July 12, 2002 at 15:05

1. Introduction. What follows is an implementation of the SHA-256 cryptographic message digest algorithm as defined in (still draft) FIPS PUB 180-2. It actually works, and I've put in the unit tests that came with the spec — define the preprocessor symbol `UNIT_TEST` to enable them.

This was largely an experiment on my part to see how easy and useful it is to "program literately", and I tell you — it really is a lot of fun.

But is it really useful? I mean, if you were going to write 120,000 lines of code, would it work? I imagine that Knuth thinks so, since he did TEX that way.

Let me mess with it some more. Enjoy this literary feast (well, OK, literary snack), and when you're done snacking, go ahead and run it. Kinda like eating the shell your taco salad comes in — it's a bowl as well as a snack.

— Blake Ramsdell, July 2002
Brute Squad Labs, Inc.

2. In order to digest data with SHA-256, you simply call *sha_256_init* followed by one or more calls to *sha_256_update* and then ultimately call *sha_256_final*. The resulting digest is stored in the byte array member *final_digest* of the *sha_256_context* structure.

```
#include <memory.h>
  ⟨ Type definitions 4 ⟩
  ⟨ Global constants 9 ⟩
  ⟨ Underlying functions 16 ⟩
  void sha_256_init(sha_256_context * context)
  {
    ⟨ Set H to the initial value, H^(0) 23 ⟩
    ⟨ Initialize the current message block, M^(0) 24 ⟩
  }
  void sha_256_update(sha_256_context * context, uint32 data_length, byte * data)
  {
    while (data_length > 0) {
      uint32 bytes_to_copy = min(remaining_bytes_in_block, data_length);
      ⟨ Append data to the current message block, M^(i) 32 ⟩
      ⟨ Update hash if required 33 ⟩
      data_length −= bytes_to_copy;
      data += bytes_to_copy;
    }
  }
  void sha_256_final(sha_256_context * context)
  {
    uint32 total_data_processed_bits = context→total_data_processed_bytes * 8;
    byte temp_buffer[sizeof (context→M)];
    ⟨ Append padding to the end of the message 35 ⟩
    ⟨ Append length to the end of the message 36 ⟩
    ⟨ Expand the final digest 37 ⟩
  }
```

3. Type definitions, macros and constants.

4. For SHA-256, the number of bits in a word, w, is 32. We will use the uint32 datatype for almost all variables.

⟨ Type definitions 4 ⟩ ≡
typedef unsigned int uint32;

See also sections 5 and 10.

This code is used in section 2.

5. A byte datatype is useful also.

⟨ Type definitions 4 ⟩ +≡
typedef unsigned char byte;

6. The *rotr* macro is used for rotating a **uint32**, X, N bits to the right.

#define $rotr(X,N)$ $((X \gg N) \mid (X \ll (32 - N)))$

7. Good ol' *min*. Nothing beats that. Except maybe *max*, but he's not here.

#define $min(X,Y)$ $(((X) < (Y)) \mathbin{?} (X) : (Y))$

8. *remaining_bytes_in_block* figures out the number of bytes remaining in the current message block, $M^{(i)}$.

#define *remaining_bytes_in_block* (**sizeof** (*context*→*M*) − *context*→*current_block_length_bytes*)

9. The array K corresponds to the sequence $K^{\{256\}}$ defined in FIPS 180-2 §4.2.2. According to that section, "These words represent the first thirty-two bits of the fractional parts of the cube roots of the first sixty four prime numbers. . ."

⟨ Global constants 9 ⟩ ≡
static uint32 $K[64]$ = {#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};

This code is used in section 2.

10. We will use a context object to retain information in between calls to *sha_256_update*.

⟨ Type definitions 4 ⟩ +≡
typedef struct {
⟨ Context data 11 ⟩
} **sha_256_context**;

11. The last hash value, $H^{(i-1)}$

⟨ Context data 11 ⟩ ≡
uint32 $H[8]$;

See also sections 12, 13, and 14.

This code is used in section 10.

12. The current block of the message, $M^{(i)}$, m bits long. $m = 512$ for SHA-256.

⟨ Context data 11 ⟩ +≡
  byte *M*[512/8]; /* Blocks are 512 bits long */
  uint32 *current_block_length_bytes*;

13. The total number of bytes in the message so far, $(l * 8)$, since l is in bits.

⟨ Context data 11 ⟩ +≡
  uint32 *total_data_processed_bytes*;

14. The final digest value, $H^{(N)}$

⟨ Context data 11 ⟩ +≡
  byte *final_digest*[32];

15. Underlying functions. There are several primitive functions for SHA-256.

16. The function Ch as defined in FIPS 180-2 §4.1.2.

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

⟨ Underlying functions 16 ⟩ ≡
```
uint32 Ch(uint32 x, uint32 y, uint32 z)
{
  return (x & y) ⊕ (∼x & z);
}
```
See also sections 17, 18, 19, 20, and 21.
This code is used in section 2.

17. The function Maj as defined in FIPS 180-2 §4.1.2.

$$\mathrm{Maj}(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

⟨ Underlying functions 16 ⟩ +≡
```
uint32 Maj(uint32 x, uint32 y, uint32 z)
{
  return (x & y) ⊕ (x & z) ⊕ (y & z);
}
```

18. The function *sigma0* corresponds to the function $\sigma_0^{\{256\}}$ in FIPS 180-2 §4.1.2.

$$\sigma_0^{\{256\}}(x) = (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3)$$

⟨ Underlying functions 16 ⟩ +≡
```
uint32 sigma0(uint32 x)
{
  return rotr(x, 7) ⊕ rotr(x, 18) ⊕ (x ≫ 3);
}
```

19. The function *sigma1* corresponds to the function $\sigma_1^{\{256\}}$ in FIPS 180-2 §4.1.2.

$$\sigma_1^{\{256\}}(x) = (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10)$$

⟨ Underlying functions 16 ⟩ +≡
```
uint32 sigma1(uint32 x)
{
  return rotr(x, 17) ⊕ rotr(x, 19) ⊕ (x ≫ 10);
}
```

20. The function *Sigma0* corresponds to the function $\Sigma_0^{\{256\}}$ in FIPS 180-2 §4.1.2.

$$\Sigma_0^{\{256\}}(x) = (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22)$$

⟨ Underlying functions 16 ⟩ +≡
```
uint32 Sigma0(uint32 x)
{
  return rotr(x, 2) ⊕ rotr(x, 13) ⊕ rotr(x, 22);
}
```

21. The function *Sigma1* corresponds to the function $\Sigma_1^{\{256\}}$ in FIPS 180-2 §4.1.2.

$$\Sigma_1^{\{256\}}(x) = (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25)$$

$\langle$ Underlying functions 16 $\rangle$ $+\equiv$

```
uint32 Sigma1(uint32 x)
{
  return rotr(x, 6) ⊕ rotr(x, 11) ⊕ rotr(x, 25);
}
```

22. Preprocessing. Prepares the **sha_256_context** structure for first use.

23. The current hash value, H, is set to $H^{(0)}$ per the values in FIPS 180-2 §5.3.2. According to that section, "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."

⟨ Set H to the initial value, $H^{(0)}$ 23 ⟩ ≡
context→$H[0]$ = #`6a09e667`;
context→$H[1]$ = #`bb67ae85`;
context→$H[2]$ = #`3c6ef372`;
context→$H[3]$ = #`a54ff53a`;
context→$H[4]$ = #`510e527f`;
context→$H[5]$ = #`9b05688c`;
context→$H[6]$ = #`1f83d9ab`;
context→$H[7]$ = #`5be0cd19`;

This code is used in section 2.

24. The current block size and the total number of bytes processed are zeroed.

⟨ Initialize the current message block, $M^{(0)}$ 24 ⟩ ≡
context→*current_block_length_bytes* = 0;
context→*total_data_processed_bytes* = 0;

This code is used in section 2.

25. Hash computation. The code in here is for implementing the methods of FIPS 180-2 §6.2 in order to compute the current hash value, $H^{(i)}$ for the current message block, $M^{(i)}$.

26. The general strategy is as follows

⟨ Compute the intermediate hash value, $H^{(i)}$ 26 ⟩ ≡

```
{
  uint32 a, b, c, d, e, f, g, h;
  uint32 t;
  uint32 T1, T2;
  uint32 W[64];
  ⟨ Initialize working variables from H^(i−1) 27 ⟩
  for (t = 0; t < 64; ++t) {
    ⟨ Compute the message schedule, W_t 28 ⟩
    ⟨ Compression function 29 ⟩
  }
  ⟨ Copy the intermediate hash value, H^(i) 30 ⟩
  context→current_block_length_bytes = 0;
}
```

This code is used in section 33.

27. FIPS 180-2 §6.2.2 specifies in step 2 the initialization of eight working variables $a, b, \ldots, h$ to the current value of H (which is the hash value for the previous block, and thus represents $H^{(i-1)}$.)

I believe that there is an inconsistency in the specification since the prose reads "Initialize…with the $(i-1)$ hash value" and the assignments that follow use components of $H^{(i)}$.

⟨ Initialize working variables from $H^{(i-1)}$ 27 ⟩ ≡

```
a = context→H[0];
b = context→H[1];
c = context→H[2];
d = context→H[3];
e = context→H[4];
f = context→H[5];
g = context→H[6];
h = context→H[7];
```

This code is used in section 26.

28. FIPS 180-2 §6.2.2 specifies in step 1 to compute W_t.

$$W_t \leftarrow \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

⟨ Compute the message schedule, W_t 28 ⟩ ≡

```
if (t ≤ 15) {
  W[t] = (context→M[t * 4] ≪ 24) | (context→M[(t * 4) + 1] ≪ 16) | (context→M[(t * 4) + 2] ≪ 8) |
      (context→M[(t * 4) + 3]);
}
else {
  W[t] = sigma1(W[t − 2]) + W[t − 7] + sigma0(W[t − 15]) + W[t − 16];
}
```

This code is used in section 26.

29. The compression function, as specified in FIPS 180-2 §6.2.2 step 3.

$\langle$ Compression function 29 $\rangle \equiv$
$\mathtt{T1} = h + \mathit{Sigma1}(e) + \mathit{Ch}(e, f, g) + K[t] + W[t];$
$\mathtt{T2} = \mathit{Sigma0}(a) + \mathit{Maj}(a, b, c);$
$h = g;$
$g = f;$
$f = e;$
$e = d + \mathtt{T1};$
$d = c;$
$c = b;$
$b = a;$
$a = \mathtt{T1} + \mathtt{T2};$

This code is used in section 26.

30. Finally we assign the current intermediate hash value, $H^{(i)}$ to H in the context, per FIPS 180-2 §6.2.2 step 3.

$\langle$ Copy the intermediate hash value, $H^{(i)}$ 30 $\rangle \equiv$
$\mathit{context}{\rightarrow}H[0] \mathrel{+}= a;$
$\mathit{context}{\rightarrow}H[1] \mathrel{+}= b;$
$\mathit{context}{\rightarrow}H[2] \mathrel{+}= c;$
$\mathit{context}{\rightarrow}H[3] \mathrel{+}= d;$
$\mathit{context}{\rightarrow}H[4] \mathrel{+}= e;$
$\mathit{context}{\rightarrow}H[5] \mathrel{+}= f;$
$\mathit{context}{\rightarrow}H[6] \mathrel{+}= g;$
$\mathit{context}{\rightarrow}H[7] \mathrel{+}= h;$

This code is used in section 26.

31. Message block processing.

32. Append some data to the message block, possibly filling the block.

⟨ Append data to the current message block, $M^{(i)}$ 32 ⟩ ≡

$memcpy(context\rightarrow M + context\rightarrow current\_block\_length\_bytes, data, bytes\_to\_copy);$
$context\rightarrow current\_block\_length\_bytes \mathrel{+}= bytes\_to\_copy;$
$context\rightarrow total\_data\_processed\_bytes \mathrel{+}= bytes\_to\_copy;$

This code is used in section 2.

33. In the event that the current block is full, then compute the intermediate hash value, $H^{(i)}$.

⟨ Update hash if required 33 ⟩ ≡

if $(remaining\_bytes\_in\_block \equiv 0)$ {
  ⟨ Compute the intermediate hash value, $H^{(i)}$ 26 ⟩
}

This code is used in section 2.

34. Final block processing. We need to jam a single 1 bit, followed by some number of 0 bits followed by 64 bits of the length (in bits) of the data that has been digested (from FIPS 180-2 §5.1.1). Finally, we make a byte array copy of the final digest value, $H^{(N)}$.

35. The padding length is computed to leave enough space for the eight byte length, and then added to the message.

⟨ Append padding to the end of the message 35 ⟩ ≡

```
temp_buffer[0] = #080;    /* Our one bit plus seven zero bits */
sha_256_update(context, 1, temp_buffer);
memset(temp_buffer, 0, sizeof (temp_buffer));
if (remaining_bytes_in_block < 8) {    /* Fill up this block */
  sha_256_update(context, remaining_bytes_in_block, temp_buffer);
}
sha_256_update(context, remaining_bytes_in_block − 8, temp_buffer);
```

This code is used in section 2.

36. The eight byte length is then appended to the message.

⟨ Append length to the end of the message 36 ⟩ ≡

```
temp_buffer[4] = (total_data_processed_bits ≫ 24) & #0FF;
temp_buffer[5] = (total_data_processed_bits ≫ 16) & #0FF;
temp_buffer[6] = (total_data_processed_bits ≫ 8) & #0FF;
temp_buffer[7] = (total_data_processed_bits) & #0FF;
sha_256_update(context, 8, temp_buffer);
```

This code is used in section 2.

37. Make a copy of the final digest block, $H^{(N)}$, converted to a byte array.

⟨ Expand the final digest 37 ⟩ ≡

```
{
  int counter;
  for (counter = 0; counter < (sizeof (context→H)/sizeof (context→H[0])); ++counter) {
    context→final_digest[(counter ∗ 4) + 0] = ((context→H[counter] ≫ 24) & #0ff);
    context→final_digest[(counter ∗ 4) + 1] = ((context→H[counter] ≫ 16) & #0ff);
    context→final_digest[(counter ∗ 4) + 2] = ((context→H[counter] ≫ 8) & #0ff);
    context→final_digest[(counter ∗ 4) + 3] = ((context→H[counter]) & #0ff);
  }
}
```

This code is used in section 2.

38. Unit tests. The following section implements the unit tests for this module. The unit tests will only be included in the event that the C preprocessor symbol `UNIT_TEST` is # **define**d.

```
#ifdef UNIT_TEST
  ⟨ Unit tests 39 ⟩
  int main(int argc, char **argv)
  {
    ⟨ Run unit tests 41 ⟩
  }
#endif
```

39. I'm going to define a function that makes it easy to check to see if a particular digest result matches.

```
⟨ Unit tests 39 ⟩ ≡
  void assert_byte_arrays_equal(char *test_name, byte *expected_value, int expected_value_length, byte
        *actual_value, int actual_value_length)
  {
    printf("%s␣%s\n", ((expected_value_length ≠ actual_value_length) ∨ (memcmp(expected_value,
        actual_value, actual_value_length) ≠ 0)) ? "FAIL" : "PASS", test_name);
  }
```

See also sections 40, 42, and 44.

This code is used in section 38.

40. FIPS 180-2 §B.1, hashing the string `"abc"`

```
⟨ Unit tests 39 ⟩ +≡
  void test_B1()
  {
    static byte expected_value[] = {#ba, #78, #16, #bf, #8f, #01, #cf, #ea, #41, #41, #40, #de, #5d, #ae,
        #22, #23, #b0, #03, #61, #a3, #96, #17, #7a, #9c, #b4, #10, #ff, #61, #f2, #00, #15, #ad};
    sha_256_context context;

    sha_256_init(&context);
    sha_256_update(&context, 3, "abc");
    sha_256_final(&context);
    assert_byte_arrays_equal("test_B1", expected_value, sizeof (expected_value), context.final_digest, sizeof
        (context.final_digest));
  }
```

41. ⟨ Run unit tests 41 ⟩ ≡

```
  test_B1();
```

See also sections 43 and 45.

This code is used in section 38.

42. FIPS 180-2 §B.2, hashing the string `"abcdbc..."`

⟨ Unit tests 39 ⟩ +≡

```
void test_B2()
{
  static byte expected_value[] = {#24, #8d, #6a, #61, #d2, #06, #38, #b8, #e5, #c0, #26, #93, #0c, #3e,
      #60, #39, #a3, #3c, #e4, #59, #64, #ff, #21, #67, #f6, #ec, #ed, #d4, #19, #db, #06, #c1};
  sha_256_context context;

  sha_256_init(&context);
  sha_256_update(&context, 56, "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq");
  sha_256_final(&context);
  assert_byte_arrays_equal("test_B2", expected_value, sizeof (expected_value), context.final_digest, sizeof
      (context.final_digest));
}
```

43. ⟨ Run unit tests 41 ⟩ +≡

```
test_B2();
```

44. FIPS 180-2 §B.3, hashing the byte `'a'` 1,000,000 times (1,000 calls to *sha_256_update* with 1,000 `'a'`s apiece.)

⟨ Unit tests 39 ⟩ +≡

```
void test_B3()
{
  static byte expected_value[] = {#cd, #c7, #6e, #5c, #99, #14, #fb, #92, #81, #a1, #c7, #e2, #84, #d7,
      #3e, #67, #f1, #80, #9a, #48, #a4, #97, #20, #0e, #04, #6d, #39, #cc, #c7, #11, #2c, #d0};
  sha_256_context context;
  byte data_block[1000];
  int counter = 0;

  memset(data_block, 'a', sizeof (data_block));
  sha_256_init(&context);
  for (counter = 0; counter < 1000; ++counter) {
    sha_256_update(&context, sizeof (data_block), data_block);
  }
  sha_256_final(&context);
  assert_byte_arrays_equal("test_B3", expected_value, sizeof (expected_value), context.final_digest, sizeof
      (context.final_digest));
}
```

45. ⟨ Run unit tests 41 ⟩ +≡

```
test_B3();
```

46. Index. CWEAVE likes to make it, so why should I complain? Hopefully you find what you're looking for in it.

The SHA-256 algorithm

	Section	Page