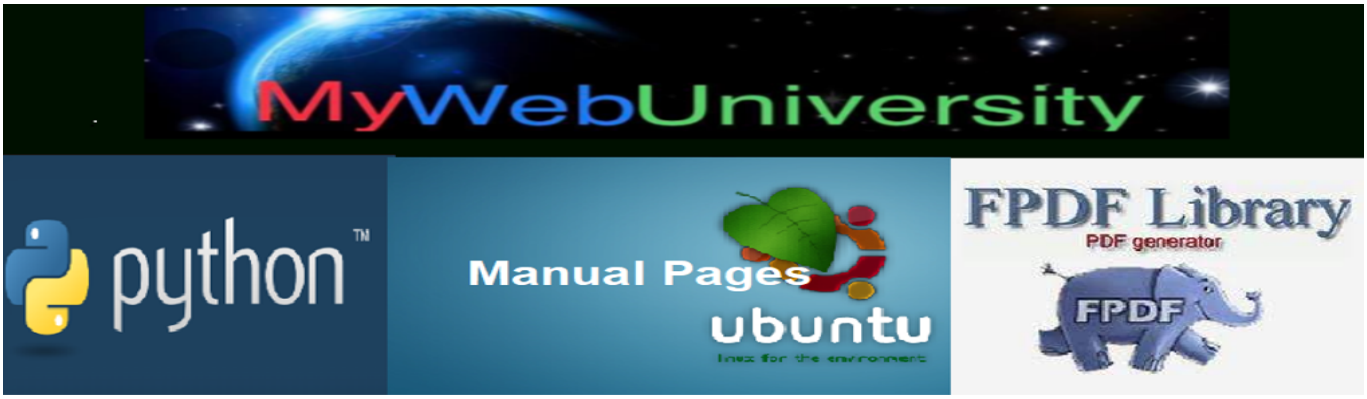




## *Rocky Enterprise Linux 9.2 Manual Pages on command 'Digest::SHA3.3pm'*

***$ man Digest::SHA3.3pm***



NAME

Digest::SHA3 - Perl extension for SHA-3

SYNOPSIS

In programs:

```
# Functional interface

use Digest::SHA3 qw(sha3_224 sha3_256_hex sha3_512_base64 ...);

$digest = sha3_224($data);
$digest = sha3_256_hex($data);
$digest = sha3_384_base64($data);
$digest = sha3_512($data);

# Object-oriented

use Digest::SHA3;

$sha3 = Digest::SHA3->new($alg);
```

```
$sha3->add($data);             # feed data into stream

$sha3->addfile(*F);
$sha3->addfile($filename);

$sha3->add_bits($bits);
$sha3->add_bits($data, $nbits);

$digest = $sha3->digest;       # compute digest
$digest = $sha3->hexdigest;
$digest = $sha3->b64digest;

     # Compute extendable-length digest

$sha3 = Digest::SHA3->new(128000)->add($data);  # SHAKE128
$digest  = $sha3->squeeze;
$digest .= $sha3->squeeze;
...

$sha3 = Digest::SHA3->new(256000)->add($data);  # SHAKE256
$digest  = $sha3->squeeze;
$digest .= $sha3->squeeze;
...
```

## ABSTRACT

Digest::SHA3 is a complete implementation of the NIST SHA-3 cryptographic hash function, as specified in FIPS 202 (SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions).

The module gives Perl programmers a convenient way to calculate SHA3-224, SHA3-256, SHA3-384, and SHA3-512 message digests, as well as variable-length hashes using SHAKE128 and SHAKE256. Digest::SHA3 can handle all types of input, including partial-byte data.

## DESCRIPTION

Digest::SHA3 is written in C for speed. If your platform lacks a C compiler, perhaps you can find the module in a binary form compatible with your particular processor and operating system.

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::SHA and you'd prefer the newer flavor of the NIST standard, 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 SHA3-256 digest of "hello world" using each approach:

```
use Digest::SHA3 qw(sha3_256_hex);

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

# all-at-once (Functional style)
$digest1 = sha3_256_hex($data);

# in-stages (OOP style)
$state = Digest::SHA3->new(256);
for (@frags) { $state->add($_) }
$digest2 = $state->hexdigest;

print $digest1 eq $digest2 ?
      "that's the ticket!\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 SHA3-512 digest:

```
use Digest::SHA3;
$bits = "110" x 148 . "11";
$sha3 = Digest::SHA3->new(512)->add_bits($bits);
print $sha3->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.

## 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-3 that are specified to operate on sequences of bytes.

The rule by which Digest::SHA3 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::SHA3 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:

```
$str1 = pack('U*', (0..255));
print sha3_224_hex($str1);          # ok

$str2 = pack('U*', (0..256));
print sha3_224_hex($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.

## 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, sha3_256_base64("abc") is computed to be

```
Ophdp0/iJbIEXBcta9OQvYVfCG4+nVJbRr/iRRFDFTI
```

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

```
Ophdp0/iJbIEXBcta9OQvYVfCG4+nVJbRr/iRRFDFTI=
```

## 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 any of them.

In the interest of simplicity, maintainability, and small code size, it's unlikely that future versions of this module will support a 32-bit implementation. Older platforms

using 32-bit-only compilers should continue to favor 32-bit hash implementations such as SHA-1, SHA-224, or SHA-256. The desire to use the SHA-3 hash standard, dating from 2015, should reasonably require that one's compiler adhere to programming language standards dating from at least 1999.

Functional style

sha3_224($data, ...)

sha3_256($data, ...)

sha3_384($data, ...)

sha3_512($data, ...)

shake128($data, ...)

shake256($data, ...)

Logically joins the arguments into a single string, and returns its SHA3-0/224/256/384/512 digest encoded as a binary string.

The digest size for shake128 is 1344 bits (168 bytes); for shake256, it's 1088 bits (136 bytes). To obtain extendable-output from the SHAKE algorithms, use the object-oriented interface with repeated calls to the squeeze method.

sha3_224_hex($data, ...)

sha3_256_hex($data, ...)

sha3_384_hex($data, ...)

sha3_512_hex($data, ...)

shake128_hex($data, ...)

shake256_hex($data, ...)

Logically joins the arguments into a single string, and returns its SHA3-0/224/256/384/512 or SHAKE128/256 digest encoded as a hexadecimal string.

sha3_224_base64($data, ...)

sha3_256_base64($data, ...)

sha3_384_base64($data, ...)

sha3_512_base64($data, ...)

shake128_base64($data, ...)

shake256_base64($data, ...)

Logically joins the arguments into a single string, and returns its SHA3-0/224/256/384/512 or SHAKE128/256 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::SHA3 object. Allowed values for $alg are 224, 256, 384, and 512 for the SHA3 algorithms; or 128000 and 256000 for SHAKE128 and SHAKE256, respectively. If the argument is missing, SHA3-224 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 224, 256, 384, 512, 1344, and 1088 for SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, and SHAKE256, respectively.

algorithm

Returns the digest algorithm for this object. The values are 224, 256, 384, 512,

128000, and 256000 for SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, and SHAKE256, 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:

```
$sha3->add("a"); $sha3->add("b"); $sha3->add("c");
$sha3->add("a")->add("b")->add("c");
$sha3->add("a", "b", "c");
$sha3->add("abc");
```

The return value is the updated object itself.

add_bits($data, $nbits [, $lsb])

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. Setting the optional $lsb flag to a true value indicates that the final (partial) byte of $data is aligned with the least-significant bit; by default it's aligned with the most-significant bit, as required by the parent Digest module.

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

```
$sha3->add_bits(pack("B*", $bits), length($bits));
```

So, the following three statements do the same thing:

```
$sha3->add_bits("111100001010");
$sha3->add_bits("\xF0\xA0", 12);
$sha3->add_bits("\xF0\x0A", 12, 1);
```

SHA-3 uses least-significant-bit ordering for its internal operation. This means that

```
$sha3->add_bits("110");
```

is equivalent to

```
$sha3->add_bits("0")->add_bits("1")->add_bits("1");
```

Many public test vectors for SHA-3, such as the Keccak known-answer tests, are delivered in least-significant-bit format. Using the optional $lsb flag in these cases allows your code to be simpler and more efficient. See the test directory for examples.

The fact that SHA-2 and SHA-3 employ opposite bit-ordering schemes has caused noticeable confusion in the programming community. Exercise caution if your code examines individual bits in data streams.

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

"U" use universal newlines

"0" use BITS mode

The "U" mode is modeled on Python's "Universal Newlines" concept, whereby DOS and Mac OS line terminators are converted internally to UNIX newlines before processing. This ensures consistent digest values when working simultaneously across multiple file systems. The "U" mode influences only text files, namely those passing Perl's -T test; binary files are processed 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.

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::SHA3 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.

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.

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.

squeeze

Returns the next 168 (136) bytes of the SHAKE128 (SHAKE256) digest encoded as a binary string. The squeeze method may be called repeatedly to construct digests of any desired length.

This method is applicable only to SHAKE128 and SHAKE256 objects.

## SEE ALSO

Digest, Digest::SHA, Digest::Keccak

The FIPS 202 SHA-3 Standard can be found at:

<http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>

The Keccak/SHA-3 specifications can be found at:

<http://keccak.noekeon.org/Keccak-reference-3.0.pdf>
<http://keccak.noekeon.org/Keccak-submission-3.pdf>

## AUTHOR

Mark Shelor <mshelor@cpan.org>

## ACKNOWLEDGMENTS

The author is particularly grateful to

Guido Bertoni

Joan Daemen

Michael Peeters

Chris Skiscim

Gilles Van Assche

"Nothing is more fatiguing nor, in the long run, more exasperating than the daily effort to believe things which daily become more incredible. To be done with this effort is an indispensible condition of secure and lasting happiness." - Bertrand Russell

## COPYRIGHT AND LICENSE