CRSCE: Cross Sums Compression and Expansion

Specification

Version 1.0

Sam Caldwell

mail@samcaldwell.net

(512) 712-3095

MIT License

(c) 2023 Sam Caldwell. MIT License.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Table of Contents

|  |  |  |  |
| --- | --- | --- | --- |
| 1.0. Summary | | | |
|  | 1.1. | Purpose and Intent | |
|  | 1.2. | Design Goals | |
|  | 1.3. | Comparing Prior Art | |
|  | 1.4. | Novel Approach | |
|  | 1.5. | Algorithm Overview | |
|  |  | 1.5.1. | Compressing a Signal |
|  |  | 1.5.2. | Decompressing a Signal |
|  |  | 1.5.3. | Parallel Processing Opportunities |

Table of Contents

|  |  |  |  |
| --- | --- | --- | --- |
| 2.0 | Theory of Operation | | |
|  | 2.1. | Objective | |
|  | 2.2 | Two Variants | |
|  | 2.3 | Cross-Sums Matrix (CSM) | |
|  |  | 2.3.1. | CSM Defined |
|  |  | 2.3.2. | Concurrent Message Block Processing |
|  |  | 2.3.3. | Message Block and CSM Size |
|  |  | 2.3.4. | Message Block Padding (P) |
|  | 2.4. | Cross-Sums Matrices (LSM, VSM) | |
|  | 2.5. | Cross-Sum Width (b) | |
|  | 2.6. | Hash Arrays (LHASH, VHASH) | |
|  | 2.7. | Cross-Sum Calculation | |
|  | 2.8. | Hash Calculations and Collisions | |
|  | 2.9. | Compression Rate Calculation (Cr) | |
|  | 2.10. | Compression Rate Curve | |
|  | 2.11. | Minimum Signal Size (Ms) | |
|  | 2.12. | Compressed Signal Format | |
|  |  | 2.12.1. | Overview |
|  |  | 2.12.2. | Header |
|  |  | 2.12.3. | Row Structure |
|  |  | 2.12.4. | CRSCE-1 Column Structure |
|  |  | 2.12.5. | CRSCE-2 Column Structure |
|  | 2.13. | Endianness | |

Table of Contents

|  |  |  |  |
| --- | --- | --- | --- |
| 3.0 | Compression Process | | |
|  | 3.1. | CrsceCompressor Class | |
|  |  | 3.1.1. | Overview |
|  |  | 3.1.2. | Properties |
|  |  | 3.1.3. | Class Constructor |
|  |  | 3.1.4. | Class Destructor |
|  |  | 3.1.5. | Compress Method |
|  |  | 3.1.6. | GetRate Method |
|  | 3.2. | VirtualCSM Class | |
|  |  | 3.2.1. | Overview |
|  |  | 3.2.2. | Properties |
|  |  | 3.2.3. | Class Constructor |
|  |  | 3.2.4. | Class Destructor |
|  |  | 3.2.5. | Asynchronous Reader Method |
|  |  | 3.2.6. | Pop Method |
|  | 3.3. | CrossSumArray Class | |
|  |  | 3.3.1. | Overview |
|  |  | 3.3.2. | Properties |
|  |  | 3.3.3. | Class Constructor |
|  |  | 3.3.4. | Class Destructor |
|  |  | 3.3.5. | Increment Method |
|  |  | 3.3.6. | Flush Method |
|  | 3.4. | HashArray Class | |
|  |  | 3.4.1. | Overview |
|  |  | 3.4.2. | Properties |
|  |  | 3.4.3. | Class Constructor |
|  |  | 3.4.4. | Class Destructor |
|  |  | 3.4.5. | Update Method |
|  |  | 3.4.6. | Flush Method |

Table of Contents

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 4.0 | Decompression Process | | |  |
|  | 4.1. | CrsceDecompressor Class | |  |
|  |  | 4.1.1. | Overview |  |
|  |  | 4.1.2. | Properties |  |
|  |  | 4.1.3. | Class Constructor |  |
|  |  | 4.1.4. | Class Destructor |  |
|  |  | 4.1.5. | Decompress Method |  |
|  |  | 4.1.6. | SimpleSolve Method |  |
|  |  | 4.1.7. | SiftSolve Method |  |
|  | 4.2. | CsmArray Class | |  |
|  |  | 4.2.1. | Overview |  |
|  |  | 4.2.2. | Properties |  |
|  |  | 4.2.3. | Class Constructor |  |
|  |  | 4.2.4. | Class Destructor |  |
|  |  | 4.2.5. | Linear Addressing and Cache Performance |  |
|  |  | 4.2.6. | GetSolved Method |  |
|  |  | 4.2.7. | Set Method |  |
|  |  | 4.2.8. | Clear Method |  |
|  |  | 4.2.9. | Get Method |  |
|  |  | 4.2.10. | isSolved Method |  |
|  | 4.3. | CSMElement Class | |  |
|  |  | 4.3.1. | Overview |  |
|  |  | 4.3.2. | Properties |  |
|  |  | 4.3.3. | Class Constructor |  |
|  |  | 4.3.4. | Class Destructor |  |
|  |  | 4.3.5. | Set Method |  |
|  |  | 4.3.6. | Clear Method |  |
|  |  | 4.3.7. | Get Method |  |
|  |  | 4.3.8. | Solve Method |  |
|  |  | 4.3.9. | isSolved Method |  |

Table of Contents

|  |  |  |  |
| --- | --- | --- | --- |
| 5.0 | General Data Types | | |
|  | 5.1. | BitReader Class | |
|  |  | 5.1.1. | Overview |
|  |  | 5.1.2. | Properties |
|  |  | 5.1.3. | Class Constructor |
|  |  | 5.1.4. | Class Destructor |
|  |  | 5.1.5. | Start Method |
|  |  | 5.1.6. | IsRunning Method |
|  |  | 5.1.7. | Pop Method |
|  |  | 5.1.8. | Stop Method |
|  | 5.2. | BitWriter Class | |
|  |  | 5.2.1. | Overview |
|  |  | 5.2.2. | Properties |
|  |  | 5.2.3. | Class Constructor |
|  |  | 5.2.4. | Class Destructor |
|  |  | 5.2.5. | Push Method |
|  |  | 5.2.6. | Start Method |
|  |  | 5.2.7. | Stop Method |
|  |  | 5.2.8. | isRunning Method |

Table of Contents

|  |  |  |  |
| --- | --- | --- | --- |
| 6.0 | CRSCE Command line specification | | |
|  | 6.1. | Compress Tool (compress) | |
|  |  | 6.1.1. | Usage |
|  | 6.2. | Decompress Tool (decompress) | |
|  |  | 6.2.1. | Usage |

1. Summary
   1. Purpose and Intent

This specification defines a new data compression algorithm—promising a consistent, predictable and lossless compression, regardless of content format or entropy.

* 1. Design Goals

The algorithm will allow a user to accurately calculate the compression rate[[1]](#footnote-1) based on input signal size regardless of actual content.

The algorithm will deliver at least 50% lossless compression regardless of input signal entropy[[2]](#footnote-2). For example, given a random (high-entropy) binary input the compression rate will be the same as if given a low-entropy binary input.

The algorithm will have adequate safeguards to prevent collisions.

The compression and decompression process will allow for a concurrent or distributed implementation to efficiently process large data sets.

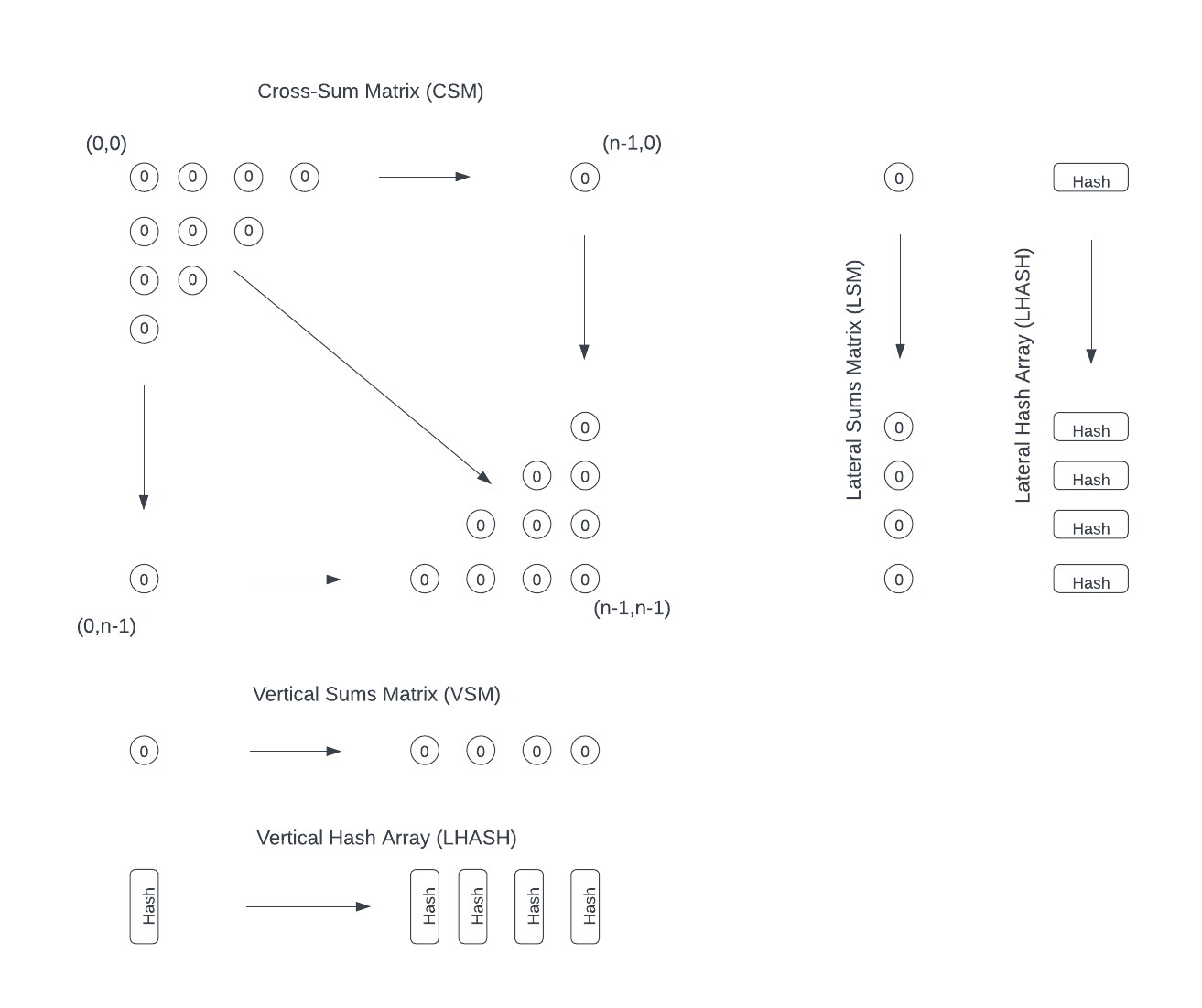
* 1. Comparing Prior Art

Version 1.0 of the algorithm has demonstrated an ability to meet or exceed the design goals. However, any persons familiar with data compression will undoubtedly read the design goals (see 1.2) and believe that the assertions are more than ambitious. The words “impossible” or “impractical” may be used, and not without reason. Conventional compression algorithms follow the same basic pattern, creating a short-hand for data, using some form of dictionary or code book. For example, Lempel-Ziv-Welch (LZW) compression[[3]](#footnote-3) uses a dynamic dictionary to encode sequences of data into variable-length short-hand codes while Run-Length Encoding (RLE)[[4]](#footnote-4) simply replaces repeated instances of a character with a single instance of the character and the number of times it is repeated in the specific instance. Likewise, Huffman[[5]](#footnote-5) encoding analyzes the input signal to construct a binary tree code book of patterns used to rewrite the original message in its short-hand form. Throughout the spectrum of compression algorithms, we find varied solutions following this theme, and they have served the information technology industry well in many ways, except when it comes to high-entropy data sources. In the case of the high entropy data set, every conventional algorithm fails more or less. The user often celebrates if a high-entropy input can achieve a consistent 20% compression rate. This is because when entropy is high, the frequency of repeated patterns is low and thus this short-hand approach fails. This is best explained by the works of Claude Shannon, the father of information theory, whose 1948 work “A Mathematical Theory of Communication”[[6]](#footnote-6) outlined many limitations of conventional compression algorithms. Shannon’s work, among other things, suggests that conventional compression algorithms using fixed-sized codes or dictionary-based methods cannot compress an input signal beyond a certain point defined by the entropy of the source. This limitation arises from the fact that entropy represents the minimum average number of bits needed to encode each symbol, and it cannot be reduced further using these simple techniques. Claude Shannon was correct with respect to the conventional compression approaches taken over the last century. However, this document asserts that some of Shannon’s limitations on the potential of compression, even when dealing with high entropy solutions can be overcome through an approach that does not use a dictionary or a code book, relying instead upon a single two-dimensional bit field, two arrays of whole numbers and two arrays of hashes. We call this approach the “Cross Sums Compression and Expansion” (CRSCE) algorithm.

* 1. Novel Approach

CRSCE deviates from conventional compression and exploits a simple mathematical pattern to eliminate the dictionary or code book. In doing so, it makes compression rates a function of message (or message block) size, regardless of content. Accordingly, the predictable compression rate based on size implies that the algorithm can perform equally well regardless of the entropy of the input signal. However, this is not without limitations. First, CRSCE only works with a signal size larger than some predetermined minimum signal size derived from the algorithm’s implementation. Second, CRSCE compression is fast. But decompression requires significant computational effort, performing best when implemented using GPUs on a modern computer system or otherwise distributed across multiple concurrent workers. As the message block size increases, the complexity and time required to decompress a signal will increase exponentially.

* 1. Algorithm Overview
     1. Compressing a Signal



Imagine a stream of uncompressed data being read into a pipeline one bit at a time. Each bit read from the source is grouped into some 8N number of bits (N bytes) representing a single message block (CSM), allowing the pipeline to compress an infinitely long stream of bits.

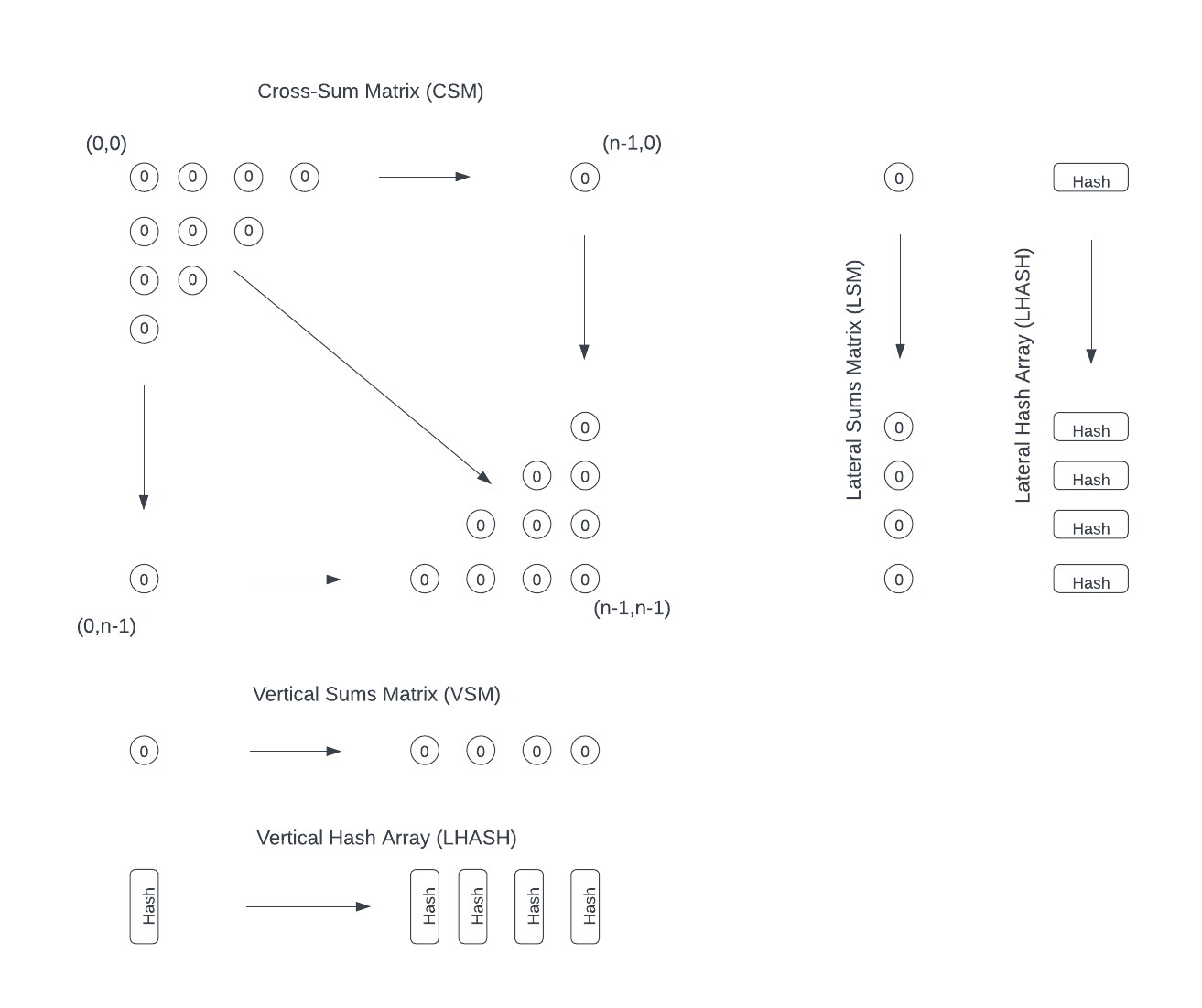
On the other end of this bit-stream pipeline, imagine the bits are written from left to right and top to bottom into a two-dimensional bit-field (array) where each element of this bit field is one bit of uncompressed data. At 8N bits, the pipeline is paused to allow the bitfield to be compressed.

A separate process then traverses the rows and columns of this two-dimensional bit field. Where the bit at each (x, y) coordinate is set, a corresponding counter for the row [y] and column [x] is incremented. Then the bit is pushed onto a queue of bits and passed to a hashing function assigned to the corresponding row [y] and column [x].

Once the two-dimensional bit field is traversed, the result is a set of four-one dimensional arrays. Two of these arrays contain the counts of bits in each corresponding row or column of the original two-dimensional bit field. Two of these arrays contain hashes of their corresponding rows and columns. From this we observe that (a) we now have a representation of the two-dimensional bitfield, both as a set of bit counts and hashes from two perpendicular perspectives (row and column), and (b) the combined length of these four one-dimensional arrays (2nb+256n bits[[7]](#footnote-7), where) is less than the length of the two-dimensional array (n2 bits).

The four one-dimensional arrays can then be written to an output file and the process can be repeated with subsequent message blocks until the entire input signal is processed and a short (160-bit) header can be added to the final result. If, given the combined length of 2nb+256n bits is less than n2 bits this shorter resulting string of bits is guaranteed to be unique to this given output, one can conclude that the process has effectively compressed the signal. Further assuming that the process can be reversed where these “cross-sum” arrays and hashes can be used to generate the unique original inputs, the algorithm is a viable general-use compression algorithm.

* + 1. Decompressing a Signal



Starting with the output of the process described in 1.5.1, above, imagine a blank two-dimensional array of n2 bytes where all values are set to 0x00 (NULL) initially. Adjacent to this array are the two cross-sum arrays and their corresponding hash arrays from the previous discussion.

The decompression process iterates over the range and test LSM[p] and VSM[p] to see if either sum is a zero or n-value. This is our SimpleSolve phase. If any sum is zero, the program knows the associated row or column is also zero in the two-dimensional bit field. Likewise, if any sum is equal to n (the maximum count value), then all bits in the corresponding row are set (value=1). This process is repeated until no row or column is solved. In the case where the entire original input consists of many zero-value bits, this simple solver can reduce the problem space significantly before the more expensive compute tasks occur.

In the above SimpleSolve process, like with any of the decompression processes, when an element in the message block (represented by the two-dimensional array) is solved, the least-significant bit is set to the solved value and the most-significant bit is set to true indicating the element is solved. Once an element is marked as solved, it should not be changed unless the process has catastrophically failed. Likewise, an element should not be marked as solved until the entire row or column under consideration is solved.

Once the SimpleSolve process has completed, the next stage is the SiftSolve. In the SiftSolve process, a number of workers is spawned to solve either a specific row or column. The rows and columns targeted by the SiftSolve process start with those having either the highest LSM or VSM count or the lowest LSM or VSM count. Each worker running SimpleSolve against a target row or column performs three tasks: (a) generate a random bit field of n bits with only the allowed number of set (1) bits as determined by LSM[y] or VSM[x] and verify the bit field has not been used before against the target row/column, (b) test whether the pattern satisfies both LSM and VSM, then (c) if the cross sums are satisfied, the hashes will be tested. If the hashes are not satisfied, the process is repeated up to five-thousand times. As each worker completes, the parent process dispatches another worker to another row/column. If all rows and columns have been covered. The process returns to SimpleSolve for a quick pass.

While the SimpleSolve and SiftSolve works to compute the final two-dimensional bit field of each message block, the order in which a block is solved is not guaranteed as each block may be processed in parallel. Blocks are, therefore, written to temporary files. A separate “assembler” process then monitors these solved message blocks, starting with block Id 0 and working through in strict order, loading and merging the temporary file data into the final decompressed result.

* + 1. Parallel Processing Opportunities

The above compression and decompression described in this document is a simplified approach. The actual algorithm in this specification, as defined over the next several pages, requires support for parallel processing of an arbitrary signal.

Where an arbitrary signal can be subdivided into smaller message blocks, each message block can be processed in parallel both for compression and decompression.

Additionally, during compression or decompression of a message block it is conceivable that the process could be mapped out to many worker processes at a row or column level to solve the message block even faster, especially where GPUs are used to perform the tasks.

* + - 1. Parallelism in Compression

Where we can create a byte queue of N bytes (a complete message block), it is possible that the BitReader could be instantiated as the virtual message block. This means multiple BitReader objects can be created, one per message block once the byte queue is allocated. However, doing this would cost significant memory resources.

Instead, if the data source size is known, the compressor could map a BitReader to a given start and end byte in the larger stream and attack the input signal in parallel without the cost of memory.

* + - 1. Parallelism in Decompression

For decompression, both SimpleSolve and SiftSolve can be made parallel by allocating rows and columns to independent solver worker instances.

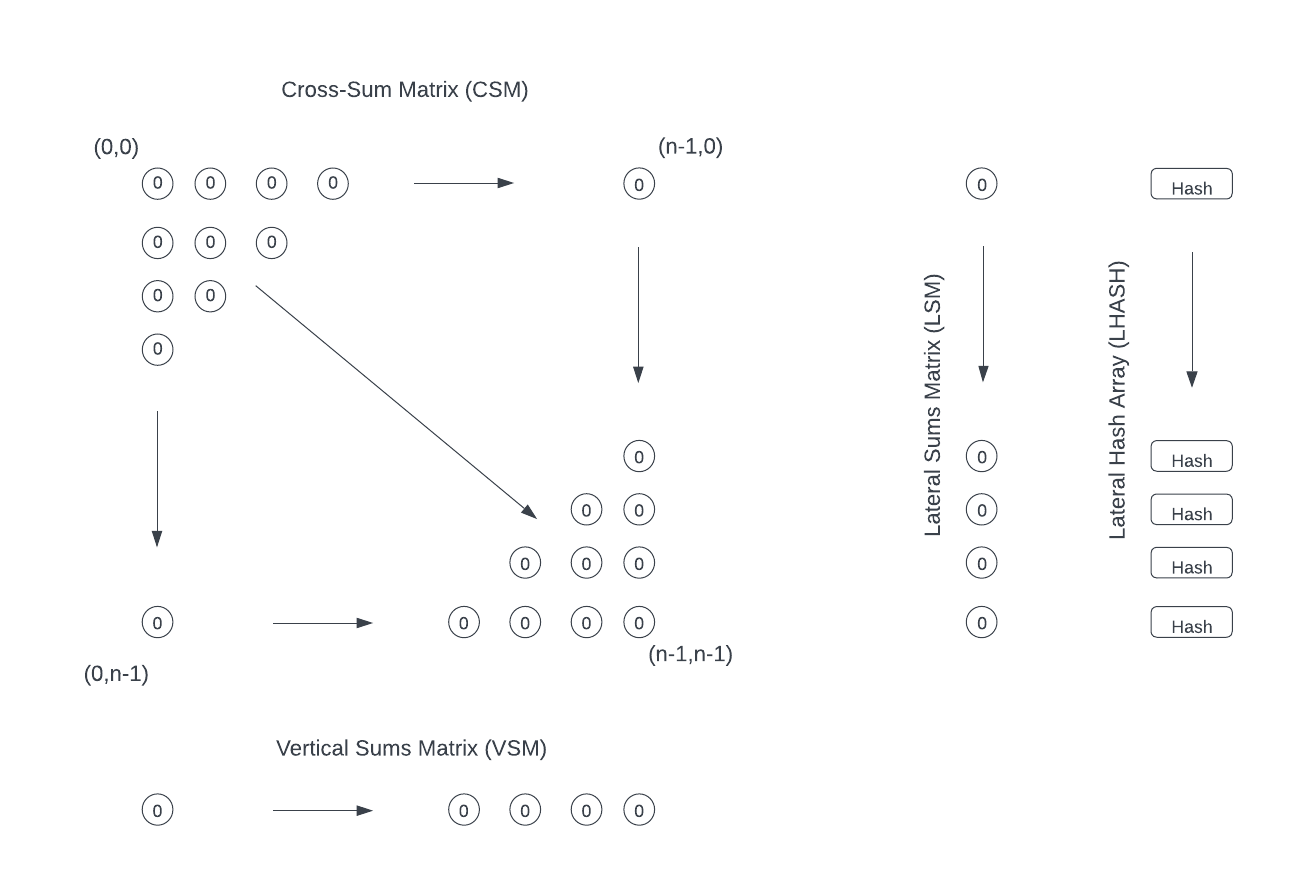
1. Theory of Operation
   1. Objective

This section will establish a high-level theoretical view of the compression and decompression processes and specify the structures and processes necessary for implementation of the algorithm.

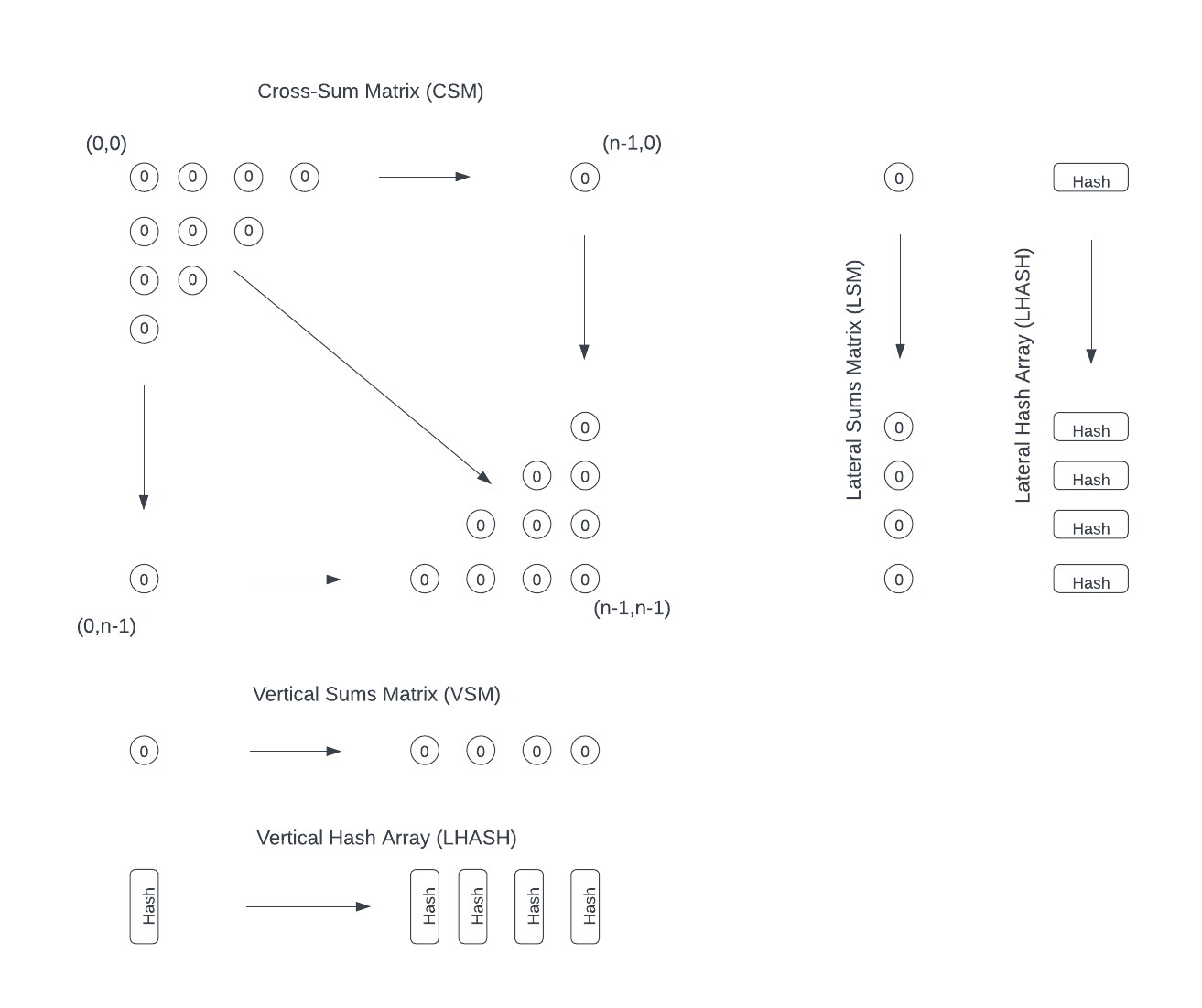
* 1. Two Variants

There are two variants of the CRSCE algorithm: CRSCE-1 and CRSCE-2.

The two variants differ only in the hash arrays used and resulting compression rate formulas realized from their different output overhead requirements.



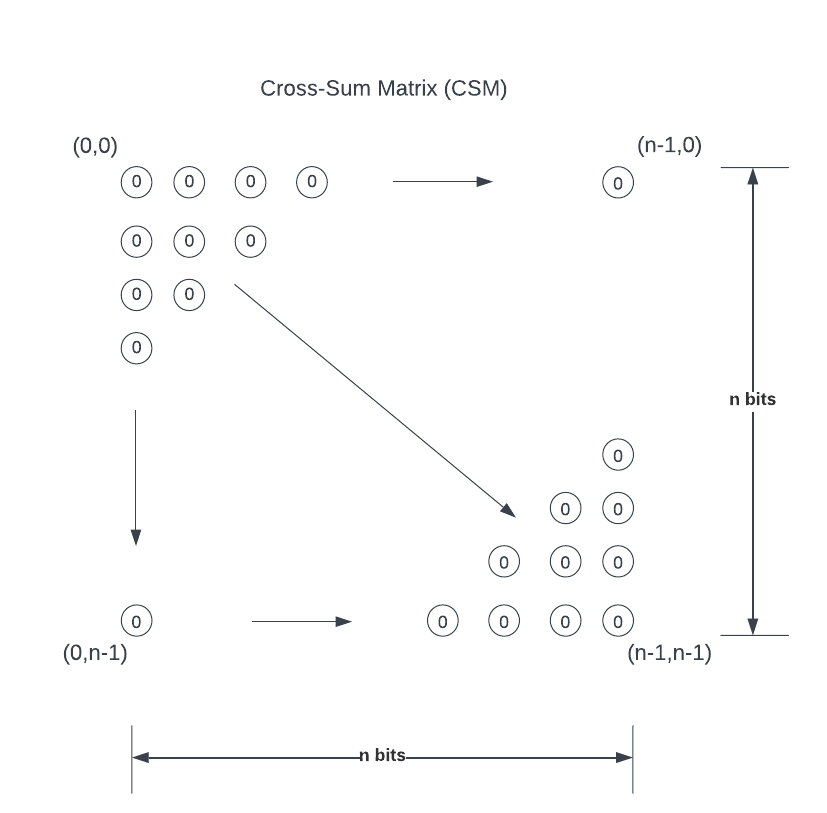
CRSCE-1 uses a single hash array (LHASH) to prevent collisions, localize decompression errors to the row level and parallelize decompression operations at the row level. This results in a lower overhead and smaller minimum signal size (approximately 19KB) as well as higher performance on a 1MB CSM (90% compression).



CRSCE-2 uses two hash arrays (LHASH, VHASH) to prevent collisions, localize decompression errors to the row and column and parallelize decompression operations by row or by column to improve decompression speed on larger CSM. The result of this additional 256n bit overhead is a higher minimum signal size (approximately 71KB) as well as lower performance on a 1MB CSM (81% compression).

* 1. Cross-Sums Matrix (CSM)
     1. CSM Defined

Assume there exists a two-dimensional bit field called the “Cross-Sums Matrix” (CSM), as illustrated below:



This bit field is some n-number of bits, squared, addressed using an ordered pair (x, y) where and , respectively. The bit field represents a single compression block.

* + 1. Concurrent Message Block Processing

Where a signal of some length is larger than the block size of the algorithm’s implementation, the algorithm shall support a means of subdividing the input signal into chunks, where each chunk represents a single “message block.”

Where an input signal can be subdivided into multiple message blocks, an implementation shall be not be precluded from compressing each message block concurrently. This implies that compressed output from any given message block may be written to the output file in any order.

* + 1. Message Block and CSM Size

The size of a message block is N bytes (8N bits).

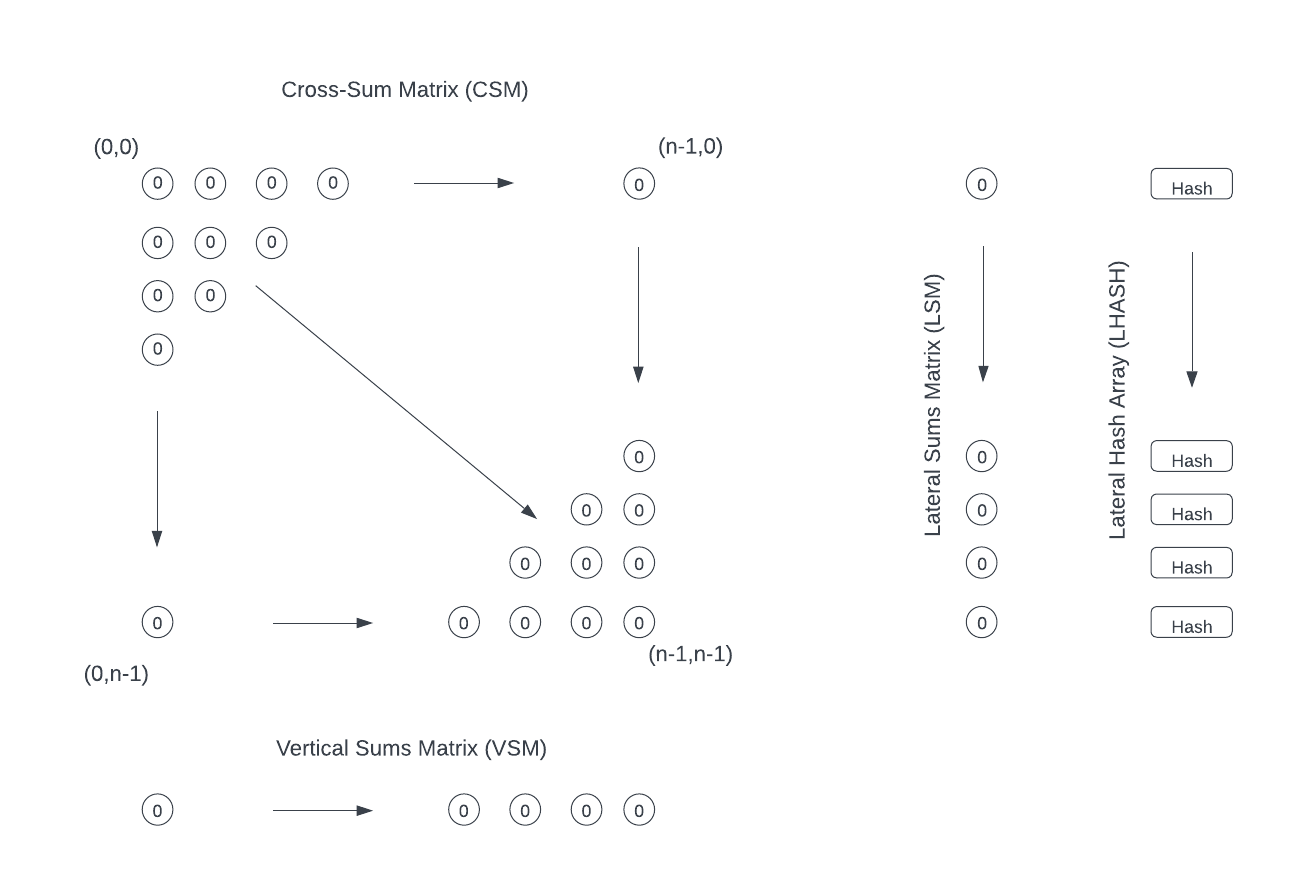
Deriving from this message block length, the size of the CSM must be greater than or equal to 8N bits. Where CSM is a square of n x n bits, the value of n is—

* + - 1. Message Padding (P)

Because the probability the message block length (8N bits) is not a perfect square, there will most likely be at least some number of padding bits (P) at the end of the CSM, where—

The last message block processed in a multi-block signal may be significantly smaller than the CSM size used for other message blocks. Where the message block stops, the rest of CSM will be zero-values and these values will be compressed and reflected in the output.

* 1. Cross-Sums Matrices (LSM, VSM)



Define two one-dimensional arrays consisting of n whole numbers (LSM and VSM).[[8]](#footnote-8)

The first array, called the “Lateral Sums Matrix” (LSM), is depicted to the right of CSM. Its size is n-elements where each element’s size in bits (b)[[9]](#footnote-9) is a function of n.

The second array, called “Vertical Sums Matrix” (VSM), is depicted below the CSM. Its size is n-elements where each element’s size in bits (b) [[10]](#footnote-10) is a function of n.

The second is depicted below the CSM and called the “Vertical Sums Matrix” (VSM).

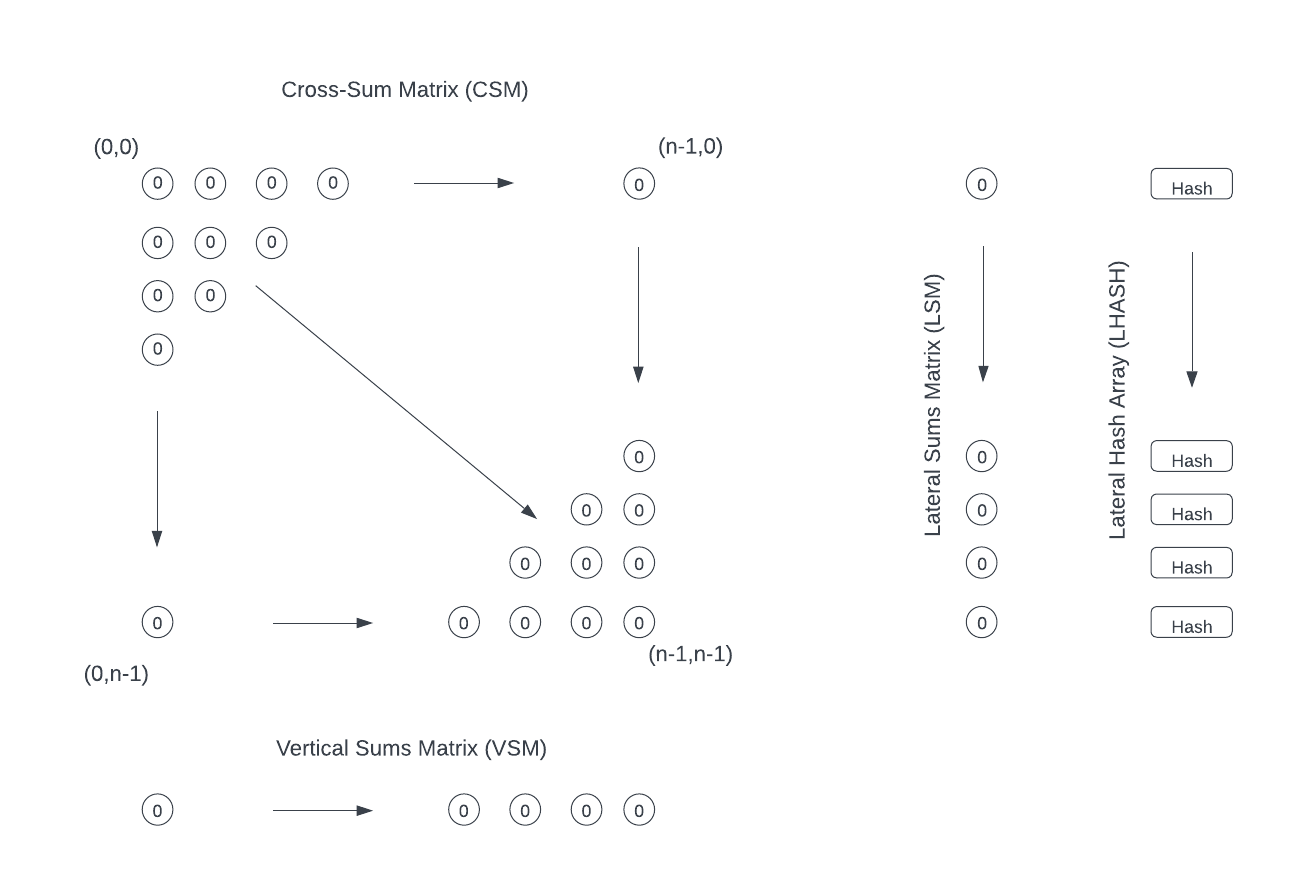
* 1. Cross Sum Width (b)

The number of bits (b) used by the cross sums matrices (LSM, VSM) is a function of the message block (CSM) size (n), stated as follows:

Where an implementation sets a limit on CSM size (such as 1MB for CRSCE-1), the implementation may define LSM and VSM elements using 16-bit unsigned integers (uint16) to conserve system memory. However, a 1MB CSM will only require 12-bits of this 16-bit unsigned integer, since—

The implementation should support an ability to pack the LSM and VSM data to this smaller b-sized bitfield or store LSM in the next-largest byte-aligned value. For example, if byte alignment is enabled using the Byte-Alignment field (see 3.3.1.4.2) the LSM and VSM data for a 1MB CRSCE-1 CSM where b=12 would store a 16-bit word to the output stream rather than a packed 12-bit value. This byte alignment trades a small amount of data compression for faster file read/write performance.

* 1. Hash Arrays (LHASH, VHASH)



Two one-dimensional arrays of n-elements are defined with each element holding a 32-byte (256-bit) hash.[[11]](#footnote-11) These arrays are collectively called “Hash Arrays,” used to hash the rows and columns of CSM.

There are two variants of the CRSCE algorithm: CRSCE-1 and CRSCE-2. The only difference between the two variants is the Hash Arrays allocated.

For CRSCE-1 and CRSCE-2, there exists a Hash Array, called “Lateral Hash Array” (LHASH), depicted parallel to LSM. Each element of this array, identified as LHASH[y] should contain a hash of row y of the CSM.

For CRSCE-2, there also exists a second Hash Array, called “Vertical Hash Array” (VHASH), depicted parallel to VSM. Each element of this array, identified as VHASH[x] should contain a hash of column x of the CSM.

The total size (in bits) of the hash arrays (h) depends upon the variant of CRSCE being implemented. Assuming SHA-256 or SHA3-256 as the hash algorithm, we can define h as—

|  |  |
| --- | --- |
| CRSCE-1 | CRSCE-2 |
|  |  |

* 1. Cross Sum Calculation

The basic CRSCE compression operation is the calculation of the cross sums (LSM, VSM)

Assume the CSM is loaded left to right, top to bottom with bits from an input signal of size 8N bits.

Next iterate over the CSM bit field using (x, y) coordinates and for every CSM [x, y] element which is set (value = 1), increment LSM[y] and VSM[x].

The result is a cross sum of the input message block of size 2nb bits.

* 1. Hash Calculations and Collisions.

When iterating over CSM to calculate the Cross Sums, each element of CSM should be recorded into a hash calculation to determine LHASH[y] and VHASH[x].

The end state should be 256n bits of hash data representing the vertical and lateral hashes of the CSM bit field to prevent collisions in the decompression process and to localize decompression errors.

The purpose of LHASH and the CRSCE-2 VHASH is to both prevent collisions, guaranteeing a unique input-output pairing as well as to aid in identifying when the decompression process has made an incorrect decision during the solving process.

Rather than allocate a single hash at the end of the algorithm to detect a collision, LHASH, for example, allows a row-specific hash to detect an incorrect solution at the specific row. Likewise, for CRSCE-2, VHASH permits problem detection at a specific column.

Row and column-level hashing further collision-proofs CRSCE. Given that SHA3-256 has thus far (as of 2023) proven itself collision resistant and each row (and column when using CRSCE-2) has its own hash, the likelihood of a hash on reasonably sized CSM is statistically unlikely. Further, in the case of CRSCE-2, the likelihood that a CSM collision would be missed both at the row and column level is even more unlikely. This assumes that CSM is reasonably sized.[[12]](#footnote-12)

* 1. Compression Rate Calculation (Cr)
     1. Overview

A key design goal of the CRSCE algorithm is the ability to predict compression rates given only a message block size. This compression rate is the ratio of the compressed output signal to the raw input, expressed as a percentage. A higher percentage indicates greater performance.

As discussed, below, there are several considerations in determine the actual compression rate formula. But the final result is this—

|  |  |  |
| --- | --- | --- |
| CRSCE-1 | CRSCE-2 | |
|  | | | |  | |
| Variable | Definition | | |
| S | Number of Message Blocks in output signal | | |
| N | Cross-sum size. | | |
| B | Cross-sum width. | | |
| H | Hash size. For CRSCE-1 this is 256, for CRSCE-2 is 2 x 256 = 512[[13]](#footnote-13) | | |

* + 1. Base Cross-Sum Hash Compression Rate

The base compression rate is the sum of outputs for the cross sums and hashes over the total size of CSM (in bits):

This calculation assumes h represents the total hash size based on the CRSCE variant being evaluated. For CRSCE-1 the output contains only LHASH while for CRSCE-2 both LHASH and VHASH must be accounted for.

Assume a single hash is 32-bytes (256 bits). The total size of LHASH elements in a row is 256n (CRSCE-1 or CRSCE-2). But in CRSCE-1 there is no VHASH. Thus, h=256.

But for CRSCE-2 there are 256n bits of hashes for LHASH and another 256n bits for VHASH. Thus, the total hash length is twice the size of CRSCE-1 (256n + 256n) and therefore h=512.

|  |  |
| --- | --- |
| CRSCE-1 | CRSCE-2 |
|  | |  |

* + 1. Per-Block Calculation

Extending on the previous section, CRSCE is designed to allow an arbitrary-length signal to span some number of fixed-length message blocks(S). This means the compression rate becomes—

Where message blocks can be compressed in parallels and the order in which a message block is written to the output file cannot be guaranteed, a BlockId identifying message block order must be added to each. This further restates the compression rate formula, as illustrated:

Algebraically the fixed-length message block size means the above could be reduced. The number of message blocks at this point does not affect the final compression rate. But as demonstrated in the next section, this value S is nevertheless important.

* + 1. Header Overhead

For every compressed signal, CRSCE prepends the multi-block message with a fixed-length 160-bit header, consisting of the fields—

|  |  |
| --- | --- |
| Magic number | 16 bits |
| Version number | 8 bits |
| Options Flag field | 8 bits |
| Message Length | 64 bits |
| Message Block Count | 64 bits |
| Total Header Size | 160 bits |

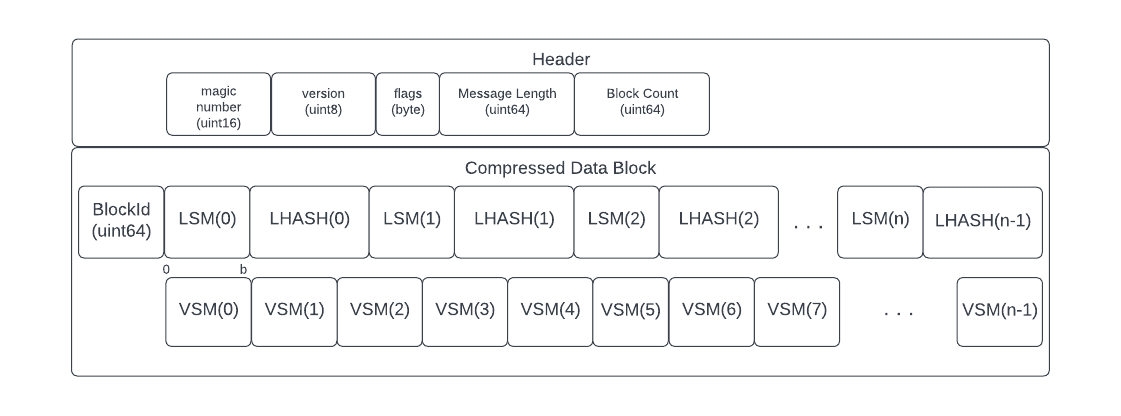
Thus, Cr is finally restated as its final form—

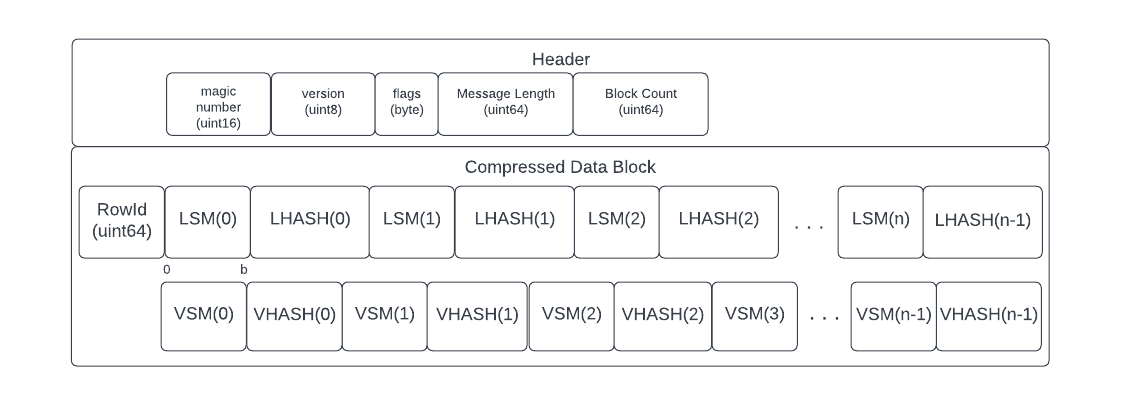
* + 1. Row/Column Structure Overhead

The algorithm’s output is formatted in such a way that we can process signals during both compression and decompression as independent message blocks, stored in arbitrary order.

This requires that each message block have a row and column data structure representing the LSM-LHASH pairs and VSM in the case of CRSCE-1 or LSM-LHASH and VSM-VHASH pairs in the case of CRSCE-2, and each of these row and column structures must be storable

To do this, each message block record must have an associated BlockId so that decompression can restate the uncompressed solution in proper order. These BlockId values each add 64 bits to the overhead.





* 1. Compression Rate Curve

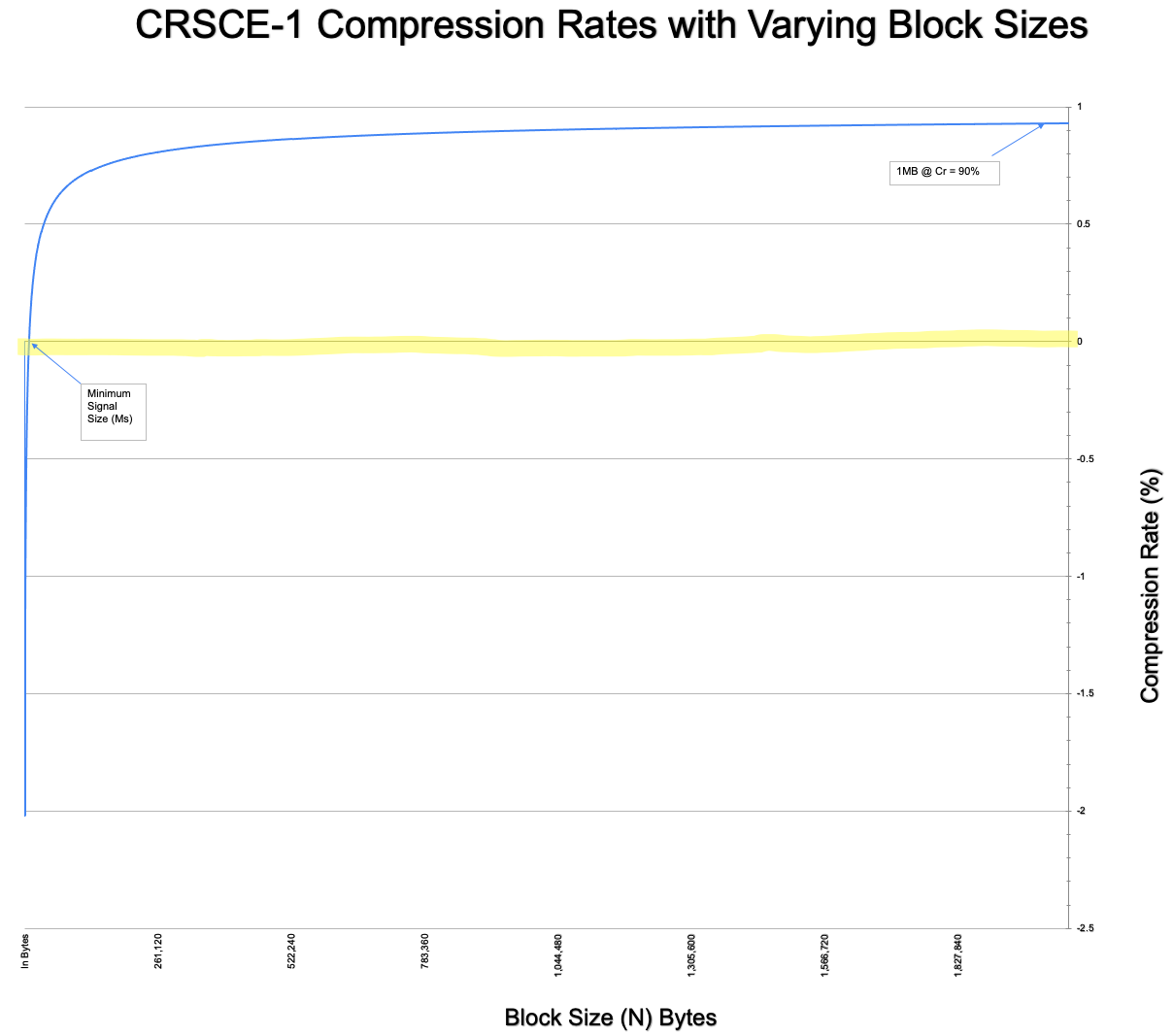
The compression rate curve (a graph of compression rates with varying message block sizes) shows that compression increase logarithmically for both CRSCE-1 and CRSCE-2, using the formula—

|  |  |
| --- | --- |
| CRSCE-1 | CRSCE-2 |
|  | |  |

The curves for the two variants also show that the minimum signal size before compression begins increases for CRSCE-2 over CRSCE-1. Below this minimum signal size, the algorithm fails to compress the signal (see section 2.2.10, below).

* + 1. CRSCE-1 Compression Curve and Data Table

The following graph depicts the compression rates for CRSCE-1:



In this graph we see the three points of concern for evaluating success of CRSCE-1: Minimum Signal Size (Ms), the 50% compression mark and the compression rate for a 1MB message block.

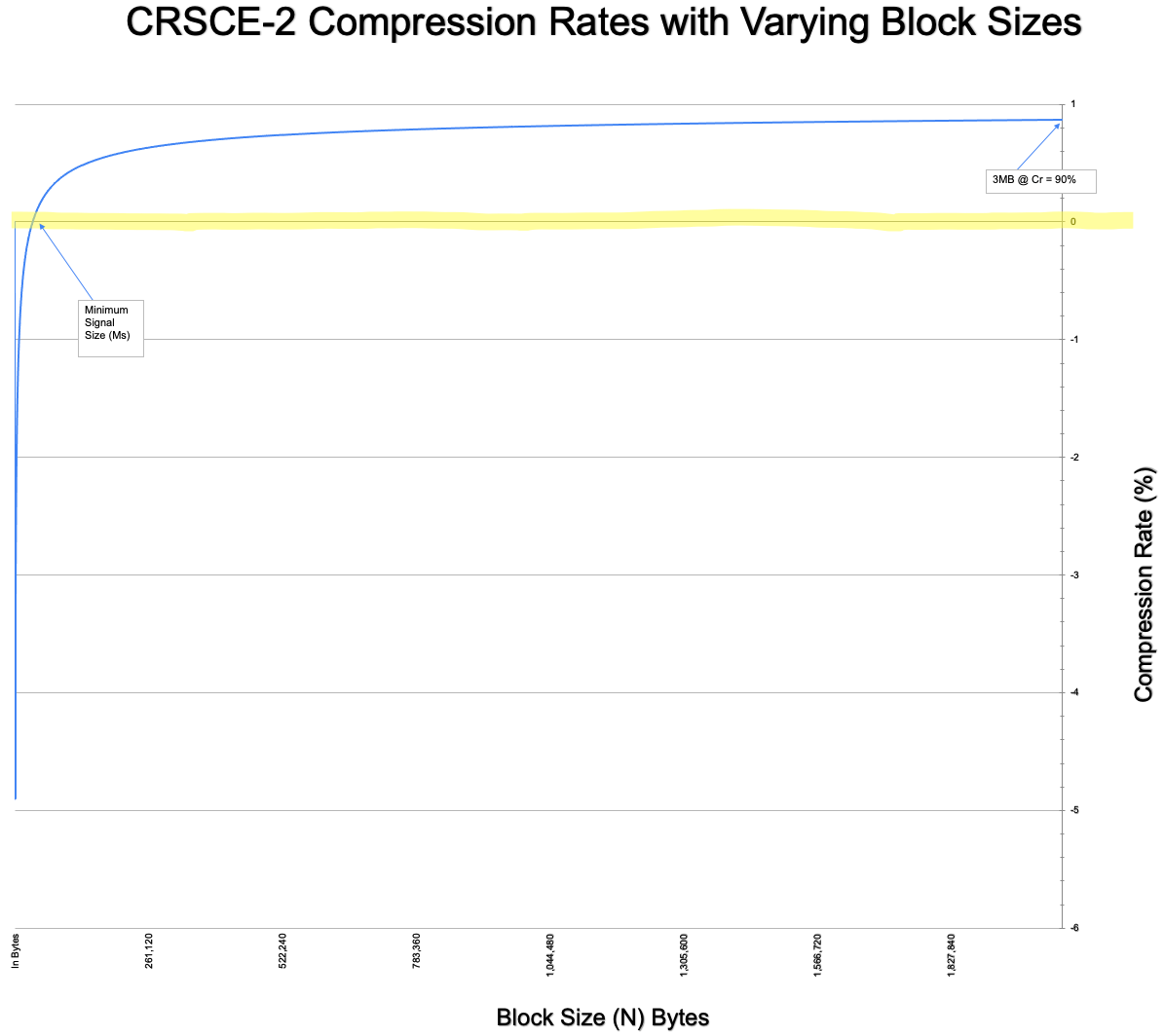
This graph is created from data summarized in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **CRSCE-1 Compression Rates** | | | |
| **Input Size (N) (in bytes)** | **CSM Size (in bits)** | **Compression** | **Output (in bytes)** |
|
| 1,024 | 8,281 | -199% | 3,099 |
| 2,048 | 16,384 | -112% | 4,348 |
| 3,072 | 24,649 | -74% | 5,366 |
| 4,096 | 33,124 | -50% | 6,216 |
| 5,120 | 41,209 | -35% | 6,930 |
| 6,144 | 49,284 | -23% | 7,576 |
| 7,168 | 57,600 | -14% | 8,188 |
| 8,192 | 65,536 | -7% | 8,732 |
| 9,216 | 73,984 | -1% | 9,344 |
| 10,240 | 82,369 | 4% | 9,858 |
| 11,264 | 90,601 | 9% | 10,337 |
| 12,288 | 98,596 | 13% | 10,783 |
| 13,312 | 106,929 | 16% | 11,228 |
| 14,336 | 114,921 | 19% | 11,639 |
| 15,360 | 123,201 | 22% | 12,050 |
| 16,384 | 131,769 | 24% | 12,461 |
| 32,768 | 262,144 | 46% | 17,564 |
| 65,536 | 525,625 | 62% | 25,041 |
| 131,072 | 1,048,576 | 73% | 35,356 |
| 262,144 | 2,099,601 | 81% | 50,381 |
| 524,288 | 4,194,304 | 86% | 71,196 |
| 1,048,576 | 8,392,609 | 90% | 101,423 |
| 2,097,152 | 16,777,216 | 93% | 143,388 |
| 4,194,304 | 33,558,849 | 95% | 204,231 |
| 8,388,608 | 67,108,864 | 97% | 288,796 |
| 16,777,216 | 134,235,396 | 98% | 411,331 |
| 33,554,432 | 268,435,456 | 98% | 581,660 |

For CRSCE-1 to achieve the minimum design goal of 50% compression, N=38KB.

* + 1. CRSCE-2 Compression Curve and Data Table

The following graph depicts the compression rates for CRSCE-2:



In this graph we see the three points of concern for evaluating success of CRSCE-2: Minimum Signal Size (Ms), the 50% compression mark and the compression rate for a 1MB message block.

This graph is created from data summarized in the following table[[14]](#footnote-14):

|  |  |  |  |
| --- | --- | --- | --- |
| CRSCE-2 | | | |
| **Input Size (N) (in bytes)** | **CSM Size (in bits)** | **Compression** | **Output (in bytes)** |
|  |
| 1,024 | 8,281 | -481% | 6,011 |  |
| 2,048 | 16,384 | -312% | 8,444 |  |
| 3,072 | 24,649 | -237% | 10,390 |  |
| 4,096 | 33,124 | -191% | 12,040 |  |
| 5,120 | 41,209 | -161% | 13,426 |  |
| 6,144 | 49,284 | -138% | 14,680 |  |
| 7,168 | 57,600 | -120% | 15,868 |  |
| 8,192 | 65,536 | -107% | 16,924 |  |
| 9,216 | 73,984 | -95% | 18,048 |  |
| 10,240 | 82,369 | -85% | 19,042 |  |
| 11,264 | 90,601 | -76% | 19,969 |  |
| 12,288 | 98,596 | -69% | 20,831 |  |
| 13,312 | 106,929 | -62% | 21,692 |  |
| 14,336 | 114,921 | -57% | 22,487 |  |
| 15,360 | 123,201 | -51% | 23,282 |  |
| 16,384 | 131,769 | -46% | 24,077 |  |
| 32,768 | 262,144 | -4% | 33,948 |  |
| 65,536 | 525,625 | 27% | 48,241 |  |
| 131,072 | 1,048,576 | 48% | 68,124 |  |
| 262,144 | 2,099,601 | 63% | 96,749 |  |
| 524,288 | 4,194,304 | 74% | 136,732 |  |
| 1,048,576 | 8,392,609 | 81% | 194,127 |  |
| 2,097,152 | 16,777,216 | 87% | 274,460 |  |
| 4,194,304 | 33,558,849 | 91% | 389,607 |  |
| 8,388,608 | 67,108,864 | 93% | 550,940 |  |
| 16,777,216 | 134,235,396 | 95% | 782,083 |  |
| 33,554,432 | 268,435,456 | 97% | 1,105,948 |  |

For CRSCE-2 to achieve the minimum design goal of 50% compression, N=137KB.

* 1. Minimum Signal Size (Ms)

The compression rate formula for Cr suggests that there is some message block size (N) below which CRSCE does not compress data but in fact *expands* data. This value is defined as the minimum signal size (Ms).

The Minimum Signal Size implies that for a message block size (N) there exists an output message size (S) and the following rules:

|  |
| --- |
|  |
|  |
|  |

Because the formula for Cr is a function of N and not dependent on message content, Ms is dependent on the formula for Cr and thus the CRSCE algorithm itself.

The value Ms is a value of N, where Cr = 0.

But because the calculation of Cr is a function of n and b, where both are ceiling functions, the answer must be determined by some numeric solver. To this end, the following program solves for Ms[[15]](#footnote-15)—

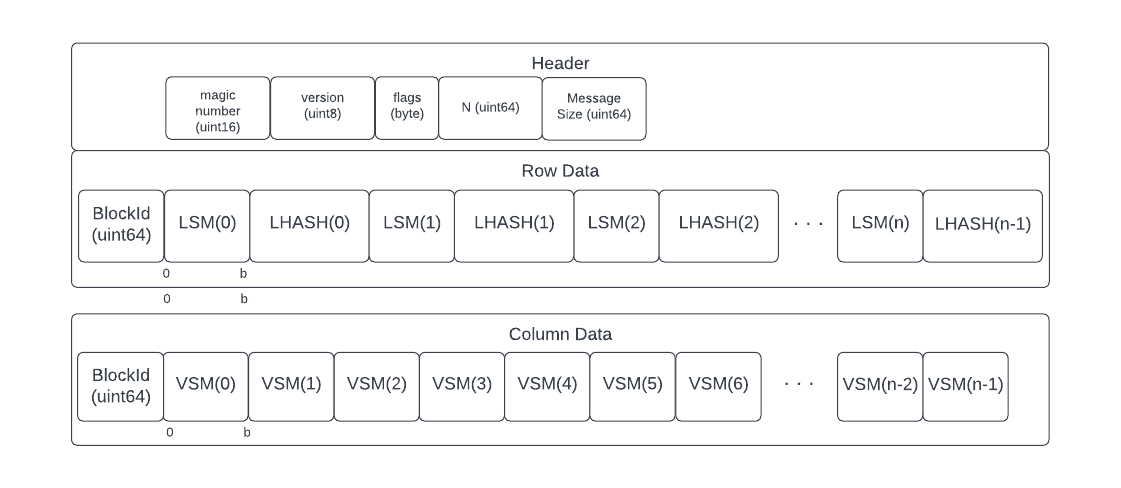
|  |
| --- |
| #include <iostream>  #include <cstdlib>  using namespace std;  const int hashSize = 256;  const int headerSize = 160;  const int blockIdSz = 64;  uint64\_t calculateMinimum(uint64\_t start, uint64\_t stop, int crsceAlgorithm) {  // Minimum signal size  uint64\_t Ms = 0;  // Hash size (based on CRSCE variant)  uint64\_t h = hashSize;  if (crsceAlgorithm == 2) {  h \*= 2; // increase h to 2 x hashSize (LHASH+VHASH) assuming CRSCE-2  }  for (uint64\_t N = start; N < stop; N++) {  // Cross Sum Size  uint64\_t n =static\_cast<int64\_t>( std::ceil( std::sqrt(8 \* N) ) );  // Cross Sum Width  uint64\_t b = static\_cast<int64\_t>( std::ceil( std::log2(n) ) );  // Compressed Signal Size  uint64\_t numerator = 2 \* b \* n + h \* n + blockIdSz + headerSize;  // Csm Array Size.  uint64\_t denominator = n \* n;  // Padding size  uint64\_t P = ( n \* n - 8 \* N);  if ( (P < n) && (numerator < denominator) ) {  Ms = N;  return Ms;  }  }  throw std::runtime\_error("ms cannot be determined");  }  int main(int argc, char \*argv[]) {  if (argc != 2) {  std::cerr << "Usage: " << argv[0] << " CRSCE\_ALGORITHM (1|2)" << std::endl;  return 1;  }  int crsceAlgorithm = std::stoi(argv[1]);  if (crsceAlgorithm != 1 && crsceAlgorithm != 2) {  std::cerr << "Invalid input. Expects 1 or 2" << std::endl;  return 1;  }  uint64\_t startSz = 1024; // Upper search bounds  uint64\_t stopSize = 1048576; // Lower search bounds  try {  uint64\_t Ms = calculateMinimum(startSz, stopSize, crsceAlgorithm);  uint64\_t minimumSignalSizeInBits = Ms \* 8;  uint64\_t minimumSignalSizeInBytes = Ms;  std::cout << " minimum signal size(bytes) (Ms): " << minimumSignalSizeInBytes << std::endl;  std::cout << " minimum signal size(bits): " << minimumSignalSizeInBits << std::endl;  } catch (const std::runtime\_error &e) {  std::cerr << "Error: " << e.what() << std::endl;  }  return 0;  } |

The above program produces the following results:

|  |
| --- |
| samcaldwell@mbp monorepo % clear; g++ -o build/calculate-minimum cmd/crsce/calculate-minimum/main.cpp    samcaldwell@mbp monorepo % ./build/calculate-minimum 1  minimum signal size(bytes) (Ms): 9419  minimum signal size(bits): 75352  samcaldwell@mbp monorepo % ./build/calculate-minimum 2  minimum signal size(bytes) (Ms): 35445  minimum signal size(bits): 283560  samcaldwell@mbp monorepo % |

Thus, for CRSCE-1, the minimum signal size is 9,419 bytes. For CRSCE-2, the minimum signal is 35,445 bytes.

* 1. Compressed Signal Format
     1. Overview



The compressed signal from CRSCE-1 (above) varies from CRSCE-2 only in the Column data where VSM becomes a VSM-VHASH pair. Otherwise, the structure is the same.

We see the length of the signal is bits per message block. Where there exist some number (S) message blocks in the final output, the total size is .

Further because each row or column structure has its relative block Id affixed.

* + 1. Header
       1. Structure

Every compressed signal will start with a 160-bit header.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Magic Number | Version | | Option Flags | Message Size | Block Count |
| (uint16) | (byte) | (byte) | | (uint64) | (uint64) |

* + - 1. Magic Number

The magic number identifies the file format.

The magic number is an unsigned 16-bit integer.

Provisionally, the CRSCE magic number 1337 will be written as 0x37 0x13.

* + - 1. Version

The CRSCE version number 0 will be reserved for any in-development version.

The first release will use the version 0x01.

Version numbers will be assigned sequentially.

* + - 1. Option Flags

The CRSCE output header will include an Option flags field indicating which options were selected when compressing the source file.

The following bit values are assigned for the Option Flags field:

|  |  |
| --- | --- |
| Bit Position | Description |
| 0 | Variant Flag (3.3.1.4.1) |
| 1 | Byte-Alignment Flag (3.3.1.4.2) |
| 2 | Reserved for Future Use (3.3.1.4.3) |
| 3 | Reserved for Future Use (3.3.1.4.3) |
| 4 | Reserved for Future Use (3.3.1.4.3) |
| 5 | Reserved for Future Use (3.3.1.4.3) |
| 6 | Reserved for Future Use (3.3.1.4.3) |
| 7 | Reserved for Future Use (3.3.1.4.3) |

* + - 1. Option Flag Field: Variant Flag

The variant flag identifies the algorithm variant used to compress the signal described in the output file.

0 = CRSCE-1

1 = CRSCE-2

* + - 1. Option Flag Field: Byte-Alignment Flag

The byte-alignment flag identifies whether or not the LSM-LHASH (CRSCE-1, -2) and VSM-VHASH (CRSCE-2) are packed or written using byte-aligned values.

0 = packed

1 = byte-aligned.

The byte-alignment flag identifies whether the algorithm will write LSM and VSM values using their packed (b-sized) value or using a byte-aligned encoding.

For example, a 1MB signal in CRSCE-1 has a 12-bit cross-sum width (b). But where b=12, the result is not aligned on a byte (8-bit) boundary. Accordingly in a packed (non-byte aligned) scenario, both LSM[y] and LHASH[y] must be written to the output as bit streams using more compute cycles per LSM-LHASH pair. In a byte-aligned scenario, where b=12, the actual data written to output would be an unsigned 16-bit integer (uint16) since 16 is the next byte-aligned integer size.

* + - 1. Reserved Future Use Flags

All other flag bits should be set to zero (0) and any secure implementation of CRSCE should verify this zero value for reserved-used flags to prevent potential abuses.

* + - 1. Message Size

The message length (L) field is an unsigned 64-bit integer representing the total byte-length of the uncompressed signal.

This value is needed to prune any padding from the last message block in the decompression process.

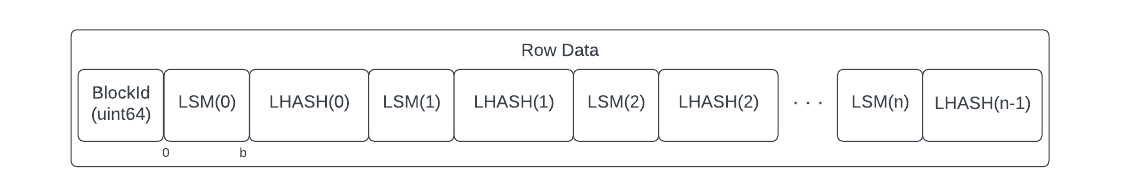
From this value, we can calculate the size of the message blocks—

From this value N, CRSCE can derive both n and b, respectively to decompress the message block.

* + - 1. Block Count

The CRSCE header includes a Block Count indicating the number of CSM message blocks in the compressed output file which must be processed to recreate the original uncompressed signal.

* + 1. Row Structure



* + - 1. Structure
         1. BlockId

The BlockId is a 64-bit unsigned integer representing the position of the message block in the original signal structure, starting at zero (0).

* + - * 1. LSM Count

The LSM count is a field of b bits (packed) or a byte-aligned equivalent representing the number of set bits in the associated row.

* + - * 1. LHASH Byte Array

The LHASH Byte Array is a 32-byte (256 bit) packed or byte-aligned field representing the hash of a single row.

* + - 1. LSM-LHASH Pairs

The entire row structure of all LSM and LHASH pairs is an atomic unit for storage to the output signal.

The LSM-LHASH pairs must be written in their row order.

Because the length of the row structure is predictable as nb+256n[[16]](#footnote-16), the structure can be pre-allocated and LSM-LHASH pairs can be written in arbitrary order to their appropriate bit positions in the image.

* + - 1. Benefits of Interleaving

By interleaving LSM and LHASH as an atomic unit, the row structure can be pre-allocated as a unit of memory and each row can potentially be compressed independently by parallel processes or execution threads. The LSM and LHASH can then be packed or written as byte-aligned units without respect for the other rows.

Since a row is processed exactly once by the compression algorithm, interleaving the LSM-LHASH pairs means only a single LSM and LHASH structure needs to be allocated for each compressing thread rather than an entire LSM and LHASH array of n elements. For a single threaded compressor, this saves (n-1)(b+256n) bits of memory.

* + - 1. Packed or Byte-Aligned Output

Flushing the LSM-LHASH pair can occur in one of two ways: packed arrays or unpacked arrays, as determined by bit 1 in the header flags. See 2.12.1.

But because LSM elements only keep b bits of content (not the full value of LSM[y]), the value flushed to the output stream is not byte aligned. For example, in CRSCE-1, an LSM[y] element would be as small as uint16. But the actual value would only be 12-bits (b=12 where N=1MB). This means LSM[y] must be flushed to the output stream as a bitwise operation.

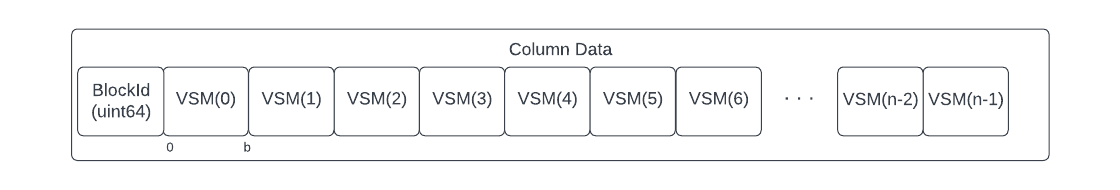
LHASH[y] is a 32-byte (256-bit) element. If output is byte-aligned the flushing operation is a simple byte array copy to the target. But if the output is to be packed, it must be written bit by bit to the output stream.

* + 1. CRSCE-1 Column Structure

In CRSCE-1, the entire VSM must remain in memory for the duration of a message block’s compression process since every column must be processed one time for every row in the CSM.

CRSCE-1 stores column (VSM only) data as either a packed or byte-aligned bit field of VSM elements. Byte-alignment is determined by the Option Flag in the header.

The Column data record for a given message block is identified by the BlockId, where the block Id is the index of the message block order in the original input signal.

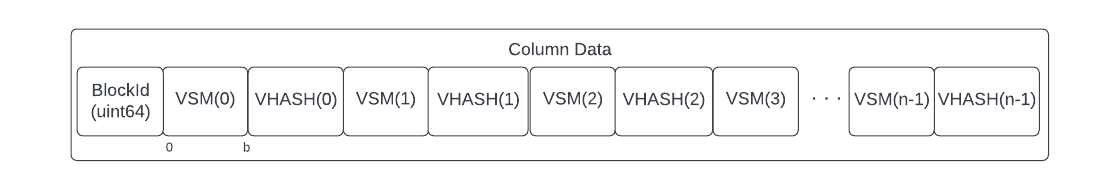


* + - 1. CRSCE-2 Column Data

In CRSCE-1, the entire VSM must remain in memory for the duration of a message block’s compression process since every column must be processed one time for every row in the CSM.

CRSCE-2 stores column (VSM, VHASH) data as either packed or byte aligned fields depending on the value of the byte-alignment flag in the header’s Option Flags byte. The storage of interleaved VSM-VHASH pairs is the same as the Row Data definition in 3.3.2.

The Column data record for a given message block is identified by the BlockId, where the block Id is the index of the message block order in the original input signal.



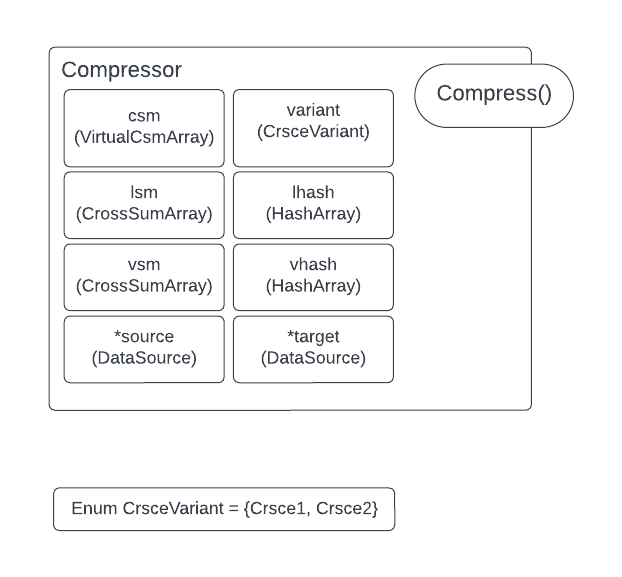
* 1. Endianness

Output files written using little endian.

1. Compression
   1. CrsceCompressor Class
      1. Overview

The CRSCE class represents the top-level class for implementing CRSCE-1 and CRSCE-2.

Given an arbitrary data source and target, the class should be able to read from the signal as a series of message blocks, compress or decompress the source into the target.



* + 1. Properties

|  |  |  |
| --- | --- | --- |
| Property | Type | Description |
| source | BitReader | This is a readable data source from which bytes can be read. |
| target | BitWriter | This is a writable data target to which bytes can be written. |
| variant | CrsceVariant (Enum) | 0=Crsce-1  1=crsce-2 |

* + 1. Class Constructor

The class constructor will initialize the BitReader and BitWriter and other internal state for the class. This includes starting the asynchronous reader/writer threads (BitReader::Start and BitWriter::Start)

The class constructor takes as input a source, target, variant and byteAlign value matching the properties of the class.

|  |  |  |
| --- | --- | --- |
| Input | Type | Description |
| \*source | BitReader | Reference to a bit reader |
| \*target | BitWriter | Reference to a bit writer |
| variant | CrsceVariant | Crsce algorithm variant |

* + 1. Class Destructor

The class destructor will close the BitReader/writer objects, stop the reader/writer threads and free any memory allocated.

* + 1. Compress Method

The compress method will start the process of compressing a given data source into the data target using the assigned CRSCE variant.

This method takes no inputs and returns nothing.

The compress method will…

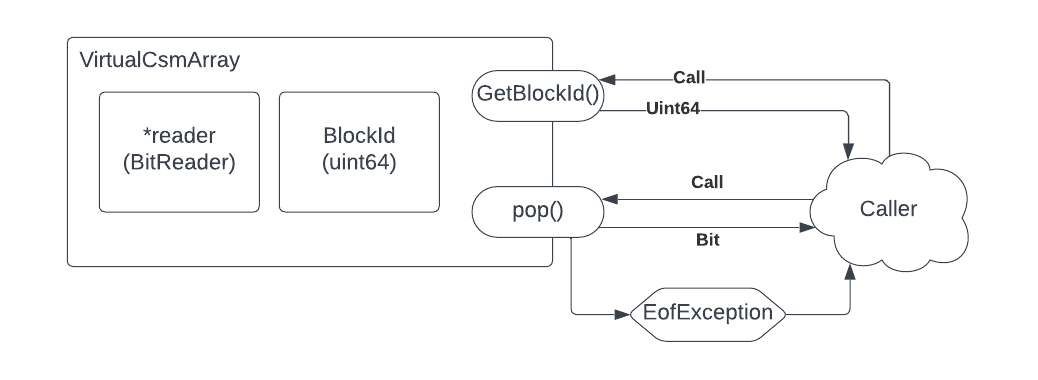
* Query csm for block size (n) and block count (S).
* Initialize LSM, VSM, LHASH and VHASH using n.
* On EofException:
  + terminate loops,
  + flush
  + exit.
* Iterate over
  + Loop over
    - Loop over
      * if bit is set…
        + increment LSM[y]
        + Increment VSM[x]
    - At the end of the row…
      * Call LSM.Flush(y)
      * Call LHASH.Flush(y)
  + At the end of the last row of the last column…
    - Call VSM.Flush(x)
    - Call VHASH.Flush(x)
* Close-out source
* Close-out target
  + 1. GetRate Method

The CrsceCompressor::GetRate() method takes a block size (uint16) and CrsceVariant input and returns the expected compression rate as float64.

* 1. VirtualCSMArray Class
     1. Overview

The message block (CSM) does not need to be loaded into memory for compression as a single block of data. Instead, it can be read as a bit stream into the process.[[17]](#footnote-17)

The CSM should be implemented as an object class programmatically with a constructor that opens the data source and allows the data source to be read into its internal state byte-by-byte asynchronously to an internal queue then presented to the compression function bit-by-bit.



To improve the performance of decompression, the version 1.0 implementation of CRSCE will be constrained to—

for CRSCE-1.

* + 1. Properties

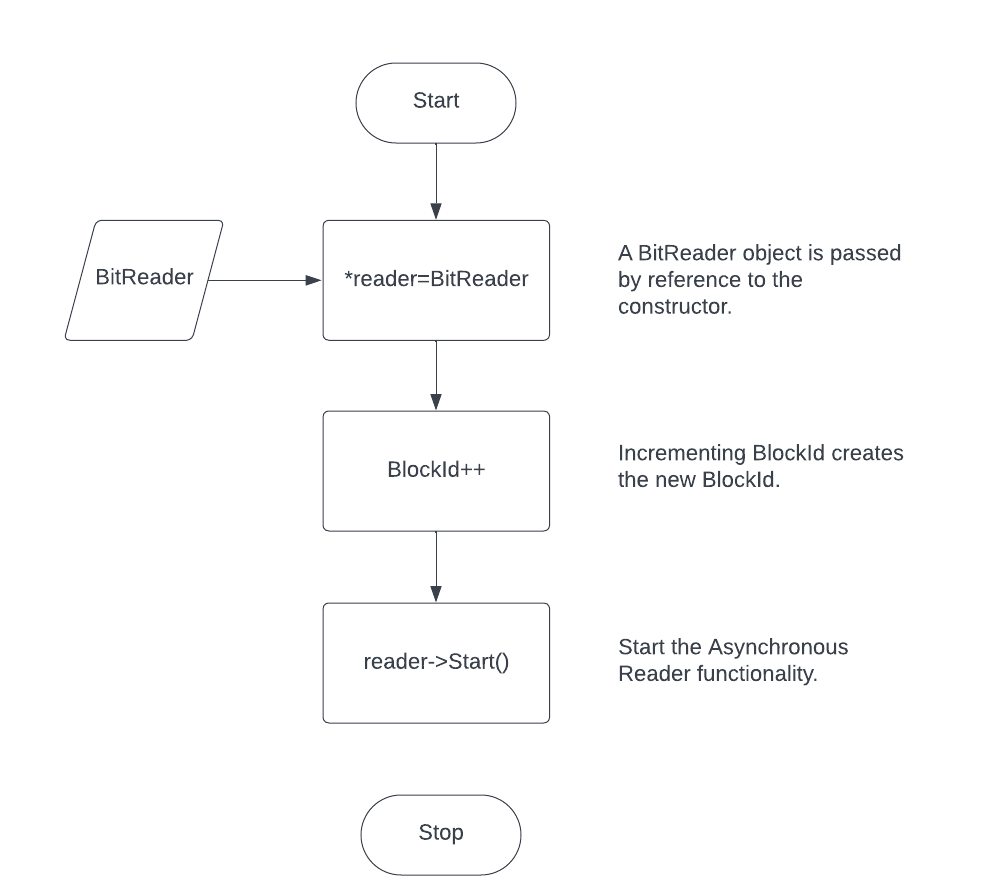
The CSM virtual array class will have the following internal states:

|  |  |  |
| --- | --- | --- |
| Reader | BitReader | The class will maintain a private BitReader instance which will be used as the Asynchronous Reader to read from the data source. |
| BlockId | Uint64 | This property identifies each class instance’s block id.  This BlockId must be exposed through the GetBlockId() method |

* + 1. Class Constructor

When the CSM virtual array class is instantiated, it will initialize the BitReader object with a data source and launch its asynchronous reader functionality.

|  |  |  |
| --- | --- | --- |
| Inputs | Type | Description |
| \*reader | BitReader reference | This is a reference to a BitReader which will allow the class to read from the common data source. |
| BlockId | Uint64 | This is the new instance’s block id. |



* + 1. Class Destructor

When class instance is no longer needed, it should execute its class destructor, closing the network connection, terminating the BitReader and freeing all memory allocated by the class.

* + 1. GetBlockId Method

This method will return the unsigned 64-bit integer representing the current class instance BlockId.

This method takes no input.

* + 1. Pop Method

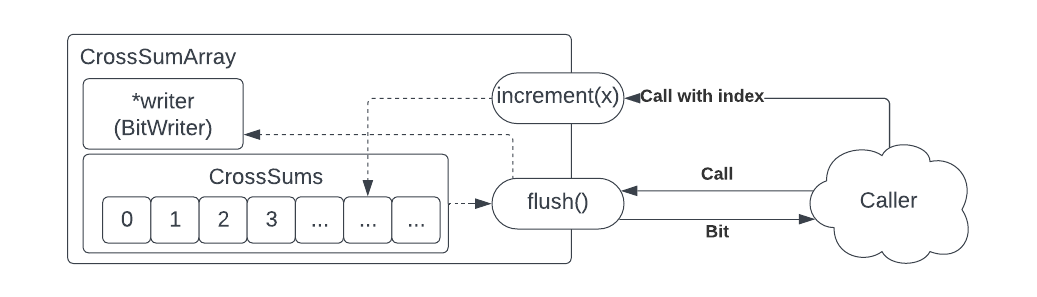
For each invocation of the Pop method, the same shall return one bit. If the BitReader is at the end of its file (EOF), an EofException will be thrown by the BitReader. The VirtualCsmArray Class will—

- Invoke BitReader::Stop() to terminate the Asynchronous Reader

- Throw the EofException for the caller to catch.

* 1. CrossSumArray Class
     1. Overview

The CrossSumArrray Class is used to create LSM and VSM cross-sum arrays.



* + 1. Properties

|  |  |  |
| --- | --- | --- |
| Property | Type | Description |
| crossSum | Array of unsigned integers. | The crossSum is an unsigned integer of size . |
| Output | BitWriter pointer | A reference to a BitWriter Class instance |

* + 1. Class Constructor

The CrossSumArray class constructor will initialize a cross sum array of n elements, define its output stream (by reference) and configure its byteAlign setting, as described.

|  |  |  |
| --- | --- | --- |
| Input Name | Input Type | Description |
| n | uint16 | Number of cross sum elements to allocate for the class instance.  As required by 3.4.4 (LSM-LHASH Pairs) n=1 when the class is used for LSM, allocating only one cross sum element.  For VSM, however, n shall be set to the value defined in 3.2.3— |
| Output | BitWriter pointer | The output stream object is passed by reference, creating a shared output stream to which information can be written when the Flush method is called.  The Output stream must provide a bit, byte and byte-array write capability. |

* + 1. Class Destructor

The class shall include a class destructor to free all memory allocated to the class.

* + 1. Increment Method

The class shall include a public method called “increment” which will allow the caller to increment the count of a specific cross sum value.

The increment method will accept one uint16 parameter which will indicate which cross sum value will be incremented.

|  |  |  |
| --- | --- | --- |
| Input | Type | Description |
| x | uint16 | X represents the number of elements in the cross sums. |

Increment has no return value.

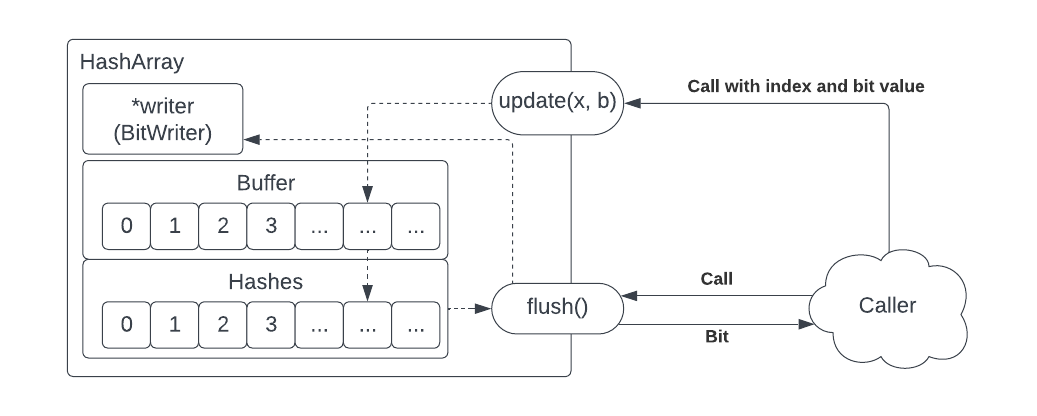
* + 1. Flush Method

The Flush method will iterate over the internal cross sum array of count elements and write each element to the BitWriter.

This assumes the BitWriter will handle any decision to pack bits or byte-align the bits being written to the output.

* 1. HashArray Class
     1. Overview

The HashArray class is used to store an array of updatable hashes (e.g., LHASH, VHASH). Each instance of the class contains an array of hash objects, both wrapping the hash algorithm for ease of replacement.



* + 1. Properties

|  |  |  |
| --- | --- | --- |
| Property | Type | Description |
| Buffer | Byte Array | This is a bit buffer (byte) array of n-number of bytes.  For every hash element, when a new bit is written to the HashArray, that bit is pushed into the corresponding Buffer byte. When that byte is full, the byte is pushed to the hash object.  When the flush method is called, the current state of the Buffer is pushed to the hash object as-is so the hash can be calculated. |
| hash | An internal hash object. | This is a SHA3-256 hash generator object which will accept multiple updates to calculate a hash of the overall signal. |
| Output | BitWriter pointer | A reference to a BitWriter Class instance |
| Closed | Boolean | When the Flush method is invoked, the Closed state is set to true.  When the Increment method is invoked, if Closed is true, an exception is thrown.  Since the flush method first flushes the current Buffer to the hash object before writing the resulting hash to the output object, this property ensures no further information can be added to the Buffer and hash object since it would produce an incorrect result. |

* + 1. Class Constructor

The class constructor will initialize an array of n hash elements and a shared BitWriter facility for writing hash array values to the compressed output target, using a packed or byte-aligned approach depending on the ByteAlign value.

|  |  |  |
| --- | --- | --- |
| Input Name | Input Type | Description |
| n | uint16 | The number of hash array and Buffer (byte) elements in the internal class state. |
| Output | BitWriter pointer | The output stream object is passed by reference, creating a shared output stream to which information can be written when the Flush method is called.  The Output stream must provide a bit, byte and byte-array write capability. |

* + 1. Class Destructor

The class shall include a class destructor to free all memory allocated to the class.

* + 1. Update Method

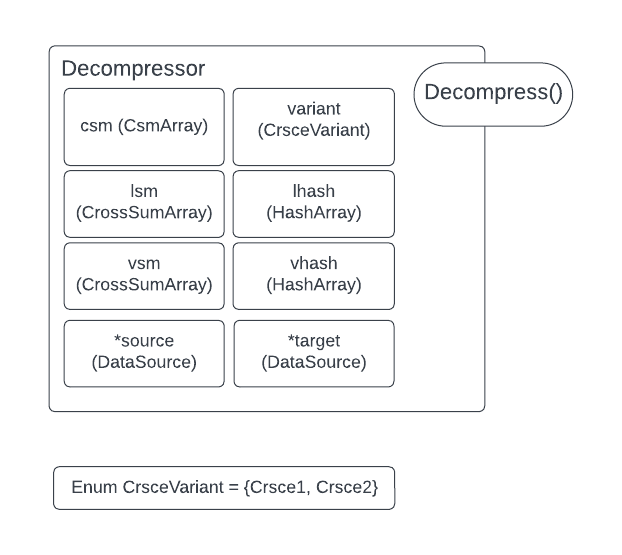
The update method accepts a bit as its input, and the value of this bit is pushed onto the byte buffer property until it is filled with eight bits, at which time the byte buffer is written to the hash property (updating the hash calculation). When byte buffer is flushed to the hash property, the byte buffer will be reset to 0x00 for the next invocation.

* + 1. Flush Method

The Flush method render a bytes array of the current hash state then write the byte array either as a packed bit stream or byte-aligned string to the output (BitWriter) stream.

1. Decompression
   * 1. CrsceDecompressor Class
        1. Overview

The CrsceDecompressor class implements the CRSCE decompression process.



* + - 1. Properties

|  |  |  |
| --- | --- | --- |
| Property | Type | Description |
| \*csm | CsmArray | The CSM array into which a message block will be decompressed. |
| \*lsm[] | CrossSumArray | An array of CrossSumArray objects for LSM.  LSM[0] –Initial LSM  LSM[1]—Actual LSM  Each element of \*lsm will contain 0,…,n cross sum counts |
| \*vsm[] | CrossSumArray | An array of CrossSumArray objects for VSM.  VSM[0] –Initial VSM  VSM[1]—Actual VSM  Each element of \*vsm will contain 0,…,n cross sum counts |
| \*source | BitReader | A data source reader |
| \*target | BitWriter | A data source writer |
| Variant | CrsceVariant | The CRSCE variant indicator |
| \*lhash | HashArray | The lateral hash array |
| \*vhash | HashArray | The vertical hash array |

* + - 1. Class Constructor

The class constructor takes the following inputs and initializes the class instance state:

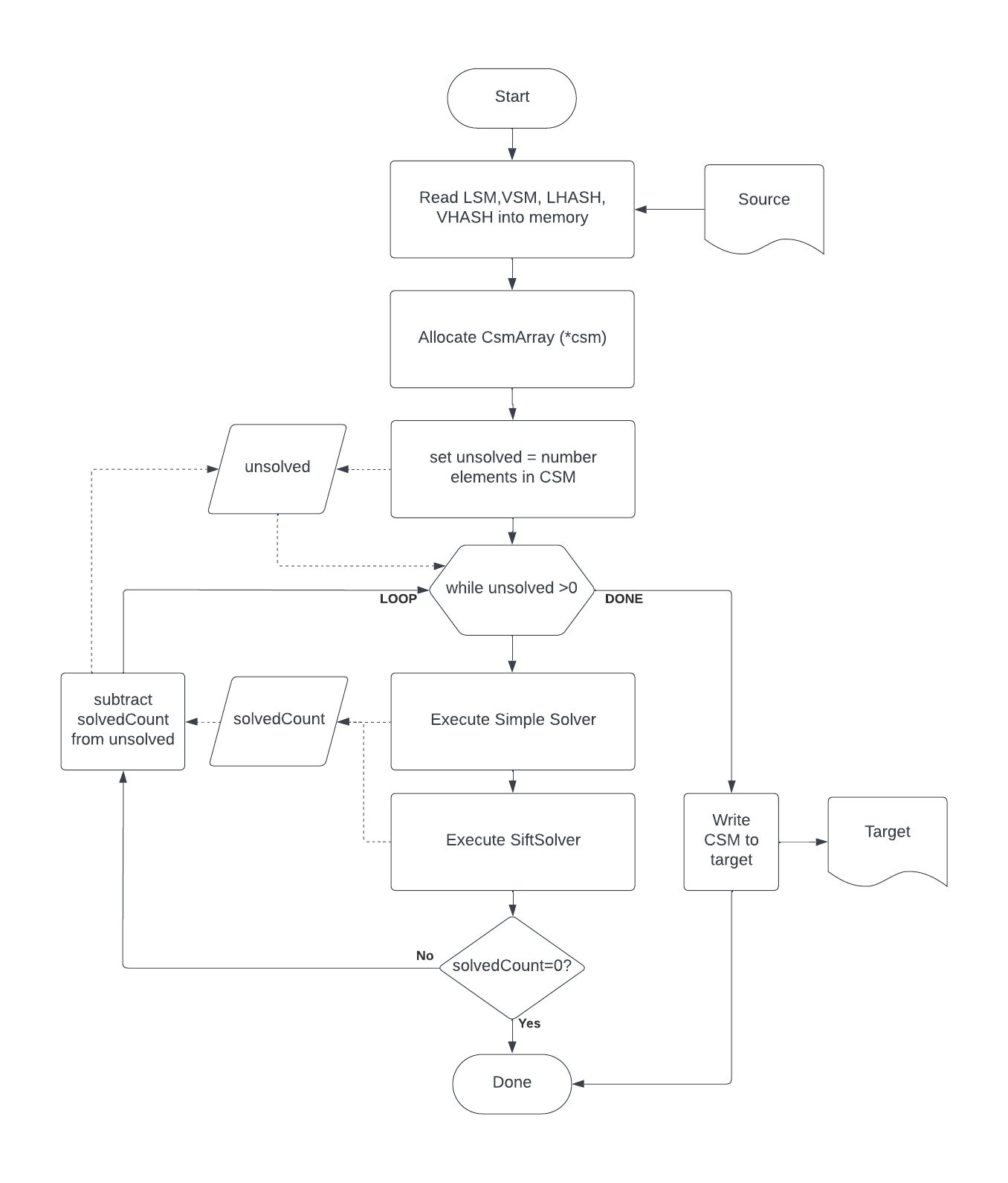
|  |  |  |
| --- | --- | --- |
| Property | Type | Description |
| \*source | BitReader | A data source reader |
| \*target | BitWriter | A data source writer |
| Variant | CrsceVariant | The CRSCE variant indicator |
| \*lhash | HashArray | The lateral hash array |
| \*vhash | HashArray | The vertical hash array |

* + - 1. Class Destructor

The class destructor frees any connection to allocated memory or external memory objects.

* + - 1. Decompress Method

The decompress method executes two private methods (SimpleSolve and SiftSolve) some number of times to complete the decompression process, as illustrated.



* + 1. SimpleSolve Method

The SimpleSolver class implements the first stage of decompression.

The SimpleSolve solution iterates over the LSM and VSM cross sum arrays to identify any LSM or VSM element with a value of 0 or n.

SimpleSolver applies these rules:

If LSM[y] == 0: CSM[0..n, y] = 0

If LSM[y]== 1: CSM[0…n, y] = 1

f VSM[y] == 0: CSM[0..n, y] = 0

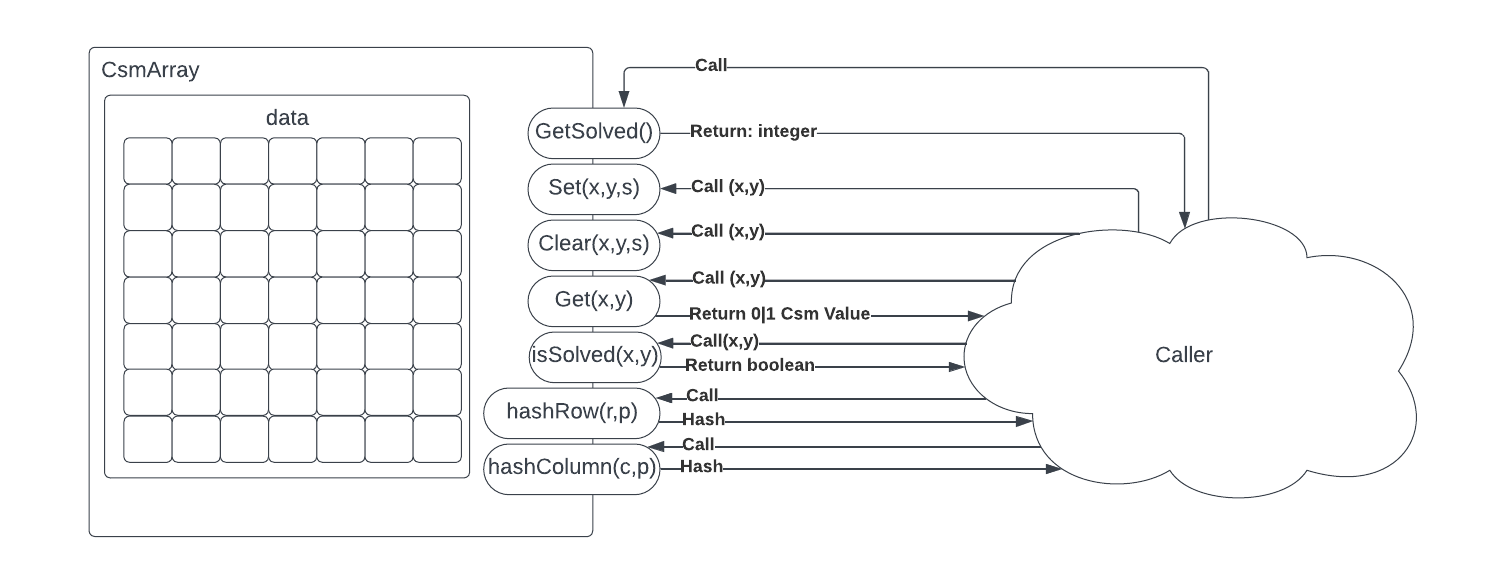
If VSM[y]== 1: CSM[0…n, y] = 1

Where SimpleSolver solves any row or column, it should continue with a subsequent pass since solving one row or column may cause other rows or columns to be solved with another evaluation.

SimpleSolver is the least expensive of the decompression stages, much less expensive than performing bitwise hashes of rows and columns.

* + 1. SiftSolve Method
  1. CsmArray Class
     + 1. Overview

The CsmArray is the in-memory representation of a single CSM message block at various stages of decompression up to and including the final result.[[18]](#footnote-18)



* + 1. Properties

|  |  |  |
| --- | --- | --- |
| Property | Type | Description |
| Data | Array of CsmElement | A linear array of n2 CsmElements |

* + 1. Class Constructor

The class constructor will allocate the data array of n2 elements with each element being w bits wide. See CsmElement.

|  |  |  |
| --- | --- | --- |
| Input | Type | Description |
| n | Uint16 | The number of rows and columns in CSM. |

* + 1. Class Destructor

The class destructor will free all memory assigned to the class.

* + 1. Linear Addressing and Cache Performance

The CsmArray::data array is represented as a two-dimensional array. Mathematically, CRSCE processes CSM as a two-dimensional array.

The CsmArray class stores the CSM elements as a one-dimensional array to ensure optimal cache performance. Because the one-dimensional array is stored sequentially in memory, the CPU can easily predict which memory will be accessed next as the decompression process traverses CSM bit field. This could also have benefits in the compiler where loop unrolling and vectorization can be more efficient.

One caveat to the above is the possibility of decompressing vertically, where the cache may not anticipate the coordinate translation to predict what memory may be needed next.

* + 1. GetSolved Method

The CsmArray::GetSolved() method returns the count of solved CSM elements. This is the count of CSM elements with their solved bit set.

For performance this uses an internal solved counter to avoid the need to inspect the solved bit of each CsmArray::data element. To make this work, when the CsmArray::Set() or ::Clear() methods are called with their s parameter true, the counter must be incremented.

* + 1. Set Method

The CsmArray::Set(x, y, s) method will set (1) the bit at point (x,y) in the CsmArray::data array.

An optional Boolean s parameter will set the solved bit for the CSM element if s==true. By default, s==false.

|  |  |  |
| --- | --- | --- |
| Input | Type | Description |
| x | Uint16 | Column coordinate |
| y | Uint16 | Row coordinate |
| s | Boolean | Optional indicator to mark the element as “solved.” |

* + 1. Clear Method

The CsmArray::Clear(x, y, s) method will clear (0) the bit at point (x,y) in the CsmArray::data array.

An optional Boolean s parameter will set the solved bit for the CSM element if s==true. By default, s==false.

|  |  |  |
| --- | --- | --- |
| Input | Type | Description |
| x | Uint16 | Column coordinate |
| y | Uint16 | Row coordinate |
| s | Boolean | Optional indicator to mark the element as “solved.” |

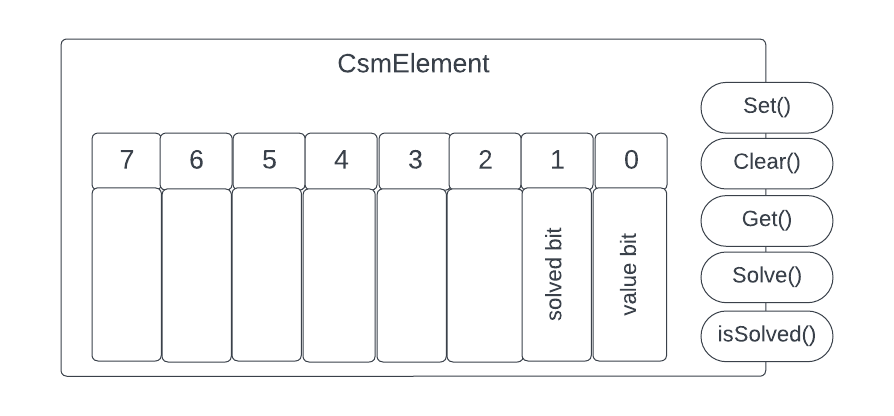
* + 1. Get Method

The CsmArray::Get(x, y) method will look up the current value of the CSM element at (x, y) and return its bit value.

* + 1. isSolved Method

The CsmArray::IsSolved(x, y) will return a Boolean result if the solved bit is set (1) or clear (0)

* 1. CsmElement Class
     1. Overview



The CsmArray class contains an array of elements (CsmElement), where each element represents a the current state of CSM during decompression.

The CsmElement stores its state as a single byte with a “solved bit” indicating the element is solved (and thus should be read-only) as well as a “value bit” indicating the current value (solved or proposed) for the element.

* + 1. Properties

The only internal state of CsmElement is a single byte.

* + 1. Class Constructor

The class constructor has no operation.

* + 1. Class Destructor

The class destructor has no operation.

* + 1. Set Method

The CsmElement::Set() method will set the bit value of the class instance.

An exception will be thrown if CsmElement::Set() is called and the solved bit is already set (1).

* + 1. Clear Method

The CsmElement::Clear() method will clear the bit value of the class instance.

An exception will be thrown if CsmElement:: Clear () is called and the solved bit is already set (1).

* + 1. Get Method

The CsmElement::Get() method will return the bit value of the class instance.

The result returned from Get() will mask out the solved bit and return only bit[0].

* + 1. Solve Method

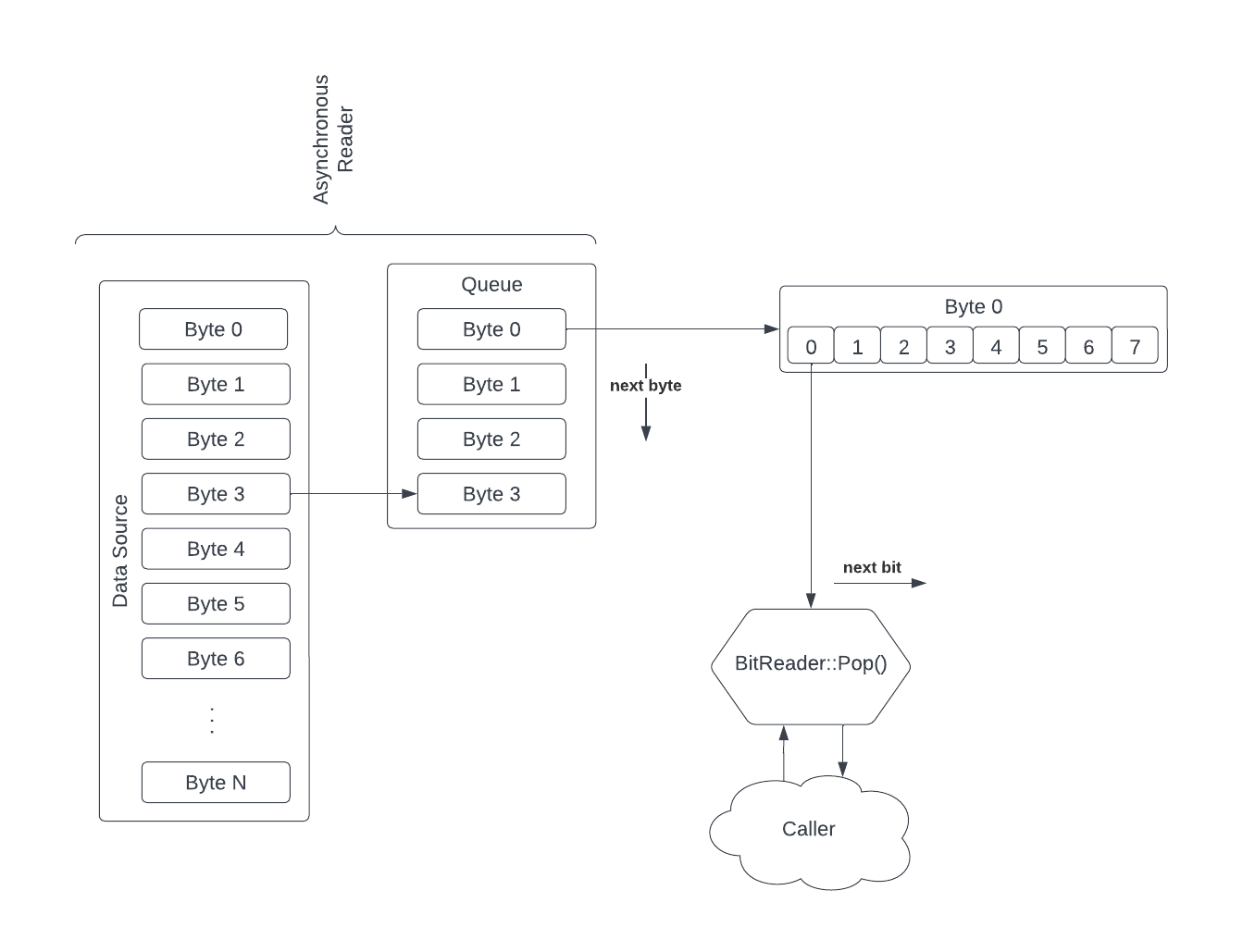
The CsmElement::Solve() method will set the solved bit (position 1).

The solved bit is WORM (Write-once, read many). Once the Solve() method is invoked for a CsmElement class instance, the solved bit cannot be cleared. This is intentional as the decompression process should not set this bit until it is certain of an element’s state.

* + 1. isSolved Method

The CsmElement::isSolved() method will return the value of bit(1)—the solved bit.

1. General Data Structures
   1. BitReader Class
      1. Overview



The BitReader is a class which opens a connection to some data source (e.g., a file) and reads a set of bytes from this data source asynchronously, writing each byte to a queue in the background. The consumer can then call the Pop method to obtain a single bit from the byte stream.

* + 1. Properties

|  |  |  |
| --- | --- | --- |
| Property | Type | Description |
| source | File handle or other data connector | This is a data connector capable of reading from some arbitrary file source. |
| Queue | Queue | A thread-safe queue of bytes |
| CurrentByte | Byte | The current byte being read by the Pop method |
| EOF | Bool | End of File flag |

* + 1. Class Constructor

The class constructor takes a data source handle (e.g., file handle) and queue size (integer) input and initializes the internal state of the class.

* + 1. Class Destructor

The class destructor frees all memory allocated by the class and stops the asymmetric reader thread/process.

* + 1. Start Method

The start method spawns a thread/process which will read from the class data source handle and writes the result to a message queue.

Invoking the Start method should not be a blocking operation.

The start method should return true if the Asynchronous Reader is started successfully.

The start method should return false if the Asynchronous Reader failed to start.

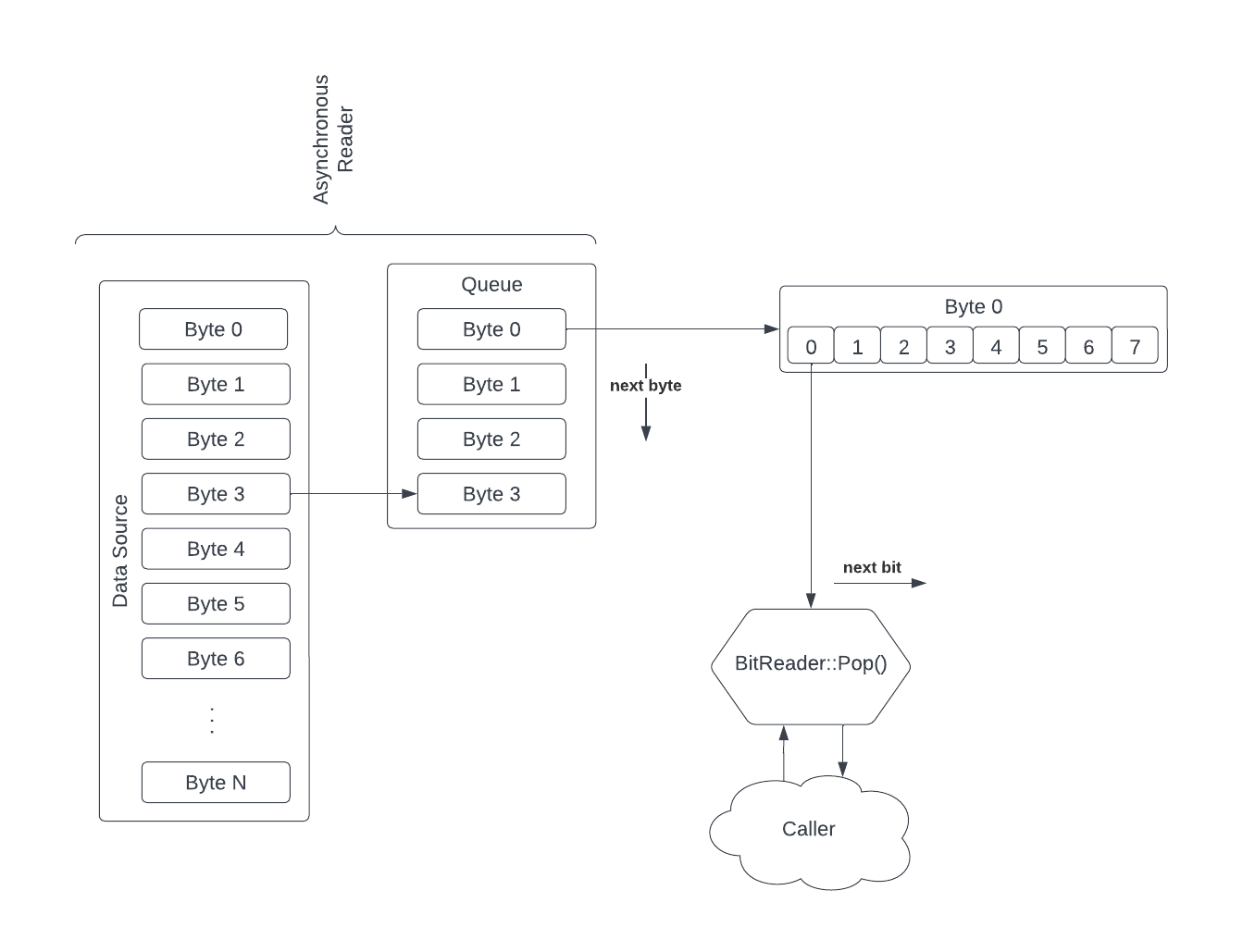
* + 1. IsRunning Method

The BitReader::IsRunning method will return true if the Asynchronous Reader is running.

The BitReader::IsRunning method will return false if the Asynchronous Reader has terminated.

* + 1. Pop Method

The Pop method returns a single bit from the current byte (value of the internal buffer) and when the byte boundary is reached, a new byte is loaded into the buffer from the internal queue structure.

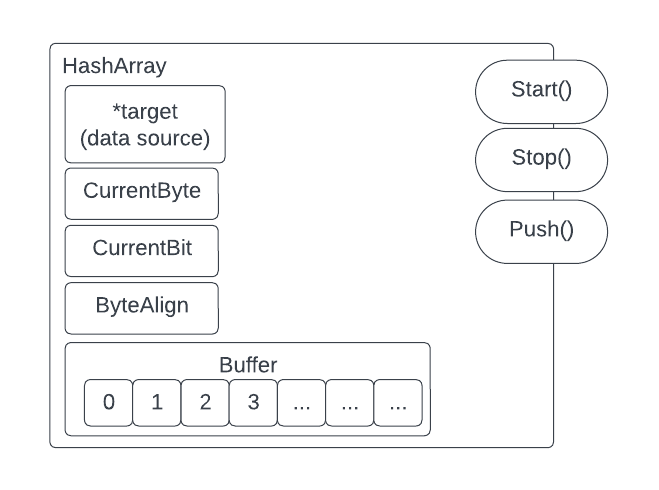


* + 1. Stop Method

The BitReader::Stop method will terminate the BitReader‘s asynchronous reader thread/process.

* 1. BitWriter Class
     1. Overview

The BitWriter class writes bitstream data, single bytes or byte arrays to the output data target.



* + 1. Properties

|  |  |  |
| --- | --- | --- |
| Property | Type | Description |
| target | File handle or other data connector | This is a writable data target to which bytes of information will be written. |
| Buffer | Queue | A thread-safe queue of bytes |
| CurrentByte | Byte | When packing bits rather than using byte-aligned writes, this byte is used to marshal bits from ::Push into a single byte which can then be pushed to the Queue |
| CurrentBit | Uint8 | When packing bits, this private property keeps track of the current bit position for the push method |
| ByteAlign | Bool | This property indicates whether data received will be packed or written as byte-aligned values. |

* + 1. Class Constructor

The class constructor will initialize the state of the class, ensuring the data target is writable and preparing the class to write data to the same.

We pass the target by reference in an initialized form.

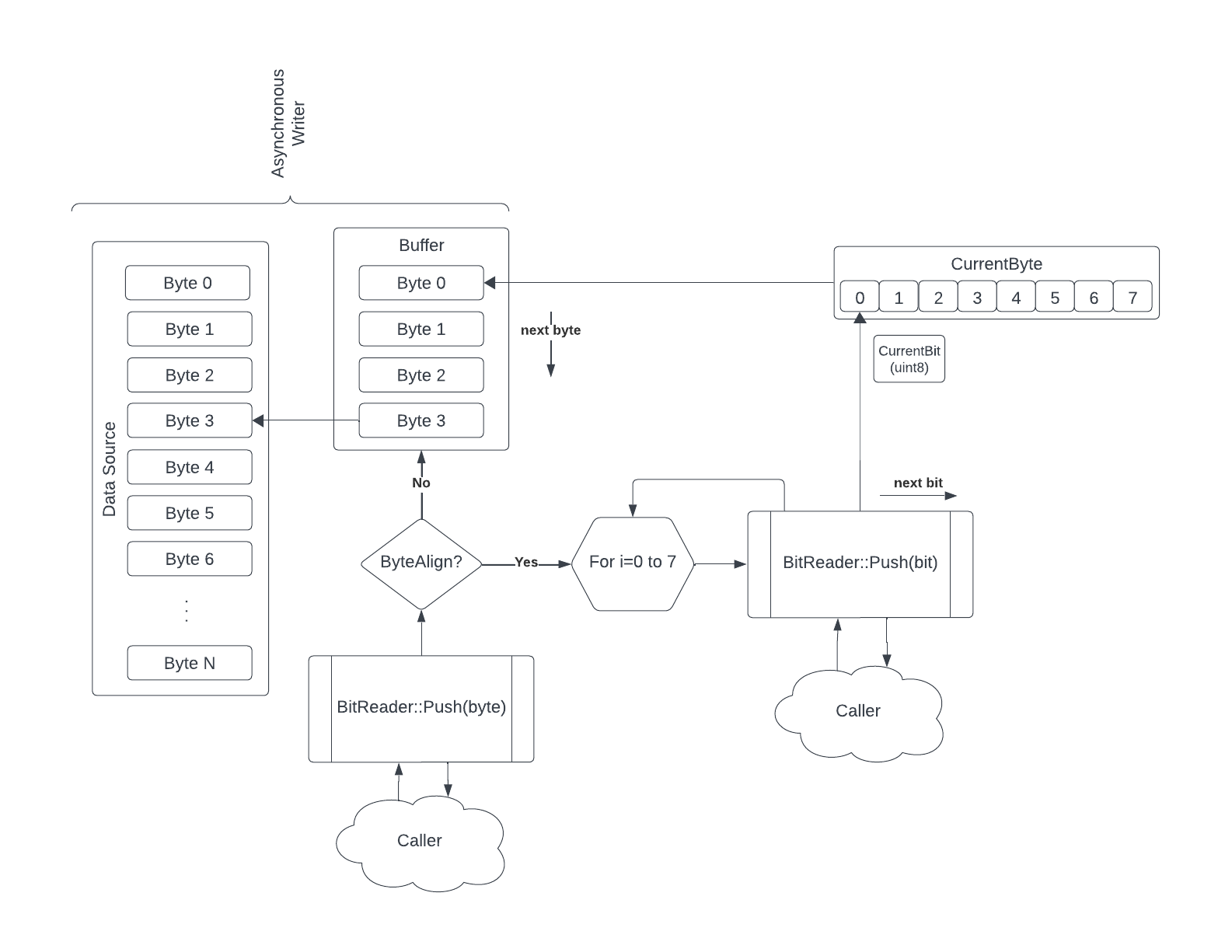
The target allows abstraction to support files, network devices, etc.

|  |  |  |
| --- | --- | --- |
| Input | Type | Description |
| \*target | DataSource Reference | A pointer to the writable data target. |
| bufferSz | uint16 | Define the size of the buffer (queue) for writing information to the target |
| byteAlign | Boolean | This flag determines whether information will be written to the target byte-aligned or packed bitwise. |

* + 1. Class Destructor

The class destructor will stop the asynchronous writer then free all memory allocated to the class.

* + 1. Push Methods



**Bit Inputs**

The Push method will receive a Boolean input representing a single bit value.

If the byteAlign property is true, this method will throw an exception or return an error, since bit values are not byte-aligned.

If the byteAlign property is false, this method will push the bit onto the CurrentByte property and when that byte is filled, the same will be flushed to the buffer (queue) and the CurrentByte will be set to 0x00.

**Byte Array Input**

The Push method will receive an array of one or more bytes of information.

If the ByteAlign property is true, each byte will be pushed to the buffer (queue).

If the ByteAlign property is false, each byte will be read one by one and then bit by bit into CurrentByte[[19]](#footnote-19). Each time CurrentByte is filled it will be flushed to the buffer (queue), reset to 0x00 and the bitwise processing will continue.

* + 1. Start Method

The BitWriter::Start method will launch an asynchronous thread/process to read from the buffer (queue) and write the bytes from that queue to the data target.

* + 1. Stop Method

The BitWriter::Stop method will terminate the BitWriter’s asynchronous writer thread/process.

* + 1. IsRunning Method

The BitWriter::isRunning() method will return a Boolean value indicating whether the BitWriter Asynchronous writer is in a healthy (e.g., running) state.

1. CRSCE Command line Specifications
   1. Compress Tool
      1. Usage

|  |
| --- |
| CRSCE Compression Tool  compress -h | --help show this message  compress --in <file> --out <file> [options]  Options:  --timings print timings for various operations  --color use ANSI color output  --debug print detailed messages |

* 1. Decompress Tool
     1. Usage

|  |
| --- |
| CRSCE Decompression Tool  decompress -h | --help show this message  decompress --in <file> --out <file> [options]  Options:  --workers number of workers to process decompress (default:64)  --timings print timings for various operations  --color use ANSI color output  --debug print detailed messages |

1. “Compression Rate” is defined herein as the ratio of output signal divided by input signal using the formula , where O represents the output signal length and N represents the input signal length. [↑](#footnote-ref-1)
2. Research shows that with the CRSCE-1 variant, a 64KB or 1MB input, the algorithm will yield 62% and 90% compression, respectively. [↑](#footnote-ref-2)
3. Welch, Terry. "A Technique for High-Performance Data Compression," in Computer, vol. 17, no. 6, pp. 8-19, June 1984, doi: 10.1109/MC.1984.1659158. [↑](#footnote-ref-3)
4. A. H. Robinson and C. Cherry, "Results of a prototype television bandwidth compression scheme," in Proceedings of the IEEE, vol. 55, no. 3, pp. 356-364, March 1967, doi: 10.1109/PROC.1967.5493. C. Cherry, M. H. Kubba, D. E. Pearson and M. P. Barton, "An experimental study of the possible bandwidth compression of visual image signals", Proc. IEEE, vol. 51, pp. 1507-1517, November 1963. [↑](#footnote-ref-4)
5. D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," in Proceedings of the IRE, vol. 40, no. 9, pp. 1098-1101, Sept. 1952, doi: 10.1109/JRPROC.1952.273898. [↑](#footnote-ref-5)
6. Shannon Claude E. “A Mathematical Theory of Communication.” Bell System Technical Journal Volume 27, Issue 3. Pages 379-423. 1948. [↑](#footnote-ref-6)
7. This “raw compression” ignores any overhead for representing a multi-block output where the output includes blockIds and a message header, as discussed later in this specification. [↑](#footnote-ref-7)
8. Or programmatically speaking, unsigned integers. [↑](#footnote-ref-8)
9. See Cross Sum Width, below. [↑](#footnote-ref-9)
10. See Cross Sum Width, below. [↑](#footnote-ref-10)
11. The element size 32-bytes assumes SHA-256 or SHA3-256. [↑](#footnote-ref-11)
12. See “Message Block Sizing” [↑](#footnote-ref-12)
13. 256 bits 32 bytes assumes SHA-256 or SHA3-256. [↑](#footnote-ref-13)
14. This data table is a summary of a much larger data set. Nonetheless this table illustrates both the minimum signal size and breaking point for the algorithm as well as the growth of the compression rates with larger block sizes. [↑](#footnote-ref-14)
15. See <https://github.com/sam-caldwell/go/blob/main/cmd/crsce/calculate-minimum/main.cpp>. This solver supports both CRSCE-1 and CRSCE-2 [↑](#footnote-ref-15)
16. Here b is either the packed logarithmic function or its byte-aligned value. [↑](#footnote-ref-16)
17. This creates a linear stream of memory allocations and accesses and helps encourage the CPU to correctly predict and cache the next memory block to be read. Were the entire CSM loaded into memory and a cartesian (x, y) coordinate system used, the memory overhead would be higher and allocated bytes may not be contiguous. Lookup operations would consume time and the solution would simply not be as performant. [↑](#footnote-ref-17)
18. VirtualCSM used in compression does not load the entire message block into memory for efficiency since compression of a signal is a linear process of iterating over the bit stream and calculating the cross sums and hashes. However, decompression requires a full in-memory message block since multiple iterations over the problem space are necessary for the SiftSolve process. [↑](#footnote-ref-18)
19. It is important that all bitwise push method implementations write to CurrentByte since a single bit could be written in one invocation and the next invocation could write a byte or byte array. If all bitwise operations do not use this same CurrentByte as a buffer for packing bits into the output it is likely the bit order will not be maintained. [↑](#footnote-ref-19)