NAME

Digest::SHA - Perl extension for SHA-1/224/256/384/512

SYNOPSIS

In programs:

```
 # Functional interface

use Digest::SHA qw(sha1 sha1_hex sha1_base64 ...);

$digest = sha1($data);
$digest = sha1_hex($data);
$digest = sha1_base64($data);

$digest = sha256($data);
$digest = sha384_hex($data);
$digest = sha512_base64($data);

 # Object-oriented

use Digest::SHA;

$sha = Digest::SHA->new($alg);

$sha->add($data);  # feed data into stream

$sha->addfile(*F);
$sha->addfile($filename);

$sha->add_bits($bits);
$sha->add_bits($data, $nbits);

$sha_copy = $sha->clone; # if needed, make copy of
$sha->dump($file);  # current digest state,
$sha->load($file);  # or save it on disk

$digest = $sha->digest;  # compute digest
$digest = $sha->hexdigest;
$digest = $sha->b64digest;
```

From the command line:

```
$ shasum files

$ shasum --help
```

SYNOPSIS (HMAC-SHA)

```
 # Functional interface only

use Digest::SHA qw(hmac_sha1 hmac_sha1_hex ...);
```

```
$digest = hmac_sha1($data, $key);
$digest = hmac_sha224_hex($data, $key);
$digest = hmac_sha256_base64($data, $key);
```

ABSTRACT

Digest::SHA is a complete implementation of the NIST Secure Hash Standard. It gives Perl programmers a convenient way to calculate SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256 message digests. The module can handle all types of input, including partial-byte data.

DESCRIPTION

Digest::SHA is written in C for speed. If your platform lacks a C compiler, you can install the functionally equivalent (but much slower) *Digest::SHA::PurePerl* module.

The programming interface is easy to use: it's the same one found in CPAN's *Digest* module. So, if your applications currently use *Digest::MD5* and you'd prefer the stronger security of SHA, it's a simple matter to convert them.

The interface provides two ways to calculate digests: all-at-once, or in stages. To illustrate, the following short program computes the SHA-256 digest of "hello world" using each approach:

```
use Digest::SHA qw(sha256_hex);

$data = "hello world";
@frags = split(//, $data);

# all-at-once (Functional style)
$digest1 = sha256_hex($data);

# in-stages (OOP style)
$state = Digest::SHA->new(256);
for (@frags) { $state->add($_) }
$digest2 = $state->hexdigest;

print $digest1 eq $digest2 ?
  "whew!\n" : "oops!\n";
```

To calculate the digest of an n-bit message where *n* is not a multiple of 8, use the *add_bits()* method. For example, consider the 446-bit message consisting of the bit-string "110" repeated 148 times, followed by "11". Here's how to display its SHA-1 digest:

```
use Digest::SHA;
$bits = "110" x 148 . "11";
$sha = Digest::SHA->new(1)->add_bits($bits);
print $sha->hexdigest, "\n";
```

Note that for larger bit-strings, it's more efficient to use the two-argument version *add_bits($data, $nbits)*, where *$data* is in the customary packed binary format used for Perl strings.

The module also lets you save intermediate SHA states to disk, or display them on standard output. The *dump()* method generates portable, human-readable text describing the current state of computation. You can subsequently retrieve the file with *load()* to resume where the calculation left off.

To see what a state description looks like, just run the following:

```
use Digest::SHA;
Digest::SHA->new->add("Shaw" x 1962)->dump;
```

As an added convenience, the Digest::SHA module offers routines to calculate keyed hashes using the HMAC-SHA-1/224/256/384/512 algorithms. These services exist in functional form only, and mimic the style and behavior of the *sha()*, *sha_hex()*, and *sha_base64()* functions.

```
# Test vector from draft-ietf-ipsec-ciph-sha-256-01.txt


use Digest::SHA qw(hmac_sha256_hex);
print hmac_sha256_hex("Hi There", chr(0x0b) x 32), "\n";
```

UNICODE AND SIDE EFFECTS

Perl supports Unicode strings as of version 5.6. Such strings may contain wide characters, namely, characters whose ordinal values are greater than 255. This can cause problems for digest algorithms such as SHA that are specified to operate on sequences of bytes.

The rule by which Digest::SHA handles a Unicode string is easy to state, but potentially confusing to grasp: the string is interpreted as a sequence of byte values, where each byte value is equal to the ordinal value (viz. code point) of its corresponding Unicode character. That way, the Unicode string 'abc' has exactly the same digest value as the ordinary string 'abc'.

Since a wide character does not fit into a byte, the Digest::SHA routines croak if they encounter one. Whereas if a Unicode string contains no wide characters, the module accepts it quite happily. The following code illustrates the two cases:

```
$str1 = pack('U*', (0..255));
print sha1_hex($str1);  # ok


$str2 = pack('U*', (0..256));
print sha1_hex($str2);  # croaks
```

Be aware that the digest routines silently convert UTF-8 input into its equivalent byte sequence in the native encoding (cf. utf8::downgrade). This side effect influences only the way Perl stores the data internally, but otherwise leaves the actual value of the data intact.

NIST STATEMENT ON SHA-1

NIST acknowledges that the work of Prof. Xiaoyun Wang constitutes a practical collision attack on SHA-1. Therefore, NIST encourages the rapid adoption of the SHA-2 hash functions (e.g. SHA-256) for applications requiring strong collision resistance, such as digital signatures.

ref. *http://csrc.nist.gov/groups/ST/hash/statement.html*

PADDING OF BASE64 DIGESTS

By convention, CPAN Digest modules do **not** pad their Base64 output. Problems can occur when feeding such digests to other software that expects properly padded Base64 encodings.

For the time being, any necessary padding must be done by the user. Fortunately, this is a simple operation: if the length of a Base64-encoded digest isn't a multiple of 4, simply append "=" characters to the end of the digest until it is:

```
while (length($b64_digest) % 4) {
 $b64_digest .= '=';
}
```

To illustrate, *sha256_base64("abc")* is computed to be

```
ungWv48Bz+pBQUDeXa4iI7ADYaOWF3qctBD/YfIAFa0
```

which has a length of 43. So, the properly padded version is

```
ungWv48Bz+pBQUDeXa4iI7ADYaOWF3qctBD/YfIAFa0=
```

EXPORT

None by default.

EXPORTABLE FUNCTIONS

Provided your C compiler supports a 64-bit type (e.g. the *long long* of C99, or *__int64* used by Microsoft C/C++), all of these functions will be available for use. Otherwise, you won't be able to perform the SHA-384 and SHA-512 transforms, both of which require 64-bit operations.

Functional style

sha1($data, ...)

sha224($data, ...)

sha256($data, ...)

sha384($data, ...)

sha512($data, ...)

sha512224($data, ...)

sha512256($data, ...)

Logically joins the arguments into a single string, and returns its SHA-1/224/256/384/512 digest encoded as a binary string.

sha1_hex($data, ...)

sha224_hex($data, ...)

sha256_hex($data, ...)

sha384_hex($data, ...)

sha512_hex($data, ...)

sha512224_hex($data, ...)

sha512256_hex($data, ...)

Logically joins the arguments into a single string, and returns its SHA-1/224/256/384/512 digest encoded as a hexadecimal string.

sha1_base64($data, ...)

sha224_base64($data, ...)

sha256_base64($data, ...)

sha384_base64($data, ...)

sha512_base64($data, ...)

sha512224_base64($data, ...)

sha512256_base64($data, ...)

Logically joins the arguments into a single string, and returns its SHA-1/224/256/384/512 digest encoded as a Base64 string.

It's important to note that the resulting string does **not** contain the padding characters typical of Base64 encodings. This omission is deliberate, and is done to maintain compatibility with the family of CPAN Digest modules. See *PADDING OF BASE64 DIGESTS* for details.

OOP style

new($alg)

>   Returns a new Digest::SHA object. Allowed values for *$alg* are 1, 224, 256, 384, 512, 512224, or 512256. It's also possible to use common string representations of the algorithm (e.g. "sha256", "SHA-384"). If the argument is missing, SHA-1 will be used by default.

>   Invoking *new* as an instance method will not create a new object; instead, it will simply reset the object to the initial state associated with *$alg*. If the argument is missing, the object will continue using the same algorithm that was selected at creation.

reset($alg)

>   This method has exactly the same effect as *new($alg)*. In fact, *reset* is just an alias for *new*.

hashsize

>   Returns the number of digest bits for this object. The values are 160, 224, 256, 384, 512, 224, and 256 for SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256, respectively.

algorithm

>   Returns the digest algorithm for this object. The values are 1, 224, 256, 384, 512, 512224, and 512256 for SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256, respectively.

clone

>   Returns a duplicate copy of the object.

add($data, ...)

>   Logically joins the arguments into a single string, and uses it to update the current digest state. In other words, the following statements have the same effect:

```
$sha->add("a"); $sha->add("b"); $sha->add("c");
$sha->add("a")->add("b")->add("c");
$sha->add("a", "b", "c");
$sha->add("abc");
```

>   The return value is the updated object itself.

add_bits($data, $nbits)

add_bits($bits)

>   Updates the current digest state by appending bits to it. The return value is the updated object itself.

>   The first form causes the most-significant *$nbits* of *$data* to be appended to the stream. The *$data* argument is in the customary binary format used for Perl strings.

>   The second form takes an ASCII string of "0" and "1" characters as its argument. It's equivalent to

```
$sha->add_bits(pack("B*", $bits), length($bits));
```

>   So, the following two statements do the same thing:

```
$sha->add_bits("111100001010");
$sha->add_bits("\xF0\xA0", 12);
```

addfile(*FILE)

>   Reads from *FILE* until EOF, and appends that data to the current state. The return value is the updated object itself.

addfile($filename [, $mode])

Reads the contents of *$filename*, and appends that data to the current state. The return value is the updated object itself.

By default, *$filename* is simply opened and read; no special modes or I/O disciplines are used. To change this, set the optional *$mode* argument to one of the following values:

```
"b" read file in binary mode

"p" use portable mode

"0" use BITS mode
```

The "p" mode ensures that the digest value of *$filename* will be the same when computed on different operating systems. It accomplishes this by internally translating all newlines in text files to UNIX format before calculating the digest. Binary files are read in raw mode with no translation whatsoever.

The BITS mode ("0") interprets the contents of *$filename* as a logical stream of bits, where each ASCII '0' or '1' character represents a 0 or 1 bit, respectively. All other characters are ignored. This provides a convenient way to calculate the digest values of partial-byte data by using files, rather than having to write programs using the *add_bits* method.

dump($filename)

Provides persistent storage of intermediate SHA states by writing a portable, human-readable representation of the current state to *$filename*. If the argument is missing, or equal to the empty string, the state information will be written to STDOUT.

load($filename)

Returns a Digest::SHA object representing the intermediate SHA state that was previously dumped to *$filename*. If called as a class method, a new object is created; if called as an instance method, the object is reset to the state contained in *$filename*. If the argument is missing, or equal to the empty string, the state information will be read from STDIN.

digest

Returns the digest encoded as a binary string.

Note that the *digest* method is a read-once operation. Once it has been performed, the Digest::SHA object is automatically reset in preparation for calculating another digest value. Call *$sha->clone->digest* if it's necessary to preserve the original digest state.

hexdigest

Returns the digest encoded as a hexadecimal string.

Like *digest*, this method is a read-once operation. Call *$sha->clone->hexdigest* if it's necessary to preserve the original digest state.

This method is inherited if *Digest::base* is installed on your system. Otherwise, a functionally equivalent substitute is used.

b64digest

Returns the digest encoded as a Base64 string.

Like *digest*, this method is a read-once operation. Call *$sha->clone->b64digest* if it's necessary to preserve the original digest state.

This method is inherited if *Digest::base* is installed on your system. Otherwise, a functionally equivalent substitute is used.

It's important to note that the resulting string does **not** contain the padding characters typical of Base64 encodings. This omission is deliberate, and is done to maintain compatibility with the family of CPAN Digest modules. See *PADDING OF BASE64 DIGESTS* for details.

HMAC-SHA-1/224/256/384/512

hmac_sha1($data, $key)

hmac_sha224($data, $key)

hmac_sha256($data, $key)

hmac_sha384($data, $key)

hmac_sha512($data, $key)

hmac_sha512224($data, $key)

hmac_sha512256($data, $key)

> Returns the HMAC-SHA-1/224/256/384/512 digest of *$data*/*$key*, with the result encoded as a binary string. Multiple *$data* arguments are allowed, provided that *$key* is the last argument in the list.

hmac_sha1_hex($data, $key)

hmac_sha224_hex($data, $key)

hmac_sha256_hex($data, $key)

hmac_sha384_hex($data, $key)

hmac_sha512_hex($data, $key)

hmac_sha512224_hex($data, $key)

hmac_sha512256_hex($data, $key)

> Returns the HMAC-SHA-1/224/256/384/512 digest of *$data*/*$key*, with the result encoded as a hexadecimal string. Multiple *$data* arguments are allowed, provided that *$key* is the last argument in the list.

hmac_sha1_base64($data, $key)

hmac_sha224_base64($data, $key)

hmac_sha256_base64($data, $key)

hmac_sha384_base64($data, $key)

hmac_sha512_base64($data, $key)

hmac_sha512224_base64($data, $key)

hmac_sha512256_base64($data, $key)

> Returns the HMAC-SHA-1/224/256/384/512 digest of *$data*/*$key*, with the result encoded as a Base64 string. Multiple *$data* arguments are allowed, provided that *$key* is the last argument in the list.
>
> It's important to note that the resulting string does **not** contain the padding characters typical of Base64 encodings. This omission is deliberate, and is done to maintain compatibility with the family of CPAN Digest modules. See *PADDING OF BASE64 DIGESTS* for details.

SEE ALSO

Digest, *Digest::SHA::PurePerl*

The Secure Hash Standard (Draft FIPS PUB 180-4) can be found at:

http://csrc.nist.gov/publications/drafts/fips180-4/Draft-FIPS180-4_Feb2011.pdf

The Keyed-Hash Message Authentication Code (HMAC):

http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf

AUTHOR

```
Mark Shelor <mshelor@cpan.org>
```

ACKNOWLEDGMENTS

The author is particularly grateful to

```
Gisle Aas
Sean Burke
Chris Carey
Alexandr Ciornii
Jim Doble
Thomas Drugeon
Julius Duque
Jeffrey Friedl
Robert Gilmour
Brian Gladman
Adam Kennedy
Andy Lester
Alex Muntada
Steve Peters
Chris Skiscim
Martin Thurn
Gunnar Wolf
Adam Woodbury
```

"who by trained skill rescued life from such great billows and such thick darkness and moored it in so perfect a calm and in so brilliant a light" - Lucretius

COPYRIGHT AND LICENSE

Copyright (C) 2003-2013 Mark Shelor

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

perlartistic