A software interface for KECCAK

In this note, we propose an interface to KECCAK at the level of the sponge and duplex constructions, and inside KECCAK at the level of the KECCAK-f permutation. The purpose is twofold.

First, it allows users of KECCAK making best use of its flexibility. As focused on by the SHA-3 contest [9, 10], KECCAK is sometimes viewed solely as a hash function and some implementations are inherently restricted to the traditional fixed-output-length instances. Instead, the proposed interface reflects the features of the sponge and duplex constructions, from the arbitrary output length to the flexibility of choosing security-speed trade-offs.

Second, it simplifies the set of optimized implementations on different platforms. Nearly all the processing of KECCAK takes place in the evaluation of the KECCAK-f permutation as well as in adding (using bitwise addition of vectors in GF(2)) input data into the state and extracting output data from it. The interface helps isolate the part that needs to be most optimized, while the rest of the code can remain generic. If they share the same interface, optimized implementations can be interchanged and a developer can select the best one for a given platform.

As a concrete exercise, we adapted some implementations from [6] to the proposed interface and posted them in a new "KECCAK Code Package" [8]. For the optimized implementations, it appears that the impact on the throughput is negligible while it significantly improves development flexibility and simplicity.

In the rest of this note, we first give an overview of the proposed interface. We then say some words about the implementations in the "KECCAK Code Package". Finally, we discuss some possible refinements and extensions.

1 Overview of the proposed interface

Figure 1 shows the different levels of functionality, from the modes of use down to the cryptographic permutation. In this note, we wish to propose and describe interfaces

- to the permutation;
- to the sponge construction;
- to the duplex construction.

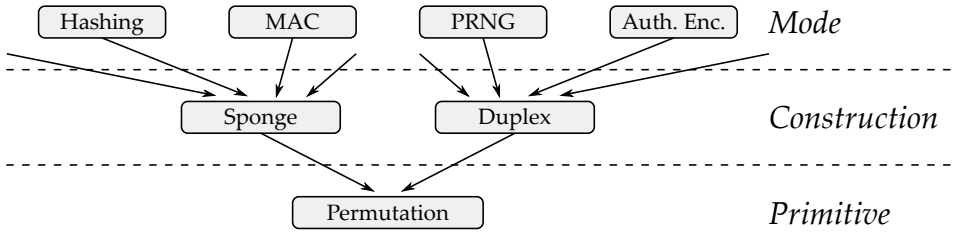

Figure 1: From modes to constructions to primitive

We leave interfaces to modes outside of the scope. Instead, we take modes into account in their use of the sponge or duplex construction.

As another remark, it has been shown in [2, Duplexing-sponge lemma] that a duplex object can be formally reduced to a sponge function. Hence, one could expect an arrow from duplex to sponge, or duplex to be on top of sponge. However, Figure 1 aims at showing the

layering as in the definitions and in implementations—it would be completely inefficient to implement the duplex construction via the sponge construction.

We now overview the proposed interface in a bottom-up fashion: from primitive to constructions.

1.1 Permutation and state management

Both the sponge and duplex constructions operate on a state. They apply the (KECCAK-f) permutation to it, add data into it or extract data from it. Therefore we define a layer that supports these three operations and their combination: the permutation and state management. We are aware that this slightly deviates from the layering depicted in Figure 1, as we support in fact operations that sponge and duplex need to perform on the state, including applying the permutation.

The proposed interface hides the value and layout of the state from the layer above. This allows using an optimized implementation of KECCAK-f that relies on a specific representation of the state. The functions and their parameters of the interface favor an implementation organized in lanes, while not assuming anything on how lanes are represented. This is consistent with the software-oriented techniques of [7], where in *lane complementing*, a subset of the lanes are stored with their bits complemented, and in *bit interleaving* each lane is stored with its bits transposed and in multiple words.

The proposed interface supports:

- **lane-aligned access**, where data can be added to, or extracted from, with the granularity of a lane and starting from the origin;
- **byte-wise, intra-lane access**, where a small number of bytes can be added to or extracted from within a single lane;
- **bit-level complementing**, where the value of a single bit can be completed, in particular at position $r - 1$ for the second bit of multi-rate padding;
- and of course the application of the **permutation** on the value of the state.

It is described in Appendix A.1.

Input, permutation and output operations can be combined in a single call for optimal performance. For instance, if data is added into lanes, the value of these lanes has to be loaded in the processor's register. Hence, applying the permutation immediately after the XORs can take advantage of these values already loaded. A similar argument applies when extracting data from the state. This is the idea behind the combined function `KeccakF1600_StateXORPermuteExtract()` (see Section A.1.9): it enables the sequence of input, permutation and output in a single function call.

As currently proposed, the input and output data are restricted to whole number of lanes, again assuming that the implementation of KECCAK-f is lane-oriented. We found this restriction to be a natural trade-off with simplification. Of course, when the data size is not a multiple of the lane size, trailing bytes can be added before or extracted after calling the combined function.

The combined function can be equally well used for sponge functions in absorbing phase (input data only) and in squeezing phase (output data only) or for duplex objects (both input and output data in the same call).

In a simple implementation respecting this interface, one may decide to write a function taking care only of the permutation and then write the combined function as a sequence of

input, permutation and output functions. In other cases, one may decide to favor a tightly optimized combined function and write the permutation function as a special case, invoking the combined function with no input or output data. In more extreme case, one could write one or more combined functions dedicated to a fixed number of input and/or output lanes. E.g., such a function dedicated to 21 input lanes can lead to optimal performances for the absorbing phase of KECCAK$[r = 1344, c = 256]$.

1.2 The sponge construction

The interface allows

- **initializing** the sponge function for chosen rate and capacity parameters;
- **absorbing input data**, with the input data provided in chunk sizes of the caller's choice (and so independently of lane alignment);
- **squeezing out output data** in chunks of arbitrary sizes.

Hence the interface provides the next layer with a queue both for input and output data. It is described in details in Appendix A.2.

In the main absorbing and squeezing functions, the data lengths are given in bytes, instead of bits. This seems rather natural as the input/output buffers are given as pointers to bytes—and this also avoids "bit attacks" [1]. Input strings whose size is not a multiple of 8 bits are covered by a dedicated function that absorbs the last trailing bits not filling a byte. For instance, as in [4, Section 6.4], a fixed bit string is appended as suffix for domain separation. When everything else is byte-aligned, this comes down to calling the dedicated function to absorb the fixed suffix.

The trailing bits are given delimited with reversible padding in a byte (see Section A.2.3), so that the number of bits does not have to be given as an explicit parameter. In practice, the reversible padding in the byte matches the first bit of the 10*1 padding, and this can be used to just add the delimited byte into the state. For example, see the function `Keccak_SpongeAbsorbLastFewBits()` in [8].

1.3 The duplex construction

The interface allows

- **initializing** the duplex object for chosen rate and capacity parameters;
- performing a **duplexing call** to the duplex object.

It is described in details in Appendix A.3.

Similarly as for the sponge construction, the interface distinguishes the input bytes from the input trailing bits. The former are given as a pointer to a buffer, while the latter are given as a byte value. This can separate application-level data from mode-level frame bits. For instance, in the SPONGEWRAP mode [2], the data blocks to be enciphered and/or authenticated are first appended with a frame bit 0 or 1 to provide domain separation between keystream and MAC. It would be a pity if, in the implementation of the SPONGEWRAP mode, one has to first copy the data blocks into a buffer, append the frame bit, and then make a duplexing call with that buffer. Instead, the data block can be given as a pointer to its original location and the frame bit can be given in a separate parameter.

2 Revisiting the KECCAK implementations

The "KECCAK Reference and Optimized Code in C" [6] package contains many different implementations, not all using the same base. For example, some implementations of that package use a dedicated buffer for the input/output data queue, others implement a queue directly in the state, and yet others do not implement any queue functionality at all. As another example, some implementations are restricted to a rate multiple of the lane size, others do not have that restriction.

In the new "KECCAK Code Package" [8], we wrote code for the sponge and duplex constructions calling the permutation and state management interface proposed in Section 1.1. This provides a common and coherent base for all implementations. We then adapted some reference and optimized codes for KECCAK-$f[1600]$ to this interface. The platform-dependency is thus now limited to the permutation and state management.

As the code for the sponge and duplex constructions is meant to be common for all (reference or optimized) implementations, we chose the options that are the most flexible to the user. As a by-product, this also proof-tested the suitability of the permutation and state management interface. For instance, in the sponge construction, we chose the option where the message queue directly uses the state, minimizing the memory consumption. In the absorbing phase, the chunks of input data are added into the state and a byte index keeps track of where the next chunk has to go. (As a reminder, in sponge functions, the message blocks can be directly added into the state and do not need to be kept in memory.) In the squeezing phase, the byte index tells where the next output bytes are to be taken from.

This new "KECCAK Code Package" is a *work in progress*. At the time of this writing, we only adapted a small number of implementations of KECCAK-$f[1600]$, namely:

- the reference implementation with 64-bit words;
- the reference implementation with 32-bit words using bit interleaving (BI);
- the optimized implementation in plain C with round unrolling and optional lane complementing;
- the optimized in-place implementation with 32-bit BI, minimizing memory usage for constrained devices.

Adapting to other KECCAK-f instances, such as KECCAK-$f[800]$ or KECCAK-$f[200]$, would not take too much effort; e.g., the sponge and duplex constructions refer to the lane size as a compile-time parameter.

Contributions are of course welcome and can be sent via `github`.

3 Refinements and perspectives

In this section, we discuss some possible refinements to the interface and how the existing functions can be used in some specific cases.

3.1 Considering parallel evaluations

On some platforms as well as in some use cases, it can be advantageous to execute the KECCAK-f permutation on several state values in parallel as it can result in faster processing per input/output data unit than when using sequential executions. One such use case is parallel (or tree) hashing; another is the production of several independent key streams.

We now discuss a possible extension of the interface for parallel executions of KECCAK-f.

We focus on the case where the different permutation instances are parallelized on a single execution flow, i.e., on a single core. In other situations, such as independent cores, processors or machines, we expect that synchronizing the different processes or threads on each call to KECCAK-f would be too fine-grained and would result in inefficient implementations. So, in other words, we focus on *microscopic* parallelism as explained in [4, Section 5.3].

A possible extension could define an opaque structure that gathers n states, and each function would have an additional parameter to specify on which instance number (say from 0 to $n-1$) the function works. In addition to this, one would add a function performing the KECCAK-f permutation on the n state instances in parallel. On a platform that does not benefit from parallelism, this multi-permutation function could just call the single-permutation implementation n times as a fall-back. And at the level of the construction, one can imagine adding functions that process n sponge functions or duplex objects in parallel.

The reason for making the structure opaque is to allow an optimized implementation organizing the n states in a favorable way. For instance, an implementation using 128-bit SIMD instructions could store the 64-bit lane (x, y) of state #0 immediately followed by the 64-bit lane (x, y) of state #1 so as to be able to load the two lanes in one shot, as proposed in [7, Section 3.1.3].

The need for a simple fall-back serial implementation goes more smoothly with parallel or tree hashing modes that process blocks of r bits per call to KECCAK-f or, in the language of [5, 4], that use an interleaving block size equal to (a multiple of) the rate, i.e., $I = nr$. The reason is the following. The faster functions to absorb input data (or to squeeze output data) assume data to be organized in a block of consecutive bits, and the most favorable situation is to absorb (or squeeze) exactly r bits. If one uses an interleaving factor of, say, $I = 64$ bits, this could be advantageous on a 128-bit SIMD unit for the same reasons as described above, but when the same tree has to be evaluated sequentially (on a platform not able to exploit parallelism), the input data are not organized in contiguous blocks. Note that the figures in [7, Section 3.1.3] are given for $I = 64$ bits, but our experiments suggest that the performance impact when using $I = r$ instead is negligible.

At the time of this writing, there is no code for parallel evaluations of KECCAK-$f[1600]$ in [8], but it would be interesting to add it in the future.

3.2 Mode-specific features

There are two other mode-specific features that could be added to the permutation and state management interface, although we do not include them at this time.

The first feature is the ability to retrieve the added input. This could be useful in a duplexing call when used in the SpongeWrap mode [2]. In that mode, the encryption of a given plaintext block is done by bitwise adding the output of the previous duplexing call. This plaintext block is then used as input in the next duplexing call, which bitwise-adds it into the state. These two additions turn out to yield the same value, namely the cipher text block on the one hand and a part of the state on the other hand. Hence, a function that retrieves the added input could be used to do the encryption for free.

The second feature is the ability to set the value of the outer part of the state instead of adding data. This would find an application in the Overwrite mode [2], which can be used to save memory between permutation calls.

For each feature, one would also need to make it visible up to the mode level, hence to add or adapt functions at the level of the duplex construction.

The KECCAK Team, July 2013
Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche

References

[1] D. J. Bernstein, *Bit attacks*, First SHA-3 candidate conference, 2009.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Duplexing the sponge: single-pass authenticated encryption and other applications*, Selected Areas in Cryptography (SAC), 2011.

[3] ———, *The* KECCAK *reference*, January 2011, `http://keccak.noekeon.org/`.

[4] ———, *Sakura: a flexible coding for tree hashing*, Cryptology ePrint Archive, Report 2013/231, 2013, `http://eprint.iacr.org/`.

[5] ———, *Sufficient conditions for sound tree and sequential hashing modes*, Cryptology ePrint Archive, Report 2009/210 (revised April 2013), 2013, `http://eprint.iacr.org/`.

[6] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, *Reference and optimized implementations of* KECCAK, 2012, `http://keccak.noekeon.org/`.

[7] ———, KECCAK *implementation overview*, May 2012, `http://keccak.noekeon.org/`.

[8] ———, KECCAK *code package*, June 2013, `https://github.com/gvanas/KeccakCodePackage`.

[9] NIST, *Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family*, Federal Register Notices **72** (2007), no. 212, 62212–62220, `http://csrc.nist.gov/groups/ST/hash/index.html`.

[10] ———, *NIST selects winner of secure hash algorithm (SHA-3) competition*, October 2012, `http://www.nist.gov/itl/csd/sha-100212.cfm`.

A Interface used in the "KECCAK Code Package"

This section documents how the proposed interface was concretely laid out in C in "KECCAK Code Package" [8]. It is currently only limited to sponge and duplex instances based on KECCAK-f[1600], although this could be extended to other KECCAK-f instances or even to other permutations.

The division of the state into lanes is specific to KECCAK-f. In the case of other permutations, the concept of "lane" could nevertheless be remapped to other data units in which the permutation can be naturally cut.

In addition, the interface is described in detail in the C language. We think that the general structure is more important than the details specific to the C language and we believe that adapting it to another programming language is easy.

A.1 Permutation and state management

A.1.1 `KeccakF1600_Initialize()`

```
void KeccakF1600_Initialize( void );
```

Function called at least once before any use of the other `KeccakF1600_*` functions, possibly to initialize global variables.

A.1.2 `KeccakF1600_StateInitialize()`

```
KeccakF1600_StateInitialize(void *state);
```

Function to initialize the state to the logical value 0^{1600}.

`state`	Pointer to the state to initialize

A.1.3 `KeccakF1600_StateXORBytesInLane()`

```
void KeccakF1600_StateXORBytesInLane(void *state, unsigned int lanePosi-
tion, const unsigned char *data, unsigned int offset, unsigned int length);
```

Function to XOR data given as bytes into the state. The bits to modify are restricted to be consecutive and to be in the same lane. The bit positions that are affected by this function are from `lanePosition` $\times 64 +$ `offset` $\times 8$ to `lanePosition` $\times 64 +$ `offset` $\times 8 +$ `length` $\times 8$. (The bit positions, the x, y, z coordinates and their link are defined in [3].)

`state`	Pointer to the state.
`lanePosition`	Index of the lane to be modified ($x + 5y$, or bit position divided by 64).
`data`	Pointer to the input data.
`offset`	Offset in bytes within the lane.
`length`	Number of bytes.

Preconditions:

- $0 \leq$ `lanePosition` < 25
- $0 \leq$ `offset` < 8
- $0 \leq$ `offset` $+$ `length` ≤ 8

A.1.4 KeccakF1600_StateXORLanes()

```
void KeccakF1600_StateXORLanes(void *state, const unsigned char *data, unsigned int laneCount);
```

Function to XOR data given as bytes into the state. The bits to modify are restricted to start from the bit position 0 and to span a whole number of lanes (i.e., multiple of 8 bytes).

`state`	Pointer to the state.
`data`	Pointer to the input data.
`laneCount`	The number of lanes, i.e., the length of the data divided by 64 bits.

Preconditions:

- $0 \leq$ `laneCount` ≤ 25

A.1.5 KeccakF1600_StateComplementBit()

```
void KeccakF1600_StateComplementBit(void *state, unsigned int position);
```

Function to complement the value of a given bit in the state. This function is typically used to XOR the second bit of the multi-rate padding into the state.

`state`	Pointer to the state.
`position`	The position of the bit to complement.

Preconditions:

- $0 \leq$ `position` < 1600

A.1.6 KeccakF1600_StatePermute()

```
void KeccakF1600_StatePermute(void *state);
```

Function to apply KECCAK-$f[1600]$ on the state.

`state`	Pointer to the state.

A.1.7 KeccakF1600_StateExtractBytesInLane()

```
void KeccakF1600_StateExtractBytesInLane(const void *state, unsigned int lanePosition, unsigned char *data, unsigned int offset, unsigned int length);
```

Function to retrieve data from the state into bytes. The bits to output are restricted to be consecutive and to be in the same lane. The bit positions that are retrieved by this function are from `lanePosition` $\times 64 +$ `offset` $\times 8$ to `lanePosition` $\times 64 +$ `offset` $\times 8 +$ `length` $\times 8$. (The bit positions, the x, y, z coordinates and their link are defined in [3].)

`state`	Pointer to the state.
`lanePosition`	Index of the lane to be read ($x + 5y$, or bit position divided by 64).
`data`	Pointer to the area where to store output data.
`offset`	Offset in byte within the lane.
`length`	Number of bytes.

Preconditions:

- $0 \leq$ `lanePosition` < 25
- $0 \leq$ `offset` < 8

- $0 \leq$ `offset` + `length` ≤ 8

A.1.8 `KeccakF1600_StateExtractLanes()`

```
void KeccakF1600_StateExtractLanes(const void *state, unsigned char *data,
unsigned int laneCount);
```

Function to retrieve data from the state into bytes. The bits to output are restricted to start from the bit position 0 and to span a whole number of lanes (i.e., multiple of 8 bytes).

`state`	Pointer to the state.
`data`	Pointer to the area where to store output data.
`laneCount`	The number of lanes, i.e., the length of the data divided by 64 bits.

Preconditions:

- $0 \leq$ `laneCount` ≤ 25

A.1.9 `KeccakF1600_StateXORPermuteExtract()`

```
void KeccakF1600_StateXORPermuteExtract(void *state, const unsigned char
*inData, unsigned int inLaneCount, unsigned char *outData, unsigned int
outLaneCount);
```

Function to sequentially XOR data bytes, apply the KECCAK-$f[1600]$ permutation and retrieve data bytes from the state. The bits to modify and to output are restricted to start from the bit position 0 and to span a whole number of lanes (i.e., multiple of 8 bytes). Its effect should be functionally identical to calling in order:

1. `KeccakF1600_StateXORLanes(state, inData, inLaneCount);`
2. `KeccakF1600_StatePermute(state);`
3. `KeccakF1600_StateExtractLanes(state, outData, outLaneCount);`

`state`	Pointer to the state.
`inData`	Pointer to the input data.
`inLaneCount`	The number of lanes, i.e., the length of the input data divided by 64 bits.
`outData`	Pointer to the area where to store output data.
`outLaneCount`	The number of lanes, i.e., the length of the output data divided by 64 bits.

Preconditions:

- $0 \leq$ `inLaneCount` ≤ 25
- $0 \leq$ `outLaneCount` ≤ 25

A.2 The sponge construction

A.2.1 `Keccak_SpongeInitialize()`

```
int Keccak_SpongeInitialize(Keccak_SpongeInstance *spongeInstance, unsigned
int rate, unsigned int capacity);
```

Function to initialize the state of the KECCAK$[r, c]$ sponge function. The phase of the sponge function is set to absorbing.

`spongeInstance`	Pointer to the sponge instance to be initialized.
`rate`	The value of the rate r.
`capacity`	The value of the capacity c.

Preconditions:

- One must have $r + c = 1600$ and the rate a multiple of 8 bits (one byte) in this implementation.

Returns: Zero if successful, 1 otherwise.

A.2.2 `Keccak_SpongeAbsorb()`

```
int Keccak_SpongeAbsorb(Keccak_SpongeInstance *spongeInstance, const unsigned char *data, unsigned long long dataByteLen);
```

Function to give input data bytes for the sponge function to absorb.

`spongeInstance`	Pointer to the sponge instance initialized by `Keccak_SpongeInitialize()`.
`data`	Pointer to the input data.
`dataByteLen`	The number of input bytes provided in the input data.

Preconditions:

- The sponge function must be in the absorbing phase, i.e., `Keccak_SpongeSqueeze()` or `Keccak_SpongeAbsorbLastFewBits()` must not have been called before.

Returns: Zero if successful, 1 otherwise.

A.2.3 `Keccak_SpongeAbsorbLastFewBits()`

```
int Keccak_SpongeAbsorbLastFewBits(Keccak_SpongeInstance *spongeInstance, unsigned char delimitedData);
```

Function to give input data bits for the sponge function to absorb and then to switch to the squeezing phase.

`spongeInstance`	Pointer to the sponge instance initialized by `Keccak_SpongeInitialize()`.
`delimitedData`	Byte containing from 0 to 7 trailing bits that must be absorbed. These n bits must be in the least significant bit positions. These bits must be delimited with a bit 1 at position n (counting from 0=LSB to 7=MSB) and followed by bits 0 from position $n + 1$ to position 7. Some examples: • If no bits are to be absorbed, then `delimitedData` must be `0x01`. • If the 2-bit sequence $0, 0$ is to be absorbed, `delimitedData` must be `0x04`. • If the 5-bit sequence $0, 1, 0, 0, 1$ is to be absorbed, `delimitedData` must be `0x32`. • If the 7-bit sequence $1, 1, 0, 1, 0, 0, 0$ is to be absorbed, `delimitedData` must be `0x8B`.

Preconditions:

- The sponge function must be in the absorbing phase, i.e., `Keccak_SpongeSqueeze()`

or `Keccak_SpongeAbsorbLastFewBits()` must not have been called before.

- `delimitedData` $\neq$ `0x00`

Returns: Zero if successful, 1 otherwise.

A.2.4 `Keccak_SpongeSqueeze()`

```
int Keccak_SpongeSqueeze(Keccak_SpongeInstance *spongeInstance, unsigned
char *data, unsigned long long dataByteLen);
```

Function to squeeze output data from the sponge function. If the sponge function was in the absorbing phase, this function switches it to the squeezing phase as if `Keccak_SpongeAbsorbLastFewBits(spongeInstance, 0x01)` was called.

`spongeInstance`	Pointer to the sponge instance initialized by `Keccak_SpongeInitialize()`.
`data`	Pointer to the buffer where to store the output data.
`dataByteLen`	The number of output bytes desired.

Returns: Zero if successful, 1 otherwise.

A.3 The duplex construction

A.3.1 `Keccak_DuplexInitialize()`

```
int Keccak_DuplexInitialize(Keccak_DuplexInstance *duplexInstance, unsigned
int rate, unsigned int capacity);
```

Function to initialize a duplex object DUPLEX[KECCAK-f[1600], pad10*1, r].

`duplexInstance`	Pointer to the duplex instance to be initialized.
`rate`	The value of the rate r.
`capacity`	The value of the capacity c.

Preconditions:

- One must have $r + c = 1600$ in this implementation.
- $3 \leq$ `rate` ≤ 1600, and otherwise the value of the rate is unrestricted.

Returns: Zero if successful, 1 otherwise.

A.3.2 `Keccak_Duplexing()`

```
int Keccak_Duplexing(Keccak_DuplexInstance *duplexInstance, const unsigned
char *sigmaBegin, unsigned int sigmaBeginByteLen, unsigned char *Z, unsigned
int ZByteLen, unsigned char delimitedSigmaEnd);
```

Function to make a duplexing call to the duplex object initialized with `Keccak_DuplexInitialize()`.

`duplexInstance`	Pointer to the duplex instance initialized by `Keccak_DuplexInitialize()`.
`sigmaBegin`	Pointer to the first part of the input σ given as bytes. Trailing bits are given in `delimitedSigmaEnd`.
`sigmaBeginByteLen`	The number of input bytes provided in `sigmaBegin`.
`Z`	Pointer to the buffer where to store the output data Z.
`ZByteLen`	The number of output bytes desired for Z. If `ZByteLen` $\times 8$ is greater than the rate r, the last byte contains only r mod 8 bits, in the least significant bits.
`delimitedSigmaEnd`	Byte containing from 0 to 7 trailing bits that must be appended to the input data in `sigmaBegin`. These $n = \|\sigma\|$ mod 8 bits must be in the least significant bit positions. These bits must be delimited with a bit 1 at position n (counting from 0=LSB to 7=MSB) and followed by bits 0 from position $n+1$ to position 7. Some examples: • If $\|\sigma\|$ is a multiple of 8, then `delimitedSigmaEnd` must be `0x01`. • If $\|\sigma\|$ mod 8 is 1 and the last bit is 1 then `delimitedSigmaEnd` must be `0x03`. • If $\|\sigma\|$ mod 8 is 4 and the last 4 bits are 0,0,0,1 then `delimitedSigmaEnd` must be `0x18`. • If $\|\sigma\|$ mod 8 is 6 and the last 6 bits are 1,1,1,0,0,1 then `delimitedSigmaEnd` must be `0x67`.

The input bits σ are the result of the concatenation of the bytes in `sigmaBegin` and the bits in `delimitedSigmaEnd` before the delimiter.

Preconditions:

- `delimitedSigmaEnd` $\neq$ `0x00`
- `sigmaBeginByteLen` $\times 8 + n \leq r - 2$
- `ZByteLen` $\leq \lceil \frac{r}{8} \rceil$

Returns: Zero if successful, 1 otherwise.