
 *    This file will exercise the HMAC SHA1 code performing
 *      the seven tests documented in RFCs 2202 and 4231.
 *
 *    To run the tests and just see PASSED/FAILED, use the -p option.
 *
 *    Other options exercise:
 *      hashing an arbitrary string
 *      hashing a file's contents
 *      a few error test checks
 *      printing the results in raw format
 *
 *  Portability Issues:
 *    None.
 *
 */

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "sha.h"

static int xgetopt(int argc, char **argv, const char *optstring);
extern char *xoptarg;
static int scasecmp(const char *s1, const char *s2);
```

```
/*
 *  Define patterns for testing
 */
#define TEST1    "abc"
#define TEST2_1  \
        "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
#define TEST2_2a \
        "abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmn"
#define TEST2_2b \
        "hijklmnoijklmnopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu"
#define TEST2_2  TEST2_2a TEST2_2b
#define TEST3    "a"                            /* times 1000000 */
#define TEST4a   "01234567012345670123456701234567"
#define TEST4b   "01234567012345670123456701234567"
    /* an exact multiple of 512 bits */
#define TEST4   TEST4a TEST4b                   /* times 10 */

#define TEST7_1 \
  "\x49\xb2\xae\xc2\x59\x4b\xbe\x3a\x3b\x11\x75\x42\xd9\x4a\xc8"
#define TEST8_1 \
  "\x9a\x7d\xfd\xf1\xec\xea\xd0\x6e\xd6\x46\xaa\x55\xfe\x75\x71\x46"
#define TEST9_1 \
  "\x65\xf9\x32\x99\x5b\xa4\xce\x2c\xb1\xb4\xa2\xe7\x1a\xe7\x02\x20" \
  "\xaa\xce\xc8\x96\x2d\xd4\x49\x9c\xbd\x7c\x88\x7a\x94\xea\xaa\x10" \
  "\x1e\xa5\xaa\xbc\x52\x9b\x4e\x7e\x43\x66\x5a\x5a\xf2\xcd\x03\xfe" \
  "\x67\x8e\xa6\xa5\x00\x5b\xba\x3b\x08\x22\x04\xc2\x8b\x91\x09\xf4" \
  "\x69\xda\xc9\x2a\xaa\xb3\xaa\x7c\x11\xa1\xb3\x2a"
#define TEST10_1 \
  "\xf7\x8f\x92\x14\x1b\xcd\x17\x0a\xe8\x9b\x4f\xba\x15\xa1\xd5\x9f" \
  "\x3f\xd8\x4d\x22\x3c\x92\x51\xbd\xac\xbb\xae\x61\xd0\x5e\xd1\x15" \
  "\xa0\x6a\x7c\xe1\x17\xb7\xbe\xea\xd2\x44\x21\xde\xd9\xc3\x25\x92" \
  "\xbd\x57\xed\xea\xe3\x9c\x39\xfa\x1f\xe8\x94\x6a\x84\xd0\xcf\x1f" \
  "\x7b\xee\xad\x17\x13\xe2\xe0\x95\x98\x97\x34\x7f\x67\xc8\x0b\x04" \
  "\x00\xc2\x09\x81\x5d\x6b\x10\xa6\x83\x83\x6f\xd5\x56\x2a\x56\xca" \
  "\xb1\xa2\x8e\x81\xb6\x57\x66\x54\x63\x1c\xf1\x65\x66\xb8\x6e\x3b" \
  "\x33\xa1\x08\xb0\x53\x07\xc0\x0a\xff\x14\xa7\x68\xed\x73\x50\x60" \
  "\x6a\x0f\x85\xe6\xa9\x1d\x39\x6f\x5b\x5c\xbe\x57\x7f\x9b\x38\x80" \
  "\x7c\x7d\x52\x3d\x6d\x79\x2f\x6e\xbc\x24\xa4\xec\xf2\xb3\xa4\x27" \
  "\xcd\xbb\xfb"
#define TEST7_224 \
  "\xf0\x70\x06\xf2\x5a\x0b\xea\x68\xcd\x76\xa2\x95\x87\xc2\x8d"
#define TEST8_224 \
  "\x18\x80\x40\x05\xdd\x4f\xbd\x15\x56\x29\x9d\x6f\x9d\x93\xdf\x62"
#define TEST9_224 \
  "\xa2\xbe\x6e\x46\x32\x81\x09\x02\x94\xd9\xce\x94\x82\x65\x69\x42" \
  "\x3a\x3a\x30\x5e\xd5\xe2\x11\x6c\xd4\xa4\xc9\x87\xfc\x06\x57\x00" \
  "\x64\x91\xb1\x49\xcc\xd4\xb5\x11\x30\xac\x62\xb1\x9d\xc2\x48\xc7" \
  "\x44\x54\x3d\x20\xcd\x39\x52\xdc\xed\x1f\x06\xcc\x3b\x18\xb9\x1f" \
```

```
  "\x3f\x55\x63\x3e\xcc\x30\x85\xf4\x90\x70\x60\xd2"
#define TEST10_224 \
  "\x55\xb2\x10\x07\x9c\x61\xb5\x3a\xdd\x52\x06\x22\xd1\xac\x97\xd5" \
  "\xcd\xbe\x8c\xb3\x3a\xa0\xae\x34\x45\x17\xbe\xe4\xd7\xba\x09\xab" \
  "\xc8\x53\x3c\x52\x50\x88\x7a\x43\xbe\xbb\xac\x90\x6c\x2e\x18\x37" \
  "\xf2\x6b\x36\xa5\x9a\xe3\xbe\x78\x14\xd5\x06\x89\x6b\x71\x8b\x2a" \
  "\x38\x3e\xcd\xac\x16\xb9\x61\x25\x55\x3f\x41\x6f\xf3\x2c\x66\x74" \
  "\xc7\x45\x99\xa9\x00\x53\x86\xd9\xce\x11\x12\x24\x5f\x48\xee\x47" \
  "\x0d\x39\x6c\x1e\xd6\x3b\x92\x67\x0c\xa5\x6e\xc8\x4d\xee\xa8\x14" \
  "\xb6\x13\x5e\xca\x54\x39\x2b\xde\xdb\x94\x89\xbc\x9b\x87\x5a\x8b" \
  "\xaf\x0d\xc1\xae\x78\x57\x36\x91\x4a\xb7\xda\xa2\x64\xbc\x07\x9d" \
  "\x26\x9f\x2c\x0d\x7e\xdd\xd8\x10\xa4\x26\x14\x5a\x07\x76\xf6\x7c" \
  "\x87\x82\x73"
#define TEST7_256 \
  "\xbe\x27\x46\xc6\xdb\x52\x76\x5f\xdb\x2f\x88\x70\x0f\x9a\x73"
#define TEST8_256 \
  "\xe3\xd7\x25\x70\xdc\xdd\x78\x7c\xe3\x88\x7a\xb2\xcd\x68\x46\x52"
#define TEST9_256 \
  "\x3e\x74\x03\x71\xc8\x10\xc2\xb9\x9f\xc0\x4e\x80\x49\x07\xef\x7c" \
  "\xf2\x6b\xe2\x8b\x57\xcb\x58\xa3\xe2\xf3\xc0\x07\x16\x6e\x49\xc1" \
  "\x2e\x9b\xa3\x4c\x01\x04\x06\x91\x29\xea\x76\x15\x64\x25\x45\x70" \
  "\x3a\x2b\xd9\x01\xe1\x6e\xb0\xe0\x5d\xeb\xa0\x14\xeb\xff\x64\x06" \
  "\xa0\x7d\x54\x36\x4e\xff\x74\x2d\xa7\x79\xb0\xb3"
#define TEST10_256 \
  "\x83\x26\x75\x4e\x22\x77\x37\x2f\x4f\xc1\x2b\x20\x52\x7a\xfe\xf0" \
  "\x4d\x8a\x05\x69\x71\xb1\x1a\xd5\x71\x23\xa7\xc1\x37\x76\x00\x00" \
  "\xd7\xbe\xf6\xf3\xc1\xf7\xa9\x08\x3a\xa3\x9d\x81\x0d\xb3\x10\x77" \
  "\x7d\xab\x8b\x1e\x7f\x02\xb8\x4a\x26\xc7\x73\x32\x5f\x8b\x23\x74" \
  "\xde\x7a\x4b\x5a\x58\xcb\x5c\x5c\xf3\x5b\xce\xe6\xfb\x94\x6e\x5b" \
  "\xd6\x94\xfa\x59\x3a\x8b\xeb\x3f\x9d\x65\x92\xec\xed\xaa\x66\xca" \
  "\x82\xa2\x9d\x0c\x51\xbc\xf9\x33\x62\x30\xe5\xd7\x84\xe4\xc0\xa4" \
  "\x3f\x8d\x79\xa3\x0a\x16\x5c\xba\xbe\x45\x2b\x77\x4b\x9c\x71\x09" \
  "\xa9\x7d\x13\x8f\x12\x92\x28\x96\x6f\x6c\x0a\xdc\x10\x6a\xad\x5a" \
  "\x9f\xdd\x30\x82\x57\x69\xb2\xc6\x71\xaf\x67\x59\xdf\x28\xeb\x39" \
  "\x3d\x54\xd6"
#define TEST7_384 \
  "\x8b\xc5\x00\xc7\x7c\xee\xd9\x87\x9d\xa9\x89\x10\x7c\xe0\xaa"
#define TEST8_384 \
  "\xa4\x1c\x49\x77\x79\xc0\x37\x5f\xf1\x0a\x7f\x4e\x08\x59\x17\x39"
#define TEST9_384 \
  "\x68\xf5\x01\x79\x2d\xea\x97\x96\x76\x70\x22\xd9\x3d\xa7\x16\x79" \
  "\x30\x99\x20\xfa\x10\x12\xae\xa3\x57\xb2\xb1\x33\x1d\x40\xa1\xd0" \
  "\x3c\x41\xc2\x40\xb3\xc9\xa7\x5b\x48\x92\xf4\xc0\x72\x4b\x68\xc8" \
  "\x75\x32\x1a\xb8\xcf\xe5\x02\x3b\xd3\x75\xbc\x0f\x94\xbd\x89\xfe" \
  "\x04\xf2\x97\x10\x5d\x7b\x82\xff\xc0\x02\x1a\xeb\x1c\xcb\x67\x4f" \
  "\x52\x44\xea\x34\x97\xde\x26\xa4\x19\x1c\x5f\x62\xe5\xe9\xa2\xd8" \
  "\x08\x2f\x05\x51\xf4\xa5\x30\x68\x26\xe9\x1c\xc0\x06\xce\x1b\xf6" \
  "\x0f\xf7\x19\xd4\x2f\xa5\x21\xc8\x71\xcd\x23\x94\xd9\x6e\xf4\x46" \
```

```
  "\x8f\x21\x96\x6b\x41\xf2\xba\x80\xc2\x6e\x83\xa9"
#define TEST10_384 \
  "\x39\x96\x69\xe2\x8f\x6b\x9c\x6d\xbc\xbb\x69\x12\xec\x10\xff\xcf" \
  "\x74\x79\x03\x49\xb7\xdc\x8f\xbe\x4a\x8e\x7b\x3b\x56\x21\xdb\x0f" \
  "\x3e\x7d\xc8\x7f\x82\x32\x64\xbb\xe4\x0d\x18\x11\xc9\xea\x20\x61" \
  "\xe1\xc8\x4a\xd1\x0a\x23\xfa\xc1\x72\x7e\x72\x02\xfc\x3f\x50\x42" \
  "\xe6\xbf\x58\xcb\xa8\xa2\x74\x6e\x1f\x64\xf9\xb9\xea\x35\x2c\x71" \
  "\x15\x07\x05\x3c\xf4\xe5\x33\x9d\x52\x86\x5f\x25\xcc\x22\xb5\xe8" \
  "\x77\x84\xa1\x2f\xc9\x61\xd6\x6c\xb6\xe8\x95\x73\x19\x9a\x2c\xe6" \
  "\x56\x5c\xbd\xf1\x3d\xca\x40\x38\x32\xcf\xcb\x0e\x8b\x72\x11\xe8" \
  "\x3a\xf3\x2a\x11\xac\x17\x92\x9f\xf1\xc0\x73\xa5\x1c\xc0\x27\xaa" \
  "\xed\xef\xf8\x5a\xad\x7c\x2b\x7c\x5a\x80\x3e\x24\x04\xd9\x6d\x2a" \
  "\x77\x35\x7b\xda\x1a\x6d\xae\xed\x17\x15\x1c\xb9\xbc\x51\x25\xa4" \
  "\x22\xe9\x41\xde\x0c\xa0\xfc\x50\x11\xc2\x3e\xcf\xfe\xfd\xd0\x96" \
  "\x76\x71\x1c\xf3\xdb\x0a\x34\x40\x72\x0e\x16\x15\xc1\xf2\x2f\xbc" \
  "\x3c\x72\x1d\xe5\x21\xe1\xb9\x9b\xa1\xbd\x55\x77\x40\x86\x42\x14" \
  "\x7e\xd0\x96"
#define TEST7_512 \
  "\x08\xec\xb5\x2e\xba\xe1\xf7\x42\x2d\xb6\x2b\xcd\x54\x26\x70"
#define TEST8_512 \
  "\x8d\x4e\x3c\x0e\x38\x89\x19\x14\x91\x81\x6e\x9d\x98\xbf\xf0\xa0"
#define TEST9_512 \
  "\x3a\xdd\xec\x85\x59\x32\x16\xd1\x61\x9a\xa0\x2d\x97\x56\x97\x0b" \
  "\xfc\x70\xac\xe2\x74\x4f\x7c\x6b\x27\x88\x15\x10\x28\xf7\xb6\xa2" \
  "\x55\x0f\xd7\x4a\x7e\x6e\x69\xc2\xc9\xb4\x5f\xc4\x54\x96\x6d\xc3" \
  "\x1d\x2e\x10\xda\x1f\x95\xce\x02\xbe\xb4\xbf\x87\x65\x57\x4c\xbd" \
  "\x6e\x83\x37\xef\x42\x0a\xdc\x98\xc1\x5c\xb6\xd5\xe4\xa0\x24\x1b" \
  "\xa0\x04\x6d\x25\x0e\x51\x02\x31\xca\xc2\x04\x6c\x99\x16\x06\xab" \
  "\x4e\xe4\x14\x5b\xee\x2f\xf4\xbb\x12\x3a\xab\x49\x8d\x9d\x44\x79" \
  "\x4f\x99\xcc\xad\x89\xa9\xa1\x62\x12\x59\xed\xa7\x0a\x5b\x6d\xd4" \
  "\xbd\xd8\x77\x78\xc9\x04\x3b\x93\x84\xf5\x49\x06"
#define TEST10_512 \
  "\xa5\x5f\x20\xc4\x11\xaa\xd1\x32\x80\x7a\x50\x2d\x65\x82\x4e\x31" \
  "\xa2\x30\x54\x32\xaa\x3d\x06\xd3\xe2\x82\xa8\xd8\x4e\x0d\xe1\xde" \
  "\x69\x74\xbf\x49\x54\x69\xfc\x7f\x33\x8f\x80\x54\xd5\x8c\x26\xc4" \
  "\x93\x60\xc3\xe8\x7a\xf5\x65\x23\xac\xf6\xd8\x9d\x03\xe5\x6f\xf2" \
  "\xf8\x68\x00\x2b\xc3\xe4\x31\xed\xc4\x4d\xf2\xf0\x22\x3d\x4b\xb3" \
  "\xb2\x43\x58\x6e\x1a\x7d\x92\x49\x36\x69\x4f\xcb\xba\xf8\x8d\x95" \
  "\x19\xe4\xeb\x50\xa6\x44\xf8\xe4\xf9\x5e\xb0\xea\x95\xbc\x44\x65" \
  "\xc8\x82\x1a\xac\xd2\xfe\x15\xab\x49\x81\x16\x4b\xbb\x6d\xc3\x2f" \
  "\x96\x90\x87\xa1\x45\xb0\xd9\xcc\x9c\x67\xc2\x2b\x76\x32\x99\x41" \
  "\x9c\xc4\x12\x8b\xe9\xa0\x77\xb3\xac\xe6\x34\x06\x4e\x6d\x99\x28" \
  "\x35\x13\xdc\x06\xe7\x51\x5d\x0d\x73\x13\x2e\x9a\x0d\xc6\xd3\xb1" \
  "\xf8\xb2\x46\xf1\xa9\x8a\x3f\xc7\x29\x41\xb1\xe3\xbb\x20\x98\xe8" \
  "\xbf\x16\xf2\x68\xd6\x4f\x0b\x0f\x47\x07\xfe\x1e\xa1\xa1\x79\x1b" \
  "\xa2\xf3\xc0\xc7\x58\xe5\xf5\x51\x86\x3a\x96\xc9\x49\xad\x47\xd7" \
  "\xfb\x40\xd2"
#define SHA1_SEED "\xd0\x56\x9c\xb3\x66\x5a\x8a\x43\xeb\x6e\xa2\x3d" \
```

```
  "\x75\xa3\xc4\xd2\x05\x4a\x0d\x7d"
#define SHA224_SEED "\xd0\x56\x9c\xb3\x66\x5a\x8a\x43\xeb\x6e\xa2" \
  "\x3d\x75\xa3\xc4\xd2\x05\x4a\x0d\x7d\x66\xa9\xca\x99\xc9\xce\xb0" \
  "\x27"
#define SHA256_SEED "\xf4\x1e\xce\x26\x13\xe4\x57\x39\x15\x69\x6b" \
  "\x5a\xdc\xd5\x1c\xa3\x28\xbe\x3b\xf5\x66\xa9\xca\x99\xc9\xce\xb0" \
  "\x27\x9c\x1c\xb0\xa7"
#define SHA384_SEED "\x82\x40\xbc\x51\xe4\xec\x7e\xf7\x6d\x18\xe3" \
  "\x52\x04\xa1\x9f\x51\xa5\x21\x3a\x73\xa8\x1d\x6f\x94\x46\x80\xd3" \
  "\x07\x59\x48\xb7\xe4\x63\x80\x4e\xa3\xd2\x6e\x13\xea\x82\x0d\x65" \
  "\xa4\x84\xbe\x74\x53"
#define SHA512_SEED "\x47\x3f\xf1\xb9\xb3\xff\xdf\xa1\x26\x69\x9a" \
  "\xc7\xef\x9e\x8e\x78\x77\x73\x09\x58\x24\xc6\x42\x55\x7c\x13\x99" \
  "\xd9\x8e\x42\x20\x44\x8d\xc3\x5b\x99\xbf\xdd\x44\x77\x95\x43\x92" \
  "\x4c\x1c\xe9\x3b\xc5\x94\x15\x38\x89\x5d\xb9\x88\x26\x1b\x00\x77" \
  "\x4b\x12\x27\x20\x39"

#define TESTCOUNT 10
#define HASHCOUNT 5
#define RANDOMCOUNT 4
#define HMACTESTCOUNT 7

#define PRINTNONE 0
#define PRINTTEXT 1
#define PRINTRAW 2
#define PRINTHEX 3
#define PRINTBASE64 4

#define PRINTPASSFAIL 1
#define PRINTFAIL 2

#define length(x) (sizeof(x)-1)

/* Test arrays for hashes. */
struct hash {
    const char *name;
    SHAversion whichSha;
    int hashsize;
    struct {
        const char *testarray;
        int length;
        long repeatcount;
        int extrabits;
        int numberExtrabits;
        const char *resultarray;
    } tests[TESTCOUNT];
    const char *randomtest;
    const char *randomresults[RANDOMCOUNT];
```

```
} hashes[HASHCOUNT] = {
  { "SHA1", SHA1, SHA1HashSize,
    {
      /* 1 */ { TEST1, length(TEST1), 1, 0, 0,
        "A9993E364706816ABA3E25717850C26C9CD0D89D" },
      /* 2 */ { TEST2_1, length(TEST2_1), 1, 0, 0,
        "84983E441C3BD26EBAAE4AA1F95129E5E54670F1" },
      /* 3 */ { TEST3, length(TEST3), 1000000, 0, 0,
        "34AA973CD4C4DAA4F61EEB2BDBAD27316534016F" },
      /* 4 */ { TEST4, length(TEST4), 10, 0, 0,
        "DEA356A2CDDD90C7A7ECEDC5EBB563934F460452" },
      /* 5 */ { "", 0, 0, 0x98, 5,
        "29826B003B906E660EFF4027CE98AF3531AC75BA" },
      /* 6 */ { "\x5e", 1, 1, 0, 0,
        "5E6F80A34A9798CAFC6A5DB96CC57BA4C4DB59C2" },
      /* 7 */ { TEST7_1, length(TEST7_1), 1, 0x80, 3,
        "6239781E03729919C01955B3FFA8ACB60B988340" },
      /* 8 */ { TEST8_1, length(TEST8_1), 1, 0, 0,
        "82ABFF6605DBE1C17DEF12A394FA22A82B544A35" },
      /* 9 */ { TEST9_1, length(TEST9_1), 1, 0xE0, 3,
        "8C5B2A5DDAE5A97FC7F9D85661C672ADBF7933D4" },
      /* 10 */ { TEST10_1, length(TEST10_1), 1, 0, 0,
        "CB0082C8F197D260991BA6A460E76E202BAD27B3" }
    }, SHA1_SEED, { "E216836819477C7F78E0D843FE4FF1B6D6C14CD4",
        "A2DBC7A5B1C6C0A8BCB7AAA41252A6A7D0690DBC",
        "DB1F9050BB863DFEF4CE37186044E2EEB17EE013",
        "127FDEDF43D372A51D5747C48FBFFE38EF6CDF7B"
     } },
  { "SHA224", SHA224, SHA224HashSize,
    {
      /* 1 */ { TEST1, length(TEST1), 1, 0, 0,
        "23097D223405D8228642A477BDA255B32AADBCE4BDA0B3F7E36C9DA7" },
      /* 2 */ { TEST2_1, length(TEST2_1), 1, 0, 0,
        "75388B16512776CC5DBA5DA1FD890150B0C6455CB4F58B1952522525" },
      /* 3 */ { TEST3, length(TEST3), 1000000, 0, 0,
        "20794655980C91D8BBB4C1EA97618A4BF03F42581948B2EE4EE7AD67" },
      /* 4 */ { TEST4, length(TEST4), 10, 0, 0,
        "567F69F168CD7844E65259CE658FE7AADFA25216E68ECA0EB7AB8262" },
      /* 5 */ { "", 0, 0, 0x68, 5,
        "E3B048552C3C387BCAB37F6EB06BB79B96A4AEE5FF27F51531A9551C" },
      /* 6 */ { "\x07", 1, 1, 0, 0,
        "00ECD5F138422B8AD74C9799FD826C531BAD2FCABC7450BEE2AA8C2A" },
      /* 7 */ { TEST7_224, length(TEST7_224), 1, 0xA0, 3,
        "1B01DB6CB4A9E43DED1516BEB3DB0B87B6D1EA43187462C608137150" },
      /* 8 */ { TEST8_224, length(TEST8_224), 1, 0, 0,
        "DF90D78AA78821C99B40BA4C966921ACCD8FFB1E98AC388E56191DB1" },
      /* 9 */ { TEST9_224, length(TEST9_224), 1, 0xE0, 3,
        "54BEA6EAB8195A2EB0A7906A4B4A876666300EEFBD1F3B8474F9CD57" },
```

```
      /* 10 */ { TEST10_224, length(TEST10_224), 1, 0, 0,
        "0B31894EC8937AD9B91BDFBCBA294D9ADEFAA18E09305E9F20D5C3A4" }
    }, SHA224_SEED, { "100966A5B4FDE0B42E2A6C5953D4D7F41BA7CF79FD"
        "2DF431416734BE", "1DCA396B0C417715DEFAAE9641E10A2E99D55A"
        "BCB8A00061EB3BE8BD", "1864E627BDB2319973CD5ED7D68DA71D8B"
        "F0F983D8D9AB32C34ADB34", "A2406481FC1BCAF24DD08E6752E844"
        "709563FB916227FED598EB621F"
     } },
  { "SHA256", SHA256, SHA256HashSize,
  {
      /* 1 */ { TEST1, length(TEST1), 1, 0, 0, "BA7816BF8F01CFEA4141"
        "40DE5DAE2223B00361A396177A9CB410FF61F20015AD" },
      /* 2 */ { TEST2_1, length(TEST2_1), 1, 0, 0, "248D6A61D20638B8"
        "E5C026930C3E6039A33CE45964FF2167F6ECEDD419DB06C1" },
      /* 3 */ { TEST3, length(TEST3), 1000000, 0, 0, "CDC76E5C9914FB92"
        "81A1C7E284D73E67F1809A48A497200E046D39CCC7112CD0" },
      /* 4 */ { TEST4, length(TEST4), 10, 0, 0, "594847328451BDFA"
        "85056225462CC1D867D877FB388DF0CE35F25AB5562BFBB5" },
      /* 5 */ { "", 0, 0, 0x68, 5, "D6D3E02A31A84A8CAA9718ED6C2057BE"
        "09DB45E7823EB5079CE7A573A3760F95" },
      /* 6 */ { "\x19", 1, 1, 0, 0, "68AA2E2EE5DFF96E3355E6C7EE373E3D"
        "6A4E17F75F9518D843709C0C9BC3E3D4" },
      /* 7 */ { TEST7_256, length(TEST7_256), 1, 0x60, 3, "77EC1DC8"
        "9C821FF2A1279089FA091B35B8CD960BCAF7DE01C6A7680756BEB972" },
      /* 8 */ { TEST8_256, length(TEST8_256), 1, 0, 0, "175EE69B02BA"
        "9B58E2B0A5FD13819CEA573F3940A94F825128CF4209BEABB4E8" },
      /* 9 */ { TEST9_256, length(TEST9_256), 1, 0xA0, 3, "3E9AD646"
        "8BBBAD2AC3C2CDC292E018BA5FD70B960CF1679777FCE708FDB066E9" },
      /* 10 */ { TEST10_256, length(TEST10_256), 1, 0, 0, "97DBCA7D"
        "F46D62C8A422C941DD7E835B8AD3361763F7E9B2D95F4F0DA6E1CCBC" },
    }, SHA256_SEED, { "83D28614D49C3ADC1D6FC05DB5F48037C056F8D2A4CE44"
        "EC6457DEA5DD797CD1", "99DBE3127EF2E93DD9322D6A07909EB33B6399"
        "5E529B3F954B8581621BB74D39", "8D4BE295BB64661CA3C7EFD129A2F7"
        "25B33072DBDDE32385B9A87B9AF88EA76F", "40AF5D3F9716B040DF9408"
        "E31536B70FF906EC51B00447CA97D7DD97C12411F4"
    } },
  { "SHA384", SHA384, SHA384HashSize,
    {
      /* 1 */ { TEST1, length(TEST1), 1, 0, 0,
        "CB00753F45A35E8BB5A03D699AC65007272C32AB0EDED163"
        "1A8B605A43FF5BED8086072BA1E7CC2358BAECA134C825A7" },
      /* 2 */ { TEST2_2, length(TEST2_2), 1, 0, 0,
        "09330C33F71147E83D192FC782CD1B4753111B173B3B05D2"
        "2FA08086E3B0F712FCC7C71A557E2DB966C3E9FA91746039" },
      /* 3 */ { TEST3, length(TEST3), 1000000, 0, 0,
        "9D0E1809716474CB086E834E310A4A1CED149E9C00F24852"
        "7972CEC5704C2A5B07B8B3DC38ECC4EBAE97DDD87F3D8985" },
      /* 4 */ { TEST4, length(TEST4), 10, 0, 0,
```

```
        "2FC64A4F500DDB6828F6A3430B8DD72A368EB7F3A8322A70"
        "BC84275B9C0B3AB00D27A5CC3C2D224AA6B61A0D79FB4596" },
      /* 5 */ { "", 0, 0, 0x10, 5,
        "8D17BE79E32B6718E07D8A603EB84BA0478F7FCFD1BB9399"
        "5F7D1149E09143AC1FFCFC56820E469F3878D957A15A3FE4" },
      /* 6 */ { "\xb9", 1, 1, 0, 0,
        "BC8089A19007C0B14195F4ECC74094FEC64F01F90929282C"
        "2FB392881578208AD466828B1C6C283D2722CF0AD1AB6938" },
      /* 7 */ { TEST7_384, length(TEST7_384), 1, 0xA0, 3,
        "D8C43B38E12E7C42A7C9B810299FD6A770BEF30920F17532"
        "A898DE62C7A07E4293449C0B5FA70109F0783211CFC4BCE3" },
      /* 8 */ { TEST8_384, length(TEST8_384), 1, 0, 0,
        "C9A68443A005812256B8EC76B00516F0DBB74FAB26D66591"
        "3F194B6FFB0E91EA9967566B58109CBC675CC208E4C823F7" },
      /* 9 */ { TEST9_384, length(TEST9_384), 1, 0xE0, 3,
        "5860E8DE91C21578BB4174D227898A98E0B45C4C760F0095"
        "49495614DAEDC0775D92D11D9F8CE9B064EEAC8DAFC3A297" },
      /* 10 */ { TEST10_384, length(TEST10_384), 1, 0, 0,
        "4F440DB1E6EDD2899FA335F09515AA025EE177A79F4B4AAF"
        "38E42B5C4DE660F5DE8FB2A5B2FBD2A3CBFFD20CFF1288C0" }
    }, SHA384_SEED, { "CE44D7D63AE0C91482998CF662A51EC80BF6FC68661A3C"
        "57F87566112BD635A743EA904DEB7D7A42AC808CABE697F38F", "F9C6D2"
        "61881FEE41ACD39E67AA8D0BAD507C7363EB67E2B81F45759F9C0FD7B503"
        "DF1A0B9E80BDE7BC333D75B804197D", "D96512D8C9F4A7A4967A366C01"
        "C6FD97384225B58343A88264847C18E4EF8AB7AEE4765FFBC3E30BD485D3"
        "638A01418F", "0CA76BD0813AF1509E170907A96005938BC985628290B2"
        "5FEF73CF6FAD68DDBA0AC8920C94E0541607B0915A7B4457F7"
    } },
  { "SHA512", SHA512, SHA512HashSize,
    {
      /* 1 */ { TEST1, length(TEST1), 1, 0, 0,
        "DDAF35A193617ABACC417349AE20413112E6FA4E89A97EA2"
        "0A9EEEE64B55D39A2192992A274FC1A836BA3C23A3FEEBBD"
        "454D4423643CE80E2A9AC94FA54CA49F" },
      /* 2 */ { TEST2_2, length(TEST2_2), 1, 0, 0,
        "8E959B75DAE313DA8CF4F72814FC143F8F7779C6EB9F7FA1"
        "7299AEADB6889018501D289E4900F7E4331B99DEC4B5433A"
        "C7D329EEB6DD26545E96E55B874BE909" },
       /* 3 */ { TEST3, length(TEST3), 1000000, 0, 0,
        "E718483D0CE769644E2E42C7BC15B4638E1F98B13B204428"
        "5632A803AFA973EBDE0FF244877EA60A4CB0432CE577C31B"
        "EB009C5C2C49AA2E4EADB217AD8CC09B" },
      /* 4 */ { TEST4, length(TEST4), 10, 0, 0,
        "89D05BA632C699C31231DED4FFC127D5A894DAD412C0E024"
        "DB872D1ABD2BA8141A0F85072A9BE1E2AA04CF33C765CB51"
        "0813A39CD5A84C4ACAA64D3F3FB7BAE9" },
      /* 5 */ { "", 0, 0, 0xB0, 5,
        "D4EE29A9E90985446B913CF1D1376C836F4BE2C1CF3CADA0"
```

```
        "720A6BF4857D886A7ECB3C4E4C0FA8C7F95214E41DC1B0D2"
        "1B22A84CC03BF8CE4845F34DD5BDBAD4" },
      /* 6 */ { "\xD0", 1, 1, 0, 0,
        "9992202938E882E73E20F6B69E68A0A7149090423D93C81B"
        "AB3F21678D4ACEEEE50E4E8CAFADA4C85A54EA8306826C4A"
        "D6E74CECE9631BFA8A549B4AB3FBBA15" },
      /* 7 */ { TEST7_512, length(TEST7_512), 1, 0x80, 3,
        "ED8DC78E8B01B69750053DBB7A0A9EDA0FB9E9D292B1ED71"
        "5E80A7FE290A4E16664FD913E85854400C5AF05E6DAD316B"
        "7359B43E64F8BEC3C1F237119986BBB6" },
      /* 8 */ { TEST8_512, length(TEST8_512), 1, 0, 0,
        "CB0B67A4B8712CD73C9AABC0B199E9269B20844AFB75ACBD"
        "D1C153C9828924C3DDEDAAFE669C5FDD0BC66F630F677398"
        "8213EB1B16F517AD0DE4B2F0C95C90F8" },
      /* 9 */ { TEST9_512, length(TEST9_512), 1, 0x80, 3,
        "32BA76FC30EAA0208AEB50FFB5AF1864FDBF17902A4DC0A6"
        "82C61FCEA6D92B783267B21080301837F59DE79C6B337DB2"
        "526F8A0A510E5E53CAFED4355FE7C2F1" },
      /* 10 */ { TEST10_512, length(TEST10_512), 1, 0, 0,
        "C665BEFB36DA189D78822D10528CBF3B12B3EEF726039909"
        "C1A16A270D48719377966B957A878E720584779A62825C18"
        "DA26415E49A7176A894E7510FD1451F5" }
    }, SHA512_SEED, { "2FBB1E7E00F746BA514FBC8C421F36792EC0E11FF5EFC3"
        "78E1AB0C079AA5F0F66A1E3EDBAEB4F9984BE14437123038A452004A5576"
        "8C1FD8EED49E4A21BEDCD0", "25CBE5A4F2C7B1D7EF07011705D50C62C5"
        "000594243EAFD1241FC9F3D22B58184AE2FEE38E171CF8129E29459C9BC2"
        "EF461AF5708887315F15419D8D17FE7949", "5B8B1F2687555CE2D7182B"
        "92E5C3F6C36547DA1C13DBB9EA4F73EA4CBBAF89411527906D35B1B06C1B"
        "6A8007D05EC66DF0A406066829EAB618BDE3976515AAFC", "46E36B007D"
        "19876CDB0B29AD074FE3C08CDD174D42169D6ABE5A1414B6E79707DF5877"
        "6A98091CF431854147BB6D3C66D43BFBC108FD715BDE6AA127C2B0E79F"
    }
  }
};

/* Test arrays for HMAC. */
struct hmachash {
    const char *keyarray[5];
    int keylength[5];
    const char *dataarray[5];
    int datalength[5];
    const char *resultarray[5];
    int resultlength[5];
} hmachashes[HMACTESTCOUNT] = {
  { /* 1 */ {
      "\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b"
      "\x0b\x0b\x0b\x0b\x0b"
    }, { 20 }, {
```

```
      "\x48\x69\x20\x54\x68\x65\x72\x65" /* "Hi There" */
    }, { 8 }, {
      /* HMAC-SHA-1 */
      "B617318655057264E28BC0B6FB378C8EF146BE00",
      /* HMAC-SHA-224 */
      "896FB1128ABBDF196832107CD49DF33F47B4B1169912BA4F53684B22",
      /* HMAC-SHA-256 */
      "B0344C61D8DB38535CA8AFCEAF0BF12B881DC200C9833DA726E9376C2E32"
      "CFF7",
      /* HMAC-SHA-384 */
      "AFD03944D84895626B0825F4AB46907F15F9DADBE4101EC682AA034C7CEB"
      "C59CFAEA9EA9076EDE7F4AF152E8B2FA9CB6",
      /* HMAC-SHA-512 */
      "87AA7CDEA5EF619D4FF0B4241A1D6CB02379F4E2CE4EC2787AD0B30545E1"
      "7CDEDAA833B7D6B8A702038B274EAEA3F4E4BE9D914EEB61F1702E696C20"
      "3A126854"
    }, { SHA1HashSize, SHA224HashSize, SHA256HashSize,
      SHA384HashSize, SHA512HashSize }
  },
  { /* 2 */ {
      "\x4a\x65\x66\x65" /* "Jefe" */
    }, { 4 }, {
      "\x77\x68\x61\x74\x20\x64\x6f\x20\x79\x61\x20\x77\x61\x6e\x74"
      "\x20\x66\x6f\x72\x20\x6e\x6f\x74\x68\x69\x6e\x67\x3f"
      /* "what do ya want for nothing?" */
    }, { 28 }, {
      /* HMAC-SHA-1 */
      "EFFCDF6AE5EB2FA2D27416D5F184DF9C259A7C79",
      /* HMAC-SHA-224 */
      "A30E01098BC6DBBF45690F3A7E9E6D0F8BBEA2A39E6148008FD05E44",
      /* HMAC-SHA-256 */
      "5BDCC146BF60754E6A042426089575C75A003F089D2739839DEC58B964EC"
      "3843",
      /* HMAC-SHA-384 */
      "AF45D2E376484031617F78D2B58A6B1B9C7EF464F5A01B47E42EC3736322"
      "445E8E2240CA5E69E2C78B3239ECFAB21649",
      /* HMAC-SHA-512 */
      "164B7A7BFCF819E2E395FBE73B56E0A387BD64222E831FD610270CD7EA25"
      "05549758BF75C05A994A6D034F65F8F0E6FDCAEAB1A34D4A6B4B636E070A"
      "38BCE737"
    }, { SHA1HashSize, SHA224HashSize, SHA256HashSize,
      SHA384HashSize, SHA512HashSize }
  },
  { /* 3 */
    {
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa"
    }, { 20 }, {
```

```
      "\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd"
      "\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd"
      "\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd\xdd"
      "\xdd\xdd\xdd\xdd\xdd"
    }, { 50 }, {
      /* HMAC-SHA-1 */
      "125D7342B9AC11CD91A39AF48AA17B4F63F175D3",
      /* HMAC-SHA-224 */
      "7FB3CB3588C6C1F6FFA9694D7D6AD2649365B0C1F65D69D1EC8333EA",
      /* HMAC-SHA-256 */
      "773EA91E36800E46854DB8EBD09181A72959098B3EF8C122D9635514CED5"
      "65FE",
      /* HMAC-SHA-384 */
      "88062608D3E6AD8A0AA2ACE014C8A86F0AA635D947AC9FEBE83EF4E55966"
      "144B2A5AB39DC13814B94E3AB6E101A34F27",
      /* HMAC-SHA-512 */
      "FA73B0089D56A284EFB0F0756C890BE9B1B5DBDD8EE81A3655F83E33B227"
      "9D39BF3E848279A722C806B485A47E67C807B946A337BEE8942674278859"
      "E13292FB"
    }, { SHA1HashSize, SHA224HashSize, SHA256HashSize,
      SHA384HashSize, SHA512HashSize }
  },
  { /* 4 */ {
      "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f"
      "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19"
    }, { 25 }, {
      "\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd"
      "\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd"
      "\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd\xcd"
      "\xcd\xcd\xcd\xcd\xcd"
    }, { 50 }, {
      /* HMAC-SHA-1 */
      "4C9007F4026250C6BC8414F9BF50C86C2D7235DA",
      /* HMAC-SHA-224 */
      "6C11506874013CAC6A2ABC1BB382627CEC6A90D86EFC012DE7AFEC5A",
      /* HMAC-SHA-256 */
      "82558A389A443C0EA4CC819899F2083A85F0FAA3E578F8077A2E3FF46729"
      "665B",
      /* HMAC-SHA-384 */
      "3E8A69B7783C25851933AB6290AF6CA77A9981480850009CC5577C6E1F57"
      "3B4E6801DD23C4A7D679CCF8A386C674CFFB",
      /* HMAC-SHA-512 */
      "B0BA465637458C6990E5A8C5F61D4AF7E576D97FF94B872DE76F8050361E"
      "E3DBA91CA5C11AA25EB4D679275CC5788063A5F19741120C4F2DE2ADEBEB"
      "10A298DD"
    }, { SHA1HashSize, SHA224HashSize, SHA256HashSize,
      SHA384HashSize, SHA512HashSize }
  },
```

```
  { /* 5 */ {
      "\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c"
      "\x0c\x0c\x0c\x0c\x0c"
    }, { 20 }, {
      "Test With Truncation"
    }, { 20 }, {
      /* HMAC-SHA-1 */
      "4C1A03424B55E07FE7F27BE1",
      /* HMAC-SHA-224 */
      "0E2AEA68A90C8D37C988BCDB9FCA6FA8",
      /* HMAC-SHA-256 */
      "A3B6167473100EE06E0C796C2955552B",
      /* HMAC-SHA-384 */
      "3ABF34C3503B2A23A46EFC619BAEF897",
      /* HMAC-SHA-512 */
      "415FAD6271580A531D4179BC891D87A6"
    }, { 12, 16, 16, 16, 16 }
  },
  { /* 6 */ {
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
    }, { 80, 131 }, {
      "Test Using Larger Than Block-Size Key - Hash Key First"
    }, { 54 }, {
      /* HMAC-SHA-1 */
      "AA4AE5E15272D00E95705637CE8A3B55ED402112",
      /* HMAC-SHA-224 */
      "95E9A0DB962095ADAEBE9B2D6F0DBCE2D499F112F2D2B7273FA6870E",
      /* HMAC-SHA-256 */
      "60E431591EE0B67F0D8A26AACBF5B77F8E0BC6213728C5140546040F0EE3"
      "7F54",
      /* HMAC-SHA-384 */
      "4ECE084485813E9088D2C63A041BC5B44F9EF1012A2B588F3CD11F05033A"
      "C4C60C2EF6AB4030FE8296248DF163F44952",
      /* HMAC-SHA-512 */
      "80B24263C7C1A3EBB71493C1DD7BE8B49B46D1F41B4AEEC1121B013783F8"
      "F3526B56D037E05F2598BD0FD2215D6A1E5295E64F73F63F0AEC8B915A98"
      "5D786598"
    }, { SHA1HashSize, SHA224HashSize, SHA256HashSize,
      SHA384HashSize, SHA512HashSize }
  },
```

```
  { /* 7 */ {
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
      "\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa"
    }, { 80, 131 }, {
      "Test Using Larger Than Block-Size Key and "
      "Larger Than One Block-Size Data",
      "\x54\x68\x69\x73\x20\x69\x73\x20\x61\x20\x74\x65\x73\x74\x20"
      "\x75\x73\x69\x6e\x67\x20\x61\x20\x6c\x61\x72\x67\x65\x72\x20"
      "\x74\x68\x61\x6e\x20\x62\x6c\x6f\x63\x6b\x2d\x73\x69\x7a\x65"
      "\x20\x6b\x65\x79\x20\x61\x6e\x64\x20\x61\x20\x6c\x61\x72\x67"
      "\x65\x72\x20\x74\x68\x61\x6e\x20\x62\x6c\x6f\x63\x6b\x2d\x73"
      "\x69\x7a\x65\x20\x64\x61\x74\x61\x2e\x20\x54\x68\x65\x20\x6b"
      "\x65\x79\x20\x6e\x65\x65\x64\x73\x20\x74\x6f\x20\x62\x65\x20"
      "\x68\x61\x73\x68\x65\x64\x20\x62\x65\x66\x6f\x72\x65\x20\x62"
      "\x65\x69\x6e\x67\x20\x75\x73\x65\x64\x20\x62\x79\x20\x74\x68"
      "\x65\x20\x48\x4d\x41\x43\x20\x61\x6c\x67\x6f\x72\x69\x74\x68"
      "\x6d\x2e"
      /* "This is a test using a larger than block-size key and a "
          "larger than block-size data. The key needs to be hashed "
          "before being used by the HMAC algorithm." */
    }, { 73, 152 }, {
      /* HMAC-SHA-1 */
      "E8E99D0F45237D786D6BBAA7965C7808BBFF1A91",
      /* HMAC-SHA-224 */
      "3A854166AC5D9F023F54D517D0B39DBD946770DB9C2B95C9F6F565D1",
      /* HMAC-SHA-256 */
      "9B09FFA71B942FCB27635FBCD5B0E944BFDC63644F0713938A7F51535C3A"
      "35E2",
      /* HMAC-SHA-384 */
      "6617178E941F020D351E2F254E8FD32C602420FEB0B8FB9ADCCEBB82461E"
      "99C5A678CC31E799176D3860E6110C46523E",
      /* HMAC-SHA-512 */
      "E37B6A775DC87DBAA4DFA9F96E5E3FFDDEBD71F8867289865DF5A32D20CD"
      "C944B6022CAC3C4982B10D5EEB55C3E4DE15134676FB6DE0446065C97440"
      "FA8C6A58"
    }, { SHA1HashSize, SHA224HashSize, SHA256HashSize,
      SHA384HashSize, SHA512HashSize }
  }
};

/*
```

```
 * Check the hash value against the expected string, expressed in hex
 */
static const char hexdigits[] = "0123456789ABCDEF";
int checkmatch(const unsigned char *hashvalue,
  const char *hexstr, int hashsize)
{
  int i;
  for (i = 0; i < hashsize; ++i) {
    if (*hexstr++ != hexdigits[(hashvalue[i] >> 4) & 0xF])
      return 0;
    if (*hexstr++ != hexdigits[hashvalue[i] & 0xF]) return 0;
  }
  return 1;
}

/*
 * Print the string, converting non-printable characters to "."
 */
void printstr(const char *str, int len)
{
  for ( ; len-- > 0; str++)
    putchar(isprint((unsigned char)*str) ? *str : '.');
}

/*
 * Print the string, converting non-printable characters to hex "## ".
 */
void printxstr(const char *str, int len)
{
  for ( ; len-- > 0; str++)
    printf("%c%c ", hexdigits[(*str >> 4) & 0xF],
      hexdigits[*str & 0xF]);
}

/*
 * Print a usage message.
 */
void usage(const char *argv0)
{
  fprintf(stderr,
    "Usage:\n"
    "Common options: [-h hash] [-w|-x] [-H]\n"
    "Standard tests:\n"
      "\t%s [-m] [-l loopcount] [-t test#] [-e]\n"
      "\t\t[-r randomseed] [-R randomloop-count] "
        "[-p] [-P|-X]\n"
    "Hash a string:\n"
      "\t%s [-S expectedresult] -s hashstr [-k key]\n"
```

```
    "Hash a file:\n"
      "\t%s [-S expectedresult] -f file [-k key]\n"
    "Hash a file, ignoring whitespace:\n"
      "\t%s [-S expectedresult] -F file [-k key]\n"
    "Additional bits to add in: [-B bitcount -b bits]\n"
    "-h\thash to test: "
      "0|SHA1, 1|SHA224, 2|SHA256, 3|SHA384, 4|SHA512\n"
    "-m\tperform hmac test\n"
    "-k\tkey for hmac test\n"
    "-t\ttest case to run, 1-10\n"
    "-l\thow many times to run the test\n"
    "-e\ttest error returns\n"
    "-p\tdo not print results\n"
    "-P\tdo not print PASSED/FAILED\n"
    "-X\tprint FAILED, but not PASSED\n"
    "-r\tseed for random test\n"
    "-R\thow many times to run random test\n"
    "-s\tstring to hash\n"
    "-S\texpected result of hashed string, in hex\n"
    "-w\toutput hash in raw format\n"
    "-x\toutput hash in hex format\n"
    "-B\t# extra bits to add in after string or file input\n"
    "-b\textra bits to add (high order bits of #, 0# or 0x#)\n"
    "-H\tinput hashstr or randomseed is in hex\n"
    , argv0, argv0, argv0, argv0);
  exit(1);
}

/*
 * Print the results and PASS/FAIL.
 */
void printResult(uint8_t *Message_Digest, int hashsize,
    const char *hashname, const char *testtype, const char *testname,
    const char *resultarray, int printResults, int printPassFail)
{
  int i, k;
  if (printResults == PRINTTEXT) {
    putchar('\t');
    for (i = 0; i < hashsize ; ++i) {
      putchar(hexdigits[(Message_Digest[i] >> 4) & 0xF]);
      putchar(hexdigits[Message_Digest[i] & 0xF]);
      putchar(' ');
    }
    putchar('\n');
  } else if (printResults == PRINTRAW) {
    fwrite(Message_Digest, 1, hashsize, stdout);
  } else if (printResults == PRINTHEX) {
    for (i = 0; i < hashsize ; ++i) {
```

```
      putchar(hexdigits[(Message_Digest[i] >> 4) & 0xF]);
      putchar(hexdigits[Message_Digest[i] & 0xF]);
    }
    putchar('\n');
  }

  if (printResults && resultarray) {
    printf("    Should match:\n\t");
    for (i = 0, k = 0; i < hashsize; i++, k += 2) {
      putchar(resultarray[k]);
      putchar(resultarray[k+1]);
      putchar(' ');
    }
    putchar('\n');
  }

  if (printPassFail && resultarray) {
    int ret = checkmatch(Message_Digest, resultarray, hashsize);
    if ((printPassFail == PRINTPASSFAIL) || !ret)
      printf("%s %s %s: %s\n", hashname, testtype, testname,
        ret ? "PASSED" : "FAILED");
  }
}

/*
 * Exercise a hash series of functions. The input is the testarray,
 * repeated repeatcount times, followed by the extrabits. If the
 * result is known, it is in resultarray in uppercase hex.
 */
int hash(int testno, int loopno, int hashno,
  const char *testarray, int length, long repeatcount,
  int numberExtrabits, int extrabits, const unsigned char *keyarray,
  int keylen, const char *resultarray, int hashsize, int printResults,
  int printPassFail)
{
  USHAContext sha;
  HMACContext hmac;
  int err, i;
  uint8_t Message_Digest[USHAMaxHashSize];
  char buf[20];

  if (printResults == PRINTTEXT) {
    printf("\nTest %d: Iteration %d, Repeat %ld\n\t'", testno+1,
      loopno, repeatcount);
    printstr(testarray, length);
    printf("'\n\t'");
    printxstr(testarray, length);
    printf("'\n");
```

```
    printf("    Length=%d bytes (%d bits), ", length, length * 8);
    printf("ExtraBits %d: %2.2x\n", numberExtrabits, extrabits);
  }

  memset(&sha, '\343', sizeof(sha)); /* force bad data into struct */
  memset(&hmac, '\343', sizeof(hmac));
  err = keyarray ? hmacReset(&hmac, hashes[hashno].whichSha,
                             keyarray, keylen) :
                   USHAReset(&sha, hashes[hashno].whichSha);
  if (err != shaSuccess) {
    fprintf(stderr, "hash(): %sReset Error %d.\n",
            keyarray ? "hmac" : "sha", err);
    return err;
  }

  for (i = 0; i < repeatcount; ++i) {
    err = keyarray ? hmacInput(&hmac, (const uint8_t *) testarray,
                               length) :
                     USHAInput(&sha, (const uint8_t *) testarray,
                               length);
    if (err != shaSuccess) {
      fprintf(stderr, "hash(): %sInput Error %d.\n",
              keyarray ? "hmac" : "sha", err);
      return err;
    }
  }

  if (numberExtrabits > 0) {
    err = keyarray ? hmacFinalBits(&hmac, (uint8_t) extrabits,
                                   numberExtrabits) :
                     USHAFinalBits(&sha, (uint8_t) extrabits,
                                   numberExtrabits);
    if (err != shaSuccess) {
      fprintf(stderr, "hash(): %sFinalBits Error %d.\n",
              keyarray ? "hmac" : "sha", err);
      return err;
    }
  }

  err = keyarray ? hmacResult(&hmac, Message_Digest) :
                   USHAResult(&sha, Message_Digest);
  if (err != shaSuccess) {
    fprintf(stderr, "hash(): %s Result Error %d, could not "
      "compute message digest.\n", keyarray ? "hmac" : "sha", err);
    return err;
  }

  sprintf(buf, "%d", testno+1);
```

```
  printResult(Message_Digest, hashsize, hashes[hashno].name,
    keyarray ? "hmac standard test" : "sha standard test", buf,
    resultarray, printResults, printPassFail);

  return err;
}

/*
 * Exercise a hash series of functions. The input is a filename.
 * If the result is known, it is in resultarray in uppercase hex.
 */
int hashfile(int hashno, const char *hashfilename, int bits,
  int bitcount, int skipSpaces, const unsigned char *keyarray,
  int keylen, const char *resultarray, int hashsize,
  int printResults, int printPassFail)
{
  USHAContext sha;
  HMACContext hmac;
  int err, nread, c;
  unsigned char buf[4096];
  uint8_t Message_Digest[USHAMaxHashSize];
  unsigned char cc;
  FILE *hashfp = (strcmp(hashfilename, "-") == 0) ? stdin :
    fopen(hashfilename, "r");

  if (!hashfp) {
    fprintf(stderr, "cannot open file '%s'\n", hashfilename);
    return shaStateError;
  }

  memset(&sha, '\343', sizeof(sha)); /* force bad data into struct */
  memset(&hmac, '\343', sizeof(hmac));
  err = keyarray ? hmacReset(&hmac, hashes[hashno].whichSha,
                             keyarray, keylen) :
                   USHAReset(&sha, hashes[hashno].whichSha);

  if (err != shaSuccess) {
    fprintf(stderr, "hashfile(): %sReset Error %d.\n",
            keyarray ? "hmac" : "sha", err);
    return err;
  }

  if (skipSpaces)
    while ((c = getc(hashfp)) != EOF) {
      if (!isspace(c)) {
        cc = (unsigned char)c;
        err = keyarray ? hmacInput(&hmac, &cc, 1) :
                         USHAInput(&sha, &cc, 1);
```

```
        if (err != shaSuccess) {
          fprintf(stderr, "hashfile(): %sInput Error %d.\n",
                  keyarray ? "hmac" : "sha", err);
          if (hashfp != stdin) fclose(hashfp);
          return err;
        }
      }
    }
  else
    while ((nread = fread(buf, 1, sizeof(buf), hashfp)) > 0) {
      err = keyarray ? hmacInput(&hmac, buf, nread) :
                       USHAInput(&sha, buf, nread);
      if (err != shaSuccess) {
        fprintf(stderr, "hashfile(): %s Error %d.\n",
                keyarray ? "hmacInput" : "shaInput", err);
        if (hashfp != stdin) fclose(hashfp);
        return err;
      }
    }

  if (bitcount > 0)
    err = keyarray ? hmacFinalBits(&hmac, bits, bitcount) :
                     USHAFinalBits(&sha, bits, bitcount);
  if (err != shaSuccess) {
    fprintf(stderr, "hashfile(): %s Error %d.\n",
            keyarray ? "hmacResult" : "shaResult", err);
    if (hashfp != stdin) fclose(hashfp);
    return err;
  }

  err = keyarray ? hmacResult(&hmac, Message_Digest) :
                   USHAResult(&sha, Message_Digest);
  if (err != shaSuccess) {
    fprintf(stderr, "hashfile(): %s Error %d.\n",
            keyarray ? "hmacResult" : "shaResult", err);
    if (hashfp != stdin) fclose(hashfp);
    return err;
  }

  printResult(Message_Digest, hashsize, hashes[hashno].name, "file",
    hashfilename, resultarray, printResults, printPassFail);

  if (hashfp != stdin) fclose(hashfp);
  return err;
}

/*
 * Exercise a hash series of functions through multiple permutations.
```

```
 * The input is an initial seed. That seed is replicated 3 times.
 * For 1000 rounds, the previous three results are used as the input.
 * This result is then checked, and used to seed the next cycle.
 * If the result is known, it is in resultarrays in uppercase hex.
 */
void randomtest(int hashno, const char *seed, int hashsize,
    const char **resultarrays, int randomcount,
    int printResults, int printPassFail)
{
  int i, j; char buf[20];
  unsigned char SEED[USHAMaxHashSize], MD[1003][USHAMaxHashSize];

  /* INPUT: Seed - A random seed n bits long */
  memcpy(SEED, seed, hashsize);
  if (printResults == PRINTTEXT) {
    printf("%s random test seed= '", hashes[hashno].name);
    printxstr(seed, hashsize);
    printf("'\n");
  }

  for (j = 0; j < randomcount; j++) {
    /* MD0 = MD1 = MD2 = Seed; */
    memcpy(MD[0], SEED, hashsize);
    memcpy(MD[1], SEED, hashsize);
    memcpy(MD[2], SEED, hashsize);
    for (i=3; i<1003; i++) {
      /* Mi = MDi-3 || MDi-2 || MDi-1; */
      USHAContext Mi;
      memset(&Mi, '\343', sizeof(Mi)); /* force bad data into struct */
      USHAReset(&Mi, hashes[hashno].whichSha);
      USHAInput(&Mi, MD[i-3], hashsize);
      USHAInput(&Mi, MD[i-2], hashsize);
      USHAInput(&Mi, MD[i-1], hashsize);
      /* MDi = SHA(Mi); */
      USHAResult(&Mi, MD[i]);
    }

    /* MDj = Seed = MDi; */
    memcpy(SEED, MD[i-1], hashsize);

    /* OUTPUT: MDj */
    sprintf(buf, "%d", j);
    printResult(SEED, hashsize, hashes[hashno].name, "random test",
      buf, resultarrays ? resultarrays[j] : 0, printResults,
      (j < RANDOMCOUNT) ? printPassFail : 0);
  }
}
```

```
/*
 * Look up a hash name.
 */
int findhash(const char *argv0, const char *opt)
{
  int i;
  const char *names[HASHCOUNT][2] = {
    { "0", "sha1" }, { "1", "sha224" }, { "2", "sha256" },
    { "3", "sha384" }, { "4", "sha512" }
  };

  for (i = 0; i < HASHCOUNT; i++)
    if ((strcmp(opt, names[i][0]) == 0) ||
        (scasecmp(opt, names[i][1]) == 0))
      return i;

  fprintf(stderr, "%s: Unknown hash name: '%s'\n", argv0, opt);
  usage(argv0);
  return 0;
}

/*
 * Run some tests that should invoke errors.
 */
void testErrors(int hashnolow, int hashnohigh, int printResults,
    int printPassFail)
{
  USHAContext usha;
  uint8_t Message_Digest[USHAMaxHashSize];
  int hashno, err;

  for (hashno = hashnolow; hashno <= hashnohigh; hashno++) {
    memset(&usha, '\343', sizeof(usha)); /* force bad data */
    USHAReset(&usha, hashno);
    USHAResult(&usha, Message_Digest);
    err = USHAInput(&usha, (const unsigned char *)"foo", 3);
    if (printResults == PRINTTEXT)
      printf ("\nError %d. Should be %d.\n", err, shaStateError);
    if ((printPassFail == PRINTPASSFAIL) ||
        ((printPassFail == PRINTFAIL) && (err != shaStateError)))
      printf("%s se: %s\n", hashes[hashno].name,
        (err == shaStateError) ? "PASSED" : "FAILED");

    err = USHAFinalBits(&usha, 0x80, 3);
    if (printResults == PRINTTEXT)
      printf ("\nError %d. Should be %d.\n", err, shaStateError);
    if ((printPassFail == PRINTPASSFAIL) ||
        ((printPassFail == PRINTFAIL) && (err != shaStateError)))
```

```
      printf("%s se: %s\n", hashes[hashno].name,
        (err == shaStateError) ? "PASSED" : "FAILED");

    err = USHAReset(0, hashes[hashno].whichSha);
    if (printResults == PRINTTEXT)
       printf("\nError %d. Should be %d.\n", err, shaNull);
    if ((printPassFail == PRINTPASSFAIL) ||
        ((printPassFail == PRINTFAIL) && (err != shaNull)))
       printf("%s usha null: %s\n", hashes[hashno].name,
        (err == shaNull) ? "PASSED" : "FAILED");

    switch (hashno) {
      case SHA1: err = SHA1Reset(0); break;
      case SHA224: err = SHA224Reset(0); break;
      case SHA256: err = SHA256Reset(0); break;
      case SHA384: err = SHA384Reset(0); break;
      case SHA512: err = SHA512Reset(0); break;
    }
    if (printResults == PRINTTEXT)
       printf("\nError %d. Should be %d.\n", err, shaNull);
    if ((printPassFail == PRINTPASSFAIL) ||
        ((printPassFail == PRINTFAIL) && (err != shaNull)))
       printf("%s sha null: %s\n", hashes[hashno].name,
        (err == shaNull) ? "PASSED" : "FAILED");
  }
}

/* replace a hex string in place with its value */
int unhexStr(char *hexstr)
{
  char *o = hexstr;
  int len = 0, nibble1 = 0, nibble2 = 0;
  if (!hexstr) return 0;
  for ( ; *hexstr; hexstr++) {
    if (isalpha((int)(unsigned char)(*hexstr))) {
      nibble1 = tolower(*hexstr) - 'a' + 10;
    } else if (isdigit((int)(unsigned char)(*hexstr))) {
      nibble1 = *hexstr - '0';
    } else {
      printf("\nError: bad hex character '%c'\n", *hexstr);
    }
    if (!*++hexstr) break;
    if (isalpha((int)(unsigned char)(*hexstr))) {
      nibble2 = tolower(*hexstr) - 'a' + 10;
    } else if (isdigit((int)(unsigned char)(*hexstr))) {
      nibble2 = *hexstr - '0';
    } else {
      printf("\nError: bad hex character '%c'\n", *hexstr);
```

```
    }
    *o++ = (char)((nibble1 << 4) | nibble2);
    len++;
  }
  return len;
}

int main(int argc, char **argv)
{
  int i, err;
  int loopno, loopnohigh = 1;
  int hashno, hashnolow = 0, hashnohigh = HASHCOUNT - 1;
  int testno, testnolow = 0, testnohigh;
  int ntestnohigh = 0;
  int printResults = PRINTTEXT;
  int printPassFail = 1;
  int checkErrors = 0;
  char *hashstr = 0;
  int hashlen = 0;
  const char *resultstr = 0;
  char *randomseedstr = 0;
  int runHmacTests = 0;
  char *hmacKey = 0;
  int hmaclen = 0;
  int randomcount = RANDOMCOUNT;
  const char *hashfilename = 0;
  const char *hashFilename = 0;
  int extrabits = 0, numberExtrabits = 0;
  int strIsHex = 0;

  while ((i = xgetopt(argc, argv, "b:B:ef:F:h:Hk:l:mpPr:R:s:S:t:wxX"))
         != -1)
    switch (i) {
      case 'b': extrabits = strtol(xoptarg, 0, 0); break;
      case 'B': numberExtrabits = atoi(xoptarg); break;
      case 'e': checkErrors = 1; break;
      case 'f': hashfilename = xoptarg; break;
      case 'F': hashFilename = xoptarg; break;
      case 'h': hashnolow = hashnohigh = findhash(argv[0], xoptarg);
        break;
      case 'H': strIsHex = 1; break;
      case 'k': hmacKey = xoptarg; hmaclen = strlen(xoptarg); break;
      case 'l': loopnohigh = atoi(xoptarg); break;
      case 'm': runHmacTests = 1; break;
      case 'P': printPassFail = 0; break;
      case 'p': printResults = PRINTNONE; break;
      case 'R': randomcount = atoi(xoptarg); break;
      case 'r': randomseedstr = xoptarg; break;
```

```
      case 's': hashstr = xoptarg; hashlen = strlen(hashstr); break;
      case 'S': resultstr = xoptarg; break;
      case 't': testnolow = ntestnohigh = atoi(xoptarg) - 1; break;
      case 'w': printResults = PRINTRAW; break;
      case 'x': printResults = PRINTHEX; break;
      case 'X': printPassFail = 2; break;
      default: usage(argv[0]);
      }

  if (strIsHex) {
    hashlen = unhexStr(hashstr);
    unhexStr(randomseedstr);
    hmaclen = unhexStr(hmacKey);
  }
  testnohigh = (ntestnohigh != 0) ? ntestnohigh:
               runHmacTests ? (HMACTESTCOUNT-1) : (TESTCOUNT-1);
  if ((testnolow < 0) ||
      (testnohigh >= (runHmacTests ? HMACTESTCOUNT : TESTCOUNT)) ||
      (hashnolow < 0) || (hashnohigh >= HASHCOUNT) ||
      (hashstr && (testnolow == testnohigh)) ||
      (randomcount < 0) ||
      (resultstr && (!hashstr && !hashfilename && !hashFilename)) ||
      ((runHmacTests || hmacKey) && randomseedstr) ||
      (hashfilename && hashFilename))
    usage(argv[0]);

  /*
   *  Perform SHA/HMAC tests
   */
  for (hashno = hashnolow; hashno <= hashnohigh; ++hashno) {
    if (printResults == PRINTTEXT)
      printf("Hash %s\n", hashes[hashno].name);
    err = shaSuccess;

    for (loopno = 1; (loopno <= loopnohigh) && (err == shaSuccess);
         ++loopno) {
      if (hashstr)
        err = hash(0, loopno, hashno, hashstr, hashlen, 1,
          numberExtrabits, extrabits, (const unsigned char *)hmacKey,
          hmaclen, resultstr, hashes[hashno].hashsize, printResults,
          printPassFail);

      else if (randomseedstr)
        randomtest(hashno, randomseedstr, hashes[hashno].hashsize, 0,
          randomcount, printResults, printPassFail);

      else if (hashfilename)
        err = hashfile(hashno, hashfilename, extrabits,
```

```
                    numberExtrabits, 0,
                    (const unsigned char *)hmacKey, hmaclen,
                    resultstr, hashes[hashno].hashsize,
                    printResults, printPassFail);

      else if (hashFilename)
        err = hashfile(hashno, hashFilename, extrabits,
                    numberExtrabits, 1,
                    (const unsigned char *)hmacKey, hmaclen,
                    resultstr, hashes[hashno].hashsize,
                    printResults, printPassFail);

      else /* standard tests */ {
        for (testno = testnolow;
             (testno <= testnohigh) && (err == shaSuccess); ++testno) {
          if (runHmacTests) {
            err = hash(testno, loopno, hashno,
                    hmachashes[testno].dataarray[hashno] ?
                    hmachashes[testno].dataarray[hashno] :
                    hmachashes[testno].dataarray[1] ?
                    hmachashes[testno].dataarray[1] :
                    hmachashes[testno].dataarray[0],
                    hmachashes[testno].datalength[hashno] ?
                    hmachashes[testno].datalength[hashno] :
                    hmachashes[testno].datalength[1] ?
                    hmachashes[testno].datalength[1] :
                    hmachashes[testno].datalength[0],
                    1, 0, 0,
                    (const unsigned char *)(
                     hmachashes[testno].keyarray[hashno] ?
                     hmachashes[testno].keyarray[hashno] :
                     hmachashes[testno].keyarray[1] ?
                     hmachashes[testno].keyarray[1] :
                     hmachashes[testno].keyarray[0]),
                    hmachashes[testno].keylength[hashno] ?
                    hmachashes[testno].keylength[hashno] :
                    hmachashes[testno].keylength[1] ?
                    hmachashes[testno].keylength[1] :
                    hmachashes[testno].keylength[0],
                    hmachashes[testno].resultarray[hashno],
                    hmachashes[testno].resultlength[hashno],
                    printResults, printPassFail);
          } else {
            err = hash(testno, loopno, hashno,
                    hashes[hashno].tests[testno].testarray,
                    hashes[hashno].tests[testno].length,
                    hashes[hashno].tests[testno].repeatcount,
                    hashes[hashno].tests[testno].numberExtrabits,
```

```
                        hashes[hashno].tests[testno].extrabits, 0, 0,
                        hashes[hashno].tests[testno].resultarray,
                        hashes[hashno].hashsize,
                        printResults, printPassFail);
          }
        }

        if (!runHmacTests) {
          randomtest(hashno, hashes[hashno].randomtest,
            hashes[hashno].hashsize, hashes[hashno].randomresults,
            RANDOMCOUNT, printResults, printPassFail);
        }
      }
    }
  }

  /* Test some error returns */
  if (checkErrors) {
    testErrors(hashnolow, hashnohigh, printResults, printPassFail);
  }

  return 0;
}

/*
 * Compare two strings, case independently.
 * Equivalent to strcasecmp() found on some systems.
 */
int scasecmp(const char *s1, const char *s2)
{
  for (;;) {
    char u1 = tolower(*s1++);
    char u2 = tolower(*s2++);
    if (u1 != u2)
      return u1 - u2;
    if (u1 == '\0')
      return 0;
   }
}

/*
 * This is a copy of getopt provided for those systems that do not
 * have it. The name was changed to xgetopt to not conflict on those
 * systems that do have it. Similarly, optarg, optind and opterr
 * were renamed to xoptarg, xoptind and xopterr.
 *
 * Copyright 1990, 1991, 1992 by the Massachusetts Institute of
 * Technology and UniSoft Group Limited.
```

```
 *
 * Permission to use, copy, modify, distribute, and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear in
 * supporting documentation, and that the names of MIT and UniSoft not
 * be used in advertising or publicity pertaining to distribution of
 * the software without specific, written prior permission.  MIT and
 * UniSoft make no representations about the suitability of this
 * software for any purpose.  It is provided "as is" without express
 * or implied warranty.
 *
 * $XConsortium: getopt.c,v 1.2 92/07/01 11:59:04 rws Exp $
 * NB: Reformatted to match above style.
 */

char    *xoptarg;
int     xoptind = 1;
int     xopterr = 1;

static int xgetopt(int argc, char **argv, const char *optstring)
{
  static int avplace;
  char    *ap;
  char    *cp;
  int     c;

  if (xoptind >= argc)
    return EOF;

  ap = argv[xoptind] + avplace;

  /* At beginning of arg but not an option */
  if (avplace == 0) {
    if (ap[0] != '-')
      return EOF;
    else if (ap[1] == '-') {
      /* Special end of options option */
      xoptind++;
      return EOF;
    } else if (ap[1] == '\0')
      return EOF;  /* single '-' is not allowed */
  }

  /* Get next letter */
  avplace++;
  c = *++ap;
```

```
  cp = strchr(optstring, c);
  if (cp == NULL || c == ':') {
    if (xopterr)
      fprintf(stderr, "Unrecognised option -- %c\n", c);
    return '?';
  }

  if (cp[1] == ':') {
    /* There should be an option arg */
    avplace = 0;
    if (ap[1] == '\0') {
      /* It is a separate arg */
      if (++xoptind >= argc) {
        if (xopterr)
          fprintf(stderr, "Option requires an argument\n");
        return '?';
      }
      xoptarg = argv[xoptind++];
    } else {
      /* is attached to option letter */
      xoptarg = ap + 1;
      ++xoptind;
    }
  } else {
    /* If we are out of letters then go to next arg */
    if (ap[1] == '\0') {
      ++xoptind;
      avplace = 0;
    }

    xoptarg = NULL;
  }
  return c;
}
```

9. Security Considerations

This document is intended to provides the Internet community convenient access to source code that implements the United States of America Federal Information Processing Standard Secure Hash Algorithms (SHAs) [FIPS180-2] and HMACs based upon these one-way hash functions. See license in Section 1.1. No independent assertion of the security of this hash function by the authors for any particular use is intended.

10. Normative References

[FIPS180-2] "Secure Hash Standard", United States of America, National Institute of Standards and Technology, Federal Information Processing Standard (FIPS) 180-2, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

11. Informative References

[RFC2202] Cheng, P. and R. Glenn, "Test Cases for HMAC-MD5 and HMAC-SHA-1", RFC 2202, September 1997.

[RFC3174] Eastlake 3rd, D. and P. Jones, "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, September 2001.

[RFC3874] Housley, R., "A 224-bit One-way Hash Function: SHA-224", RFC 3874, September 2004.

[RFC4086] Eastlake, D., 3rd, Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[RFC4231] Nystrom, M., "Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512", RFC 4231, December 2005.

[SHAVS] "The Secure Hash Algorithm Validation System (SHAVS)", http://csrc.nist.gov/cryptval/shs/SHAVS.pdf.

Authors' Addresses

Donald E. Eastlake, 3rd
Motorola Laboratories
155 Beaver Street
Milford, MA 01757 USA

Phone: +1-508-786-7554 (w)
EMail: donald.eastlake@motorola.com

Tony Hansen
AT&T Laboratories
200 Laurel Ave.
Middletown, NJ 07748 USA

Phone: +1-732-420-8934 (w)
EMail: tony+shs@maillennium.att.com

Full Copyright Statement

Copyright (C) The Internet Society (2006).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Acknowledgement

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).