[HMAC HWIP Technical Specification](https://opentitan.org/book/hw/ip/hmac/index.html" \l "hmac-hwip-technical-specification)

[hmac](https://reports.opentitan.org/hw/ip/hmac/dv/latest/report.html):

[Overview](https://opentitan.org/book/hw/ip/hmac/index.html#overview)

This document specifies HMAC hardware IP functionality. This module conforms to the [OpenTitan guideline for peripheral device functionality.](https://opentitan.org/book/doc/contributing/hw/comportability/index.html) See that document for integration overview within the broader OpenTitan top level system.

[Features](https://opentitan.org/book/hw/ip/hmac/index.html#features)

* Two modes: SHA-2 | HMAC based on SHA-2
* Multiple digest sizes supported (for both modes): SHA-2 256/384/512 hashing algorithm
* Configurable key length 128/256/384/512/1024-bit secret key for HMAC mode
* Support for context switching (via saving and restoring) across multiple message streams
* 32 x 32-bit message FIFO buffer

[Description](https://opentitan.org/book/hw/ip/hmac/index.html#description)

The HMAC module is a [SHA-2](https://csrc.nist.gov/publications/detail/fips/180/4/final) hash-based authentication code generator to check the integrity of an incoming message and a signature signed with the same secret key. It supports SHA-2 256/384/512 and 128/256/384/512/1024-bit keys in HMAC mode, so long as the key length does not exceed the block size of the configured SHA-2 digest size, i.e., 1024-bit keys are not supported for SHA-2 256 where the block size is 512-bit. It generates a different authentication code with the same message if the secret key is different.

This HMAC implementation is not hardened against side channel or fault injection attacks. It is meant purely for hashing acceleration. If hardened MAC operations are required, users should use either [KMAC](https://opentitan.org/book/hw/ip/kmac/index.html) or a software implementation.

The secret key is written in [KEY\_0-KEY\_31](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#key), and the key length relevant to the HMAC operation is configured in [CFG.key\_length](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#cfg--key_length). For example, to use a 256-bit secret key, [CFG.key\_length](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#cfg--key_length) needs to be configured (as per register documentation) and then only the relevant secret key registers, only [KEY\_0-KEY\_7](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#key) in this case, are consumed for the HMAC operation. The digest size required is configured in [CFG.digest\_size](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#cfg--digest_size). The message to authenticate is written to [MSG\_FIFO](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#msg_fifo) and the HMAC generates a 256/384/512-bit digest value (depending on the digest size configuration provided) which can be read from [DIGEST\_0-DIGEST\_7](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#digest) for SHA-2 256, or from [DIGEST\_0-DIGEST\_12](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#digest) for SHA-2 384, or from [DIGEST\_0-DIGEST\_15](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#digest) for SHA-2 512. The hmac\_done interrupt is raised to report to software that the final digest is available.

This module allows software to save and restore the hashing context so that different message streams can be interleaved; please check the [Programmer’s Guide](https://opentitan.org/book/hw/ip/hmac/doc/programmers_guide.html#saving-and-restoring-the-context) for more information.

The HMAC IP can run in SHA-2 only mode, whose purpose is to check the correctness of the received message. The same digest registers above are used to hold the final hash result. SHA-2 mode does not use the given secret key. It generates the same result with the same message every time.

The software does not need to provide the message length. The HMAC IP will calculate the length of the message received between 1 being written to [CMD.hash\_start](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#cmd) and 1 being written to [CMD.hash\_process](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#cmd).

This version does not have many defense mechanisms but is able to wipe internal variables such as the secret key, intermediate hash results, digest and the internal message scheduling array. It does not wipe the message FIFO, which SW writes the message to (but cannot read from). The software can wipe the internal variables and secret key by writing a 32-bit random value into [WIPE\_SECRET](https://opentitan.org/book/hw/ip/hmac/doc/registers.html#wipe_secret) register. The internal variables and secret key will be reset to the written value. For SHA-2 384/512 modes that operate on 64-bit words, the 32-bit random value is replicated and concatenated to create the 64-bit value. This version of the HMAC does not have an internal pseudo-random number generator to derive the random number from the written seed number.

A later update may provide an interface for external hardware IPs, such as a key manager, to update the secret key. It will also have the ability to send the digest directly to a shared internal bus.

**Theory of Operation**

The HMAC block diagram above shows that the HMAC core converts the secret key registers into an inner padded key and an outer padded key which are fed to the SHA-2 hash engine (which is a SHA-2 engine primitive instantiated with the multi-mode feature enabled) when appropriate. The module also feeds the result of the first round message (which uses the inner padded key) from the SHA-2 hash engine into the 32x32b message FIFO for the second round (which uses the outer padded key). The message length is automatically updated to reflect the size of the outer padded key and first round digest result for the second round. See [Design Details](https://github.com/lowRISC/opentitan/blob/master/hw/ip/hmac/doc/theory_of_operation.md#design-details) for more information.

The SHA-2 engine block diagram shows the message scheduling FIFO array, hash registers, digest registers, and SHA-2 compression function inside SHA-2 engine. The message scheduling FIFO is not software accessible but is fed from the 32x32b message FIFO seen in the HMAC block diagram via the HMAC core. The HMAC core can forward the message directly from the 32x32b message FIFO if HMAC is not enabled. The message words are padded with the message length appended to fit either the 512-bit or 1024-bit block size (depending on the configured digest size) as described in the [SHA-256 specification](https://csrc.nist.gov/publications/detail/fips/180/4/final).

With the 512-bit block (for SHA-2 256), the compression function runs 64 rounds to calculate the block hash, which is stored in the hash registers above. After 64 rounds are completed, the SHA-2 256 updates the digest registers with the addition of the hash result and the previous digest registers. With the 1024-bit block (for SHA-2 384/512), the compression function runs 80 rounds instead. SHA-2 384 is a truncated version of SHA-2 512 where the last 128 bits of the final digest output are truncated to reduce the digest size to 384 bits.

**Design Details**

**SHA-2 message feed and pad**

A message is fed via a memory-mapped message FIFO. Any write access to the memory-mapped window [MSG\_FIFO](https://github.com/lowRISC/opentitan/blob/master/hw/ip/hmac/doc/registers.md#msg_fifo) updates the message FIFO. If the FIFO is full, the HMAC block will block any writes leading to back-pressure on the interconnect (as opposed to dropping those writes or overwriting existing FIFO contents). It is recommended to avoid this back-pressure by not writing to the memory-mapped message FIFO when it is full. To avoid doing so, software can read the [STATUS.fifo\_full](https://github.com/lowRISC/opentitan/blob/master/hw/ip/hmac/doc/registers.md#status) register.

The logic assumes the input message is little-endian. It converts the byte order of the word right before writing to SHA-2 storage as SHA-2 treats the incoming message as big-endian. If SW wants to convert the message byte order, SW should set [CFG.endian\_swap](https://github.com/lowRISC/opentitan/blob/master/hw/ip/hmac/doc/registers.md#cfg) to **1**. The byte order of the digest registers, from [DIGEST\_0-DIGEST\_15](https://github.com/lowRISC/opentitan/blob/master/hw/ip/hmac/doc/registers.md#digest) can be configured with [CFG.digest\_swap](https://github.com/lowRISC/opentitan/blob/master/hw/ip/hmac/doc/registers.md#cfg--digest_swap).

Small writes to [MSG\_FIFO](https://github.com/lowRISC/opentitan/blob/master/hw/ip/hmac/doc/registers.md#msg_fifo) are coalesced into 32-bit words by the [packer logic]({{< relref "hw/ip/prim/doc/prim\_packer" >}}). These words are fed into the internal message scheduling FIFO. While passing writes to the packer logic, the block also counts the number of bytes that are being passed. This computes the received message length, which is used in the HMAC and SHA-2 hash computation logic.

The SHA-2 engine computes an intermediate hash for every 512-bit or 1024-bit block depending on the configured digest size. The message must be padded to fill the 512/1024-bit blocks. This is done with an initial **1** bit after the actual message bits, followed by enough **0** padding bits, and then the 64/128-bit message length at the end of the block. The number of **0** padding bits should be enough such that the full block size (512 or 1024 bits) is achieved. The [SHA-256 specification](https://csrc.nist.gov/publications/detail/fips/180/4/final) describes this in more detail. An example is shown below. The padding logic handles this so software only needs to write the actual message bits into the message FIFO.

For example, for SHA-2 256, if the message is empty, the message length is 64-bit 0. In this case, the padding logic gives 0x80000000 into the SHA-2 module first. Then it sends (512 - 32 - 64)/32, 13 times of 0x00000000 for Padding 0x00. Lastly, it returns the message length which is 64-bit 0x00000000\_00000000. If incomplete words are written, the packet logic appends 0x80 in the proper byte location, such as 0xXX800000 for the message length % 4B == 1 case. This similarly occurs for SHA-2 384/512 but with a 128-bit message length and block size of 1024 bits.

**SHA-2 computation**

For SHA-2 256, the SHA-2 engine receives 16 32-bit words from the message FIFO or the HMAC core, which get padded into 16 64-bit words for the SHA-2 engine (upper 32 bits of each data word are all-zero padded), and then begin 64 rounds of the hash computation which is also called *compression*. Alternatively for SHA-2 384/512, the SHA-2 engine receives 32 32-bit words from message FIFO, which get packed into 16 64-bit words for the SHA-2 engine, and then begin the 80 compression rounds. In each round, the compression function fetches a 64-bit word from the buffer and computes the internal variables. The first 16 rounds are fed by the words from the message FIFO or the HMAC core. Input for later rounds comes from shuffling the given 512/1024-bit block. Details are well described in [Wikipedia](https://en.wikipedia.org/wiki/SHA-2) and the [SHA-256 specification](https://csrc.nist.gov/publications/detail/fips/180/4/final).

With the given hash values, 4-byte (or 8-byte) message word, and round constants, the compression function computes the next round hash values. The round constants for the different digest sizes are hard-wired in the design. After the compression at the last round is finished, the resulting hash values are added into the digest. The digest, again, is used as initial hash values for the next block compression. During the compression rounds, it doesn't fetch data from the message FIFO. The software can push up to 16 (or 32 for SHA-2 384/512) entries to the FIFO for the next hash computation.

**HMAC computation**

HMAC can be used with any hash algorithm but this version of HMAC IP uses SHA-2 256/384/512. The first phase of HMAC calculates the SHA-2 hash of the inner secret key concatenated with the actual message to be authenticated. This inner secret key is created with the 128/256/384/512/1024-bit (hashed) secret key (depending on the configured key length) and 0x36 padding to complete the corresponding block size of the configured digest size. For example, for SHA-2 256 with 256-bit key, 512-bit inner secret key is created with the 256-bit secret key with 256-bit zero padding, XORed with 64{0x36}.

The message length used in the SHA-2 module is calculated by the HMAC core by adding the block size to the original message length (to account for the length of inner\_pad\_key, which has been prepended to the message).

The first round digest is fed into the second round in HMAC. The second round computes the hash of the outer secret key concatenated with the first round digest. In case of SHA-2 256 with 256-bit key, as the digest result is 256-bit, it must be zero-padded to fit into 512-bit block size.

In the second round, the message length is a fixed 768 bits (512-bit size of outer secret key + 256-bit first round digest size).

HMAC supports a secret key of length 128/256/384/512/1024-bit, so long as the key length does not exceed the block size of the configured digest, i.e., for SHA-2 256 a maximum length of 512-bit key is supported. The byte order of the key registers is big-endian by default, can be swapped to little endian by setting [CFG.key\_swap](https://github.com/lowRISC/opentitan/blob/master/hw/ip/hmac/doc/registers.md#cfg--key_swap) to 1. To support any arbitrary key length, the software should configure the HMAC to the next largest supported key length, e.g. for an 80-bit key, HMAC should be configured with an 128-bit key length and fed with the 80-bit key. It is also up to the software to shrink the key to the supported key length (up to 512-bit for SHA-2 256 and up to 1024-bit for SHA-2 384/512) using a hash function when setting up the HMAC. For example, common key sizes may be 2048-bit or 4096-bit. Software is expected to hash these into the supported key length and write the hashed result as the configured key to the HMAC IP.

**Performance in SHA-2 mode and HMAC mode**

The SHA-2 256 hash algorithm computes 512 bits of data at a time. The first 16 rounds need the actual 16 x 32-bit message and the following 48 rounds need some value derived from the message.

In these 48 rounds, the software can feed the next 16 x 32-bit message block. But, once the FIFO gets full, the software cannot push more data until the current block is processed. This version of the IP fetches the next 16 x 32-bit message into the internal message scheduling array only after completing the current block. As such, it takes 80 cycles to complete a block. The effective throughput considering this is 64 byte / 80 clk or 16 clk / 80 clk, 20% of the maximum throughput. For instance, if the clock frequency is 100MHz, the SHA-2 256 can hash out 80MB/s at most.

For SHA-2 384/512, the algorithm computes 1024 bits of data a time and runs for 80 rounds where the first 16 rounds consume the actual 16 x 64-bit message. It takes 96 cycles to complete a 1024-bit block. If the clock frequency is 100MHz, the SHA-2 384/512 can hash out 133MB/s at most.

This throughput could be enhanced in a future version by feeding the message into the internal buffer when the round hits 48, eliminating the extra 16 cycles to feed the message after completing a block.

If HMAC mode is turned on, it introduces extra latency due to the second round of computing the final hash of the outer key and the result of the first round using the inner key. This adds an extra 240 cycles (80 for the inner key, 80 for the outer key, and 80 for the result of the first round) to complete a HMAC SHA-2 256 digest of a message. For instance, if an empty message is given then it takes 360 cycles (80 for msg itself and 240 for the extra) to get the HMAC authentication token.

**MSG\_FIFO**

The MSG\_FIFO in the HMAC IP has a wide address range not just one 4 byte address. Any writes to the address range go into the single entry point of the prim\_packer. Then prim\_packer compacts the data into the word-size if not a word-write then writes to the MSG\_FIFO. This is different from a conventional memory-mapped FIFO.

By having wide address range pointing to a single entry point, the FIFO can free software from the fixed address restriction. For instance, the core can use "store multiple" commands to feed the message fifo efficiently. Also, a DMA engine which might not have the ability to be configured to the fixed write and incremental read may benefit from this behavior.