Internet Engineering Task Force (IETF)
Request for Comments: 6234
Obsoletes: 4634
Updates: 3174
Category: Informational
ISSN: 2070-1721

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

May 2011

US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)

Безопасные алгоритмы хэширования США (SHA, HMAC и HKDF на основе SHA)

Аннотация

В США внедрён набор алгоритмов безопасного хэширования (Secure Hash Algorithm или SHA), в том числе 4 алгоритма сверх SHA-1 (SHA-224, SHA-256, SHA-384, SHA-512), в рамках Федерального стандарта обработки информации (Federal Information Processing Standard или FIPS). В этом документе приведён исходный код функций SHA, доступный для сообщества Internet. Пример кода поддерживает входные строки произвольного размера. Большая часть представленного здесь текста заимствована авторами из FIPS 180-2.

Этот документ заменяет RFC 4634, исправляя ошибки и добавляя код функции получения ключа (Key Derivation Function) на основе HMAC (HKDF, RFC 5869). Как и в RFC 4634, включён код для хэшированного кода аутентификации сообщений (Hashed Message Authentication Code или HMAC) на основе SHA.

Статус документа

Документ не относится к категории Internet Standards Track и публикуется с информационными целями.

Документ является результатом работы IETF[1] и представляет согласованный взгляд сообщества IETF. Документ прошёл открытое обсуждение и был одобрен для публикации IESG[2]. Не все документы, одобренные IESG, претендуют на статус стандартов Internet, см. раздел 2 в RFC 5741.

Информацию о текущем статусе документа, ошибках и способах обратной связи можно найти по ссылке http://www.rfc-editor.org/info/rfc6234.

Авторские права

Copyright (c) 2011. Авторские права принадлежат IETF Trust и лицам, указанным в качестве авторов документа. Все права защищены.

К документу применимы права и ограничения, указанные в BCP 78 и IETF Trust Legal Provisions и относящиеся к документам IETF (http://trustee.ietf.org/license-info), на момент публикации данного документа. Прочтите упомянутые документы внимательно, поскольку они могут описывать ваши права и ограничения применительно к этому документу. Компоненты кода, извлекаемые из этого документа, должны включать текст Simplified BSD License, как указано в параграфе 4.e Trust Legal Provisions и предоставляться без каких-либо гарантий, как указано в Simplified BSD License.

Оглавление

[1] Internet Engineering Task Force - комиссия по решению инженерных задач Internet.
[2] Internet Engineering Steering Group - комиссия по инженерным разработкам Internet.

1. Обзор содержимого

Этот документ содержит спецификации алгоритмов безопасного хэширования (SHA) в соответствии с Федеральным стандартом обработки информации (FIPS) США, код для реализации SHA, HMAC [RFC2104]) на основе SHA и HKDF на основе HMAC [RFC5869]. Спецификации HMAC и HKDF не включены, поскольку они представлены в других RFC [RFC2104] [RFC5869].

Примечание. Значительная часть текста заимствована из [SHS], заявления о безопасности описанных алгоритмов хэширования сделаны правительством США и автором [SHS], а не авторами этого документа. Вопросы безопасности SHA-1 рассмотрены в [RFC6194].

Приведённый ниже текст задаёт алгоритмы SHA-224 [RFC3874], SHA-256, SHA-384 и SHA-512 для расчёта свёртки сообщений или файлов данных (SHA-1 задан в [RFC3174]). При подаче на вход описанных алгоритмов сообщения произвольного размера не более 2^64 (для SHA-224 и SHA-256) или 2^128 битов (для SHA-384 и SHA-512) результатом обработки будет дайджест сообщения. Размеры дайджестов составляют от 224 до 512 битов в зависимости от алгоритма. Алгоритмы SHA обычно применяются с криптографическими алгоритмами, такими как алгоритмы цифровой подписи и алгоритмы хэшированных кодов аутентификации с ключом, а также для генерации случайных чисел [RFC4086] и функций вывода ключей.

Заданные в документе алгоритмы считаются безопасными потому, что вычислительными средствами невозможно (1) найти сообщение, соответствующее любому данному дайджесту и (2) найти два разных сообщения с одинаковыми дайджестами. Любое изменение сообщения в процессе передачи с очень высокой вероятностью приведёт к изменению дайджеста. Это приведёт к отрицательному результату проверки с помощью алгоритма SHA, применяемого с алгоритмом цифровой подписи или алгоритмом аутентификации сообщения с хэшированием и ключом.

Представленный здесь код поддерживает на входе строки с произвольным числом битов. Пример кода SHA-1 из [RFC3174] обновлён для работы со входными строками произвольной длины. Предоставляется право на любое использование кода (коммерческое или некоммерческое).

Документ отменяет действие [RFC4634], а отличия от него кратко перечислены в Приложении.

Ниже приведены идентификаторы объектов ASN.1 OID для алгоритмов SHA, взятые из [RFC4055].

```
id-sha1  OBJECT IDENTIFIER  ::=  { iso(1)
                     identified-organization(3) oiw(14)
                     secsig(3) algorithms(2) 26 }
 id-sha224  OBJECT IDENTIFIER  ::=  {{ joint-iso-itu-t(2)
                     country(16) us(840) organization(1) gov(101)
                     csor(3) nistalgorithm(4) hashalgs(2) 4 }
 id-sha256  OBJECT IDENTIFIER  ::=  { joint-iso-itu-t(2)
                     country(16) us(840) organization(1) gov(101)
                     csor(3) nistalgorithm(4) hashalgs(2) 1 }
 id-sha384  OBJECT IDENTIFIER  ::=  { joint-iso-itu-t(2)
                     country(16) us(840) organization(1) gov(101)
                     csor(3) nistalgorithm(4) hashalgs(2) 2 }
 id-sha512  OBJECT IDENTIFIER  ::=  { joint-iso-itu-t(2)
                     country(16) us(840) organization(1) gov(101)
                     csor(3) nistalgorithm(4) hashalgs(2) 3 }
```

В разделе 2 определены термины и функции, используемые в качестве базовых элементов для формирования алгоритмов. В разделе 3 описаны базовые операции со словами, на основе которых построены алгоритмы. В разделе 4 описано, как сообщения дополняются до размера, кратного требуемому размеру блока, а затем разбиваются на блоки. В разделе 5 определены константы и составные функции, используемые для алгоритмов хэширования. В разделе 6 приведены фактические спецификации функций SHA-224, SHA-256, SHA-384 и SHA-512. В разделе 7 приведены ссылки на спецификации кодов аутентификации сообщений HMAC и функции вывода ключей на основе HMAC. Раздел 8 содержит примеры кода для алгоритмов SHA, HMAC на основе SHA и функции вывода ключей на основе HMAC.

2. Обозначения битовых строк и целых чисел

Ниже приведены обозначения, используемые для представления битовых строк и целых чисел.

a) Шестнадцатеричными цифрами являются элементы множества {0, 1, ... , 9, A, ... , F}. Шестнадцатеричная цифра представляется строкой из 4 битов, например, 7 = 0111, A = 1010.

b) Слово эквивалентно 32- или 64-битовой строке, которую можно представить последовательностью из 8 или 16 шестнадцатеричных цифр, соответственно. Для преобразования слова в шестнадцатеричные цифры каждая строка из 4 битов преобразуется в шестнадцатеричную цифру, как указано в п. (a). Например,

```
1010 0001 0000 0011 1111 1110 0010 0011 = A103FE23
```

В этом документе для преобразования 32- и 64-битовых слов применяется порядок big-endian, где старший бит каждого слова размещается в самой левой позиции.

c) Целое число может быть представлено словом или парой слов.

Целые числа от 0 до 2^32-1 (включительно) можно представить 32-битовым словом. Четыре младших бита такого числа представляются самой правой шестнадцатеричной цифрой в слове. Например, целое число 291 = 2^8+2^5+2^1+2^0 = 256+32+2+1 представляется шестнадцатеричным словом 00000123.

Такой же подход применяется для чисел от 0 до 2^64-1 включительно, которые можно представить 64-битовым словом. Если Z - целое число и 0 <= z < 2^64, то z = (2^32)x + y, где 0 <= x < 2^32 и 0 <= y < 2^32. Поскольку x и y можно представить словами X и Y, соответственно, z можно представить парой слов (X,Y). Здесь также применяется порядок битов big-endian и старшее слово размещается в самой левой позиции.

d) block - это строка из 512 или 1024 битов. Блок (например, B) можно представить как последовательность 32- или 64-битовых слов.

3. Операции со словами

Ниже описаны побитовые операции, применяемые к словам во всех описанных здесь операциях хэширования. SHA-224 и SHA-256 работают с 32-битовыми словами, а SHA-384 и SHA-512 - с 64-битовыми.

В операции x<<n сначала отбрасываются n битов x слева, затем добавляется n нулевых битов справа (в результате число битов не меняется). В операции x>>n отбрасывается n битов x справа и добавляется n нулевых битов слева.

a) Побитовые логические операции над словами:

X AND Y = побитовая логическая операция И для X и Y

X OR Y = побитовая логическая операция ИЛИ для X и Y

X XOR Y = побитовая логическая операция Исключающее-ИЛИ для X и Y

NOT X = побитовое логическое «дополнение» X.

Например,

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

b) Операция сложения X + Y определяется следующим образом: слова X и Y представляются w-битовыми целыми числами x и y (0 <= x < 2^w и 0 <= y < 2^w). Для положительных целых чисел n и m значение (n mod m) является остатком от деления n на m. Рассчитывается z = (x + y) mod 2^w, где 0 <= z < 2^w. Затем z преобразуется в слово Z и определяется Z = X + Y.

c) Сдвиг вправо SHR^n(x), где x - w-битовое слово, а n - целое число (0 <= n < w) определяется как x>>n

d) Кольцевой сдвиг вправо ROTR^n(x), где x - w-битовое слово, а n - целое число (0 <= n < w), определяется как

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

e) Кольцевой сдвиг влево ROTL^n(x), где x - w-битовое слово, а n - целое число (0 <= n < w), определяется как

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

Отметим показанные ниже отношения эквивалентности при фиксированном значении w.

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

4. Дополнение и разбор сообщений

Описанные здесь хэш-функции служат для расчёта дайджестов сообщений или файлов данных, поданных на вход. Входное сообщение или файл следует считать строкой битов, число битов которой считается размером сообщения или файла (пустое сообщение имеет размер 0). Если число битов кратно 8, сообщение представляется для краткости в шестнадцатеричном формате. Для приведения сообщения к размеру, кратному 512 (SHA-224 и SHA-256) или 1024 (SHA-384 и SHA-512), применяется дополнение. При дополнении в конец сообщения подставляется бит со значением 1, затем m нулей (0) и 64- или 128-битовое целое число, указывающее размер исходного сообщения. Дополненное сообщение (512*n или 1024*n) обрабатывается функцией хэширования как n блоков по 512 или 1024 бита.

4.1. SHA-224 и SHA-256

Предположим, что сообщение имеет размер L < 2^64. Перед подачей сообщения на вход функции хэширования оно дополняется справа, как показано ниже.

a) Добавляется бит со значением 1. Например, сообщение 01010000 после дополнения станет 010100001.

b) Затем добавляется K битов со значением 0, где K - наименьшее неотрицательное решение уравнения

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

c) Затем добавляется 64-битовый блок с двоичным представлением L, после чего размер сообщения становится кратным 512.

Предположим, что исходное сообщение имело вид

```
01100001 01100010 01100011 01100100 01100101
```

После п. (a) оно принимает вид

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

Поскольку L = 40, число битов после предыдущего шага становится 41 и добавляется K = 407 битов со значением 0, что даёт размер 448 и сообщение принимает вид

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

```
00000000 00000000 00000000 00000000
00000000 00000000
```

В конец сообщения добавляется 64-битовое представление числа L = 40 (00000000 00000028) и сообщение принимает вид

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

4.2. SHA-384 и SHA-512

Предположим, что сообщение имеет размер L < 2^128. Перед подачей сообщения на вход функции хэширования оно дополняется справа, как показано ниже.

a) Добавляется бит со значением 1. Например, сообщение 01010000 после дополнения станет 01010000 1.

b) Затем добавляется K битов со значением 0, где K - наименьшее неотрицательное решение уравнения

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

c) Затем добавляется 128-битовый блок с двоичным представлением L, после чего размер сообщения становится кратным 1024.

Предположим, что исходное сообщение имело вид

```
01100001 01100010 01100011 01100100 01100101
```

После п. (a) оно принимает вид

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

Поскольку L = 40, число битов после предыдущего шага становится 41 и добавляется K = 855 битов со значением 0, что даёт размер 896 и сообщение принимает вид

```
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
```

В заключение в конец сообщения добавляется 128-битовое представление числа L = 40 (00000000 00000000 00000000 00000028) и сообщение принимает вид

```
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. Используемые функции и константы

В последующих параграфах описываются шесть логических функций и таблица используемых ими констант.

5.1. SHA-224 и SHA-256

В SHA-224 и SHA-256 используется шесть логических функций, каждая из которых оперирует 32-битовыми словами x, y, z, возвращая в результате новое 32-битовое слово.

```
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 и SHA-256 используют один набор из 64 констант в форме 32-битовых слов K0, K1, ..., K63. Эти слова представляют первые 32 бита дробных частей кубического корня из первых 64 простых чисел. Шестнадцатеричное представление слов приведено ниже (слева направо).

```
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 и SHA-512

В SHA-384 и SHA-512 используется шесть логических функций, каждая из которых оперирует 64-битовыми словами x, y, z, возвращая в результате новое 64-битовое слово.

```
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 и SHA-512 используют один набор из 80 констант в форме 64-битовых слов K0, K1, ... K79. Эти слова представляют первые 64 бита дробных частей кубического корня из первых 80 простых чисел. Шестнадцатеричное представление слов приведено ниже (слева направо).

```
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. Расчёт дайджеста сообщения

Выходом каждой из безопасных хэш-функций после применения к сообщению из N блоков является хэш-значение H(N). Для SHA-224 и SHA-256 можно рассматривать H(i) как восемь 32-битовых слов H(i)0, H(i)1, ... H(i)7, для SHA-384 и SHA-512 - как восемь 64-битовых слов H(i)0, H(i)1, ..., H(i)7.

Как описано ниже, хэш-слова инициализируются, изменяются при обработке каждого блока сообщения, а после обработки последнего блока выполняется конкатенация. В SHA-256 и SHA-512 объединяются все переменные H(N), а в SHA-224 и SHA-384 некоторые переменные исключаются при финальной конкатенации.

6.1. Инициализация SHA-224 и SHA-256

Для SHA-224 начальное хэш-значение H(0) состоит из показанных ниже 32-битовых слов, которые были получены взятием второго 32-битового слова дробной части квадратного корня из простых чисел с девятого по шестнадцатое.[1]

```
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
```

Для SHA-256 начальное хэш-значение H(0) состоит из показанных ниже 32-битовых слов, которые были получены взятием первых 32 битов дробной части квадратного корня из первых 8 простых чисел.

```
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. Обработка SHA-224 и SHA-256

В SHA-224 и SHA-256 выполняется идентичная обработка блоков сообщений и отличаются лишь инициализация H(0) и создание финального результата. Функции могут служить для хэширования сообщений M размером L битов, где 0 <= L < 2^64. Алгоритм использует (1) планы (schedule) сообщения из 64 слов по 32 бита (W0, W1, ..., W63), (2) восемь рабочих переменных по 32 бита (a, b, c, d, e, f, g, h) и (3) хэш-значение из 8 слов по 32 бита (H(i)0, H(i)1, ..., H(i)7). Хэш-значение после обработки каждого блока сообщения заменяется промежуточным результатом H(i), а после обработки всех N блоков устанавливается значение H(N). Применяется также два временных слова T1 и T2.

Входное сообщение дополняется, как описано в параграфе 4.1, и разбивается на блоки по 512 битов, которые считаются состоящими из 16 слов по 32 бита (M(i)0, M(i)1, ..., M(i)15). Над этими словами выполняются указанные ниже операции (по модулю 2^32) для i от 1 до N.

1. Подготовка плана сообщения W

[1] В оригинале не был указан источник значений. См. https://errata.rfc-editor.org/eid5240/. *Прим. перев.*

```
For t = 0 to 15
   Wt = M(i)t
For t = 16 to 63
   Wt = SSIG1(W(t-2)) + W(t-7) + SSIG0(w(t-15)) + W(t-16)
```

2. Инициализация рабочих переменных

```
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. Расчёт основного хэш-значения

```
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. Расчёт промежуточного хэш-значения 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
```

После выполнения указанных выше операций для всех блоков сообщения вычисляется финальный результат. Для SHA-256 это конкатенация значений H(N)0, H(N)1, ... H(N)7, для SHA-224 - конкатенация H(N)0, H(N)1, ... H(N)6.

6.3. Инициализация SHA-384 и SHA-512

Для SHA-384 начальное хэш-значение H(0) состоит из показанных ниже 64-битовых слов, которые были получены взятием первых 64 битов дробной части квадратного корня из простых чисел с девятого по шестнадцатое.

```
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
```

Для SHA-512 начальное хэш-значение H(0) состоит из показанных ниже 64-битовых слов, которые были получены взятием первых 64 битов дробной части квадратного корня из первых 8 простых чисел.

```
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. Обработка SHA-384 и SHA-512

В SHA-384 и SHA- 512 выполняется идентичная обработка блоков сообщений и отличается лишь инициализация H(0) и создание финального результата. Функции могут служить для хэширования сообщений M размером L битов, где 0 <= L < 2^128. Алгоритм использует (1) планы (schedule) сообщения из 64-битовых слов (W0, W1, ..., W79), (2) восемь рабочих переменных по 64 бита (a, b, c, d, e, f, g, h) и (3) хэш-значение из 8 слов по 64 бита (H(i)0, H(i)1, ..., H(i)7). Хэш-значение после обработки каждого блока сообщения заменяется промежуточным результатом H(i), а после обработки всех N блоков устанавливается значение H(N).

Входное сообщение дополняется, как описано в параграфе 4.2 и разбивается на блоки по 1024 бита, которые считаются состоящими из 16 слов по 64 бита (M(i)0, M(i)1, ..., M(i)15). Над этими словами выполняются указанные ниже операции (по модулю 2^64) для i от 1 до N.

1. Подготовка плана сообщения W

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

2. Инициализация рабочих переменных

```
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. Расчёт основного хэш-значения

```
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. Расчёт промежуточного хэш-значения 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
```

После выполнения указанных выше операций для всех блоков сообщения вычисляется финальный результат. Для SHA- 512 это конкатенация значений H(N)0, H(N)1, ... H(N)7, для SHA- 384 - конкатенация H(N)0, H(N)1, ... H(N)6.

7. HKDF и HMAC на основе SHA

Ниже приведены краткие описания и ссылки на более подробные описания и код для (1) HMAC на основе SHA и (2) функции вывода ключей на основе HMAC. HKDF и HMAC разработаны Hugo Krawczyk.

7.1. HMAC на основе SHA

Метод HMAC служит для расчёта кода аутентификации сообщения (MAC или Message Authentication Code) с ключом с помощью хэш-функции, как описано в [RFC2104]. Ключ смешивается с входными данными для создания финального хэша.

В параграфе 8.3 приведён пример кода для расчёта HMAC на основе любого из описанных здесь алгоритмов SHA. Пример кода в [RFC2104] был написан для заданного размера текста. Поскольку SHA определены для произвольного числа битов, пример кода HMAC также был создан для обработки произвольного числа октетов и битов. Предусмотрен также интерфейс для фиксированного размера.

7.2. HKDF

HKDF - это функция получения ключа (Key Derivation Function или KDF), создающая по исходному ключевому материалу один или несколько криптостойких секретных ключей. HKDF, описанная в [RFC5869], основана на HMAC.

Пример кода HKDF приведён в параграфе 8.4.

8. Код C для SHA, HMAC и HKDF

Ниже приведены примеры реализации защищённых хэш-функций на языке C. В параграфе 8.1 представлен заголовочный файл sha.h, где объявлены все константы, структуры и функции, используемые функциями SHA и HMAC. Файл включает условные операторы на основе состояния определения USE_32BIT_ONLY для исключения 64-битовых операций, когда это требуется. Параграф также содержит файл sha-private.h с некоторыми объявлениями, общими для всех функций SHA. В параграфе 8.2 приведён код C для sha1.c, sha224-256.c, sha384-512.c, usha.c. В параграфе 8.3 представлен код C для функций HMAC, а в параграфе 8.4 - для HKDF. В параграфе 8.5 приведён тестовый драйвер для проверки кода.

Для каждого размера дайджеста $$$ имеется набор указанных ниже констант, структур и функций.

Константы

SHA$$$HashSize
: Число октетов в хэш-значении.

SHA$$$HashSizeBits
: Число битов в хэш-значении.

SHA$$$_Message_Block_Size
: Число октетов в промежуточных блоках сообщения.

Большинство функций возвращает одно из приведённых ниже перечисляемых значений (enum)

shaSuccess(0)
: Успешное выполнение.

shaNull(1)
: Представлен параметр с пустым (null) указателем.

shaInputTooLong(2)
: Слишком длинные данные на входе.

shaStateError(3)
Вызов SHA$$$Input после SHA$$$FinalBits или SHA$$$Result.

Структура

typedef SHA$$$Context
Неанализируемая структура, содержащая полное состояние для создания хэш-значения.

Функции

int SHA$$$Reset(SHA$$$Context *context);
Сброс состояния контекста хэша.

int SHA$$$Input(SHA$$$Context *context, const uint8_t *octets, unsigned int bytecount);
Встраивание bytecount октетов в хэш.

int SHA$$$FinalBits(SHA$$$Context *, const uint8_t octet, unsigned int bitcount);
Встраивание bitcount битов в хэш (в верхнюю часть октета). После этой функции нельзя вызывать SHA$$$Input().

int SHA$$$Result(SHA$$$Context *, uint8_t Message_Digest[SHA$$$HashSize]);
Финальный расчёт хэш-значения и копирование его в Message_Digest.

Функции с префиксом USHA принимают значение SHAversion (SHA$$$) для выбора набора функций SHA. Ниже перечислены константы, структуры и функции для них.

Константы

shaBadParam(4)
Значение, возвращаемое функциями USHA при неверном значении SHAversion (SHA$$$) или ином недопустимом параметре.

USAMaxHashSize
Максимальный размер хэша SHA.

SHA$$$
Перечисляемые значения SHAversion, используемые функциями USHA, HMAC, HKDF для выбора набора функций SHA.

Константы

typedef USHAContext
Неанализируемая структура, содержащая полное состояние для создания хэша.

Функции

int USHAReset(USHAContext *context, SHAversion whichSha);
Сброс состояния контекста хэша.

int USHAInput(USHAContext context*, const uint8_t *bytes, unsigned int bytecount);
Встраивание bytecount октетов в хэш.

int USHAFinalBits(USHAContext *context, const uint8_t bits, unsigned int bitcount);
Встраивание bitcount битов в хэш.

int USHAResult(USHAContext *context, uint8_t Message_Digest[USHAMaxHashSize]);
Финальный расчёт хэш-значения и копирование его в Message_Digest. Октеты Message_Digest за пределами USHAHashSize(whichSha) не затрагиваются.

int USHAHashSize(enum SHAversion whichSha);
Число октетов в данном хэше.

int USHAHashSizeBits(enum SHAversion whichSha);
Число битов в данном хэше.

int USHABlockSize(enum SHAversion whichSha);
Внутренний размер блока для данного хэша.

const char *USHAHashName(enum SHAversion whichSha);
Возвращает имя данного алгоритма SHA в форме строки.

Указанные ниже функции HMAC следуют той же схеме и принимают на входе текст любого размера.

Структура

typedef HMACContext
Неанализируемая структура, содержащая полное состояние для создания дайджеста сообщения с ключом (MAC).

Функции

int hmacReset(HMACContext *ctx, enum SHAversion whichSha, const unsigned char *key, int key_len);
Сброс состояния контекста MAC.

int hmacInput(HMACContext *ctx, const unsigned char *text, int text_len);
Встраивание text_len октетов в MAC.

int hmacFinalBits(HMACContext *ctx, const uint8_t bits, unsigned int bitcount);
Встраивание bitcount битов в MAC.

int hmacResult(HMACContext *ctx, uint8_t Message_Digest[USHAMaxHashSize]);
Финальный расчёт MAC и копирование его в Message_Digest. Октеты Message_Digest за пределами USHAHashSize(whichSha) не затрагиваются.

Обеспечивается также комбинированный интерфейс, похожий на показанный в [RFC2104], который позволяет обрабатывать входные данные фиксированного размера.

int hmac(SHAversion whichSha, const unsigned char *text, int text_len, const unsigned char *key, int key_len, uint8_t Message_Digest[USHAMaxHashSize]);
Расчёт дайджеста для данного текста и ключа с возвратом полученного кода MAC. Октеты Message_Digest за пределами USHAHashSize(whichSha) не затрагиваются.

Указанные ниже функции HKDF следуют той же схеме и принимают на входе текст любого размера.

Структура

typedef HKDFContext
Неанализируемая структура, содержащая полное состояние для создания ключевого материала.

Функции

int hkdfReset(HKDFContext *context, enum SHAversion whichSha, const unsigned char *salt, int salt_len)
Сброс состояния вывода ключей и его инициализация с использованием salt_len октетов необязательного salt.

int hkdfInput(HKDFContext *context, const unsigned char *ikm, int ikm_len)
Встраивание ikm_len октетов в средство извлечения энтропии.

int hkdfFinalBits(HKDFContext *context, uint8_t ikm_bits, unsigned int ikm_bit_count)
Встраивание ikm_bit_count битов в средство извлечения энтропии.

int hkdfResult(HKDFContext *context, uint8_t prk[USHAMaxHashSize], const unsigned char *info, int info_len, uint8_t okm[ ], int okm_len)
Завершение извлечения HKDF и финальное расширение HKDF с сохранением okm_len октетов в выходной ключевой материал (okm). Может также сохраняться псевдослучайный ключ (prk) генерируемый внутри.

Кроме того, предоставляются комбинированные интерфейсы, похожие на приведённый в [RFC5869], которые позволяют принимать на входе текст фиксированного размера.

int hkdfExtract(SHAversion whichSha, const unsigned char *salt, int salt_len, const unsigned char *ikm, int ikm_len, uint8_t prk[USHAMaxHashSize])
Извлечение HKDF путём объединения salt_len октетов необязательного параметра salt с ikm_len октетов входного ключевого материала (ikm) для формирования псевдослучайного ключа prk. Размер prk должен быть достаточным для данного типа хэша.

int hkdfExpand(SHAversion whichSha, const uint8_t prk[ ], int prk_len, const unsigned char *info, int info_len, uint8_t okm[ ], int okm_len)
Выполняет расширение HKDF, объединяя prk_len октетов псевдослучайного ключа prk с info_len октетов info для формирования okm_len октетов, сохраняеиых в okm.

int hkdf(SHAversion whichSha, const unsigned char *salt, int salt_len, const unsigned char *ikm, int ikm_len, const unsigned char *info, int info_len, uint8_t okm[ ], int okm_len)
Этот комбинированный интерфейс выполняет извлечение и расширение HKDF, используя те же переменные, что и функции hkdfExtract() и hkdfExpand().

8.1. Заголовочные файлы

8.1.1. Файл sha.h

Файл sha.h file предполагает наличие в системе заголовочного файла <stdint.h>. Если это не так, следует включить файл <stdint-example.h> из параграфа 8.1.2 или что-то подобное.

```
/***************************** sha.h *****************************/
/***************** См. детали в RFC 6234. ***********************/
/*
   Авторские права (Copyright (c) 2011) принадлежат IETF Trust и
   лицам, указанным как авторы кода. Все права защищены.

   Распространение и использование в исходной или двоичной форме
   (с изменениями или без таковых) разрешено при условии:

   - В исходном коде сохранено указание авторских прав, данный
     список условий и заявление об отказе от ответственности.

   - При распространении в двоичной форме должно воспроизводиться
     указание авторских прав, данный список условий и заявление об
     отказе от ответственности в документации или иных материалах.

   - Ни Internet Society, ни IETF, ни IETF Trust, ни конкретные
     участники работы не могут быть использованы для продвижения
     продукции, созданной на основе этого кода без предварительного
     письменного разрешения.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
   CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
   DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
   CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
   NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
   LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
   HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
   OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
   EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/
#ifndef _SHA_H_
#define _SHA_H_

/*
 *  Описание:
 *      Эта функция реализует алгоритмы безопасного хэширования
 *      в соответствии со стандартом FIPS PUB 180-3, опубликованным
 *      в октябре 2008 г. и его предшественниками FIPS PUB 180-1
 *      и FIP PUB 180-2.
 *
 *      Объединённый документ с описанием всех алгоритмов
 *   http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf
 *
```

```
 *      Определены 5 хэшей разного размера
 *              SHA-1           20 байтов / 160 битов
 *              SHA-224         28 байтов / 224 бита
 *              SHA-256         32 байта / 256 битов
 *              SHA-384         48 байтов / 384 бита
 *              SHA-512         64 байта / 512 битов
 *
 *  Примечания по компиляции
 *    Файлы можно компилировать с двумя опциями:
 *        USE_32BIT_ONLY - использовать только 32-битовую арифметику
 *                         в системах без 64-битовых целых чисел
 *
 *        USE_MODIFIED_MACROS - использовать другую форму макросов
 *                         SHA_Ch() и SHA_Maj(), которая на многих
 *                         системах может быть быстрее.
 */

#include <stdint.h>
/*
 * При отсутствии стандартного (ISO) заголовочного файла stdint.h
 * требуется задать приведённые ниже typedef.
 *  Имя              Назначение
 *  uint64_t         64-битовое целое число без знака
 *  uint32_t         32-битовое целое число без знака
 *  uint8_t          8-битовое целое число без знака (unsigned char)
 *  int_least16_t    целое число не менее (>=) 16 битов
 *
 * См. файл stdint-example.h
 */

#ifndef _SHA_enum_
#define _SHA_enum_
/*
 *  Все функции SHA возвращают одно из указанных ниже значений
 */
enum {
    shaSuccess = 0,
    shaNull,            /* Параметр с пустым (null) указателем */
    shaInputTooLong,    /* Слишком много входных данных */
    shaStateError,      /* Вызов Input после FinalBits или Result */
    shaBadParam         /* Передан непригодный параметр */
};
#endif /* _SHA_enum_ */

/*
 *  Константы для размеров в каждой операции хэширования SHA
 */
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
};

/*
 *  Константы для функций USHA (Unified SHA).
 */
typedef enum SHAversion {
    SHA1, SHA224, SHA256, SHA384, SHA512
} SHAversion;

/*
 *  Структура данных контекста для операции хэширования SHA-1
 */
typedef struct SHA1Context {
    uint32_t Intermediate_Hash[SHA1HashSize/4]; /* Дайджест сообщения */

    uint32_t Length_High;               /* Размер сообщения в битах */
    uint32_t Length_Low;                /* Размер сообщения в битах */

    int_least16_t Message_Block_Index;  /* Индекс массива Message_Block */
                                        /* 512-битовых блоков сообщения */
    uint8_t Message_Block[SHA1_Message_Block_Size];

    int Computed;                   /* Расчёт хеша? */
    int Corrupted;                  /* Кумулятивный код повреждения */
} SHA1Context;
```

```
/* Структура с данными контекста для операций хэширования SHA-256. */
typedef struct SHA256Context {
    uint32_t Intermediate_Hash[SHA256HashSize/4]; /* Дайджест сообщения */

    uint32_t Length_High;               /* Размер сообщения в битах */
    uint32_t Length_Low;                /* Размер сообщения в битах */

    int_least16_t Message_Block_Index;  /* Индекс массива Message_Block */
                                        /* 512-битовых блоков сообщения */
    uint8_t Message_Block[SHA256_Message_Block_Size];

    int Computed;                   /* Хэш повреждён? */
    int Corrupted;                  /* Кумулятивный код повреждения */
} SHA256Context;

/* Структура с данными контекста для операций хэширования SHA-512. */
typedef struct SHA512Context {
#ifdef USE_32BIT_ONLY
    uint32_t Intermediate_Hash[SHA512HashSize/4]; /* Дайджест сообщения */
    uint32_t Length[4];                 /* Размер сообщения в битах */
#else /* !USE_32BIT_ONLY */
    uint64_t Intermediate_Hash[SHA512HashSize/8]; /* Дайджест сообщения */
    uint64_t Length_High, Length_Low;   /* Размер сообщения в битах */
#endif /* USE_32BIT_ONLY */

    int_least16_t Message_Block_Index;  /* Индекс массива Message_Block */
                                        /* 1024-битовых блоков сообщения */
    uint8_t Message_Block[SHA512_Message_Block_Size];

    int Computed;                   /* Расчёт хэша?*/
    int Corrupted;                  /* Кумулятивный код повреждения */
} SHA512Context;

/*
 *  Структура с данными контекста для операций хэширования SHA-224.
 *  Для расчётов применяется структура SHA-256.
 */
typedef struct SHA256Context SHA224Context;

/*
 *  Структура с данными контекста для операций хэширования SHA-384.
 *  Для расчётов применяется структура SHA-512.
 */
typedef struct SHA512Context SHA384Context;

/*
 *  Структура с данными контекста для всех операций хэширования SHA.
 */
typedef struct USHAContext {
    int whichSha;               /* Используемый вариант SHA */
    union {
      SHA1Context sha1Context;
      SHA224Context sha224Context; SHA256Context sha256Context;
      SHA384Context sha384Context; SHA512Context sha512Context;
    } ctx;
} USHAContext;

/* Структура с данными контекста для операций хэширования HMAC. */
typedef struct HMACContext {
    int whichSha;               /* Используемый вариант SHA */
    int hashSize;               /* Размер хэша применяемого SHA */
    int blockSize;              /* Размер блока применяемого SHA */
    USHAContext shaContext;     /* Контекст SHA */
    unsigned char k_opad[USHA_Max_Message_Block_Size];
                        /* Внешнее заполнение key XOR opad */
    int Computed;               /* Это расчёт MAC? */
    int Corrupted;              /* Кумулятивный код повреждения */

} HMACContext;

/* Структура с данными контекста для операций HKDF. */
typedef struct HKDFContext {
    int whichSha;               /* Используемый вариант SHA */
    HMACContext hmacContext;
    int hashSize;               /* Размер хэша применяемого SHA */
    unsigned char prk[USHAMaxHashSize];
                        /* Псевдослучайный ключ - вывод hkdfInput */
    int Computed;               /* Это расчёт ключевого материала? */
    int Corrupted;              /* Кумулятивный код повреждения */
} HKDFContext;

/* Прототипы функций */

/* SHA-1 */
extern int SHA1Reset(SHA1Context *);
extern int SHA1Input(SHA1Context *, const uint8_t *bytes,
```

```
                    unsigned int bytecount);
extern int SHA1FinalBits(SHA1Context *, uint8_t bits,
                         unsigned int bit_count);
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 *, uint8_t bits,
                           unsigned int bit_count);
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 *, uint8_t bits,
                           unsigned int bit_count);
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 *, uint8_t bits,
                           unsigned int bit_count);
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 *, uint8_t bits,
                           unsigned int bit_count);
extern int SHA512Result(SHA512Context *,
                        uint8_t Message_Digest[SHA512HashSize]);

/* Унифицированные функции SHA, определяемые whichSha */
extern int USHAReset(USHAContext *context, SHAversion whichSha);
extern int USHAInput(USHAContext *context,
                     const uint8_t *bytes, unsigned int bytecount);
extern int USHAFinalBits(USHAContext *context,
                         uint8_t bits, unsigned int bit_count);
extern int USHAResult(USHAContext *context,
                      uint8_t Message_Digest[USHAMaxHashSize]);
extern int USHABlockSize(enum SHAversion whichSha);
extern int USHAHashSize(enum SHAversion whichSha);
extern int USHAHashSizeBits(enum SHAversion whichSha);
extern const char *USHAHashName(enum SHAversion whichSha);

/*
 * HMAC для аутентификации сообщения (RFC 2104) со всеми SHA.
 * Этот интерфейс позволяет подавать текст фиксированного размера.
 */
extern int hmac(SHAversion whichSha, /* Используемый вариант SHA */
    const unsigned char *text,     /* Указатель на поток данных */
    int text_len,                  /* Размер потока данных */
    const unsigned char *key,      /* Указатель на ключ аутентификации */
    int key_len,                   /* Размер ключа аутентификации */
    uint8_t digest[USHAMaxHashSize]); /* Дайджест для заполнения */

/*
 * HMAC для аутентификации сообщения (RFC 2104) со всеми SHA.
 * Этот интерфейс позволяет подавать текст любого размера.
 */
extern int hmacReset(HMACContext *context, enum SHAversion whichSha,
                     const unsigned char *key, int key_len);
extern int hmacInput(HMACContext *context, const unsigned char *text,
                     int text_len);
extern int hmacFinalBits(HMACContext *context, uint8_t bits,
                         unsigned int bit_count);
extern int hmacResult(HMACContext *context,
                      uint8_t digest[USHAMaxHashSize]);

/*
 * Функция вывода ключей (HKDF) на основе HMAC (RFC 5869) для всех SHA.
 */
extern int hkdf(SHAversion whichSha, const unsigned char *salt,
                int salt_len, const unsigned char *ikm, int ikm_len,
                const unsigned char *info, int info_len,
                uint8_t okm[ ], int okm_len);
```

```
extern int hkdfExtract(SHAversion whichSha, const unsigned char *salt,
                       int salt_len, const unsigned char *ikm,
                       int ikm_len, uint8_t prk[USHAMaxHashSize]);
extern int hkdfExpand(SHAversion whichSha, const uint8_t prk[ ],
                      int prk_len, const unsigned char *info,
                      int info_len, uint8_t okm[ ], int okm_len);

/*
 * Функция вывода ключей (HKDF) на основе HMAC (RFC 5869) для всех SHA.
 * Этот интерфейс позволяет подавать текст любого размера.
 */
extern int hkdfReset(HKDFContext *context, enum SHAversion whichSha,
                     const unsigned char *salt, int salt_len);
extern int hkdfInput(HKDFContext *context, const unsigned char *ikm,
                     int ikm_len);
extern int hkdfFinalBits(HKDFContext *context, uint8_t ikm_bits,
                         unsigned int ikm_bit_count);
extern int hkdfResult(HKDFContext *context,
                      uint8_t prk[USHAMaxHashSize],
                      const unsigned char *info, int info_len,
                      uint8_t okm[USHAMaxHashSize], int okm_len);
#endif /* _SHA_H_ */
```

8.1.2. stdint-example.h

Если в системе нет файла <stdint.h>, приведённый ниже код послужит заменой при компиляции.

```
/********************** stdint-example.h **********************/
/**** Используется при отсутствии в системе файла stdint.h. ****/
/********************** См. RFC 6234. *************************/
#ifndef STDINT_H
#define STDINT_H

typedef unsigned long long uint64_t;    /* 64-битовое целое без знака */
typedef unsigned int       uint32_t;    /* 32-битовое целое без знака */
typedef unsigned char      uint8_t;     /* 8-битовое целое без знака */
typedef int int_least32_t;              /* целое с числом битов не менее 32 */
typedef short int_least16_t;            /* целое с числом битов не менее 16 */

#endif /* STDINT_H */
```

8.1.3. sha-private.h

Файл sha-private.h содержит определения, требуемые лишь для других файлов из этого документа. Он не потребуется для приложений, использующих представленный в этом документе код.

```
/********************** sha-private.h **********************/
/********************** См. RFC 6234. *************************/
#ifndef _SHA_PRIVATE__H
#define _SHA_PRIVATE__H
/*
 * Определения из параграфа 4.1 в FIPS 180-3, Ch() и Maj()
 * заданы в параграфах 4.1.1, 4.1.2, 4.1.3.
 *
 * Определения применяются в FIPS 180-3, как показано ниже.
 */

#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 */
/*
 * Приведённые ниже определения эквивалентны, но могут быть быстрее.
 */

#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 */
```

8.2. Код SHA

Этот код приведён в основном для пояснений и может быть оптимизирован. Например, можно рассматривать присвоение переменных a, b, ..., h как цикл, обходясь без явного копирования. Имеются другие представления функций Ch() и Maj() с использованием ifdef.

8.2.1. sha1.c

```
/*************************** sha1.c ***************************/
/***************** См. детали в RFC 6234. *********************/
/* Авторские права (Copyright (c) 2011) принадлежат IETF Trust */
/* и лицам, указанным как авторы кода. Все права защищены.     */
/* Распространение описано в файле sha.h.                      */
/*
```

```
 *  Описание
 *      Эта функция реализует алгоритмы безопасного хэширования
 *      в соответствии со стандартом FIPS PUB 180-3, опубликованным
 *      в октябре 2008 г., и его предшественниками FIPS PUB 180-1
 *      и FIP PUB 180-2.
 *
 *   Объединённый документ с описанием всех алгоритмов
 *   http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf
 *
 *      Алгоритм SHA-1 создаёт 160-битовый дайджест сообщения для входного
 *      потока данных. Дайджест может служить «оттиском» сообщения.
 *
 *  Переносимость
 *      SHA-1 определён с использованием 32-битовых слов. В этом коде
 *      используется файл <stdint.h> (включён в sha.h) для определения 32-
 *      и 8-битовых целых чисел без знака. Код не будет транслироваться,
 *      если компилятор C не поддерживает 32-битовые целые без знака.
 *
 *  Предостережения
 *      SHA-1 разработан для работы с сообщениями размером меньше 2^64
 *      битов. Эта реализация использует SHA1Input() для хэширования
 *      числа битов кратного 8 (октет) а затем может применять
 *      SHA1FinalBits() для хэширования оставшихся входных битов.
 */

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

/* Макрос циклического сдвига влево для SHA1 */
#define SHA1_ROTL(bits,word) \
                (((word) << (bits)) | ((word) >> (32-(bits))))

/*
 * Добавляет length к размеру, устанавливает Corrupted при переполнении.
 */
static uint32_t addTemp;
#define SHA1AddLength(context, length)                      \
    (addTemp = (context)->Length_Low,                       \
     (context)->Corrupted =                                 \
        (((context)->Length_Low += (length)) < addTemp) &&  \
        (++(context)->Length_High == 0) ? shaInputTooLong   \
                                        : (context)->Corrupted )

/* Прототипы локальных функций */
static void SHA1ProcessMessageBlock(SHA1Context *context);
static void SHA1Finalize(SHA1Context *context, uint8_t Pad_Byte);
static void SHA1PadMessage(SHA1Context *context, uint8_t Pad_Byte);

/*
 *  SHA1Reset
 *
 *  Описание
 *      Инициализация SHA1Context для подготовки к расчёту нового
 *      дайджеста SHA1.
 *
 *  Параметры
 *      context: [in/out] - контекст для сброса.
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */
int SHA1Reset(SHA1Context *context)
{
  if (!context) return shaNull;

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

  /* Исходные хэш-значения из параграфа 5ю3ю1 в FIPS 180-3 */
  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  = shaSuccess;

  return shaSuccess;
}

/*
 *  SHA1Input
 *
 *  Описание
 *      Функция принимает массив октетов как следующую часть сообщения.
```

```
 *
 *  Параметры
 *      context: [in/out] - обновляемый контекст SHA.
 *      message_array[ ]: [in]
 *          Массив октетов, представляющий следующую часть сообщения.
 *      length: [in] - размер сообщения в message_array.
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */
int SHA1Input(SHA1Context *context,
    const uint8_t *message_array, unsigned length)
{
  if (!context) return shaNull;
  if (!length) return shaSuccess;
  if (!message_array) return shaNull;
  if (context->Computed) return context->Corrupted = shaStateError;
  if (context->Corrupted) return context->Corrupted;

  while (length--) {
    context->Message_Block[context->Message_Block_Index++] =
      *message_array;

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

    message_array++;
  }

  return context->Corrupted;
}

/*
 * SHA1FinalBits
 *
 * Описание
 *   Добавляет все финальные биты из сообщения.
 *
 * Параметры
 *   context: [in/out] - обновляемый контекст SHA.
 *   message_bits: [in]
 *     Заключительные биты сообщения в старшей части байта (при
 *     3 битах ### следует использовать 0b###00000, а не 0b00000###).
 *   length: [in] - число битов в message_bits (1 - 7).
 *
 * Возвращаемое значение
 *     Код ошибки sha.
 */
int SHA1FinalBits(SHA1Context *context, uint8_t message_bits,
    unsigned int length)
{
  static 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
  };

  static 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 (!context) return shaNull;
  if (!length) return shaSuccess;
  if (context->Corrupted) return context->Corrupted;
  if (context->Computed) return context->Corrupted = shaStateError;
  if (length >= 8) return context->Corrupted = shaBadParam;

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

  return context->Corrupted;
}

/*
 * SHA1Result
 *
 * Описание
 *   Возвращает 160-битовый дайджест сообщения в массиве
 *   Message_Digest, предоставленном при вызове.
```

```
 *   Примечание.
 *    Первый октет хэша сохраняется в элементе с индексом 0,
 *    последний имеет индекс 19.
 *
 * Параметры
 *   context: [in/out] — контекст для расчёта хэша SHA-1.
 *   Message_Digest[ ]: [out] - массив для возврата дайджеста.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA1Result(SHA1Context *context,
    uint8_t Message_Digest[SHA1HashSize])
{
  int i;

  if (!context) return shaNull;
  if (!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;
}

/*
 * SHA1ProcessMessageBlock
 *
 * Описание
 *   Обработка первых 512 битов сообщения из массива Message_Block.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *
 *  Возвращаемое значение
 *   Нет.
 *
 * Комментарии
 *   Имена многих переменных (особенно 1-символьные) взяты из
 *   Secure Hash Standard.
 */
static void SHA1ProcessMessageBlock(SHA1Context *context)
{
  /* Константы, заданные в параграфе 4.2.1 FIPS 180-3 */
  const uint32_t K[4] = {
      0x5A827999, 0x6ED9EBA1, 0x8F1BBCDC, 0xCA62C1D6
  };

  int        t;              /* Счетчик цикла */
  uint32_t   temp;           /* Временное значение слова */
  uint32_t   W[80];          /* Последовательность слов */
  uint32_t   A, B, C, D, E;  /* Буферы слов */

  /* Инициализация первых 16 слов в массиве 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;
}

/*
 * SHA1Finalize
 *
 * Описание
 *   Вспомогательная функция, завершающая расчёт дайджеста.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   Pad_Byte: [in]
 *     Последний байт, добавляемый в сообщение перед заполнением
 *     нулями и указанием размера. Этот байт содержит последние биты
 *     сообщения, за которыми следует ещё 1 бит. Если размер сообщения
 *     кратен 8, Pad_Byte будет иметь значение 0x80.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
static void SHA1Finalize(SHA1Context *context, uint8_t Pad_Byte)
{
  int i;
  SHA1PadMessage(context, Pad_Byte);
  /* Сообщение может быть конфиденциальным, бит сбрасывается */
  for (i = 0; i < SHA1_Message_Block_Size; ++i)
    context->Message_Block[i] = 0;
  context->Length_High = 0;     /* Размер сбрасывается */
  context->Length_Low = 0;
  context->Computed = 1;
}

/*
 * SHA1PadMessage
 *
 * Описание
 *   По стандарту сообщение должно дополняться до размера, кратного
 *   512 битам. Первый бит дополнения должен иметь значение 1,
 *   последние 64 указывают размер исходного сообщения, остальные биты
 *   следует заполнить 0. Эта функция дополняет сообщение по указанным
 *   правилам, заполняя массив Message_Block. По завершении функции
 *   можно считать дайджест сообщения рассчитанным.
 *
 * Параметры
 *   context: [in/out] - контекст для дополнения.
 *   Pad_Byte: [in]
 *     Последний байт для добавления в блок сообщения до заполнения
 *     нулями и указания размера. Этот байт содержит последние биты
 *     сообщения и ещё 1 бит. Если размер сообщения кратен 8 битам,
 *     Pad_Byte = 0x80.
 *
 *  Возвращаемое значение
 *   Нет.
 */
static void SHA1PadMessage(SHA1Context *context, uint8_t Pad_Byte)
{
```

```
  /*
   * Если текущего блока сообщения недостаточно для включения битов
   * заполнения и размера, добавляется ещё один блок и заполнение
   * продолжается в нем.
   */
  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;

  /* Запись размера сообщения в последние 8 октетов */
  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);
}
```

8.2.2. sha224-256.c

```
/************************ sha224-256.c ************************/
/***************** См. детали в RFC 6234. *********************/
/* Авторские права (Copyright (c) 2011) принадлежат IETF Trust */
/* и лицам, указанным как авторы кода. Все права защищены.     */
/* Распространение описано в файле sha.h.                      */
/*
 *  Описание
 *    Реализация алгоритмов безопасного хэширования SHA-224 и SHA-256
 *    в соответствии со стандартом FIPS PUB 180-3, опубликованным в
 *    2008 г., и его предшественниками FIPS PUB 180-1 и FIP PUB 180-2.
 *
 *    Объединённый документ с описанием всех алгоритмов
 *    http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf
 *
 *    Алгоритмы SHA-224 и SHA-256 создают дайджест сообщения размером
 *    224 и 256 битов (соответственно) для потока данных. Для поиска
 *    сообщения с таким же дайджестом, как у данного, потребуется около
 *    2**n шагов и 2**(n/2) шагов - для поиска 2 сообщений с одинаковым
 *    дайджестом, где n - размер дайджеста в битах. Эти алгоритмы могут
 *    применяться для создания «оттисков» (fingerprint) сообщений.
 *
 * Переносимость
 *    SHA-224 и SHA-256 определены с использованием 32-битовых слов. В
 *    этом коде используется файл <stdint.h> (включён в sha.h) для 32-
 *    и 8-битовых целых чисел без знака. Код не будет транслироваться,
 *    если компилятор C не поддерживает 32-битовые целые без знака.
 *
 * Предостережения
 *    SHA-224 и SHA-256 разработаны для работы с сообщениями размером
 *    меньше 2^64 битов. Эта реализация использует SHA224/256Input() для
 *    хэширования числа битов кратного 8 (октет) а затем может применять
 *    SHA224/256FinalBits() для хэширования оставшихся входных битов.
 */

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

/* Макросы сдвига, а также циклического сдвига влево и вправо */
#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))))

/* Макросы SHA SIGMA и sigma */
#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))
```

```
/*
 * Добавляет length к размеру, устанавливает Corrupted при переполнении.
 */
static uint32_t addTemp;
#define SHA224_256AddLength(context, length)                \
  (addTemp = (context)->Length_Low, (context)->Corrupted = \
    (((context)->Length_Low += (length)) < addTemp) &&     \
    (++(context)->Length_High == 0) ? shaInputTooLong :    \
                                      (context)->Corrupted )

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

/* Исходные хэш-значения: FIPS 180-3, параграф 5.3.2 */
static uint32_t SHA224_H0[SHA256HashSize/4] = {
    0xC1059ED8, 0x367CD507, 0x3070DD17, 0xF70E5939,
    0xFFC00B31, 0x68581511, 0x64F98FA7, 0xBEFA4FA4
};

/* Исходные хэш-значения: FIPS 180-3, параграф 5.3.3 */
static uint32_t SHA256_H0[SHA256HashSize/4] = {
  0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,
  0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19
};

/*
 * SHA224Reset
 *
 * Описание
 *   Инициализация SHA224Context для подготовки к расчёту нового
 *   дайджеста SHA224.
 *
 * Параметры
 *   context: [in/out] - контекст для сброса
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA224Reset(SHA224Context *context)
{
  return SHA224_256Reset(context, SHA224_H0);
}

/*
 * SHA224Input
 *
 * Описание
 *   Принимает массив октетов следующей части сообщения.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_array[ ]: [in]
 *     Массив октетов, представляющий следующую часть сообщения.
 *   length: [in]
 *     Размер сообщения в message_array.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA224Input(SHA224Context *context, const uint8_t *message_array,
    unsigned int length)
{
  return SHA256Input(context, message_array, length);
}

/*
 * SHA224FinalBits
 *
 * Описание
 *   Добавление финальных битов в сообщение.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_bits: [in]
 *     Заключительные биты сообщения в старшей части байта (при
 *     3 битах ### следует использовать 0b###00000, а не 0b00000###).
 *   length: [in] - число битов в message_bits (1 - 7).
 *
 *  Возвращаемое значение
```

```
 *   Код ошибки sha.
 */
int SHA224FinalBits(SHA224Context *context,
                    uint8_t message_bits, unsigned int length)
{
  return SHA256FinalBits(context, message_bits, length);
}

/*
 * SHA224Result
 *
 * Описание
 *   Возвращает 224-битовый дайджест сообщения в массиве
 *   Message_Digest, представленном вызывающим.
 *   Примечание
 *    Первый октет хэша сохраняется в элементе массива с индексом 0,
 *    последний - в элементе с индексом 27.
 *
 * Параметры
 *   context: [in/out] - контекст для расчёта хэша SHA.
 *   Message_Digest[ ]: [out] - массив для возврата дайджеста.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA224Result(SHA224Context *context,
    uint8_t Message_Digest[SHA224HashSize])
{
  return SHA224_256ResultN(context, Message_Digest, SHA224HashSize);
}

/*
 * SHA256Reset
 *
 * Описание
 *   Инициализация SHA256Context для подготовки к расчёту нового
 *   дайджеста SHA256.
 *
 * Параметры
 *   context: [in/out] - контекст для сброса.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA256Reset(SHA256Context *context)
{
  return SHA224_256Reset(context, SHA256_H0);
}

/*
 * SHA256Input
 *
 * Описание
 *   Принимает массив октетов как следующую часть сообщения.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_array[ ]: [in]
 *     Массив октетов, представляющий следующую часть сообщения.
 *   length: [in] — размер сообщения в message_array.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA256Input(SHA256Context *context, const uint8_t *message_array,
    unsigned int length)
{
  if (!context) return shaNull;
  if (!length) return shaSuccess;
  if (!message_array) return shaNull;
  if (context->Computed) return context->Corrupted = shaStateError;
  if (context->Corrupted) return context->Corrupted;

  while (length--) {
    context->Message_Block[context->Message_Block_Index++] =
            *message_array;

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

    message_array++;
  }

  return context->Corrupted;
}
```

```
/*
 * SHA256FinalBits
 *
 * Описание
 *   Добавление финальных битов в сообщение.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_bits: [in]
 *     Заключительные биты сообщения в старшей части байта (при
 *     3 битах ### следует использовать 0b###00000, а не 0b00000###).
 *   length: [in] - число битов в message_bits (1 - 7).
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA256FinalBits(SHA256Context *context,
                    uint8_t message_bits, unsigned int length)
{
  static 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
  };
  static 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 (!context) return shaNull;
  if (!length) return shaSuccess;
  if (context->Corrupted) return context->Corrupted;
  if (context->Computed) return context->Corrupted = shaStateError;
  if (length >= 8) return context->Corrupted = shaBadParam;

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

  return context->Corrupted;
}

/*
 * SHA256Result
 *
 * Описание
 *   Возвращает 256-битовый дайджест сообщения в массиве
 *   Message_Digest, представленном вызывающим.
 *   Примечание
 *    Первый октет хэша сохраняется в элементе массива с индексом 0,
 *    последний - в элементе с индексом 31.
 *
 * Параметры
 *   context: [in/out] - контекст для расчёта хэша SHA.
 *   Message_Digest[ ]: [out] - массив для возврата дайджеста.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA256Result(SHA256Context *context,
                 uint8_t Message_Digest[SHA256HashSize])
{
  return SHA224_256ResultN(context, Message_Digest, SHA256HashSize);
}

/*
 * SHA224_256Reset
 *
 * Описание
 *   Инициализация SHA256Context для подготовки к расчёту нового
 *   дайджеста SHA-224 или SHA-256.
 *
 * Параметры
 *   context: [in/out] — контекст для сброса.
 *   H0[ ]: [in] — исходный массив значения хэша для использования.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
static int SHA224_256Reset(SHA256Context *context, uint32_t *H0)
{
  if (!context) return shaNull;
```

```
  context->Length_High = context->Length_Low = 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 = shaSuccess;

  return shaSuccess;
}

/*
 * SHA224_256ProcessMessageBlock
 *
 * Описание
 *   Обрабатывает следующие 512 битов массива Message_Block.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления
 *
 *  Возвращаемое значение
 *   Нет.
 *
 * Комментарии
 *   Имена многих переменных (особенно 1-символьные) взяты из
 *   Secure Hash Standard.
 */
static void SHA224_256ProcessMessageBlock(SHA256Context *context)
{
  /* Constants defined in FIPS 180-3, 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;                  /* Счетчик цикла */
  uint32_t   temp1, temp2;           /* Временное значение слова */
  uint32_t   W[64];                  /* Последовательность слов */
  uint32_t   A, B, C, D, E, F, G, H;  /* Буферы слов */

  /*
   * Инициализация первых 16 слов в массиве 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_256Finalize
 *
 * Описание
 *   Завершение расчёта дайджеста.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   Pad_Byte: [in]
 *     Последний байт для добавления в блок сообщения до заполнения
 *     нулями и указания размера. Этот байт содержит последние биты
 *     сообщения и ещё 1 бит. Если размер сообщения кратен 8 битам,
 *     Pad_Byte = 0x80.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
static void SHA224_256Finalize(SHA256Context *context,
    uint8_t Pad_Byte)
{
  int i;
  SHA224_256PadMessage(context, Pad_Byte);
  /* Сообщение может быть конфиденциальным, поэтому выполняется сброс */
  for (i = 0; i < SHA256_Message_Block_Size; ++i)
    context->Message_Block[i] = 0;
  context->Length_High = 0;     /* and clear length */
  context->Length_Low = 0;
  context->Computed = 1;
}

/*
 * SHA224_256PadMessage
 *
 * Описание
 *   По стандарту сообщение должно дополняться до размера, кратного
 *   512 битам. Первый бит дополнения должен иметь значение 1,
 *   последние 64 указывают размер исходного сообщения, остальные биты
 *   следует заполнить 0. Эта функция дополняет сообщение по указанным
 *   правилам, заполняя массив Message_Block. По завершении функции
 *   можно считать дайджест сообщения рассчитанным.
 *
 * Параметры
 *   context: [in/out] - контекст для дополнения.
 *   Pad_Byte: [in]
 *     Последний байт для добавления в блок сообщения до заполнения
 *     нулями и указания размера. Этот байт содержит последние биты
 *     сообщения и ещё 1 бит. Если размер сообщения кратен 8 битам,
 *     Pad_Byte = 0x80.
 *
 *  Возвращаемое значение
 *   Нет.
 */
static void SHA224_256PadMessage(SHA256Context *context,
    uint8_t Pad_Byte)
{
  /*
   * Если текущего блока сообщения недостаточно для включения битов
   * заполнения и размера, добавляется ещё один блок и заполнение
   * продолжается в нем.
   */
  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;

  /*
   * Запись размера сообщения в последние 8 октетов.
   */
  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_256ResultN
 *
 * Описание
 *   Возвращает 224- или 256-битовый дайджест сообщения в массиве
 *   Message_Digest, представленном вызывающим.
 *   Примечание
 *    Первый октет хэша сохраняется в элементе массива с индексом 0,
 *    последний - в элементе с индексом 27 или 31.
 *
 * Параметры
 *   context: [in/out] - контекст для расчёта хэша SHA.
 *   Message_Digest[ ]: [out] - массив для возврата дайджеста.
 *   HashSize: [in] - размер хэша (28 или 32).
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
static int SHA224_256ResultN(SHA256Context *context,
    uint8_t Message_Digest[ ], int HashSize)
{
  int i;

  if (!context) return shaNull;
  if (!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 ************************/
/***************** См. детали в RFC 6234. *********************/
/* Авторские права (Copyright (c) 2011) принадлежат IETF Trust */
/* и лицам, указанным как авторы кода. Все права защищены.     */
/* Распространение описано в файле sha.h.                      */
/*
 *  Описание
 *   Реализация алгоритмов безопасного хэширования SHA-384 и SHA-512
 *   в соответствии со стандартом FIPS PUB 180-3, опубликованным в
 *   2008 г., и его предшественниками FIPS PUB 180-1 и FIP PUB 180-2.
 *
 *   Объединённый документ с описанием всех алгоритмов
 *   http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf
 *
 *   Алгоритмы SHA-384 и SHA-512 создают дайджест сообщения размером
 *   224 и 256 битов (соответственно) для потока данных. Для поиска
 *   сообщения с таким же дайджестом, как у данного, потребуется около
 *   2**n шагов и 2**(n/2) шагов - для поиска 2 сообщений с одинаковым
 *   дайджестом, где n - размер дайджеста в битах. Эти алгоритмы могут
 *   применяться для создания «оттисков» (fingerprint) сообщений.
 *
 * Переносимость
 *   Алгоритмы SHA-384 и SHA-512 определены для 64-битовых слов, но
 *   при заданном USE_32BIT_ONLY код работает лишь с 32-битовыми.
 *   Код использует <stdint.h> (включён через sha.h) для задания
 *   64-, 32- and 8-битовый целых чисел без знака. Если компилятор
 *   C не поддерживает 64-битовые целые числа и не задано
 *   #define USE_32BIT_ONLY, код не будет работать.
 *
```

```
 * Предостережения
 *   SHA-384 и SHA-512 созданы для работы с сообщениями размером
 *   меньше 2^128 битов. Эта реализация использует SHA384/512Input()
 *   для хэширования числа битов, кратного 8-битовому октету, а затем
 *   может использовать SHA384/256FinalBits() для остальных битов.
 */

#include "sha.h"

#ifdef USE_32BIT_ONLY
/*
 * Задаёт 64-битовую арифметику на основе 32-битовых операций.
 * Каждое 64-битовое слово представляется массивом из 2 слов.
 * Макросы заданы так, что результат является последним параметром.
 */

/*
 * Макросы для сдвига, а также кольцевого сдвига влево и вправо.
 */
#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 )

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

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

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

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

/* 64-битовая операция 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]) )

/* Прибавляет значение 4word из word2 к 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) )

/* Прибавляет значение 2word из word2 к 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) )

/*
 * Циклический сдвиг SHA ((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)) )

/*
 * Макросы SHA SIGMA и sigma
 *
 *  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)) )

#ifndef USE_MODIFIED_MACROS
/*
 * Определения из параграфа 4.1.3 в FIPS 180-3
 *  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_MODIFIED_MACROS */
/*
 * Эти определения быстрее заданных в параграфе 4.1.3 FIPS 180-3
 *   ((x & y) ^ (~x & z)) становится
 *   ((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)) становится
 *   ((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 */

/*
 * Добавляет length к размеру, устанавливает Corrupted при переполнении.
 */
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] < (length)) &&      \
       ((context)->Length[2] == 0) && ((context)->Length[1] == 0) &&  \
       ((context)->Length[0] == 0)) ? shaInputTooLong :               \
                                      (context)->Corrupted )

/* Локальные прототипы функций */
static int SHA384_512Reset(SHA512Context *context,
                           uint32_t H0[SHA512HashSize/4]);
static void SHA384_512ProcessMessageBlock(SHA512Context *context);
static void SHA384_512Finalize(SHA512Context *context,
  uint8_t Pad_Byte);
static void SHA384_512PadMessage(SHA512Context *context,
  uint8_t Pad_Byte);
static int SHA384_512ResultN( SHA512Context *context,
  uint8_t Message_Digest[ ], int HashSize);

/* Исходные значения хэша: FIPS 180-3, параграфы 5.3.4 и 5.3.5 */
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 */

#include "sha-private.h"

/* Макросы для сдвига, а также кольцевого сдвига влево и вправо. */
#define SHA512_SHR(bits,word)  (((uint64_t)(word)) >> (bits))
#define SHA512_ROTR(bits,word) ((((uint64_t)(word)) >> (bits)) | \
                                (((uint64_t)(word)) << (64-(bits))))

/*
 * Макросы SHA SIGMA и sigma.
 *
 *  SHA512_ROTR(28,word) ^ SHA512_ROTR(34,word) ^ SHA512_ROTR(39,word)
 */
#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))

/*
 * Добавляет length к размеру, устанавливает Corrupted при переполнении.
 */
static uint64_t addTemp;
#define SHA384_512AddLength(context, length)                  \
   (addTemp = context->Length_Low, context->Corrupted =       \
    ((context->Length_Low += length) < addTemp) &&            \
    (++context->Length_High == 0) ? shaInputTooLong :         \
                                    (context)->Corrupted)

/* Локальные прототипы функций. */
static int SHA384_512Reset(SHA512Context *context,
                           uint64_t H0[SHA512HashSize/8]);
static void SHA384_512ProcessMessageBlock(SHA512Context *context);
static void SHA384_512Finalize(SHA512Context *context,
```

```
  uint8_t Pad_Byte);
static void SHA384_512PadMessage(SHA512Context *context,
  uint8_t Pad_Byte);
static int SHA384_512ResultN(SHA512Context *context,
  uint8_t Message_Digest[ ], int HashSize);

/* Исходные значения хэша: FIPS 180-3, параграфы 5.3.4 и 5.3.5 */
static uint64_t SHA384_H0[ ] = {
    0xCBBB9D5DC1059ED8ll, 0x629A292A367CD507ll, 0x9159015A3070DD17ll,
    0x152FECD8F70E5939ll, 0x67332667FFC00B31ll, 0x8EB44A876858151Bll,
    0xDB0C2E0D64F98FA7ll, 0x47B5481DBEFA4FA4ll
};
static uint64_t SHA512_H0[ ] = {
    0x6A09E667F3BCC908ll, 0xBB67AE8584CAA73Bll, 0x3C6EF372FE94F82Bll,
    0xA54FF53A5F1D36F1ll, 0x510E527FADE682D1ll, 0x9B05688C2B3E6C1Fll,
    0x1F83D9ABFB41BD6Bll, 0x5BE0CD19137E2179ll
};

#endif /* USE_32BIT_ONLY */

/*
 * SHA384Reset
 *
 * Описание
 *   Инициализация SHA384Context для подготовки к расчёту нового
 *   дайджеста SHA-384 или.
 *
 * Параметры
 *   context: [in/out] - контекст для сброса.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 *
 */
int SHA384Reset(SHA384Context *context)
{
  return SHA384_512Reset(context, SHA384_H0);
}

/*
 * SHA384Input
 *
 * Описание
 *   Принимает массив октетов как следующую часть сообщения.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_array[ ]: [in]
 *     Массив октетов, представляющий следующую часть сообщения.
 *   length: [in] — размер сообщения в message_array.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA384Input(SHA384Context *context,
    const uint8_t *message_array, unsigned int length)
{
  return SHA512Input(context, message_array, length);
}

/*
 * SHA384FinalBits
 *
 * Описание
 *   Добавление финальных битов в сообщение.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_bits: [in]
 *     Заключительные биты сообщения в старшей части байта (при
 *     3 битах ### следует использовать 0b###00000, а не 0b00000###).
 *   length: [in] - число битов в message_bits (1 - 7).
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA384FinalBits(SHA384Context *context,
                    uint8_t message_bits, unsigned int length)
{
  return SHA512FinalBits(context, message_bits, length);
}

/*
 * SHA384Result
 *
 * Описание
```

```
 *   Возвращает 384-битовый дайджест сообщения в массиве
 *   Message_Digest, представленном вызывающим.
 *   Примечание
 *    Первый октет хэша сохраняется в элементе массива с индексом 0,
 *    последний - в элементе с индексом 47.
 *
 * Параметры
 *   context: [in/out] - контекст для расчёта хэша SHA.
 *   Message_Digest[ ]: [out] - массив для возврата дайджеста.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA384Result(SHA384Context *context,
    uint8_t Message_Digest[SHA384HashSize])
{
  return SHA384_512ResultN(context, Message_Digest, SHA384HashSize);
}

/*
 * SHA512Reset
 *
 * Описание
 *   Инициализация SHA512Context для подготовки к расчёту нового
 *   дайджеста SHA-512.
 *
 * Параметры
 *   context: [in/out] - контекст для сброса.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA512Reset(SHA512Context *context)
{
  return SHA384_512Reset(context, SHA512_H0);
}

/*
 * SHA512Input
 *
 * Описание
 *   Принимает массив октетов как следующую часть сообщения.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_array[ ]: [in]
 *     Массив октетов, представляющий следующую часть сообщения.
 *   length: [in] — размер сообщения в message_array.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA512Input(SHA512Context *context,
        const uint8_t *message_array,
        unsigned int length)
{
  if (!context) return shaNull;
  if (!length) return shaSuccess;
  if (!message_array) return shaNull;
  if (context->Computed) return context->Corrupted = shaStateError;
  if (context->Corrupted) return context->Corrupted;

  while (length--) {
    context->Message_Block[context->Message_Block_Index++] =
            *message_array;

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

    message_array++;
  }

  return context->Corrupted;
}

/*
 * SHA512FinalBits
 *
 * Описание
 *   Добавление финальных битов в сообщение.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_bits: [in]
 *     Заключительные биты сообщения в старшей части байта (при
```

```
 *     3 битах ### следует использовать 0b###00000, а не 0b00000###).
 *   length: [in] - число битов в message_bits (1 - 7).
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA512FinalBits(SHA512Context *context,
                    uint8_t message_bits, unsigned int length)
{
  static 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
  };
  static 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 (!context) return shaNull;
  if (!length) return shaSuccess;
  if (context->Corrupted) return context->Corrupted;
  if (context->Computed) return context->Corrupted = shaStateError;
  if (length >= 8) return context->Corrupted = shaBadParam;

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

  return context->Corrupted;
}

/*
 * SHA512Result
 *
 * Описание
 *   Возвращает 512-битовый дайджест сообщения в массиве
 *   Message_Digest, представленном вызывающим.
 *   Примечание
 *    Первый октет хэша сохраняется в элементе массива с индексом 0,
 *    последний - в элементе с индексом 63.
 *
 * Параметры
 *   context: [in/out] - контекст для расчёта хэша SHA.
 *   Message_Digest[ ]: [out] - массив для возврата дайджеста.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int SHA512Result(SHA512Context *context,
    uint8_t Message_Digest[SHA512HashSize])
{
  return SHA384_512ResultN(context, Message_Digest, SHA512HashSize);
}

/*
 * SHA384_512Reset
 *
 * Описание
 *   Инициализация SHA512Context для подготовки к расчёту нового
 *   дайджеста SHA-3844 или SHA-512.
 *
 * Параметры
 *   context: [in/out] - контекст для сброса.
 *   H0[ ]: [in] - исходный массив хэша для использования.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
#ifdef USE_32BIT_ONLY
static int SHA384_512Reset(SHA512Context *context,
                           uint32_t H0[SHA512HashSize/4])
#else /* !USE_32BIT_ONLY */
static int SHA384_512Reset(SHA512Context *context,
                           uint64_t H0[SHA512HashSize/8])
#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] =
  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 = shaSuccess;

  return shaSuccess;
}

/*
 * SHA384_512ProcessMessageBlock
 *
 * Описание
 *   Обрабатывает следующие 1024 бита массива Message_Block.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления
 *
 *  Возвращаемое значение
 *   Нет.
 *
 * Комментарии
 *   Имена многих переменных (особенно 1-символьные) взяты из
 *   Secure Hash Standard.
 */
static void SHA384_512ProcessMessageBlock(SHA512Context *context)
{
#ifdef USE_32BIT_ONLY
  /* Constants defined in FIPS 180-3, section 4.2.3 */
  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;                  /* Счётчик цикла */
  uint32_t  temp1[2], temp2[2],       /* Временное значение словаs */
        temp3[2], temp4[2], temp5[2];
  uint32_t  W[2*80];                  /* Последовательность слов */
  uint32_t  A[2], B[2], C[2], D[2],   /* Буферы слов */
        E[2], F[2], G[2], H[2];

  /* Инициализация первых 16 слов в массиве 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 */
  /* Константы заданы в параграфе 4.2.3 FIPS 180-3 */
  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,
```

```
      0x650A73548BAF63DE1l, 0x766A0ABB3C77B2A81l, 0x81C2C92E47EDAEE61l,
      0x92722C851482353B1l, 0xA2BFE8A14CF103641l, 0xA81A664BBC4230011l,
      0xC24B8B70D0F8979111l, 0xC76C51A30654BE301l, 0xD192E819D6EF52181l,
      0xD69906245565A9101l, 0xF40E35855771202A1l, 0x106AA07032BBD1B81l,
      0x19A4C116B8D2D0C81l, 0x1E376C085141AB531l, 0x2748774CDF8EEB991l,
      0x34B0BCB5E19B48A81l, 0x391C0CB3C5C95A631l, 0x4ED8AA4AE3418ACB1l,
      0x5B9CCA4F7763E3731l, 0x682E6FF3D6B2B8A31l, 0x748F82EE5DEFB2FC1l,
      0x78A5636F43172F601l, 0x84C87814A1F0AB721l, 0x8CC702081A6439EC1l,
      0x90BEFFFA23631E281l, 0xA4506CEBDE82BDE91l, 0xBEF9A3F7B2C6791S1l,
      0xC67178F2E372532B1l, 0xCA273ECEEA26619C1l, 0xD186B8C721C0C2071l,
      0xEADA7DD6CDE0EB1E1l, 0xF57D4F7FEE6ED1781l, 0x06F067AA72176FBA1l,
      0x0A637DC5A2C898A61l, 0x113F9804BEF90DAE1l, 0x1B710B35131C471B1l,
      0x28DB77F523047D841l, 0x32CAAB7B40C7249311, 0x3C9EBE0A15C9BEBC1l,
      0x431D67C49C100D4C1l, 0x4CC5D4BECB3E42B61l, 0x597F299CFC657E2A1l,
      0x5FCB6FAB3AD6FAEC1l, 0x6C44198C4A475817l1
  };
  int        t, t8;                  /* Счётчик цикла */
  uint64_t   temp1, temp2;           /* Временное значение слова */
  uint64_t   W[80];                  /* Последовательность слов */
  uint64_t   A, B, C, D, E, F, G, H; /* Буферы слов */

  /* Инициализация первых 16 слов в массиве 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_512Finalize
 *
 * Описание
 *   Завершение расчёта дайджеста.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   Pad_Byte: [in]
 *     Последний байт для добавления в блок сообщения до заполнения
 *     нулями и указания размера. Этот байт содержит последние биты
 *     сообщения и ещё 1 бит. Если размер сообщения кратен 8 битам,
 *     Pad_Byte = 0x80.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
```

```
 */
static void SHA384_512Finalize(SHA512Context *context,
    uint8_t Pad_Byte)
{
  int_least16_t i;
  SHA384_512PadMessage(context, Pad_Byte);
  /* сообщение может быть конфиденциальным, сброс */
  for (i = 0; i < SHA512_Message_Block_Size; ++i)
    context->Message_Block[i] = 0;
#ifdef USE_32BIT_ONLY    /* сброс length */
  context->Length[0] = context->Length[1] = 0;
  context->Length[2] = context->Length[3] = 0;
#else /* !USE_32BIT_ONLY */
  context->Length_High = context->Length_Low = 0;
#endif /* USE_32BIT_ONLY */
  context->Computed = 1;
}

/*
 * SHA384_512PadMessage
 *
 * Описание
 *   По стандарту сообщение должно дополняться до размера, кратного
 *   1024 битам. Первый бит дополнения должен иметь значение 1,
 *   последние 64 указывают размер исходного сообщения, остальные биты
 *   следует заполнить 0. Эта функция дополняет сообщение по указанным
 *   правилам, заполняя массив Message_Block. По завершении функции
 *   можно считать дайджест сообщения рассчитанным.
 *
 * Параметры
 *   context: [in/out] - контекст для дополнения.
 *   Pad_Byte: [in]
 *     Последний байт для добавления в блок сообщения до заполнения
 *     нулями и указания размера. Этот байт содержит последние биты
 *     сообщения и ещё 1 бит. Если размер сообщения кратен 8 битам,
 *     Pad_Byte = 0x80.
 *
 *  Возвращаемое значение
 *   Нет.
 */
static void SHA384_512PadMessage(SHA512Context *context,
    uint8_t Pad_Byte)
{
  /*
   * Если текущего блока сообщения недостаточно для включения битов
   * заполнения и размера, добавляется ещё один блок и заполнение
   * продолжается в нем.
   */
  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;

  /* Сохранение размера сообщения в последних 16 октетах. */
#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_512ResultN
 *
 * Описание
 *   Возвращает 384- или 512битовый дайджест сообщения в массиве
 *   Message_Digest, представленном вызывающим.
 *   Примечание
 *    Первый октет хэша сохраняется в элементе массива с индексом 0,
 *    последний - в элементе с индексом 47 или 63.
 *
 * Параметры
 *   context: [in/out] - контекст для расчёта хэша SHA.
 *   Message_Digest[ ]: [out] - массив для возврата дайджеста.
 *   HashSize: [in] - размер хэша (48 или 64).
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
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) return shaNull;
  if (!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 ***************************/
/***************** См. детали в RFC 6234. *********************/
/* Авторские права (Copyright (c) 2011) принадлежат IETF Trust */
/* и лицам, указанным как авторы кода. Все права защищены.     */
/* Распространение описано в файле sha.h.                      */
/*
 *  Описание
 *    Реализация унифицированного интерфейса для алгоритмов SHA.
 */

#include "sha.h"

/*
 *  USHAReset
 *
 *  Описание
```

```
 *   Инициализация контекста SHA для подготовки к расчёту нового
 *   дайджеста SHA.
 *
 *  Параметры
 *      context: [in/out] - контекст для сброса.
 *      whichSha: [in] - задаёт SHA для сброса.
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */
int USHAReset(USHAContext *context, enum SHAversion whichSha)
{
  if (!context) return shaNull;
  context->whichSha = whichSha;
  switch (whichSha) {
    case SHA1:   return SHA1Reset((SHA1Context*)&context->ctx);
    case SHA224: return SHA224Reset((SHA224Context*)&context->ctx);
    case SHA256: return SHA256Reset((SHA256Context*)&context->ctx);
    case SHA384: return SHA384Reset((SHA384Context*)&context->ctx);
    case SHA512: return SHA512Reset((SHA512Context*)&context->ctx);
    default: return shaBadParam;
  }
}

/*
 *  USHAInput
 *
 *  Описание
 *   Принимает массив октетов как следующую часть сообщения.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_array[ ]: [in]
 *     Массив октетов, представляющий следующую часть сообщения.
 *   length: [in] — размер сообщения в message_array.
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */
int USHAInput(USHAContext *context,
              const uint8_t *bytes, unsigned int bytecount)
{
  if (!context) return shaNull;
  switch (context->whichSha) {
    case SHA1:
      return SHA1Input((SHA1Context*)&context->ctx, bytes, bytecount);
    case SHA224:
      return SHA224Input((SHA224Context*)&context->ctx, bytes,
          bytecount);
    case SHA256:
      return SHA256Input((SHA256Context*)&context->ctx, bytes,
          bytecount);
    case SHA384:
      return SHA384Input((SHA384Context*)&context->ctx, bytes,
          bytecount);
    case SHA512:
      return SHA512Input((SHA512Context*)&context->ctx, bytes,
          bytecount);
    default: return shaBadParam;
  }
}

/*
 * USHAFinalBits
 *
 * Описание
 *   Добавление финальных битов в сообщение.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_bits: [in]
 *     Заключительные биты сообщения в старшей части байта (при
 *     3 битах ### следует использовать 0b###00000, а не 0b00000###).
 *   length: [in] - число битов в message_bits (1 - 7).
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int USHAFinalBits(USHAContext *context,
                  uint8_t bits, unsigned int bit_count)
{
  if (!context) return shaNull;
  switch (context->whichSha) {
    case SHA1:
      return SHA1FinalBits((SHA1Context*)&context->ctx, bits,
                           bit_count);
```

```
    case SHA224:
      return SHA224FinalBits((SHA224Context*)&context->ctx, bits,
          bit_count);
    case SHA256:
      return SHA256FinalBits((SHA256Context*)&context->ctx, bits,
          bit_count);
    case SHA384:
      return SHA384FinalBits((SHA384Context*)&context->ctx, bits,
          bit_count);
    case SHA512:
      return SHA512FinalBits((SHA512Context*)&context->ctx, bits,
          bit_count);
    default: return shaBadParam;
  }
}

/*
 * USHAResult
 *
 * Описание
 *   Возвращает дайджест сообщения с числом битов
 *   USHAHashSizeBits(whichSHA) для значения whichSHA в предыдущем
 *   вызове USHAReset в массиве Message_Digest заданном вызывающим.
 *
 * Параметры
 *   context: [in/out] - контекст для расчёта хэша SHA-1.
 *   Message_Digest[ ]: [out] - массив для возврата дайджеста.
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int USHAResult(USHAContext *context,
               uint8_t Message_Digest[USHAMaxHashSize])
{
  if (!context) return shaNull;
  switch (context->whichSha) {
    case SHA1:
      return SHA1Result((SHA1Context*)&context->ctx, Message_Digest);
    case SHA224:
      return SHA224Result((SHA224Context*)&context->ctx,
                          Message_Digest);
    case SHA256:
      return SHA256Result((SHA256Context*)&context->ctx,
                          Message_Digest);
    case SHA384:
      return SHA384Result((SHA384Context*)&context->ctx,
                          Message_Digest);
    case SHA512:
      return SHA512Result((SHA512Context*)&context->ctx,
                          Message_Digest);
    default: return shaBadParam;
  }
}

/*
 * USHABlockSize
 *
 * Описание
 *   Возвращает размер блока для данного алгоритма SHA.
 *
 * Параметры
 *   whichSha - алгоритм SHA для запроса.
 *
 *  Возвращаемое значение
 *   Размер блока.
 */
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
 *
 * Описание
 *   Возвращает размер хэша для данного алгоритма SHA.
 *
 * Параметры
 *   whichSha - алгоритм SHA для запроса.
```

```
 *
 *  Возвращаемое значение
 *    Размер блока.
 */
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
 *
 * Описание
 *   Возвращает размер хэша для данного алгоритма SHA (в битах).
 *
 * Параметры
 *   whichSha - алгоритм SHA для запроса.
 *
 *  Возвращаемое значение
 *   Размер хэша в битах.
 */
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;
  }
}

/*
 * USHAHashName
 *
 * Описание
 *   Имя данного алгоритма SHA в виде строки.
 *
 * Параметры
 *   whichSha - алгоритм SHA для запроса.
 *
 *  Возвращаемое значение
 *   Строка символов имени алгоритма.
 */
const char *USHAHashName(enum SHAversion whichSha)
{
  switch (whichSha) {
    case SHA1:   return "SHA1";
    case SHA224: return "SHA224";
    case SHA256: return "SHA256";
    case SHA384: return "SHA384";
    default:
    case SHA512: return "SHA512";
  }
}
```

8.3. Код HMAC

```
/*************************** hmac.c ***************************/
/***************** См. детали в RFC 6234. *********************/
/* Авторские права (Copyright (c) 2011) принадлежат IETF Trust */
/* и лицам, указанным как авторы кода. Все права защищены.     */
/* Распространение описано в файле sha.h.                      */
/*
 *  Описание
 *    Этот файл реализует алгоритм HMAC (хэширование с ключом для
 *    проверки подлинности сообщения, [RFC 2104]), на основе SHA.
 */

#include "sha.h"

/*
 *  hmac
 *
 *  Описание
 *      Расчёт дайджеста HMAC для сообщения.
 *
 *  Параметры
```

```
 *      whichSha: [in] - SHA1, SHA224, SHA256, SHA384 или SHA512
 *      message_array[ ]: [in]
 *          Массив октетов, представляющих сообщение (в  RFC 2104
 *          этот параметр называется text).
 *      length: [in] - размер сообщения в message_array.
 *      key[ ]: [in] - общий секретный ключ.
 *      key_len: [in] - размер общего секретного ключа.
 *      digest[ ]: [out]
 *          Массив для возврата дайджеста.
 *          Примечание. Размер дайджеста указывает значение whichSha.
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */

int hmac(SHAversion whichSha,
    const unsigned char *message_array, int length,
    const unsigned char *key, int key_len,
    uint8_t digest[USHAMaxHashSize])
{
  HMACContext context;
  return hmacReset(&context, whichSha, key, key_len) ||
         hmacInput(&context, message_array, length) ||
         hmacResult(&context, digest);
}

/*
 *  hmacReset
 *
 *  Описание
 *      Инициализация hmacContext для подготовки к расчёту HMAC.
 *
 *  Параметры
 *      context: [in/out] - контекст для сброса.
 *      whichSha: [in] - SHA1, SHA224, SHA256, SHA384 или SHA512.
 *      key[ ]: [in] - общий секретный ключ.
 *      key_len: [in] - размер общего секретного ключа.
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */
int hmacReset(HMACContext *context, enum SHAversion whichSha,
    const unsigned char *key, int key_len)
{
  int i, blocksize, hashsize, ret;

  /* Внутреннее заполнение - key XOR ipad */
  unsigned char k_ipad[USHA_Max_Message_Block_Size];

  /* Временный буфер при key_len > blocksize */
  unsigned char tempkey[USHAMaxHashSize];

  if (!context) return shaNull;
  context->Computed = 0;
  context->Corrupted = shaSuccess;

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

  /*
   * Если key длиннее blocksize, сброс в key = HASH(key).
   */
  if (key_len > blocksize) {
    USHAContext tcontext;
    int err = USHAReset(&tcontext, whichSha) ||
              USHAInput(&tcontext, key, key_len) ||
              USHAResult(&tcontext, tempkey);
    if (err != shaSuccess) return err;

    key = tempkey;
    key_len = hashsize;
  }

  /*
   * Преобразования HMAC имеет вид:
   *
   * SHA(K XOR opad, SHA(K XOR ipad, text))
   *
   * где K - n-й байт key с дополнением 0 до blocksize байтов,
   * ipad - байт 0x36, повторенный blocksize раз,
   * opad - байт 0x5c, повторенный blocksize раз,
   * text - защищаемые данные.
   */
```

```
  /* Сохранение ключа в заполнении, XOR со значениями ipad и opad. */
  for (i = 0; i < key_len; i++) {
    k_ipad[i] = key[i] ^ 0x36;
    context->k_opad[i] = key[i] ^ 0x5c;
  }
  /* Оставшиеся байты заполнения имеют значение \0 XOR ipad и opad. */
  for ( ; i < blocksize; i++) {
    k_ipad[i] = 0x36;
    context->k_opad[i] = 0x5c;
  }

  /* Внутреннее хэширование. */
  /* Инициализация контекста для первого прохода */
  ret = USHAReset(&context->shaContext, whichSha) ||
        /* Начало внутреннего заполнения */
        USHAInput(&context->shaContext, k_ipad, blocksize);
  return context->Corrupted = ret;
}

/*
 *  hmacInput
 *
 *  Описание
 *   Принимает массив октетов как следующую часть сообщения.
 *   Может вызываться неоднократно.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_array[ ]: [in]
 *     Массив октетов, представляющий следующую часть сообщения.
 *   length: [in] — размер сообщения в text.
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */
int hmacInput(HMACContext *context, const unsigned char *text,
    int text_len)
{
  if (!context) return shaNull;
  if (context->Corrupted) return context->Corrupted;
  if (context->Computed) return context->Corrupted = shaStateError;
  /* Затем текст дейтаграммы */
  return context->Corrupted =
    USHAInput(&context->shaContext, text, text_len);
}

/*
 * hmacFinalBits
 *
 * Описание
 *   Добавление финальных битов в HMAC.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_bits: [in]
 *     Заключительные биты сообщения в старшей части байта (при
 *     3 битах ### следует использовать 0b###00000, а не 0b00000###).
 *   length: [in] - число битов в message_bits (1 - 7).
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int hmacFinalBits(HMACContext *context,
    uint8_t bits, unsigned int bit_count)
{
  if (!context) return shaNull;
  if (context->Corrupted) return context->Corrupted;
  if (context->Computed) return context->Corrupted = shaStateError;
  /* Затем финальные биты дейтаграммы */
  return context->Corrupted =
    USHAFinalBits(&context->shaContext, bits, bit_count);
}

/*
 * hmacResult
 *
 * Описание
 *   Возвращает N-байтовый дайджест сообщения в массиве
 *   Message_Digest, предоставленном вызывающим.
 *
 * Параметры
 *   context: [in/out] - контекст для расчёта хэша SHA.
 *   Message_Digest[ ]: [out] - массив для возврата дайджеста.
 *     Примечание. Размер хэша определяется значением
 *     whichSha, переданным в hmacReset().
 *
```

```
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int hmacResult(HMACContext *context, uint8_t *digest)
{
  int ret;
  if (!context) return shaNull;
  if (context->Corrupted) return context->Corrupted;
  if (context->Computed) return context->Corrupted = shaStateError;

  /* Завершение первого прохода */
  /* (дайджест используется как временный буфер) */
  ret =
    USHAResult(&context->shaContext, digest) ||
         /* Выполнение внешнего SHA */
         /* Инициализация контекста для второго прохода */
         USHAReset(&context->shaContext, context->whichSha) ||

         /* Начало внешнего заполнения */
         USHAInput(&context->shaContext, context->k_opad,
                   context->blockSize) ||

         /* Затем результат первого прохода */
         USHAInput(&context->shaContext, digest, context->hashSize) ||
         /* Завершение второго прохода */
         USHAResult(&context->shaContext, digest);

  context->Computed = 1;
  return context->Corrupted = ret;
}
```

8.4. Код HKDF

```
/************************** hkdf.c ***************************/
/***************** См. детали в RFC 6234. *********************/
/* Авторские права (Copyright (c) 2011) принадлежат IETF Trust */
/* и лицам, указанным как авторы кода. Все права защищены.     */
/* Распространение и описано в файле sha.h.                    */
/*
 *  Описание
 *      Этот файл реализует алгоритм HKDF (функция порождения ключей на
 *      основе HMAC, RFC 5869), с различными вариантами алгоритма SHA.
 */

#include "sha.h"
#include <string.h>
#include <stdlib.h>

/*
 *  hkdf
 *
 *  Описание
 *      Генерация ключевого материала с использованием HKDF.
 *
 *  Параметры
 *      whichSha: [in] - SHA1, SHA224, SHA256, SHA384 или SHA512
 *      salt[ ]: [in]
 *         Необязательная затравка (несекретное случайное значение);
 *         по умолчанию salt == NULL, устанавливается внутренне как
 *         строка HashLen(whichSha) нулей.
 *      salt_len: [in] - размер затравки (игнорируется при salt == NULL)
 *      ikm[ ]: [in] - входной ключевой материал.
 *      ikm_len: [in] - размер входного ключевого материала.
 *      info[ ]: [in]
 *         необязательный контекст и зависящая от приложения информация.
 *         Игнорируется, если info == NULL или задана пустая строка.
 *      info_len: [in]
 *          размер необязательного контекста или зависящей от приложения
 *          информации (игнорируется при info == NULL).
 *      okm[ ]: [out] - массив для записи HKDF.
 *      okm_len: [in]
 *          Размер буфера для okm. Должно быть
 *          okm_len <= 255 * USHABlockSize(whichSha)
 *
 *  Примечание
 *      Вызывает hkdfExtract() и hkdfExpand().
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */
int hkdf(SHAversion whichSha,
    const unsigned char *salt, int salt_len,
    const unsigned char *ikm, int ikm_len,
    const unsigned char *info, int info_len,
    uint8_t okm[ ], int okm_len)
{
```

```
  uint8_t prk[USHAMaxHashSize];
  return hkdfExtract(whichSha, salt, salt_len, ikm, ikm_len, prk) ||
         hkdfExpand(whichSha, prk, USHAHashSize(whichSha), info,
                    info_len, okm, okm_len);
}

/*
 *  hkdfExtract
 *
 *  Описание
 *      Извлекает HKDF.
 *
 *  Параметры
 *      whichSha: [in] - SHA1, SHA224, SHA256, SHA384 или SHA512
 *      salt[ ]: [in]
 *         Необязательная затравка (несекретное случайное значение);
 *         по умолчанию salt == NULL, устанавливается внутренне как
 *         строка HashLen(whichSha) нулей.
 *      salt_len: [in] - размер затравки (игнорируется при salt == NULL)
 *      ikm[ ]: [in] - входной ключевой материал.
 *      ikm_len: [in] - размер входного ключевого материала.
 *      prk[ ]: [out]
 *          Массив для записи извлеченного HKDF. Должен быть больше
 *          USHAHashSize(whichSha);
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */
int hkdfExtract(SHAversion whichSha,
    const unsigned char *salt, int salt_len,
    const unsigned char *ikm, int ikm_len,
    uint8_t prk[USHAMaxHashSize])
{
  unsigned char nullSalt[USHAMaxHashSize];
  if (salt == 0) {
    salt = nullSalt;
    salt_len = USHAHashSize(whichSha);
    memset(nullSalt, '\0', salt_len);
  } else if (salt_len < 0) {
    return shaBadParam;
  }
  return hmac(whichSha, ikm, ikm_len, salt, salt_len, prk);
}

/*
 *  hkdfExpand
 *
 *  Описание
 *      Получет HKDF из входных данных.
 *
 *  Параметры
 *      whichSha: [in] - SHA1, SHA224, SHA256, SHA384 или SHA512
 *      prk[ ]: [in]
 *         Псевдослучайный ключ для преобразоавния, получаемый напрямую
 *         от криптографически строгого генератора с однородным
 *         распределением или из функции hkdfExtract().
 *      prk_len: [in]
 *         Размер ключа prk, которому следует быть не меньше
 *         USHAHashSize(whichSHA).
 *      info[ ]: [in]
 *         необязательный контекст и зависящая от приложения информация.
 *         Игнорируется, если info == NULL или задана пустая строка.
 *      info_len: [in]
 *         размер необязательного контекста или зависящей от приложения
 *         информации (игнорируется при info == NULL).
 *      okm[ ]: [out] - массив для записи HKDF.
 *      okm_len: [in]
 *         Размер буфера для okm. Должно быть
 *         okm_len <= 255 * USHABlockSize(whichSha)
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */
int hkdfExpand(SHAversion whichSha, const uint8_t prk[ ], int prk_len,
    const unsigned char *info, int info_len,
    uint8_t okm[ ], int okm_len)
{
  int hash_len, N;
  unsigned char T[USHAMaxHashSize];
  int Tlen, where, i;

  if (info == 0) {
    info = (const unsigned char *)"";
    info_len = 0;
  } else if (info_len < 0) {
    return shaBadParam;
```

```
  }
  if (okm_len <= 0) return shaBadParam;
  if (!okm) return shaBadParam;

  hash_len = USHAHashSize(whichSha);
  if (prk_len < hash_len) return shaBadParam;
  N = okm_len / hash_len;
  if ((okm_len % hash_len) != 0) N++;
  if (N > 255) return shaBadParam;

  Tlen = 0;
  where = 0;
  for (i = 1; i <= N; i++) {
    HMACContext context;
    unsigned char c = i;
    int ret = hmacReset(&context, whichSha, prk, prk_len) ||
              hmacInput(&context, T, Tlen) ||
              hmacInput(&context, info, info_len) ||
              hmacInput(&context, &c, 1) ||
              hmacResult(&context, T);
    if (ret != shaSuccess) return ret;
    memcpy(okm + where, T,
           (i != N) ? hash_len : (okm_len - where));
    where += hash_len;
    Tlen = hash_len;
  }
  return shaSuccess;
}

/*
 *  hkdfReset
 *
 *  Описание
 *      Инициализация hkdfContext для подготовки к порождению ключа
 *      с использованием модульного интерфейса HKDF для входных данных
 *      произвольного размера.
 *
 *  Параметры
 *      context: [in/out] - контекст для сброса.
 *      whichSha: [in] - SHA1, SHA224, SHA256, SHA384 или SHA512
 *      salt[ ]: [in]
 *         Необязательная затравка (несекретное случайное значение);
 *         по умолчанию salt == NULL, устанавливается внутренне как
 *         строка HashLen(whichSha) нулей.
 *      salt_len: [in] - размер затравки (игнорируется при salt == NULL)
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */
int hkdfReset(HKDFContext *context, enum SHAversion whichSha,
              const unsigned char *salt, int salt_len)
{
  unsigned char nullSalt[USHAMaxHashSize];
  if (!context) return shaNull;

  context->whichSha = whichSha;
  context->hashSize = USHAHashSize(whichSha);
  if (salt == 0) {
    salt = nullSalt;
    salt_len = context->hashSize;
    memset(nullSalt, '\0', salt_len);
  }

  return hmacReset(&context->hmacContext, whichSha, salt, salt_len);
}

/*
 *  hkdfInput
 *
 *  Описание
 *     Принимает массив октетов как следующую часть входного ключевого
 *     материала и может вызываться неоднократно.
 *
 *  Параметры
 *      context: [in/out] - контекст HKDF для обновления.
 *      ikm[ ]: [in]
 *        Массив октетов, представляющий следующую часть входных данных.
 *      ikm_len: [in] - размер ikm.
 *
 *  Возвращаемое значение
 *      Код ошибки sha.
 */
int hkdfInput(HKDFContext *context, const unsigned char *ikm,
              int ikm_len)
{
  if (!context) return shaNull;
```

```
  if (context->Corrupted) return context->Corrupted;
  if (context->Computed) return context->Corrupted = shaStateError;
  return hmacInput(&context->hmacContext, ikm, ikm_len);
}

/*
 * hkdfFinalBits
 *
 * Описание
 *   Добавление финальных битов в ключевой материал.
 *
 * Параметры
 *   context: [in/out] - контекст SHA для обновления.
 *   message_bits: [in]
 *     Заключительные биты сообщения в старшей части байта (при
 *     3 битах ### следует использовать 0b###00000, а не 0b00000###).
 *   length: [in] - число битов в message_bits (1 - 7).
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int hkdfFinalBits(HKDFContext *context, uint8_t ikm_bits,
                  unsigned int ikm_bit_count)
{
  if (!context) return shaNull;
  if (context->Corrupted) return context->Corrupted;
  if (context->Computed) return context->Corrupted = shaStateError;
  return hmacFinalBits(&context->hmacContext, ikm_bits, ikm_bit_count);
}

/*
 * hkdfResult
 *
 * Описание
 *   Завершает создание HKDF и выполняет финальное извлечение HKDF.
 *
 * Параметры
 *   context: [in/out] - контекст HKDF для расчёта хэша HKDF.
 *   prk[ ]: [out]
 *     необязательный массив для записи HKDF, NULL или указатель на
 *     буфер, который должен быть больше USHAHashSize(whichSha)
 *   info[ ]: [in]
 *     необязательный контекст и зависящая от приложения информация.
 *     Игнорируется, если info == NULL или задана пустая строка.
 *   info_len: [in]
 *     размер необязательного контекста или зависящей от приложения
 *     информации (игнорируется при info == NULL).
 *   okm[ ]: [out] - массив для записи HKDF.
 *   okm_len: [in]
 *         Размер буфера для okm. Должно быть
 *         okm_len <= 255 * USHABlockSize(whichSha)
 *
 *  Возвращаемое значение
 *   Код ошибки sha.
 */
int hkdfResult(HKDFContext *context,
               uint8_t prk[USHAMaxHashSize],
               const unsigned char *info, int info_len,
               uint8_t okm[ ], int okm_len)
{
  uint8_t prkbuf[USHAMaxHashSize];
  int ret;

  if (!context) return shaNull;
  if (context->Corrupted) return context->Corrupted;
  if (context->Computed) return context->Corrupted = shaStateError;
  if (!okm) return context->Corrupted = shaBadParam;
  if (!prk) prk = prkbuf;

  ret = hmacResult(&context->hmacContext, prk) ||
        hkdfExpand(context->whichSha, prk, context->hashSize, info,
                   info_len, okm, okm_len);
  context->Computed = 1;
  return context->Corrupted = ret;
}
```

8.5. Тестовый драйвер

Ниже приведён код тестового драйвера для проверки программ из файлов sha1.c, sha224-256.c, sha384-512.c, hmac.c, hkdf.c. Драйвер может также служить программой для генерации хэш-значений. Отметим, что тесты предполагают символы в кодировке [US-ASCII] и в процессе работы будут выдаваться предупреждения, если код был скомпилирован с иной кодировкой символов.

См. также [SHAVS].

```
/************************* shatest.c ************************/
/***************** См. детали в RFC 6234. *******************/
/* Авторские права (Copyright (c) 2011) принадлежат IETF Trust */
/* и лицам, указанным как авторы кода. Все права защищены.     */
/* Распространение описано в файле sha.h.                      */
/*
 *  Описание
 *    Этот файл проверяет исполнение кода SHA по трём тестам из
 *    FIPS PUB 180-3 (http://csrc.nist.gov/publications/fips/fips180-2/
 *    fips180-2withchangenotice.pdf), один из которых вызывает
 *    SHAInput со значением, кратным 512 битам - 7 проверок, заданных
 *    для каждого алгоритма в "The Secure Hash Algorithm Validation
 *    System (SHAVS)" (http://csrc.nist.gov/cryptval/shs/SHAVS.pdf),
 *    три из которых выполняются на уровне битов.
 *
 *    Позднее эти тесты были перенесены на страницы, указанные в
 *    http://csrc.nist.gov/groups/ST/toolkit/examples.html
 *
 *    Этот файл использует код HMAC SHA1 для 7 тестов, документированных
 *    в [RFC 2202] и [RFC 4231].
 *
 *    Файл использует код HKDF  для 7 тестов из RFC 5869[1].
 *
 *    Для запуска тестов с выводом лишь результата PASSED/FAILED
 *    служит опция -p. Другие варианты тестов включают:
 *      хэширование произвольной строки;
 *      хэширование содержимого файла;
 *      несколько проверок на наличие ошибок;
 *      вывод результатов в необработанном (raw) виде.
 *
 *  Переносимость
 *    Нет.
 */

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>     /* определения getopt() и optarg */
#include "sha.h"

static int scasecmp(const char *s1, const char *s2);

/*
 *  Шаблоны тестирования
 */
#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"                                /* 1000000 раз */
#define TEST4a   "01234567012345670123456701234567"
#define TEST4b   "01234567012345670123456701234567"
    /* Размер, кратный 512 битам */
#define TEST4    TEST4a TEST4b                      /* 10 раз */
#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"
```

[1] В оригинале ошибочно указано RFC 4869. См. https://errata.rfc-editor.org/eid5179/. *Прим. перев.*

```
#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 HKDFTESTCOUNT 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)

/* Тестовые массивы для хэширования. */
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"
    }
  }
};

/* Тестовые массивы для 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"
    }, { 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"
    }, { 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"
      /* "Тест с ключом больше размера блока и данных. Такой ключ "
         "нужно хэшировать перед использованием в алгоритме HMAC." */
    }, { 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 }
  }
};

/* Тестовые массивы для HKDF. */
struct hkdfhash {
    SHAversion whichSha;
    int ikmlength;
    const char *ikmarray;
    int saltlength;
    const char *saltarray;
    int infolength;
    const char *infoarray;
    int prklength;
    const char *prkarray;
    int okmlength;
    const char *okmarray;
} hkdfhashes[HKDFTESTCOUNT] = {
    {   /* RFC 5869 A.1. Test Case 1 */
        SHA256,
        22, "\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b"
            "\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b",
        13, "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c",
        10, "\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9",
        32, "077709362C2E32DF0DDC3F0DC47BBA6390B6C73BB50F9C3122EC844A"
            "D7C2B3E5",
        42, "3CB25F25FAACD57A90434F64D0362F2A2D2D0A90CF1A5A4C5DB02D56"
            "ECC4C5BF34007208D5B887185865"
    },
    {   /* RFC 5869 A.2. Test Case 2 */
        SHA256,
        80, "\x00\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\x1a\x1b"
            "\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29"
            "\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37"
            "\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45"
            "\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f",
        80, "\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d"
            "\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b"
            "\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89"
            "\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97"
            "\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5"
            "\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf",
        80, "\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd"
            "\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb"
            "\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9"
            "\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7"
            "\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5"
            "\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff",
        32, "06A6B88C5853361A06104C9CEB35B45C"
            "EF760014904671014A193F40C15FC244",
        82, "B11E398DC80327A1C8E7F78C596A4934"
            "4F012EDA2D4EFAD8A050CC4C19AFA97C"
            "59045A99CAC7827271CB41C65E590E09"
            "DA3275600C2F09B8367793A9ACA3DB71"
            "CC30C58179EC3E87C14C01D5C1F3434F"
            "1D87"
    },
    {   /* RFC 5869 A.3. Test Case 3 */
        SHA256,
        22, "\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b"
            "\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b",
        0, "",
        0, "",
        32, "19EF24A32C717B167F33A91D6F648BDF"
            "96596776AFDB6377AC434C1C293CCB04",
        42, "8DA4E775A563C18F715F802A063C5A31"
            "B8A11F5C5EE1879EC3454E5F3C738D2D"
            "9D201395FAA4B61A96C8"
    },
    {   /* RFC 5869 A.4. Test Case 4 */
        SHA1,
```

```
        11, "\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b",
        13, "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c",
        10, "\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9",
        20, "9B6C18C432A7BF8F0E71C8EB88F4B30BAA2BA243",
        42, "085A01EA1B10F36933068B56EFA5AD81"
            "A4F14B822F5B091568A9CDD4F155FDA2"
            "C22E422478D305F3F896"
    },
    {   /* RFC 5869 A.5. Test Case 5 */
        SHA1,
        80, "\x00\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\x1a\x1b"
            "\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29"
            "\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37"
            "\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45"
            "\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f",
        80, "\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6A\x6B\x6C\x6D"
            "\x6E\x6F\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7A\x7B"
            "\x7C\x7D\x7E\x7F\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89"
            "\x8A\x8B\x8C\x8D\x8E\x8F\x90\x91\x92\x93\x94\x95\x96\x97"
            "\x98\x99\x9A\x9B\x9C\x9D\x9E\x9F\xA0\xA1\xA2\xA3\xA4\xA5"
            "\xA6\xA7\xA8\xA9\xAA\xAB\xAC\xAD\xAE\xAF",
        80, "\xB0\xB1\xB2\xB3\xB4\xB5\xB6\xB7\xB8\xB9\xBA\xBB\xBC\xBD"
            "\xBE\xBF\xC0\xC1\xC2\xC3\xC4\xC5\xC6\xC7\xC8\xC9\xCA\xCB"
            "\xCC\xCD\xCE\xCF\xD0\xD1\xD2\xD3\xD4\xD5\xD6\xD7\xD8\xD9"
            "\xDA\xDB\xDC\xDD\xDE\xDF\xE0\xE1\xE2\xE3\xE4\xE5\xE6\xE7"
            "\xE8\xE9\xEA\xEB\xEC\xED\xEE\xEF\xF0\xF1\xF2\xF3\xF4\xF5"
            "\xF6\xF7\xF8\xF9\xFA\xFB\xFC\xFD\xFE\xFF",
        20, "8ADAE09A2A307059478D309B26C4115A224CFAF6",
        82, "0BD770A74D1160F7C9F12CD5912A06EB"
            "FF6ADCAE899D92191FE4305673BA2FFE"
            "8FA3F1A4E5AD79F3F334B3B202B2173C"
            "486EA37CE3D397ED034C7F9DFEB15C5E"
            "927336D0441F4C4300E2CFF0D0900B52"
            "D3B4"
    },
    {   /* RFC 5869 A.6. Test Case 6 */
        SHA1,
        22, "\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b"
            "\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b",
        0, "",
        0, "",
        20, "DA8C8A73C7FA77288EC6F5E7C297786AA0D32D01",
        42, "0AC1AF7002B3D761D1E55298DA9D0506"
            "B9AE52057220A306E07B6B87E8DF21D0"
            "EA00033DE03984D34918"
    },
    {   /* RFC 5869 A.7. Test Case 7. */
        SHA1,
        22, "\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c"
            "\x0c\x0c\x0c\x0c\x0c\x0c\x0c\x0c",
        0, 0,
        0, "",
        20, "2ADCCADA18779E7C2077AD2EB19D3F3E731385DD",
        42, "2C91117204D745F3500D636A62F64F0A"
            "B3BAE548AA53D423B0D1F27EBBA6F5E5"
            "673A081D70CCE7ACFC48"
    }
};

/*
 * Сравнение хэш-значения с ожидаемой шестнадцатеричной строкой
 */
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;
}

/*
 * Вызов строки с представлением непечатных символов точкой.
 */
void printstr(const char *str, int len)
{
  for ( ; len-- > 0; str++)
    putchar(isprint((unsigned char)*str) ? *str : '.');
}
```

```
/*
 * Вывод строки с преобразованием всех символов в hex "## ".
 */
void printxstr(const char *str, int len)
{
  char *sep = "";
  for ( ; len-- > 0; str++) {
    printf("%s%c%c", sep, hexdigits[(*str >> 4) & 0xF],
      hexdigits[*str & 0xF]);
    sep = " ";
  }
}

/*
 * Вывод сообщения о параметрах и опциях.
 */
void usage(const char *argv0)
{
  fprintf(stderr,
    "Usage:\n"
    "Common options: [-h hash] [-w|-x|-6] [-H]\n"
    "Hash a string:\n"
      "\t%s [-S expectedresult] -s hashstr [-k key] "
           "[-i info -L okm-len]\n"
    "Hash a file:\n"
      "\t%s [-S expectedresult] -f file [-k key] "
           "[-i info -L okm-len]\n"
    "Hash a file, ignoring whitespace:\n"
      "\t%s [-S expectedresult] -F file [-k key] "
           "[-i info -L okm-len]\n"
    "Additional bits to add in: [-B bitcount -b bits]\n"
    "(If -k,-i&-L are used, run HKDF-SHA###.\n"
    " If -k is used, but not -i&-L, run HMAC-SHA###.\n"
    " Otherwise, run SHA###.)\n"
    "Standard tests:\n"
      "\t%s [-m | -d] [-l loopcount] [-t test#] [-e]\n"
      "\t\t[-r randomseed] [-R randomloop-count] "
        "[-p] [-P|-X]\n"
    "-h\thash to test: "
      "0|SHA1, 1|SHA224, 2|SHA256, 3|SHA384, 4|SHA512\n"
    "-m\tperform hmac standard tests\n"
    "-k\tkey for hmac test\n"
    "-d\tperform hkdf standard tests\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"
    "-6\toutput hash in base64 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);
}

/*
 * Вывод результата и 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) {
    printf("\nhashsize=%d\n", hashsize);
    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');
  } else if (printResults == PRINTBASE64) {
    unsigned char b;
    char *sm = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
               "0123456789+/";
    for (i = 0; i < hashsize; i += 3) {
      putchar(sm[Message_Digest[i] >> 2]);
      b = (Message_Digest[i] & 0x03) << 4;
      if (i+1 < hashsize) b |= Message_Digest[i+1] >> 4;
      putchar(sm[b]);
      if (i+1 < hashsize) {
        b = (Message_Digest[i+1] & 0x0f) << 2;
        if (i+2 < hashsize) b |= Message_Digest[i+2] >> 6;
        putchar(sm[b]);
      } else putchar('=');
      if (i+2 < hashsize) putchar(sm[Message_Digest[i+2] & 0x3f]);
      else putchar('=');
    }
    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");
  }
}

/*
 * Выполнение последовательности хэш-функций. На входе testarray
 * с повторением repeatcount раз, затем extrabits. Если результат
 * известен, он будет в resultarray (шестнадцатеричные цифры).
 */
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 unsigned char *info, int infolen, int okmlen,
  const char *resultarray, int hashsize, int printResults,
  int printPassFail)
{
  USHAContext sha;
  HMACContext hmac;
  HKDFContext hkdf;
  int err, i;
  uint8_t Message_Digest_Buf[USHAMaxHashSize];
  uint8_t *Message_Digest = Message_Digest_Buf;
  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);
  }

  if (info) Message_Digest = malloc(okmlen);
  memset(&sha, '\343', sizeof(sha)); /* force bad data into struct */
  memset(&hmac, '\343', sizeof(hmac));
  memset(&hkdf, '\343', sizeof(hkdf));

  err = info ? hkdfReset(&hkdf, hashes[hashno].whichSha,
                         keyarray, keylen) :
        keyarray ? hmacReset(&hmac, hashes[hashno].whichSha,
                         keyarray, keylen) :
                   USHAReset(&sha, hashes[hashno].whichSha);
  if (err != shaSuccess) {
    fprintf(stderr, "hash(): %sReset Error %d.\n",
            info ? "hkdf" : keyarray ? "hmac" : "sha", err);
    return err;
  }
```

```
  for (i = 0; i < repeatcount; ++i) {
    err = info ? hkdfInput(&hkdf, (const uint8_t *)testarray, length) :
          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",
              info ? "hkdf" : keyarray ? "hmac" : "sha", err);
      return err;
    }
  }

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

  err = info ? hkdfResult(&hkdf, 0, info, infolen,
                          Message_Digest, okmlen) :
        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",
      info ? "hkdf" : keyarray ? "hmac" : "sha", err);
    return err;
  }

  sprintf(buf, "%d", testno+1);
  printResult(Message_Digest, info ? okmlen : hashsize,
    hashes[hashno].name, info ? "hkdf standard test" :
    keyarray ? "hmac standard test" : "sha standard test", buf,
    resultarray, printResults, printPassFail);

  return err;
}

/*
 * Выполнение последовательности HKDF. На входе testarray
 * с повторением repeatcount раз, затем extrabits. Если результат
 * известен, об будет в resultarray (шестнадцатеричные цифры).
 */
int hashHkdf(int testno, int loopno, int hashno,
  int printResults, int printPassFail)
{
  int err;
  unsigned char prk[USHAMaxHashSize+1];
  uint8_t okm[255 * USHAMaxHashSize+1];
  char buf[20];

  if (printResults == PRINTTEXT) {
    printf("\nTest %d: Iteration %d\n\tSALT\t'", testno+1, loopno);
    printxstr(hkdfhashes[testno].saltarray,
      hkdfhashes[testno].saltlength);
    printf("'\n\tIKM\t'");
    printxstr(hkdfhashes[testno].ikmarray,
      hkdfhashes[testno].ikmlength);
    printf("'\n\tINFO\t'");
    printxstr(hkdfhashes[testno].infoarray,
      hkdfhashes[testno].infolength);
    printf("'\n");
    printf("    L=%d bytes\n", hkdfhashes[testno].okmlength);
  }

  /* Вызов hkdf() для тестовых векторов */
  err = hkdf(hkdfhashes[testno].whichSha,
             (const uint8_t *) hkdfhashes[testno].saltarray,
             hkdfhashes[testno].saltlength,
             (const uint8_t *) hkdfhashes[testno].ikmarray,
             hkdfhashes[testno].ikmlength,
             (const uint8_t *) hkdfhashes[testno].infoarray,
             hkdfhashes[testno].infolength, okm,
             hkdfhashes[testno].okmlength);
  if (err != shaSuccess) {
    fprintf(stderr, "hashHkdf(): hkdf Error %d.\n", err);
    return err;
```

```
  }
  sprintf(buf, "hkdf %d", testno+1);
  printResult(okm, hkdfhashes[testno].okmlength,
    USHAHashName(hkdfhashes[testno].whichSha), "hkdf standard test",
    buf, hkdfhashes[testno].okmarray, printResults, printPassFail);

  /* Вызов hkdfExtract() с тестовыми векторами для проверки */
  /* промежуточных результатов. */
  err = hkdfExtract(hkdfhashes[testno].whichSha,
                    (const uint8_t *) hkdfhashes[testno].saltarray,
                    hkdfhashes[testno].saltlength,
                    (const uint8_t *) hkdfhashes[testno].ikmarray,
                    hkdfhashes[testno].ikmlength, prk);
  if (err != shaSuccess) {
    fprintf(stderr, "hashHkdf(): hkdfExtract Error %d.\n", err);
    return err;
  }
  sprintf(buf, "hkdfExtract %d", testno+1);
  printResult(prk, USHAHashSize(hkdfhashes[testno].whichSha),
    USHAHashName(hkdfhashes[testno].whichSha), "hkdf standard test",
    buf, hkdfhashes[testno].prkarray, printResults, printPassFail);

  /* Вызов hkdfExpand() с тестовыми векторами для проверки */
  /* промежуточных результатов из hkdfExtract. */
  err = hkdfExpand(hkdfhashes[testno].whichSha, prk,
    USHAHashSize(hkdfhashes[testno].whichSha),
    (const uint8_t *)hkdfhashes[testno].infoarray,
    hkdfhashes[testno].infolength, okm, hkdfhashes[testno].okmlength);
  if (err != shaSuccess) {
    fprintf(stderr, "hashHkdf(): hkdfExpand Error %d.\n", err);
    return err;
  }
  sprintf(buf, "hkdfExpand %d", testno+1);
  printResult(okm, hkdfhashes[testno].okmlength,
    USHAHashName(hkdfhashes[testno].whichSha), "hkdf standard test",
    buf, hkdfhashes[testno].okmarray, printResults, printPassFail);

  return err;
}

/*
 * Выполнение последовательности хэш-функций с именем файла на входе.
 * Если результат известен, он будет в resultarray (16-ричные цифры).
 */
int hashfile(int hashno, const char *hashfilename, int bits,
  int bitcount, int skipSpaces, const unsigned char *keyarray,
  int keylen, const unsigned char *info, int infolen, int okmlen,
  const char *resultarray, int hashsize,
  int printResults, int printPassFail)
{
  USHAContext sha;
  HMACContext hmac;
  HKDFContext hkdf;
  int err, nread, c;
  unsigned char buf[4096];
  uint8_t Message_Digest_Buf[USHAMaxHashSize];
  uint8_t *Message_Digest = Message_Digest_Buf;
  unsigned char cc;
  FILE *hashfp = (strcmp(hashfilename, "-") == 0) ? stdin :
    fopen(hashfilename, "r");

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

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

  if (skipSpaces)
    while ((c = getc(hashfp)) != EOF) {
      if (!isspace(c)) {
        cc = (unsigned char)c;
        err = info ? hkdfInput(&hkdf, &cc, 1) :
```

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

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

  err = info ? hkdfResult(&hkdf, 0, info, infolen,
                          Message_Digest, okmlen) :
        keyarray ? hmacResult(&hmac, Message_Digest) :
                   USHAResult(&sha, Message_Digest);
  if (err != shaSuccess) {
    fprintf(stderr, "hashfile(): %s Error %d.\n",
            info ? "hkdf" : keyarray ? "hmacResult" :
            "shaResult", err);
    if (hashfp != stdin) fclose(hashfp);
    return err;
  }

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

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

/*
 * Выполнение последовательности хэш-функций с перестановками. На входе
 * затравка (seed) повторённая трижды. При 1000 раундов входными
 * данными служат 3 предыдущих результата. Результат проверяется
 * и служит затравкой (seed) для следующего цикла. Если результат
 * известен, он будет в resultarrays (шестнадцатеричные цифры).
 */
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];

  /* На входе затравка (seed) — случайное значение размером n битов */
  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)); /* Внесение ошибки в структуру */
      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);
  }
}

/*
 * Просмотр имени хэша.
 */
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;
}

/*
 * Запуск некоторых тестов, которые должны вызвать ошибки.
 */
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");
  }
}

/* Замена шестнадцатеричной строки её значением */
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((int)(unsigned char)(*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((int)(unsigned char)(*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;
  int runHkdfTests = 0;
  char *hmacKey = 0;
  int hmaclen = 0;
  char *info = 0;
  int infolen = 0, okmlen = 0;
  int randomcount = RANDOMCOUNT;
  const char *hashfilename = 0;
  const char *hashFilename = 0;
  int extrabits = 0, numberExtrabits = 0;
  int strIsHex = 0;

  if ('A' != 0x41) {
    fprintf(stderr, "%s: these tests require ASCII\n", argv[0]);
  }

  while ((i = getopt(argc, argv,
                     "6b:B:def:F:h:i:Hk:l:L:mpPr:R:s:S:t:wxX")) != -1)
    switch (i) {
      case 'b': extrabits = strtol(optarg, 0, 0); break;
      case 'B': numberExtrabits = atoi(optarg); break;
      case 'd': runHkdfTests = 1; break;
      case 'e': checkErrors = 1; break;
      case 'f': hashfilename = optarg; break;
      case 'F': hashFilename = optarg; break;
      case 'h': hashnolow = hashnohigh = findhash(argv[0], optarg);
        break;
      case 'H': strIsHex = 1; break;
      case 'i': info = optarg; infolen = strlen(optarg); break;
      case 'k': hmacKey = optarg; hmaclen = strlen(optarg); break;
      case 'l': loopnohigh = atoi(optarg); break;
      case 'L': okmlen = strtol(optarg, 0, 0); break;
      case 'm': runHmacTests = 1; break;
      case 'P': printPassFail = 0; break;
```

```
      case 'p': printResults = PRINTNONE; break;
      case 'R': randomcount = atoi(optarg); break;
      case 'r': randomseedstr = optarg; break;
      case 's': hashstr = optarg; hashlen = strlen(hashstr); break;
      case 'S': resultstr = optarg; break;
      case 't': testnolow = ntestnohigh = atoi(optarg) - 1; break;
      case 'w': printResults = PRINTRAW; break;
      case 'x': printResults = PRINTHEX; break;
      case 'X': printPassFail = 2; break;
      case '6': printResults = PRINTBASE64; break;
      default: usage(argv[0]);
      }

  if (strIsHex) {
    hashlen = unhexStr(hashstr);
    unhexStr(randomseedstr);
    hmaclen = unhexStr(hmacKey);
    infolen = unhexStr(info);
  }
  testnohigh = (ntestnohigh != 0) ? ntestnohigh:
               runHmacTests ? (HMACTESTCOUNT-1) :
               runHkdfTests ? (HKDFTESTCOUNT-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) ||
      (info && ((infolen <= 0) || (okmlen <= 0))) ||
      (info && !hmacKey))
    usage(argv[0]);

  /*
   *  Выполнение тестов SHA/HMAC.
   */
  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, (const uint8_t *) info, infolen, okmlen, 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,
                       (const uint8_t *) info, infolen, okmlen,
                       resultstr, hashes[hashno].hashsize,
                       printResults, printPassFail);

      else if (hashFilename)
        err = hashfile(hashno, hashFilename, extrabits,
                       numberExtrabits, 1,
                       (const unsigned char *)hmacKey, hmaclen,
                       (const uint8_t *) info, infolen, okmlen,
                       resultstr, hashes[hashno].hashsize,
                       printResults, printPassFail);

      else /* Стандартные тесты */ {
        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],
                        0, 0, 0,
                        hmachashes[testno].resultarray[hashno],
                        hmachashes[testno].resultlength[hashno],
                        printResults, printPassFail);
          } else if (runHkdfTests) {
            err = hashHkdf(testno, loopno, hashno,
                        printResults, printPassFail);
          } else { /* sha tests */
            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, 0, 0, 0,
                        hashes[hashno].tests[testno].resultarray,
                        hashes[hashno].hashsize,
                        printResults, printPassFail);
          }
        }
        if (!runHmacTests && !runHkdfTests) {
          randomtest(hashno, hashes[hashno].randomtest,
            hashes[hashno].hashsize, hashes[hashno].randomresults,
            RANDOMCOUNT, printResults, printPassFail);
        }
      }
    }
  }

  /* Тесты с ошибками */
  if (checkErrors) {
    testErrors(hashnolow, hashnohigh, printResults, printPassFail);
  }

  return 0;
}

/*
 * Сравнение двух строк без учёта регистра.
 * Эквивалент strcasecmp() в некоторых системах.
 */
int scasecmp(const char *s1, const char *s2)
{
  for (;;) {
    char u1 = tolower((int)(unsigned char)(*s1++));
    char u2 = tolower((int)(unsigned char)(*s2++));
    if (u1 != u2)
      return u1 - u2;
    if (u1 == '\0')
      return 0;
   }
}
```

9. Вопросы безопасности

Этот документ предоставляет сообществу Internet удобный доступ к открытому исходному коду алгоритмов безопасного хэширования SHA [FIPS 180-2], HMAC на основе их и HKDF. Авторы не делают заявлений о безопасности этих функций для какого-либо применения.

Вопросы безопасности SHA-1 рассмотрены в [RFC6194].

10. Благодарности

Спасибо за исправления [RFC4634], представленные Alfred Hoenes и Jan Andres, а также за комментарии Alfred Hoenes к этому документу.

Спасибо James Carlson, Russ Housley, Tero Kivinen, Juergen Quittek, Sean Turner за комментарии, приведшие к улучшению этого документа.

11. Литература

11.1. Нормативные документы

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

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, May 2010.

[SHS] "Secure Hash Standard", United States of American, National Institute of Science and Technology, Federal Information Processing Standard (FIPS) 180-3, http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf.

[US-ASCII] ANSI, "USA Standard Code for Information Interchange", X3.4, American National Standards Institute: New York, 1968.

11.2. Дополнительная литература

[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.

[RFC4055] Schaad, J., Kaliski, B., and R. Housley, "Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 4055, June 2005.

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

[RFC4634] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)", RFC 4634, July 2006.

[RFC6194] Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, March 2011.

[SHAVS] "The Secure Hash Algorithm Validation System (SHAVS)", http://csrc.nist.gov/groups/STM/cavp/documents/shs/SHAVS.pdf, July 2004.

Приложение. Отличия от RFC 4634

Ниже перечислены изменения, внесённые в RFC 4634 для создания этого документа

1 Добавлен код и краткое описание HKDF со ссылкой на [RFC5869].

2 Исправлено множество ошибок [RFC4634], указанных ниже (отметим, что старый код не приводил к некорректным значениям хэша).

2.a Некоторые некорректно возвращаемые коды ошибки shaNull заменены на shaInputTooLong.

2.b Обновлены комментарии и переменные в коде для согласованности и ясности, а также внесены другие редакторские правки.

2.c Прежний код для SHA-384 и SHA-512 останавливался после 2^93 байтов (2^96 битов), исправленный обрабатывает до 2^125 байтов (2^128 битов).

2.d Включена дополнительная проверка ошибок, включая проверку при работе тестового драйвера для обнаружения попыток запустить драйвер после компиляции с набором символов, отличным от [US-ASCII].

3 Обновлён шаблон, исключена специальная лицензия [RFC4634], поскольку новый шаблон требует упрощённую лицензию BSD.

4 Версия getopt, распространяемая по лицензии MIT, заменена новым кодом, соответствующим ограничениям лицензий IETF.

5 Добавлены ссылки на [RFC6194].

6 Внесены другие редакционные улучшения.

Адреса авторов

Donald Eastlake
Huawei
155 Beaver Street
Milford, MA 01757 USA
Telephone: +1-508-333-2270
EMail: d3e3e3@gmail.com

Tony Hansen
AT&T Laboratories
200 Laurel Ave.
Middletown, NJ 07748 USA
Telephone: +1-732-420-8934
EMail: tony+shs@maillennium.att.com

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru