CRAM codec specification (version 3.1)

samtools-devel@lists.sourceforge.net

24 Mar 2025

The master version of this document can be found at `https://github.com/samtools/hts-specs`.
This printing is version 6259dcd from that repository, last modified on the date shown above.

license: Apache 2.0

1 Introduction

This document covers the compression and decompression algorithms (codecs) specific to the CRAM format. All bar the first of these were introduced in CRAM v3.1.

It does not cover the CRAM format itself. For that see `http://samtools.github.io/hts-specs/`.

1.1 Pseudocode introduction

Various parts of this specification are written in a simplistic pseudocode. This intentionally does not make explicit use of data types and has minimal error checking. The number of operators is kept to a minimum, but some are necessary and may be language specific. Due to the lack of explicit data types, we also have different division operators to symbolise floating point and integer divisions.

The pseudocode doesn't prescribe any particular programming paradigm - functional, procedural or object oriented - but it does have a few implicit assumptions. Variables are considered to be passed between functions via unspecified means. For example the Range Coder sets *range* and *code* during creation and these are used during the decoding steps, but are not explicitly passed in as variables. We make the implicit assumption that they are simply local variables of the particular usage of this range coder. Other than ephemeral loop counters, we do not reuse variable names so the intention should be clear.

The exception to the above is occasionally we need to have multiple instances of a particular data type, such as Order-1 decoding will have many models. Here we use an object oriented way of describing the problem with *instance*.FUNCTION notation.

Note some functions may return multiple items, such as `return (`*value, length*`)`, but the calling code may assign a single variable to this result. In this case the first value *value* will be used and *length* will be discarded.

1.2 Mathematical operators

Operator	Description
$a + b$	Addition
$a - b$	Subtraction
$a \times b$	Multiplication
$a \ / \ b$	Floating point division a/b
a div b	Integer division a/b, equivalent to $\lfloor a/b \rfloor$
a mod b	Integer modulo (remainder) $a - b \times (a \text{ div } b)$
$a = b$	Compares a and b variables, yielding true if they match, false otherwise
$a \leftarrow b$	Assigns value of b to variable a
$a << b$	Bit-wise left shift a by b bits, shifting in zeros
$a >> b$	Bit-wise right shift a by b bits, shifting in zeros
a AND b	Bit-wise AND operator, joining values a, b
a OR b	Bit-wise OR operator, joining values a, b
a **or** b	Logical OR operator, joining expressions a, b
a **and** b	Logical AND operator, joining expressions a, b
$a \mathbin{+\!\!+} b$	String concatenation of a and b: ab
V_i	Element i of vector V
	The entire vector V may be passed into a function
$W_{i,j}$	Element i, j of two-dimensional vector W.
	The entire vector W or a one dimensional slice W_i (of size j) may be passed into a function.

Note that string concatenation with the $+\!\!+$ operator assumes the left and right values are converted to string form. For example "level" $+\!\!+ 42$ will convert the integer 42 to "42" and produce the string "level42".

1.3 Implicit functions

Operator	Description
MIN(a, b)	Smaller of a and b
MAX(a, b)	Larger of a and b
READUINT8	Read an 8-bit unsigned integer (1 byte) from unspecified input source
READUINT32	Read a 32-bit unsigned little-endian integer from unspecified input source
READUINT8(src)	Read an 8-bit unsigned integer (1 byte) from input src
READUINT32(src)	Read a 32-bit unsigned little-endian integer from input src
READDATA(len)	Read len bytes (8-bit unsigned) from an unspecified input source
READDATA(len, src)	Read len bytes (8-bit unsigned) from input src
READCHAR(src)	Read a single character from input src
READSTRING(src)	Read a nul-terminated string from input src
EOF	Returns true if the input source is exhausted.
CHAR(a)	Converts integer a to a single character of appropriate ASCII value
LENGTH(a)	Returns length of string a excluding any nul-termination bytes
SWAP(a, b)	Swaps the contents of a and b variables

Many of the input functions here and below are defined to read either from an unspecified input source (such as the input file descriptor, or a global buffer that has not been explicitly stated in the pseudocode), but also have forms that may decode from specified inputs / buffers. They both consume their input sources in the same manner, using an implicit offset of how many bytes so far have been read.

1.4 Other basic functions

7-bit integer encoding stores values 7-bits at a time with the top bit set if further bytes are required.

(Read a variable sized unsigned integer 7-bits at a time. Returns the value.)

1: **function** READUINT7(*source*) ▷ If *source* is unspecified then it is the default input stream
2: $value \leftarrow 0$
3: $length \leftarrow 0$
4: **repeat**

```
        c ← ReadUint8
        value ← (value << 7) + (c AND 127)
        length ← length + 1
    until c < 128
    return value
end function
```

ITF8 integer encoding stores the additional number of bytes needed in the count of the top bits set in the initial byte (ending with a zero bit), followed by any subsequent whole bytes. See the main CRAM specification for more details.

```
(Read a variable sized unsigned integer with ITF8 encoding. Returns the value.)
function ReadITF8(source)          ▷ If source is unspecified then it is the default input stream
    v ← ReadUint8
    if i >= 0xf0 then                                          ▷ 1111xxxx => +4 bytes
        v ← (v  AND 0x0f) << 28
        v ← v + ( ReadUint8 <<20)
        v ← v + ( ReadUint8 <<12)
        v ← v + ( ReadUint8 <<4)
        v ← v + ( ReadUint8 >>4)
    else if i >= 0xe0 then                                     ▷ 1110xxxx => +3 bytes
        v ← (v  AND 0x0f) << 24
        v ← v + ( ReadUint8 <<16)
        v ← v + ( ReadUint8 <<8)
        v ← v+ ReadUint8
    else if i >= 0xc0 then                                     ▷ 110xxxxx => +2 bytes
        v ← (v  AND 0x1f) << 16
        v ← v + ( ReadUint8 <<8)
        v ← v+ ReadUint8
    else if i >= 0x80 then                                     ▷ 10xxxxxx => +1 bytes
        v ← (v  AND 0x3f) << 8
        v ← v+ ReadUint8
    end if
    return v
end function
```

2 rANS 4x8 - Asymmetric Numeral Systems

This is the rANS format first defined in CRAM v3.0.

rANS is the range-coder variant of the Asymmetric Numerical Systems[1].

The structure of the external rANS codec consists of several components: meta-data consisting of compression-order, and compressed and uncompressed sizes; normalised frequencies of the alphabet systems to be encoded, either in Order-0 or Order-1 context; and the rANS encoded byte stream itself.

Here "Order" refers to the number of bytes of context used in computing the frequencies. It will be 0 or 1. Ignoring punctuation and space, an Order-0 analysis of English text may observe that 'e' is the most common letter (12-13%), and that 'u' occurs only around 2.5% of the time. If instead we consider the frequency of a letter in the context of one previous letter (Order-1) then these statistics change considerably; we know that if the previous letter was 'q' then 'e' becomes a rare letter while 'u' is the most likely.

These observed frequencies are inversely related to the amount of storage required to encode a symbol (e.g. an alphabet letter)[2].

[1] J. Duda, *Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding*, `http://arxiv.org/abs/1311.2540`

[2] C.E. Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal, vol. 27, pp. 379-423, 623-656, July, October, 1948

rANS 4x8 compressed data structure

A compressed data block consists of the following logical parts:

Value data type	Name	Description
byte	order	the order of the codec, either 0 or 1
uint32	compressed size	the size in bytes of frequency table and compressed blob
uint32	data size	raw or uncompressed data size in bytes
byte[]	frequency table	byte frequencies of input data written using RLE
byte[]	compressed blob	compressed data

2.1 Frequency table

The alphabet used here has a maximum of 256 possible symbols (all byte values), but alphabets where fewer symbols are permitted too.

The symbol frequency table indicates which symbols are present and what their relative frequencies are. The total sum of symbol frequencies are normalised to add up to 4095[3]. Given rounding differences when renormalising to a fixed sum, it is up to the encoder to decide how to distribute any remainder or remove excess frequencies. The normalised frequency tables below are examples and not prescriptive of a specific normalisation strategy.

Formally, this is an ordered alphabet $\mathbb{A}$ containing symbols s where s_i with the i-th symbol in $\mathbb{A}$, occurring with the frequency $freq_i$.

Order-0 encoding

The normalised symbol frequencies are then written out as {symbol, frequency} pairs in ascending order of symbol (0 to 255 inclusive). If a symbol has a frequency of 0 then it is omitted.

To avoid storing long consecutive runs of symbols if all are present (eg a-z in a long piece of English text) we use run-length-encoding on the alphabet symbols. If two consecutive symbols have non-zero frequencies then a counter of how many other non-zero frequency consecutive symbols is output directly after the second consecutive symbol, with that many symbols being subsequently omitted.

For example for non-zero frequency symbols ‘a’, ‘b’, ‘c’, ‘d’ and ‘e’ we would write out symbol ‘a’, ‘b’ and the value 3 (to indicate ‘c’, ‘d’ and ‘e’ are also present).

The frequency is output after every symbol (whether explicit or implicit) using ITF8 encoding. This means that frequencies 0-127 are encoded in 1 byte while frequencies 128-4095 are encoded in 2 bytes.

Finally the symbol 0 is written out to indicate the end of the symbol-frequency table.

As an example, take the string `abracadabra`.

Symbol frequency:

Symbol	Frequency
a	5
b	2
c	1
d	1
r	2

Normalised to sum to 4095:

Symbol	Frequency
a	1863
b	744
c	372
d	372
r	744

Encoded as:

[3]While the maths work fine up to 4096, for historical reasons this has always been documented as having a limit of 4095. Implementations may wish to validate decoding on <= 4096, but we recommend they use a limit of 4095 in their encoding output.

```
0x61      0x87 0x47      # 'a'           <1863>
0x62 0x02 0x82 0xe8      # 'b' <+2: c,d>  <744>
          0x81 0x74      # 'c' (implicit) <372>
          0x81 0x74      # 'd' (implicit) <372>
0x72      0x82 0xe8      # 'r'            <744>
0x00                     # <0>
```

Order-1 encoding

To encode Order-1 statistics typically requires a larger table as for an N sized alphabet we need to potentially store an NxN matrix. We store these as a series of Order-0 tables.

We start with the outer context byte, emitting the symbol if it is non-zero frequency. We perform the same run-length-encoding as we use for the Order-0 table and end the contexts with a nul byte. After each context byte we emit the Order-0 table relating to that context.

One last caveat is that we have no context for the first byte in the data stream (in fact for 4 equally spaced starting points, see "interleaving" below). We use the ASCII value ('\0') as the starting context for each interleaved rANS state and so need to consider this in our frequency table.

Consider `abracadabraabracadabraabracadabraabracadabrad` as example input. Note for the additional trailing "d" giving us 45 characters instead of 44. This can be broken into 4 approximate equal portions `abracadabra abracadabra abracadabra abracadabrad`. We operate one independent rANS stream per portion, providing us the opportunity to exploit CPU data parallelism.

Naively observed Order-1 frequencies:

Context	Symbol	Frequency
\0	a	4
a	a	3
	b	8
	c	4
	d	5
b	r	8
c	a	4
d	a	4
r	a	8

Normalised (per Order-0 statistics):

Context	Symbol	Frequency
\0	a	4095
a	a	614
	b	1638
	c	819
	d	1024
b	r	4095
c	a	4095
d	a	4095
r	a	4095

Note that the above table has redundant entries. While our complete string had three cases of two consecutive "a" characters ("...cadabr**aa**braca..."), these spanned the junction of our split streams and each rANS state is operating independently, starting with the same last character of nul (0). Hence during decode we will not need to access the table for the frequency of "a" in the context of a previous "a". A similar issue occurs for the very last byte used for each rANS state, which will not be used as a context. In extreme cases this may even be the only time that symbols occurs anywhere. While these scenarios represent unnecessary data to store, and these frequency entries can be safely omitted, their presence does not invalidate the data format and it may be simpler to use a more naive algorithm when producing the frequency tables.

The above tables are encoded as:

```
0x00                # '\0' context
0x61      0x8f 0xff # a  <4095>
0x00                # end of Order-0 table

0x61                # 'a' context
0x61      0x82 0x66 # a             <614>
0x62 0x02 0x86 0x66 # b <+2: c,d>  <1638>
          0x83 0x33 # c (implicit)  <819>
          0x84 0x00 # d (implicit) <1024>
0x00                # end of Order-0 table

0x62 0x02           # 'b' context, <+2: c, d>
0x72      0x8f 0xff # r <4095>
```

```
0x00                # end of Order-0 table

                    # 'c' context (implicit)
0x61      0x8f 0xff # a <4095>
0x00                # end of Order-0 table

                    # 'd' context (implicit)
0x61      0x8f 0xff # a <4095>
0x00                # end of Order-0 table

0x72                # 'r' context
0x61      0x8f 0xff # a <4095>
0x00                # end of Order-0 table

0x00                # end of contexts
```

2.2 rANS entropy encoding

The encoder takes a symbol s and a current state x (initially L below) to produce a new state x' with function C.

$$x' = C(s, x)$$

The decoding function D is the inverse of C such that $C(D(x)) = x$.

$$D(x') = (s, x)$$

The entire encoded message can be viewed as a series of nested C operations, with decoding yielding the symbols in reverse order, much like popping items off a stack. This is where the asymmetric part of ANS comes from.

As we encode into x the value will grow, so for efficiency we ensure that it always fits within known bounds. This is governed by

$$L \leq x < bL - 1$$

where b is the base and L is the lower-bound.

We ensure this property is true before every use of C and after every use of D. Finally to end the stream we flush any remaining data out by storing the end state of x.

Implementation specifics

We use an unsigned 32-bit integer to hold x. In encoding it is initialised to L. For decoding it is read little-endian from the input stream.

Recall $freq_i$ is the frequency of the i-th symbol s_i in alphabet $\mathbb{A}$. We define $cfreq_i$ to be cumulative frequency of all symbols up to but not including s_i:

$$cfreq_i = \begin{cases} 0 & \text{if } i < 1 \\ cfreq_{i-1} + freq_{i-1} & \text{if } i \geq 1 \end{cases}$$

We have a reverse lookup table $cfreq_to_sym_c$ from 0 to 4095 (0xfff) that maps a cumulative frequency c to a symbol s.

$$cfreq_to_sym_c = s_i \quad where \quad c: \ cfreq_i \leq c < cfreq_i + freq_i$$

The $x' = C(s, x)$ function used for the i-th symbol s is:

$$x' = (x/freq_i) \times \texttt{0x1000} + cfreq_i + (x \bmod freq_i)$$

The $D(x') = (s, x)$ function used to produce the i-th symbol s and a new state x is:

$c = x'$ AND `0xfff`
$s_i = cfreq_to_sym_c$
$x = freq_i(x'/\texttt{0x1000}) + c - cfreq_i$

Most of these operations can be implemented as bit-shifts and bit-AND, with the encoder modulus being implemented as a multiplication by the reciprocal, computed once only per alphabet symbol.

We use $L = \texttt{0x800000}$ and $b = 256$, permitting us to flush out one byte at a time (encoded and decoded in reverse order).

Before every encode $C(s, x)$ we renormalise x, shifting out the bottom 8 bits of x until $x < \texttt{0x80000} \times freq_i$. After finishing all encoding we flush 4 more bytes (lowest 8-bits first) from x.

After every decoded $D(x')$ we renormalise x', shifting in the bottom 8 bits until $x \geq \texttt{0x800000}$.

Interleaving

For efficiency, we interleave 4 separate rANS codecs at the same time[4]. For the Order-0 codecs these simply encode or decode the 4 neighbouring bytes in cyclic fashion using interleaved codec 0, 1, 2, and 3, sharing the same output buffer (so the output bytes get interleaved).

For the Order-1 codec we cannot do this as we need to know the previous byte value as the context for the next byte. We therefore split the input data into 4 approximately equal sized fragments[5] starting at 0, $\lfloor len/4 \rfloor$, $\lfloor len/4 \rfloor \times 2$ and $\lfloor len/4 \rfloor \times 3$. Each Order-1 codec operates in a cyclic fashion as with Order-0, all starting with 0 as their state and sharing the same compressed output buffer. Any remainder, when the input buffer is not divisible by 4, is processed at the end by the 4th rANS state.

We do not permit Order-1 encoding of data streams smaller than 4 bytes.

2.3 rANS decode pseudocode

A naïve implementation of a rANS decoder follows. This pseudocode is for clarity only and is not expected to be performant and we would normally rewrite this to use lookup tables for maximum efficiency. The function ReadUint8 fetches the next single unsigned byte from an unspecified input source. Similarly for ReadITF8 (variable size integer) and ReadUint32 (32-bit unsigned integer in little endian format).

```
 1: procedure RansDecode(input, output)
 2:     order  ← ReadUint8                          ▷ Implicit read from input
 3:     n_in   ← ReadUint32
 4:     n_out  ← ReadUint32
 5:
 6:     if order = 0 then
 7:         RansDecode0(output, n_out)
 8:     else
 9:         RansDecode1(output, n_out)
10:     end if
11: end procedure
```

rANS order-0

The Order-0 code is the simplest variant. Here we also define some of the functions for manipulating the rANS state, which are shared between Order-0 and Order-1 decoders.

```
    (Reads a table of Order-0 symbol frequencies F_i)
    (and sets the cumulative frequency table C_{i+1} = C_i + F_i)
 1: procedure ReadFrequencies0(F, C)
 2:     s ← ReadUint8                               ▷ Next alphabet symbol
 3:     last_sym ← s
 4:     rle ← 0
 5:     repeat
 6:         F_s ← ReadITF8
 7:         if rle > 0 then
 8:             rle ← rle − 1
```

[4]F. Giesen, *Interleaved entropy coders*, `http://arxiv.org/abs/1402.3392`

[5]This was why the '\0' → 'a' context in the example above had a frequency of 4 instead of 1.

```
            s    ← s + 1
        else
            s ← ReadUint8
            if s = last_sym + 1 then
                rle ← ReadUint8
            end if
        end if
        last_sym ← s
    until s = 0
    (Compute cumulative frequencies C_i from F_i)
    C_0 ← 0
    for s ← 0 to 255 do
        C_{s+1} ← C_s + F_s
    end for
end procedure

(Bottom 12 bits of our rANS state R are our frequency)
function RansGetCumulativeFreq(R)
    return R AND 0xfff
end function

(Convert frequency to a symbol. Find s such that C_s ≤ f < C_{s+1})
(We would normally implement this via a lookup table)
function RansGetSymbolFromFreq(C, f)
    s ← 0
    while f >= C_{s+1} do
        s ← s + 1
    end while
    return s
end function

(Compute the next rANS state R given frequency f and cumulative freq c)
function RansAdvanceStep(R, c, f)
    return f × (R >> 12) + (R AND 0xfff) − c
end function

(If too small, feed in more bytes to the rANS state R)
function RansRenorm(R)
    while R < (1 << 23) do
        R ← (R << 8)+ ReadUint8
    end while
    return R
end function

procedure RansDecode0(output, nbytes)
    ReadFrequencies0(F, C)
    for j ← 0 to 3 do                          ▷ Initialise the 4 interleaved streams
        R_j ← ReadUint32                       ▷ Unsigned 32-bit little endian
    end for
    for i ← 0 to nbytes − 1 do
        j ← i mod 4
        f ← RansGetCumulativeFreq(R_j)
        s ← RansGetSymbolFromFreq(C, f)
        output_i ← s
        R_j ← RansAdvanceStep(R_j, C_s, F_s)
        R_j ← RansRenorm(R_j)
    end for
end procedure
```

rANS order-1

As described above, the decode logic is very similar to rANS Order-0 except we have a two dimensional array of frequencies to read and the decode uses the last character as the context for decoding the next one. In the pseudocode we illustrate this by using two dimensional vectors $C_{i,j}$ and $F_{i,j}$. For simplicity, we reuse the Order-0 code by referring to C_i and F_i of the 2D vectors to get a single dimensional vector that operates in the same manner as the Order-0 code. This is not necessarily the most efficient implementation.

Note the code for dealing with the remaining bytes when an output buffer is not an exact multiple of 4 is less elegant in the Order-1 code. This is correct, but it is unfortunately a design oversight.

```
    (Reads a table of Order-1 symbol frequencies F_{i,j})
    (and sets the cumulative frequency table C_{i,j+1} = C_{i,j} + F_{i,j})
 1: procedure ReadFrequencies1(F, C)
 2:     sym ← ReadUint8                                  ▷ Next alphabet symbol
 3:     last_sym ← sym
 4:     rle ← 0
 5:     repeat
 6:         ReadFrequencies0(F_i, C_i)
 7:         if rle > 0 then
 8:             rle ← rle − 1
 9:             sym ← sym + 1
10:         else
11:             sym ← ReadUint8
12:             if sym = last_sym + 1 then
13:                 rle ← ReadUint8
14:             end if
15:         end if
16:         last_sym ← sym
17:     until sym = 0
18: end procedure

19: procedure RansDecode1(output, nbytes)
20:     ReadFrequencies1(F, C)
21:     for j ← 0 to 3 do                                ▷ Initialise 4 interleaved streams
22:         R_j ← ReadUint32                             ▷ Unsigned 32-bit little endian
23:         L_j ← 0                                      ▷ Last symbol
24:     end for
25:     i ← 0
26:     while i < ⌊nbytes/4⌋ do
27:         for j ← 0 to 3 do
28:             f ← RansGetCumulativeFreq(R_j)
29:             s ← RansGetSymbolFromFreq(C_{L_j}, f)
30:             output_{i+j×⌊nbytes/4⌋} ← s
31:             R_j ← RansAdvanceStep(R_j, C_{L_j,s}, F_{L_j,s})
32:             R_j ← RansRenorm(R_j)
33:             L_j ← s
34:         end for
35:         i ← i + 1
36:     end while
        (Now deal with the remainder if buffer size is not a multiple of 4,)
        (using rANS state 3 exclusively.)
37:     for i ← i × 4 to len − 1 do
38:         f ← RansGetCumulativeFreq(R_3)
39:         s ← RansGetSymbolFromFreq(C_{L_3}, f)
40:         output_i ← s
41:         R_3 ← RansAdvanceStep(R_3, C_{L_3,s}, F_{L_3,s})
42:         R_3 ← RansRenorm(R_3)
43:         L_3 ← s
```

44: **end for**
45: **end procedure**

3 rANS Nx16

CRAM version 3.1 defines an additional rANS entropy encoder, using 16-bit renormalisation instead of the 8-bit used in CRAM 3.0 and with inbuilt bit-packing and run-length encoding. The lower-bound and initial encoder state L is also changed to 0x8000. The Order-1 rANS Nx16 encoder has also been modified to permit a maximum frequency of 1024 instead of 4096. This offers better cache performance for the large Order-1 tables and usually has minimal impact on compression ratio.

Additionally it permits adjustment of the number of interleaved rANS states from the fixed 4 used in rANS 4x8 to either 4 or 32 states. The benefit of the 32-way interleaving is in enabling efficient use of SIMD instructions for faster encoding and decoding speeds. However it has a small cost in size and initialisation times so it is not recommended on smaller blocks of data.

Frequencies are now stored using uint7 format instead of ITF8. The tables are also stored differently, separating the list of symbols present in the alphabet (those with frequency greater than zero) from the frequencies themselves.

Finally transformations may be applied to the data prior to compression (or after decompression). These consist of stripe, for structured data where every Nth byte is sent to one of N separate compression streams, Run Length Encoding replacing repeated strings of symbols with a symbol and count, and bit-packing where reduced alphabets can combine multiple symbols into a byte prior to entropy encoding.

The initial "Order" byte is expanded with additional bits to list the transformations to be applied. The specifics of each sub-format are listed below, in the order they are applied.

- **Stripe**: rANS Nx16 with multi-way interleaving (see Section 3.6).
- **NoSize**: Do not store the size of the uncompressed data stream. This information is not required when the data stream is one of the four sub-streams in the Stripe format.
- **Cat**: If present, the order bit flag is ignored.

  The uncompressed data stream is the same as the compressed stream. This is useful for very short data where the overheads of compressing are too high.
- **N32**: Flag indicating whether to interleave 4 or 32 rANS states.
- **Order**: Bit field defining order-0 (unset) or order-1 (set) entropy encoding, as described above by the RansDecodeNx16_0 and RansDecodeNx16_1 functions.
- **RLE**: Bit field defining whether Run Length Encoding has been applied to the data. If set, the reverse transorm will be applied using DecodeRLE after Order-0 or Order-1 uncompression (see Section 3.4).
- **Pack**: Bit field indicating the data was packed prior to compression (see Section 3.5). If set, unpack the bits after any RLE decoding has been applied (if required) using the DecodePack function.

3.1 Frequency tables

Frequency tables in rANS Nx16 separate the list of symbols from their frequencies. The symbol list must be stored in ascending ASCII order, with their frequency values in the same ordering as their corresponding symbols. For the Order-1 frequency table this list of symbols is those used in any context, thus we only have one alphabet recorded for all contexts. This means in some contexts some (potentially many) symbols will have zero frequency. To reduce the Order-1 table size an additional zero run-length encoding step is used. Finally the Order-1 frequencies may optionally be compressed using the Order-0 rANS Nx16 codec.

Frequencies must always add up to a power of 2, but do not necessarily have to match the final power of two used in the Order-0 (4096) and Order-1 (1024, 4096) entropy decoder algorithm. A normalisation step is applied after reading the frequencies to scale them appropriately. This is required as the Order-1 frequencies may be scaled differently for each context.

(Reads a set of symbols A used in our alphabet)

```
1: function ReadAlphabet
2:     s ← ReadUint8
3:     last_sym ← s
4:     rle ← 0
5:     repeat
6:         A ← (A, s)                                  ▷ Append s to the symbol set A
7:         if rle > 0 then
8:             rle ← rle − 1
9:             s   ← s + 1
10:        else
11:            s ← ReadUint8
12:            if s = last_sym + 1 then
13:                rle ← ReadUint8
14:            end if
15:        end if
16:        last_sym ← s
17:    until s = 0
18:    return A
19: end function
```

(Reads a table of Order-0 symbol frequencies F_i)
(and sets the cumulative frequency table $C_{i+1} = C_i + F_i$)

```
1: procedure ReadFrequenciesNx16_0(F, C)
2:     F ← (0, ...)                                    ▷ (Set to zero for all i ∈ {0, 1, ..., 255})
3:     A ← ReadAlphabet
4:     foreach i in A do
5:         F_i ← ReadUint7
6:     end foreach
7:
8:     NormaliseFrequenciesNx16_0(F, 12)
9:
10:    C_0 ← 0
11:    for s ← 0 to 255 do                             ▷ All 256 possible byte values
12:        C_{s+1} ← C_s + F_s
13:    end for
14: end procedure
```

(Normalises a table of frequencies F_i to sum to a specified power of 2.)

```
1: procedure NormaliseFrequenciesNx16_0(F, bits)
2:     tot ← 0
3:     for i ← 0 to 255 do
4:         tot ← tot + F_i
5:     end for
6:     if tot = 0 or tot = (1 << bits) then
7:         return
8:     end if
9:
10:    shift ← 0
11:    while tot < (1 << bits) do
12:        tot ← tot * 2
13:        shift ← shift + 1
14:    end while
15:
16:    for i ← 0 to 255 do
17:        F_i ← F_i << shift
18:    end for
19: end procedure
```

The Order-1 frequencies also store the complete alphabet of observed symbols (ignoring context) followed by a table of frequencies for each symbol in the alphabet. Given many frequencies will be zero where a symbol is present in one context but not in others, all zero frequencies are followed by a run length to omit adjacent zeros.

The order-1 frequency table itself may still be quite large, so is optionally compressed using the order-0 rANS Nx16 codec with a fixed 4-way interleaving. This is specified in the bottom bit of the first byte. If this is 1, it is followed by 7-bit encoded uncompressed and compressed lengths and then the compressed frequency data. The pseudocode here differs slightly to elsewhere as it indicates the input sources, which are either the uncompressed frequency buffer or the default (unspecified) source. The top 4 bits of the first byte indicate the number of bits used for the frequency tables. Permitted values are 10 and 12.

```
    (Reads a table of Order-1 symbol frequencies F_{i,j})
    (and sets the cumulative frequency table C_{i,j+1} = C_{i,j} + F_{i,j})
 1: procedure ReadFrequenciesNx16_1(F, C, bits)
 2:     comp ← ReadUint8
 3:     bits ← comp >> 4
 4:     if (comp and 1) ≠ 0 then
 5:         u_size ← ReadUint7                          ▷ Uncompressed size
 6:         c_size ← ReadUint7                          ▷ Compressed size
 7:         c_data ← ReadData(c_size)
 8:         source ← RansDecodeNx16_0(c_data, 4)        ▷ Create u_size bytes of source from c_data
 9:     else
10:         (define source to be the default input stream)
11:     end if

12:     F ← ((0, ...), ...)                             ▷ (Set to zero for all i and j ∈ {0, 1, ..., 255})
13:     A ← ReadAlphabet(from source)
14:     foreach i in A do
15:         run ← 0
16:         foreach j in A do
17:             if run > 0 then
18:                 run ← run − 1                       ▷ F_{i,j} is implicitly zero already
19:             else
20:                 F_{i,j} ← ReadUint7(from source)
21:                 if F_{i,j} = 0 then
22:                     run ← ReadUint8(from source)
23:                 end if
24:             end if
25:         end foreach
26:         NormaliseFrequenciesNx16_0(F_i, bits)
27:
28:         C_{i,0} ← 0
29:         for j ← 0 to 255 do
30:             C_{i,j+1} ← C_{i,j} + F_{i,j}
31:         end for
32:     end foreach
33: end procedure
```

3.2 rANS Nx16 Order-0

To decode an Order-0 encoded byte stream we first decode the symbol frequencies as described above and then decode the N interleaved rANS states. This is similar to the old (4x8) rANS decoder, but the RansRenorm function is replaced by a single 16-bit renormalisation instead of a loop using 8-bit values and can interleave to different amounts.

```
1: function RansGetCumulativeFreqNx16(R, bits)
2:     return R AND ((1 << bits) − 1)
3: end function
```

function RansAdvanceStepNx16($R, c, f, bits$)
    return $f \times (R >> bits) + (R \text{ AND } ((1 << bits) - 1) - c$
end function

function RansRenormNx16(R)
    if $R < (1 << 15)$ **then**
        $R \leftarrow (R << 16)$+ ReadUint16
    end if
    return R
end function

function RansDecodeNx16_0(len, N)
    ReadFrequenciesNx16_0(F, C)
    for $j \leftarrow 0$ **to** $N - 1$ **do**
        $R_j \leftarrow$ ReadUint32
    end for
    for $i \leftarrow 0$ **to** $len - 1$ **do**
        $j \leftarrow i \bmod N$
        $f \leftarrow$ RansGetCumulativeFreqNx16($R_j,\ 12$)
        $s \leftarrow$ RansGetSymbolFromFreq($C,\ f$)
        $out_i \leftarrow s$
        $R_j \leftarrow$ RansAdvanceStepNx16($R_j,\ C_s,\ F_s,\ 12$)
        $R_j \leftarrow$ RansRenormNx16(R_j)
    end for
    return out
end function

3.3 rANS Nx16 Order-1

The Order-1 code is comparable to Order-0 but with an extra dimension (the previous value) to the F and C matrices and a more complex system for storing the frequencies. The frequencies may also add up to either 1024 or 4096 (10 or 12 bits).

The N rANS states don't operate on interleaved data either, but in distinct regions so they can utilise their previous symbols without inter-dependency between the decoded output of the rANS states. This makes the handling of data that isn't a multiple of N is a little more complex too.

function RansDecodeNx16_1($len,\ N$)
    ReadFrequenciesNx16_1(F, C, $bits$)
    for $j \leftarrow 0$ **to** $N - 1$ **do**
        $R_j \leftarrow$ ReadUint32
        $L_j \leftarrow 0$ ▷ Last symbol
    end for
(The primary unrollable loop)
    for $i \leftarrow 0$ **to** $\lfloor len/N \rfloor - 1$ **do**
        for $j \leftarrow 0$ **to** $N - 1$ **do**
            $f \leftarrow$ RansGetCumulativeFreqNx16($R_j,\ bits$)
            $s \leftarrow$ RansGetSymbolFromFreq($C_{L_j},\ f$)
            $out_{i+j \times \lfloor len/N \rfloor} \leftarrow s$
            $R_j \leftarrow$ RansAdvanceStepNx16($R_j,\ C_{L_j,s},\ F_{L_j,s},\ bits$)
            $R_j \leftarrow$ RansRenormNx16(R_j)
            $L_j \leftarrow s$
        end for
    end for
(The remainder for data not a multiple of N in size, using R_{N-1} throughout.)
    for $i \leftarrow i \times N$ **to** $len - 1$ **do**
        $f \leftarrow$ RansGetCumulativeFreqNx16($R_{N-1},\ bits$)
        $s \leftarrow$ RansGetSymbolFromFreq($C_{L_{N-1}},\ f$)
        $out_i \leftarrow s$

21: $R_{N-1} \leftarrow$ RansAdvanceStepNx16(R_{N-1}, $C_{L_{N-1},s}$, $F_{L_{N-1},s}$, $bits$)
22: $R_{N-1} \leftarrow$ RansRenormNx16(R_{N-1})
23: $L_{N-1} \leftarrow s$
24: **end for**
25: **return** *out*
26: **end function**

3.4 rANS Nx16 Run Length Encoding

For symbols that occur many times in succession, we can replace them with a single symbol and a count. In this specification, run lengths are always provided for certain symbols (even if the run length is 1) and never for the other symbols (even if many are consecutive).

The data stream is split into two parts: the meta-data holding run-lengths and the run-removed data itself.

Bytes	Type	Name	Description
?	uint7	*rle_meta_len*	RLE meta-data-size times 2. The bottom bit is a flag to indicate whether *rle_meta* is uncompressed (1) or compressed (0).
?	uint7	*rle_len*	Size of uncompressed data before DecodeRLE is applied
?	uint7	(*comp_meta_len*)	Only stored if bottom bit of *rle_meta_len* is unset. Size of compressed RLE meta data.
?	uint8[]	*rle_meta*	RLE meta-data. Decompress with RansDecodeNx16_0 if bottom bit of *rle_meta_len* is unset.

The meta-data format starts with the count of symbols which have runs associated with them (zero being interpreted as all of them) and the list of these symbol values. This is followed by the run lengths encoded as variable sized integers in the *uint7* format.

(Reads and optionally uncompresses the blob of run-lengths and the array L)
(indicating which symbols have associates run-lengths.)
1: **function** DecodeRLEMeta(N)
2: $L \leftarrow (0, ...)$ ▷ (Set to zero for all $i \in \{0, 1, ..., 255\}$)
3: *rle_meta_len* ←ReadUint7
4: *len* ←ReadUint7 ▷ Length of uncompressed O0/O1 data, pre-expansion
5: **if** *rle_meta_len* AND 1 **then**
6: *rle_meta* ←ReadData($\lfloor rle_meta_len/2 \rfloor$)
7: **else**
8: *comp_meta_len* ←ReadUint7
9: *rle_meta* ←ReadData(*comp_meta_len*)
10: *rle_meta* ←RansDecodeNx16_0(*rle_meta_len*/2, N, *source* = *rle_meta*, *source* = *rle_meta*)
11: **end if**

12: $n \leftarrow$ ReadUint8(*metadata*)
13: **if** $n = 0$ **then**
14: $n \leftarrow 256$
15: **end if**
16: **for** $i \leftarrow 0$ **to** $n - 1$ **do**
17: $s \leftarrow$ ReadUint8(*metadata*)
18: $L_s \leftarrow 1$
19: **end for**

20: **return** (L, *rle_meta*, *len*)
21: **end function**

The use of the run length meta-data occurs when expanding the uncompressed data, after Order-0 or Order-1 data decompression.

(Expands data (*in*) using run-length metadata)
1: **function** DecodeRLE(*in*, L, *metadata*, *in_len*)

```
    j ← 0
    for i ← 0 to in_len − 1 do
        sym ← ReadUint8(in)
        if L_s then
            run ← ReadUint7(metadata)
            for k ← 0 to run do
                out_{j+k} ← s
            end for
            j ← j + run + 1
        else
            out_j ← s
            j ← j + 1
        end if
    end for
    return out
end function
```

3.5 rANS Nx16 Bit Packing

If the alphabet of used symbols in the uncompressed data stream is small - no more than 16 - then we can pack multiple symbols together to form bytes prior to compression. This permits 2, 4, 8 and infinite (all symbols are the same) numbers per byte. The distinct symbol values do not need to be adjacent as a mapping table P converts mapped value x to original symbol P_x.

The packed format is split into uncompressed meta-data (below) and the compressed packed data.

Bytes	Type	Name	Description
1	byte	*nsym*	Number of distinct symbols
nsym	byte[]	P	Symbol map
?	uint7	*len*	Length of packed data

The first meta-data byte holds *nsym*, the number of distinct values, followed by *nsym* bytes to construct the P map. If $nsym = 1$ then the byte stream is a stream of constant values and no bit-packing is done (we know every value already). If $nsym = 2$ then each symbol is 1 bit (8 per byte), if $2 < nsym \leq 4$ symbols are 2 bits each (4 per byte) and if $4 < nsym \leq 16$ symbols are 4 bits each (2 per byte). It is not permitted to have $nsym > 16$ or $nsym = 0$ as bit packing is not possible. Bits are unpacked from low to high.

Decoding this meta-data is implemented by the DecodePackMeta function below.

```
function DecodePackMeta
    nsym ←ReadUint8
    for i ← 0 to nsym − 1 do
        P_i ← ReadUint8
    end for
    len ← ReadUint7
    return (P, nsym, len)
end function
```

After uncompressing the main data stream, it should be unpacked using DecodePack below. The format of this data stream is packed data as described above.

```
function DecodePack(data, P, nsym, len)
    j ← 0                                   ▷ Index into data; i is index into output
    if nsym ≤ 1 then                        ▷ Constant value
        for i ← 0 to len − 1 do
            out_i ← P_0
        end for

    else if nsym ≤ 2 then                   ▷ 1 bit per value
        for i ← 0 to len − 1 do
            if i mod 8 = 0 then
```

```
            v ← data_j
            j ← j + 1
        end if
        out_i ← P_(v AND 1)
        v = v >> 1
    end for

else if nsym ≤ 4 then                                   ▷ 2 bits per value
    for i ← 0 to len − 1 do
        if i mod 4 = 0 then
            v ← data_j
            j ← j + 1
        end if
        out_i ← P_(v AND 3)
        v = v >> 2
    end for

else if nsym ≤ 16 then                                  ▷ 4 bits per value
    for i ← 0 to len − 1 do
        if i mod 2 = 0 then
            v ← data_j
            j ← j + 1
        end if
        out_i ← P_(v AND 15)
        v = v >> 4
    end for

else
    ERROR
end if

return out
end function
```

3.6 Striped rANS Nx16

If we have a series of 32-bit values, we can often get better compression by treating it as a series of 4 8-bit values representing the first to last bytes in each 32-bit word, than we can by simply processing it as a stream of 8-bit values. Each $4th$ byte is sent to its own stream producing 4 interleaved streams, so the 1^{st} stream will hold data from byte 0, 4, 8, etc while the 2^{nd} stream will hold data from byte 1, 5, 9, etc. Each of those four streams is then itself compressed using this compression format.

For example an input block of small unsigned 32-bit little-endian numbers may use RLE for the first three streams as they are mostly zero, and a non-RLE Order-0 entropy encoder for the last stream.

In the general case we describe this as N-way interleaved streams. We can consider this interleaving process to be equivalent to a table transpose of M rows by N columns to N rows by M columns, followed by compressing each N row independently.

The byte stream consists of a 7-bit encoded uncompressed combined length, a byte holding the value of N, followed by N compressed lengths also 7-bit encoded. Finally the data sub-streams themselves, each a valid *cdata* stream, follow.

Normally our *cdata* format will include the decoded size, but with STRIPE we can omit this from the internal compressed sub-streams (using the NOSIZE flag) as given the total length we know how to compute the sub-lengths.

Reproducing the original uncompressed data involves decoding the N sub-streams and interleaving them together again (reversing the table transpose). The uncompressed data length may not necessary be an exact multiple of N, in which case the latter uncompressed sub-streams may be 1 byte shorter.

As an example starting with input data D we define the transposed data T as:

$$D = aA\underline{A}bB\underline{B}cC\underline{C}dD\underline{D}e$$

$$T = [\ abcde,\ ABCD,\ \underline{ABCD}\ ]$$

Note our example data is not a multiple of N long, missing $E\underline{E}$, which gives T fragments of length [5, 4, 4].

If D_i is the i^{th} character in D and $T_{j,i}$ is the i^{th} character of the j^{th} substring in T, transformations between D and T are defined as:

$$T_{j,i} = D_{iN+j}$$

$$D_i = T_{(i \bmod N),\ (i \operatorname{div} N)}$$

```
1:  function RansDecodeStripe(len)
2:      N ←ReadUint8
3:      for j ← 0 to N do                                  ▷ Fetch N compressed lengths
4:          clen_j ←ReadUint7
5:      end for

6:      for j ← 0 to N do                                  ▷ Decode N streams
7:          ulen_j ← (len div N) + ((len mod N) > j)       ▷ (x > y) expression being 1 if true, 0 if false
8:          T_j ←RansDecodeNx16(ulen_j)
9:      end for

10:     for j ← 0 to N − 1 do                              ▷ Stripe
11:         for i ← 0 to ulen_j − 1 do
12:             out_{i×N+j} ← T_{j,i}
13:         end for
14:     end for
15:     return out
16: end function
```

3.7 Combined rANS Nx16 Format

We combine the Order-0 and Order-1 rANS Nx16 encoder with optional run-length encoding, bit-packing and four-way interleaving into a single data stream.

Bits	Type	Name	Description
8	uint8	*flag*	Data format bit field
Unless NoSize flag is set:			
?	uint7	ulen	Uncompressed length
If Stripe flag is set:			
8	uint8	N	Number of sub-streams
?	uint7[]	clen[]	N copies of compressed sub-block length
?	uint8[]	cdata[]	N copies of Compressed data sub-block (recurse)
If Cat flag is set (and Stripe flag is unset):			
?	uint8[]	udata	Uncompressed data stream
If Pack flag is set (and neither Stripe or Cat flags are set):			
?	uint8[]	pack_meta	Pack lookup table
If RLE flag is set (and neither Stripe or Cat flags are set):			
?	uint8[]	rle_meta	RLE meta-data
If neither Stripe or Cat flags are set:			
?	uint8[]	cdata	Entropy encoded data stream (see Order flag)

The first byte of our generalised data stream is a bit-flag detailing the type of transformations and entropy

encoders to be combined, followed by optional meta-data, followed by the actual compressed data stream. The bit-flags are defined below, but note not all combinations are permitted.

Bit AND value	Code	Description
1	ORDER	Order-0 or Order-1 entropy coding
2	reserved	Reserved (for possible order-2/3)
4	N32	Interleave $N = 32$ rANS states (else $N = 4$)
8	STRIPE*	multi-way interleaving of byte streams
16	NOSIZE	original size is not recorded (for use by STRIPE)
32	CAT	Data is uncompressed
64	RLE	Run length encoding, with runs and literals encoded separately
128	PACK	Pack 2, 4, 8 or infinite symbols per byte

(∗) Not to be used in conjunction with other bit-field values except NOSIZE.

Bit-packing and run length encoding transforms have their own meta-data, which is decoded prior to the main compressed data stream. After decoding, these transforms are applied in order of RLE followed by unpack as required and in that order.

For example a *flag* data-format value of 197 indicates a byte stream should decode the pack meta-data, the RLE meta-data and the Order-1 compressed data itself, all using 32 rANS states, and then apply the RLE followed by Unpack transforms to yield the original uncompressed data.

RANSDECODENX16 describes the decoding of the generalised RANS Nx16 decoder.

```
1:  function RansDecodeNx16(len)
2:      flags ←ReadUint8
3:      if flags AND NoSize ≠ 0 then
4:          len ←ReadUint7
5:      end if
6:      if flags AND Stripe then
7:          data ←RansDecodeStripe(len)
8:          return data
9:      end if
10:     if flags AND N32 then
11:         N ← 32
12:     else
13:         N ← 4
14:     end if
                                                    ▷ Read meta-data
15:     if flags AND Pack then
16:         pack_len ← len
17:         (P, nsym, len) ←DecodePackMeta
18:     end if
19:     if flags AND RLE then
20:         rle_len ← len
21:         (L, rle_meta, len) ←DecodeRLEMeta(N)
22:     end if
                                    ▷ Uncompress main data block
23:     if flags AND Cat then
24:         data ←ReadData(len)
25:     else if flags AND Order then
26:         data ←RansDecodeNx16_1(len, N)
27:     else
28:         data ←RansDecodeNx16_0(len, N)
29:     end if
                                    ▷ Apply data transformations
30:     if flags AND RLE then
31:         data ←DecodeRLE(data, L, rle_meta, rle_len)
32:     end if
33:     if flags AND Pack then
34:         data ←DecodePack(data, P, nsym, pack_len)
```

35: **end if**
36: **return** *data*
37: **end function**

4 Range coding

The range coder is a byte-wise arithmetic coder that operates by repeatedly reducing a probability range (for example 0.0 to 1.0) one symbol (byte) at a time, with the complete compressed data being represented by any value within the final range.

This is easiest demonstrated with a worked example, so let us imagine we have an alphabet of 4 symbols, 't', 'c', 'g', and 'a' with probabilities 0.2, 0.3, 0.3 and 0.2 respectively. We can construct a cumulative distribution table and apply probability ranges to each of the symbols:

Symbol	Probability	Range low	Range high
t	0.2	0.0	0.2
c	0.3	0.2	0.5
g	0.3	0.5	0.8
a	0.2	0.8	1.0

As a *conceptual example* (note: this is not how it is implemented in practice, see below) using arbitrary precision floating point mathematics this could operate as follows.

If we wish to encode a message, such as "cat" then we will encode one symbol at a time ('c', 'a', 't') successively reducing the initial range of 0.0 to 1.0 by the cumulative distribution for that symbol. At each point the new range is adjusted to be the proportion of the previous range covered by the cumulative symbol range. See the table footnotes below for the worked mathematics.

Range low	/ high	Symbol	Sym. low	/ high	New range low	New range high
0.000	1.000	c	0.2	0.5	$0 + (1 - 0) \times .2$	$0 + (1 - 0) \times .5$
0.200	0.500	a	0.8	1.0	$.2 + (.5 - .2) \times .8$	$.2 + (.5 - .2) \times 1$
0.440	0.500	t	0.0	0.2	$.44 + (.5 - .44) \times 0$	$.44 + (.5 - .44) \times .2$
0.440	0.452	<end>				

Our final range is 0.44 to 0.452 with any value in that range representing "cat", thus 0.45 would suffice. A pictorial example of this process is below.

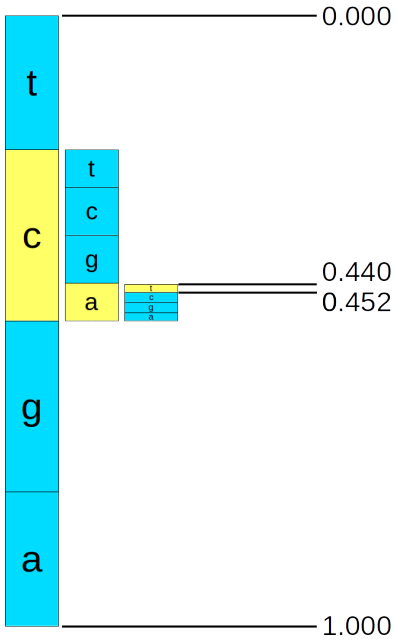

Figure 1: A pictorial demonstration of range reduction.

Decoding is simply the reverse of this. In the above picture we can see that 0.45 would read off 'c', 'a' and 't' by repeatedly comparing the symbol ranges to the current range and using those to identify the symbol and produce a new range.

Range low	Range high	Fraction into range	Symbol
0.000	1.000	0.450	c
0.200	0.500	0.833[a]	a
0.440[b]	0.500	0.167	t

a. 0.45 into range 0.2 to 0.5: $(0.45 - 0.2)/(0.5 - 0.2) = 0.833$. This falls within the 0.8 to 1.0 symbol range for 'a'.
b. 'a' symbol range 0.8 to 1.0 applied to range 0.2 to 0.5: $0.2 + 0.8 \times (0.5 - 0.2) = 0.44$ and $0.2 + 1.0 \times (0.5 - 0.2) = 0.5$.

Note: The above example is not how the actual implementation works[6]. For efficiency, we use integer values having a starting range of 0 to $2^{32} - 1$. We write out the top 8-bits of the range when low and high become the same value. Special care needs to be taken to handle small values that are numerically close but straddling a top byte value, such as 0x37ffba20 to 0x38000034. The decoder does not need to do anything special here, but the encoder must track the number of 0xff or 0x00 values to emit in order to avoid needing arbitrary precision integers.

Pseudocode for the range codec decoding follows. This implementation uses code (next few bytes in the current bit-stream) and range instead of low and high, both 32-bit unsigned integers. This specification focuses on decoding, but given the additional complexity of the precision overflows in encoder we describe this implementation too.

RangeDecodeCreate initialises the range coder, reading the first bytes of the compressed data stream.

```
1: function RangeDecodeCreate
2:     range ← 2^32 − 1                          ▷ Maximum 32-bit unsigned value
3:     code  ← 0                                 ▷ 32-bit unsigned
4:     for i ← 0 to 4 do
5:         code ← (code << 8)+ReadUint8
6:     end for
7:     code ← code AND 2^32 − 1
8:     return this range coder (range, code)
9: end function
```

Decoding each symbol is in two parts; getting the current frequency and updating the range.

```
1: function RangeGetFreq(tot_freq)
2:     range ← range div tot_freq
3:     return code div range
4: end function
1: procedure RangeDecode(sym_low, sym_freq, tot_freq)
2:     code  ← code − sym_low × range
3:     range ← range × sym_freq
4:     while range < 2^24 do                     ▷ Renormalise
5:         range ← range << 8
6:         code  ← (code << 8)+ReadUint8
7:     end while
8: end procedure
```

As mentioned above, the encoder is more complex as it cannot shift out the top byte until it has determined the value. This can take a considerable while if our current low / high (low + range) are very close but span a byte boundary, such as 0x37ffba20 to 0x38000034, where ultimately we will later emit either 0x37 or 0x38. To handle this case, when the range gets too small but the top bytes still differ, the encoder caches the top byte of low (0x37) and keeps track of how many 0xff or 0x00 values will need to be written out once we finally observe which value the range has shrunk to.

The RangeEncode function is a straight forward reversal of the RangeDecode, with the exception of the special code for shifting the top byte out of the *low* variable.

```
1: procedure RangeEncode(sym_low, sym_freq, tot_freq)
```

[6]This implementation was designed by Eugene Shelwein, based on Michael Schindler's earlier work.

```
2:    old_low ← low
3:    range   ← range div tot_freq
4:    low     ← low + sym_low × range
5:    range   ← range × sym_freq

6:    if low < old_low then
7:        carry ← 1                                              ▷ overflow
8:    end if
9:    while range < 2^24 do                                      ▷ Renormalise
10:       range ← range << 8
11:       RangeShiftLow
12:   end while
13: end procedure
```

RangeShiftLow is the main heart of the encoder renormalisation. It tracks the total number of extra bytes to emit and *carry* indicates whether they are a string of 0xFF or 0x00 values.

```
1: procedure RangeShiftLow
2:    if low < 0xff000000 or carry ≠ 0 then
3:        if carry = 0 then
4:            WriteByte(cache)                                   ▷ top byte cache plus FFs
5:            while FFnum > 0 do
6:                WriteByte(0xff)
7:                FFnum ← FFnum − 1
8:            end while
9:        else
10:           WriteByte(cache + 1)                               ▷ top byte cache + 1 plus 00s
11:           while FFnum > 0 do
12:               WriteByte(0)
13:               FFnum ← FFnum − 1
14:           end while
15:       end if
16:       cache ← low >> 24                                      ▷ Copy of top byte ready for next flush
17:       carry ← 0
18:   else
19:       FFnum ← FFnum + 1
20:   end if

21:   low ← low << 8
22: end procedure
```

For completeness, the Encoder initialisation and finish functions are below.

```
1: procedure RangeEncodeStart
2:    low    ← 0
3:    range  ← 2^32 − 1
4:    FFnum  ← 0
5:    carry  ← 0
6:    cache  ← 0
7: end procedure
```

```
1: procedure RangeEncodeEnd
2:    for i ← 0 to 4 do                                          ▷ Flush any residual state in low
3:        RangeShiftLow
4:    end for
5: end procedure
```

4.1 Adaptive Modelling

The probabilities passed to the range coder may be fixed for all scenarios (as we had in the "cat" example), or they may be adaptive and context aware. For example the letter 'u' occurs around 3% of time in English text,

but if the previous letter was 'q' it is close to 100% and if the previous letter was 'u' it is close to 0%. Using the previous letter is known as an Order-1 entropy encoder, but the context can be anything. We can also adaptively adjust our probabilities as we encode or decode, learning the likelihoods and thus avoiding needing to store frequency tables in the data stream covering all possible contexts.

To do this we use a statistical model, containing an array of symbols S and their frequencies F. The sum of these frequences must be less than $2^{16} - 16$ so after adjusting the frequencies it never go above the maximum unsigned 16-bit integer. When they get too high, they are renormalised by approximately halving the frequencies (ensuring none drop to zero).

Typically an array of models are used where the array index represents the current context.

To encode any symbol the entropy encoder needs to know the frequency of the symbol to encode, the cumulative frequencies of all symbols prior to this symbol, and the total of all frequencies. For decoding a cumulative frequency is obtained given the frequency total and the appropriate symbol is found matching this frequency. Symbol frequencies are updated after each encode or decode call and the symbols are kept in order of most-frequent symbol first in order to reduce the overhead of scanning through the cumulative frequencies.

ModelCreate initialises a model by setting every symbol to have a frequency of 1. (At no point do we permit any symbol to have zero frequency.)

```
1: function ModelCreate(num_sym)
2:     total_freq ← num_sym
3:     max_sym ← num_sym − 1
4:     for i ← 0 to max_sym do
5:         S_i ← i
6:         F_i ← 1
7:     end for
8:     return this model (total_freq, max_sym, S, F)
9: end function
```

ModelDecode is called once for each decoded symbol. It returns the next symbol and updates the model frequencies automatically.

```
1: function ModelDecode(rc)
2:     freq ← rc.RangeGetFrequency(total_freq)
3:     x ← 0
4:     acc ← 0
5:     while acc + F_x ≤ freq do
6:         acc ← acc + F_x
7:         x ← x + 1
8:     end while
9:     rc.RangeDecode(acc, F_x, total_freq)
10:    F_x ← F_x + 16                                  ▷ Update model frequencies
11:    total_freq ← total_freq + 16
12:    if total_freq > 2^16 − 17 then
13:        ModelRenormalise
14:    end if
15:    sym ← S_x
16:    if x > 0 and F_x > F_{x−1} then
17:        Swap(F_x, F_{x−1})
18:        Swap(S_x, S_{x−1})
19:    end if
20:    return sym
21: end function
```

ModelRenormalise is called whenever the total frequencies get too high. The frequencies are halved, taking sure to avoid any zero frequencies being created.

```
1: procedure ModelRenormalise
2:     total_freq ← 0
3:     for i ← 0 to max_sym do
4:         F_i ← F_i − (F_i div 2)
5:         total_freq ← total_freq + F_i
```

```
    end for
end procedure
```

4.2 Order-0 and Order-1 Encoding

We can combine the model defined above and the range coder to provide a simple function to perform Order-0 entropy decoder.

```
function DecodeOrder0(len)
    max_sym ←ReadUint8
    if max_sym = 0 then
        max_sym ← 256
    end if
    model_lit ←ModelCreate(max_sym)

    rc ←RangeDecodeCreate
    for i ← 0 to len − 1 do
        out_i ← model_lit.ModelDecode(rc)
    end for
    return out
end function
```

The Order-1 variant simply uses an array of models and selects the appropriate model based on the previous value encoded or decoded. This array index is our "context".

```
function DecodeOrder1(len)
    max_sym ←ReadUint8
    if max_sym = 0 then
        max_sym ← 256
    end if
    for i ← 0 to max_sym − 1 do
        model_lit_i ←ModelCreate(max_sym)
    end for

    rc ←RangeDecodeCreate
    last ← 0
    for i ← 0 to len − 1 do
        out_i ← model_lit_last.ModelDecode(rc)
        last ← out_i
    end for
    return out
end function
```

4.3 RLE with Order-0 and Order-1 Encoding

The DecodeOrder0 and DecodeOrder1 codecs can be expanded to include a count of how many runs of each symbol should be decoded. Both order 0 and order 1 variants are possible.

After the symbol is decoded, the run length must be decoded to indicate how many *extra* copies of this symbol occur. Long runs are broken into a series of lengths of no more than 3. If length 3 is decoded it indicates we must decode an additional length and add to the current one. The context used for the run length model is the symbol itself for the initial run, 256 for the first continuation run (if ≥ 4) and 257 for any further continuation runs. Thus encoding 10 'A' characters would first store symbol 'A' followed by run length 3 (with context 'A'), length 3 (context 256), length 3 (context 257), and length 1 (context 257).

For example, if we have the string "ABBCCCCDDDDD" we will record "A"<0> "B"<1> "C"<3,0> and "D"<3,1>.

```
function DecodeRLE0(len)
    max_sym ←ReadUint8
    if max_sym = 0 then
        max_sym ← 256
```

```
    end if
    model_lit ←ModelCreate(max_sym)
    for i ← 0 to 257 do
        model_run_i ←ModelCreate(4)
    end for

    rc ←RangeDecodeCreate
    i ← 0
    while i < len do
        out_i ← model_lit.ModelDecode(rc)
        part ← model_run_{out_i}.ModelDecode(rc)
        run ← part
        rctx ← 256
        while part = 3 do
            part ← model_run_{rctx}.ModelDecode(rc)
            rctx ← 257
            run ← run + part
        end while
        for j ← 1 to run do
            out_{i+j} ← out_i
        end for
        i ← run + 1
    end while
    return out
end function
```

The order-1 run length variant is identical to order-0 except the previous symbol is used as the context for the next literal. The context for the run length does not change.

```
function DecodeRLE1(len)
    max_sym ←ReadUint8
    if max_sym = 0 then
        max_sym ← 256
    end if
    for i ← 0 to max_sym − 1 do
        model_lit_i ←ModelCreate(max_sym)
    end for
    for i ← 0 to 257 do
        model_run_i ←ModelCreate(4)
    end for

    rc ←RangeDecodeCreate
    last ← 0
    i ← 0
    while i < len do
        out_i ← model_lit_{last}.ModelDecode(rc)
        last ← out_i
        part ← model_run_{last}.ModelDecode(rc)
        run ← part
        rctx ← 256
        while part = 3 do
            part ← model_run_{rctx}.ModelDecode(rc)
            rctx ← 257
            run ← run + part
        end while
        for j ← 1 to run do
            out_{i+j} ← last
        end for
        i ← run + 1
```

30: **end while**
31: **return** *out*
32: **end function**

We wrap up the Order-0 and 1 entropy encoder, both with and without run length encoding, into a data stream that specifies the type of encoded data and also permits a number of additional transformations to be applied. These transformations support bit packing (for example a data block with only 4 distinct values can be packed with 4 values per byte), no-op for tiny data blocks where entropy encoding would grow the data and N-way interleaving of the 8-bit components of a 32-bit value.

Bits	Type	Name	Description
8	uint8	*flag*	Data format bit field
Unless NoSize flag is set:			
?	uint7	ulen	Uncompressed length
If Stripe flag is set:			
8	uint8	N	Number of sub-streams
?	uint7[]	clen[]	N copies of compressed sub-block length
?	uint8[]	cdata[]	N copies of Compressed data sub-block (recurse)
If Cat flag is set (and Stripe flag is unset):			
?	uint8[]	udata	Uncompressed data stream
If Pack flag is set (and neither Stripe or Cat flags are set):			
?	uint8[]	pack_meta	Pack lookup table
If neither Stripe or Cat flags are set:			
?	uint8[]	cdata	Entropy encoded data stream (see Order / RLE / Ext flags)

The first byte of our generalised data stream is a bit-flag detailing the type of transformations and entropy encoders to be combined, followed by optional meta-data, followed by the actual compressed data stream. The bit-flags are defined below, but note not all combinations are permitted.

Bit AND value	Code	Description
1	Order*	Order-0 or Order-1 entropy coding
2	reserved	Reserved (for possible order-2/3)
4	Ext	"External" compression via bzip2
8	Stripe†	N-way interleaving of byte streams
16	NoSize	Original size is not recorded (used by Stripe)
32	Cat†	Data is uncompressed
64	RLE*	Run length encoding, with runs and literals encoded separately
128	Pack	Pack 2, 4, 8 or infinite symbols per byte

(∗) Has no effect when Ext flag is set.
(†) Not to be used in conjunction with other flags except Pack and NoSize.

Of these Stripe is the most complex. As with the rANS Nx16 encoder, the data is rearranged such that every N^{th} byte is adjacent in a single block producing N distinct sub-blocks. Each of the N streams is then itself compressed using this compression format.

For example an input block of small unsigned 32-bit little-endian numbers may use RLE for the first three streams as they are mostly zero, and a non-RLE Order-0 entropy encoder of the last stream. Normally our data format will include the decoded size, but with Stripe we can omit this from the internal compressed streams as we know their size will be a computable fraction of the combined data.

The data layout differs for each of these bit types, as described below in the ArithDecode function. Some of these can be used in combination, so the order needs to be observed. The Pack format has additional meta data. This is decoded first, before entropy decoding and finally expanding the specified pack transformation after decompression. For example value 193 is indicates a byte stream should be decoded with an RLE aware order-1 entropy encoder and then unpacked.

1: **function** ArithDecode(*len*)

```
    flags ←ReadUint8
    if flags AND NoSize ≠ 0 then
        len ←ReadUint7
    end if
    if flags AND Stripe then
        data ←DecodeStripe(len)
        return data
    end if
    if flags AND Pack then
        pack_len ← len
        (P, nsym, len) ←DecodePackMeta
    end if
                                                        ▷ Entropy Decoding
    if flags AND Cat then
        data ←ReadData(len)
    else if flags AND Ext then
        data ←DecodeEXT(len)
    else if flags AND RLE then
        if flags AND Order then
            data ←DecodeRLE1(len)
        else
            data ←DecodeRLE0(len)
        end if
    else
        if flags AND Order then
            data ←DecodeOrder1(len)
        else
            data ←DecodeOrder0(len)
        end if
    end if
                                                        ▷ Apply data transformations
    if flags AND Pack then
        data ←DecodePack(data, P, nsym, pack_len)
    end if
    return data
end function
```

The specifics of each sub-format are described below, in the order (minus meta-data specific shuffling) they are applied.

- **Stripe**: The byte stream consists of a 7-bit encoded uncompressed length and a byte holding the number of substreams N, and their 7-bit encoded compressed data streams lengths. This is then followed by the substreams themselves, each of which is a valid *cdata* stream as defined above, hence this offers a recursive mechanism as each substream has its own format byte.

  The total uncompressed byte stream is then an interleaving of one byte in turn from each of the N substreams (in order of 1st to Nth). Thus an array of 32-bit unsigned integers could be unpacked using Stripe to compress each of the 4 8-bit components together with their own algorithm.

```
function DecodeStripe(len)
    N ←ReadUint8
    for j ← 0 to N do                                   ▷ Fetch N compressed lengths
        clen_j ←ReadUint7
    end for

    for j ← 0 to N do                                   ▷ Decode N streams
        ulen_j ← (len div N) + ((len mod N) > j)        ▷ (x > y) expression being 1 if true, 0 if false
        T_j ←ArithDecode(ulen_j)
    end for
```

```
    for j ← 0 to N − 1 do                                          ▷ Stripe
        for i ← 0 to ulen_j − 1 do
            out_{i×N+j} ← T_{j,i}
        end for
    end for
    return out
end function
```

- **NoSize**: Do not store the size of the uncompressed data stream. This information is not required when the data stream is one of the four sub-streams in the Stripe format.
- **Cat**: If present, all other bit flags should be zero, with the possible exception of NoSize or Pack.

  The uncompressed data stream is the same as the compressed stream. This is useful for very short data where the overheads of compressing are too high.
- **Order**: Bit field defining order-0 (unset) or order-1 (set) entropy encoding, as described above by the DecodeOrder0 and DecodeOrder1 functions.
- **RLE**: Bit field defining whether the Order-0 and Order-1 encoding should also use a run-length. When set, the DecodeRLE0 and DecodeRLE1 functions will be used instead of DecodeOrder0 and DecodeOrder1.
- **Ext**: Instead of using the adaptive arithmetic coder for decompression (with or without RLE), this uses an compression codec defined in an "external" library. Currently the only supported such codec is Bzip2. In future more may be added, so the "magic number" (the file signature, typically in first few bytes of data) *must* be validated to check the external codec being used.

  Given bzip2 is already supported elsewhere in CRAM, the purpose of adding it here is to permit bzip2 compression after Pack and Stripe transformations. This may be tidied up in later CRAM releases to clarify the separation between compression codecs and data transforms, but that requires more major restructuring so for compatibility with v3.0 these have been placed into this single codec.

```
function DecodeExt(len)
    if Bzip2 magic number is present then
        return DecodeBzip2(len)
    else
        Error
    end if
end function
```

- **Pack**: Data containing only 1, 2, 4 or 16 distinct values can have multiple values packed into a single byte (infinite, 8, 4 or 2). The distinct symbol values do not need to be adjacent as a mapping table P converts mapped value x to original symbol P_x.

  The packed format is split into uncompressed meta-data (below) and the compressed packed data as described in the rANS Nx16 bit-packing section. The same DecodePackMeta and DecodePack functions are used.

5 Name tokenisation codec

Sequence names (identifiers) typically follow a structured pattern and compression based on columns within those structures usually leads to smaller sizes. The sequence name (identifier) tokenisation relies heavily on the rANS Nx16 and Adaptive arithmetic coders described above.

As an example, take a series of names:

```
I17_08765:2:123:61541:01763#9
I17_08765:2:123:1636:08611#9
I17_08765:2:124:45613:16161#9
```

We may wish to tokenise each of these into 7 tokens, e.g. "I17_08765:2:", "123", ":", "61541", ":", "01763"and "#9". Some of these are multi-byte strings, some single characters, and some numeric, possibily with a leading zero. We also observe some regularly have values that match the previous line (the initial prefix string, colons, "#9") while others are numerically very close to the value in the previous line (124 vs 123).

The name tokeniser compares each name against a previous name (which is not necessarily the one immediately prior) and encodes this name either as a series of differences to the previous name or marking it as an exact duplicate. A maximum of 128 tokens are permitted within any single read name.

Token IDs (types) are listed below.

ID	Type	Value	Description
0	TYPE	Type	Used to determine the type of token at a given position
5	DUP	Integer (distance)	The entire name is a duplicate of an earlier one. Used in position 0 only
6	DIFF	Integer (distance)	The entire name differs to earlier ones. Used in position 0 only
1	STRING	String	A nul-terminated string of characters
2	CHAR	Byte	A single character
7	DIGITS	$0 \leq$ Int $< 2^{32}$	A numerical value, not containing a leadng zero
3	DIGITS0	$0 \leq$ Int $< 2^{32}$	A numerical value possibly starting in leading zeros
4	DZLEN	Int length	Length of associated DIGITS0 token
8	DELTA	$0 \leq$ Int < 256	A numeric value being stored as the difference to the numeric value of this token on the previous name
9	DELTA0	$0 \leq$ Int < 256	As DELTA, but for numeric values starting with leading zeros
10	MATCH	(none)	This token is identical type and value to the same position in the previous name (NB: not permitted for DELTA/DELTA0)
11	NOP	(none)	Does nothing
12	END	(none)	Marks end of name

The tokens and values are stored in a 2D array of byte streams, $B_{pos,type}$, where pos 0 is reserved for name meta-data (whether it is a duplicate name) and pos 1 onwards is for the first, second and later tokens. *Type* is one of the token types listed above, corresponding to the type of data being stored. Some token types may also have associated values. $B_{pos,\texttt{TYPE}}$ (*type*) holds the token type itself and that is then used to retrieve any associated value(s) if appropriate from $B_{pos,type}$. Thus multiple types at the same token position will have their values encoded in distinct data streams, e.g. if position 5 is of type either DIGITS or DELTA then data streams will exist for $B_{5,\texttt{TYPE}}$, $B_{5,\texttt{DIGITS}}$ and $B_{5,\texttt{DELTA}}$. Decoding per name continues until a token of type END is observed.

More detail on the token types is given below.

- **TYPE**: This is the first token fetched at each token position. It holds the type of the token at this position, which in turn may then require retrieval from type-specific data streams at this position.

  For position 0, the TYPE field indicates whether this record is an exact duplicate of a prior read name or has been encoded as a delta to an earlier one.

- **DUP**, **DIFF**: These types are fetched for position 0, at the start of each new identifier. The value is an integer value describing how many reads before this (with 1 being the immediately previous name) we are comparing against. When we subsequently refer to "previous name" below, we always mean the one indicated by the DIFF field and not the one immediately prior to the current name.

  The first record will have a DIFF of zero and no delta or match operations are permitted.

- **STRING**: We fetch one byte at a time from the value byte stream, appending to the name buffer until the byte retrieved is zero. The zero byte is not stored in the name buffer. For purposes of token type MATCH, a match is defined as entirely matching the string including its length.

- **CHAR**: Fetch one single byte from the value byte stream and append to the name buffer.

- **DIGITS**: Fetch 4 bytes from the value byte stream and intrepret these as a little endian unsigned integer. This is appended to the name buffer as string of base-10 digits, most significant digit first. Larger values may be represented, but will require multiple DIGITS tokens. Negative values may be encoded by treating the minus sign as a CHAR or STRING and storing the absolute value.

- **DIGITS0**, **DZLEN**: This fetches the 4 byte value from $B_{pos,DIGITS0}$ and a 1 byte length from $B_{pos,DZLEN}$. As per DIGITS, the value is intrepreted as a little endian unsigned integer. The length indicates the total

size of the numeric value when displayed in base 10 which must be greater than $\log_{10}(value)$ with any remaining length indicating the number of leading zeros. For example if DIGITS0 value is 123 and DZLEN length is 5 the string "00123" must be appended to the name.

For purposes of the MATCH type, both value and length must match.

- **DELTA**: Fetch a 1 byte value and add this to the DIGITS value from the previous name. The token in the prior name must be of type DIGITS or DELTA.

  MATCH is not supported for this type.

- **DELTA0**: As per DELTA, but the 1 byte value retrieved is added to the DIGITS0 value in the previous name. No DZLEN value is retrieved, with the length from the previous name being used instead. The token in the prior name must be of type DIGITS0 or DELTA0.

  MATCH is not supported for this type.

- **MATCH**: This token matches the token at the same position in the previous name. (The previous name is permitted to also have a MATCH token at this position, in which case it recurses to its previous name.)

  MATCH is only valid when the token being matched against is CHAR, STRING, DIGITS, DIGITS0 or MATCH. (I.e. matching a numeric delta will not repeat the delta increment.)

  No value is needed for MATCH tokens.

- **NOP**: This token type does nothing. The purpose of this is simply to permit skipping tokens in order to keep token numbers in sync, such as when processing "10" vs "-10" with the latter needing an additional "-" token.

- **END**: Marks the end of the name. A nul byte is added to the name output buffer. No value is needed for END tokens.

Decoding needs some simple functions to read successive bytes from our token byte streams, as 8-bit characters or unsigned integers, as 32-bit unsigned integers and nul-terminated strings. We reuse the ReadUint32 and related functions with the byte array specified as input.

```
    (Convert an integer to a string form in base-10 digits, at least len bytes long with leading zeros)
1:  function LeftPadNumber(val, len)
2:      str ← val                          ▷ Implicit language-specific Integer to String conversion
3:      while Length(str) < len do
4:          str ← '0' ++ str
5:      end while
6:      return str
7:  end function
```

For the main name decoding loop, we use a single dimensional array of names decoded so far, N, and a two dimensional array of their tokens T (indexed by name number n and token position t within that name). We define a function to decode the n^{th} name (N_n) using a previous m^{th} name (N_m). The tokens T are used in `MATCH` and `DELTA` token types to copy data from when constructing the name.

Now we have the basic primitives for pulling from the B byte streams, decoding the n^{th} individual name is as follows[7]:

```
    (Decodes the n^th name, returning N_n and updating globals N_n and T_n)
1:  function DecodeSingleName(n)
2:      type ← ReadUint8(B_{0,TYPE})
3:      dist ← ReadUint32(B_{0,type})
4:      m ← n − dist
5:      if type = DUP then
6:          N_n ← N_m
7:          T_n ← T_m                      ▷ Copy for all T_{n,*}
```

[7]For simplicity of algorithm description, we take a flexible approach as to whether we read/write T in numeric or string form. For example a `DELTA` token will fetch the previous token as a string, interpret it as a numeric value, add to it, and then write it back as a string. Practical implementations may wish to separate out T into distinct integer and string arrays.

```
        return N_n
    end if

    t ← 1                                                    ▷ Token number t
    repeat
        type ← ReadUint8(B_{t,TYPE})
        if type = CHAR then
            T_{n,t} ← ReadChar(B_{t,CHAR})
        else if type = STRING then
            T_{n,t} ← ReadString(B_{t,STRING})
        else if type = DIGITS then
            T_{n,t} ← ReadUint32(B_{t,DIGITS})
        else if type = DIGITS0 then
            d ← ReadUnt32(B_{t,DIGITS0})
            l ← ReadUint8(B_{t,DZLEN})
            T_{n,t} ← LeftPadNumber(d, l)
        else if type = DELTA then
            T_{n,t} ← T_{m,t}+ ReadUint8(B_{t,DELTA})
        else if type = DELTA0 then
            d ← T_{m,t}+ ReadUint8(B_{t,DELTA0})
            l ← Length(T_{m,t})                              ▷ String length including leading zeros
            T_{n,t} ← LeftPadNumber(d, l)
        else if type = MATCH then
            T_{n,t} ← T_{m,t}
        else
            T_{n,t} ← ''
        end if
        N_n ← N_n ++ T_{n,t}
        t ← t + 1
    until type = END
    return N_n
end function
```

Given a complex name with both position and type specific values, this can lead to many separate data streams. The name tokeniser codec is a format within a format, as the multiple byte streams $B_{pos,type}$ are serialised into a single byte stream.

The serialised data stream starts with two unsigned little endian 32-bit integers holding the total size of uncompressed name buffer and the number of read names, and a flag byte indicating whether data is compressed with arithmetic coding or rANS Nx16. Note the uncompressed size is calculated as the sum of all name lengths including a termination byte per name (e.g. the nul char). This is irrespective of whether the implementation produces data in this form or whether it returns separate name and name-length arrays.

This is then followed by serialised data and meta-data for each token stream. Token types, *ttype* holds one of the token ID values listed above in the list above, plus special values to indicate certain additional flags. Bit 6 (64) set indicates that this entire token data stream is a duplicate of one earlier. Bit 7 (128) set indicates the token is the first token at a new position. This way we only need to store token types and not token positions.

The total size of the serialised stream needs to be already known, in order to determine when the token types finish.

Bytes	Type	Name	Description
4	uint32	*uncomp_length*	Length of uncompressed name buffer
4	uint32	*num_reads*	Number of read names
1	uint8	*use_arith*	Whether compression is arithmetic (1) or rANS Nx16 (0)
For each token data stream			
1	uint8	*ttype*	Token type code plus flags (64=duplicate, 128=next token position).
If ttype AND 64 (duplicate)			
1	uint8	*dup_pos*	Duplicate from this token position
1	uint8	*dup_type*	Duplicate from this token type ID
else if not duplicate			
?	uint7	*clen*	compressed length
clen	uint8[]	*cdata*	compressed data stream

A few tricks are used to remove some byte streams. In addition to the explicit marking of duplicate bytes streams, if a byte stream of token types is entirely MATCH apart from the very first value it is discarded. It is possible to regenerate this during decode by observing the other byte streams. For example if we have a byte stream $B_{5,DIGITS}$ but no $B_{5,TYPE}$ then we assume the contents of $B_{5,TYPE}$ consist of one DIGITS type followed by as many MATCH types as are needed.

The *cdata* stream itself is as described in the relevant entropy encoder section above (rANS or arithmetic coding).

```
    (Decodes and uncompresses the serialised token byte streams)
 1: function DecodeTokenByteStreams(use_arith)
 2:     sz ← 0
 3:     t ← −1
 4:     repeat
 5:         ttype ← ReadUint8
 6:         tok_new ← ttype AND 128
 7:         tok_dup ← ttype AND 64
 8:         type ← ttype AND 63
 9:         if tok_new ≠ 0 then
10:             t ← t + 1
11:             if type ≠ TYPE then
12:                 B_{t,TYPE} ← (type, TOK_MATCH, TOK_MATCH, ...)        ▷ for nnames − 1 times
13:             end if
14:         end if

15:         if tok_dup ≠ 0 then
16:             dup_pos ← ReadUint8
17:             dup_type ← ReadUint8
18:             B_{t,type} ← B_{dup_pos,dup_type}
19:         else
20:             clen ← ReadUint7
21:             data ← ReadData(clen)
22:             if use_arith then
23:                 B_{t,type} ← ArithDecode(clen, source = data)
24:             else
25:                 B_{t,type} ← RansDecodeNx16(clen, source = data)
26:             end if
27:         end if
28:     until EOF
29:     return B
30: end function
```

```
    (Decodes all names, returning N)
 1: function DecodeNames
 2:     ulen ← ReadUint32
```

```
    nnames ← ReadUint32
    use_arith ← ReadUint8
    B ← DecodeTokenByteStreams(use_arith)
    for n ← 0 to nnames − 1 do
        N_n ← DecodeSingleName(n)
    end for
    return N
end function
```

6 FQZComp quality codec

The FQZComp quality codec uses an adaptive statistical model to predict the next quality value in a given context (comprised of previous quality values, position along this sequence, whether the sequence is the second in a pair, and a running total of number of times the quality has changed in this sequence).

For each position along the sequence, the models produce probabilities for all possible next quality values, which are passed into an arithmetic entropy encoder to encode or decode the actual next quality value. The models are then updated based on the actual next quality in order to learn the statistical properties of the quality data stream. This step wise update process is identical for both encoding and decoding.

The algorithm is a generalisation on the original fqzcomp program, described in *Compression of FASTQ and SAM Format Sequencing Data* by Bonfield JK, Mahoney MV (2013). PLoS ONE 8(3): e59190. `https://doi.org/10.1371/journal.pone.0059190`

6.1 FQZComp Models

The FQZComp process utilises knowledge of the read lengths, complement (qualities reversed) status, and a generic parameter selector, but in order to maintain a strict separation between CRAM data series this knowledge is stored (duplicated) within the quality data stream itself. Note the complement model is only needed in CRAM 3.1 as CRAM 4 natively stores the quality in the original orientation already. Both reversed and duplication models have no context and are boolean values.

The parameter selector model also has no context associated with it and encodes max_sel distinct values. The selector value may be quantised further using *stab* (Selector Table) to reduce the selector to fewer sets of parameters. This is useful if we wish to use the selector bits directly in the context using the same parameters. The selector is arbitrary and may be used for distinguishing READ1 from READ2, as a precalculated "delta" instead of the running total, distinguishing perfect alignments from imperfect ones, or any other factor that is shown to improve quality predictability and increase compression ratio (average quality, number of mismatches, tile, swathe, proximity to tile edge, etc).

The quality model has a 16-bit context used to address an array of 2^{16} models, each model permitting max_sym distinct quality values. The context used is defined by the FQZcomp parameters, of which there may be multiple sets, selected using the selector model. There are 4 read length models each having max_sym of 256. Each model is used for the 4 successive bytes in a 32-bit length value.

The entropy encoder used is shared between all models, so the bit streams are multiplexed together.

The 16-bit quality value context is constructed by adding sub-contexts together consisting of previous quality values, position along the current record, a running count (per record) of how many times the quality value has differed to the previous one (delta), and an arbitrary stored selector value, each shifted to a defined location within the combined context value (*qloc*, *ploc*, *dloc* and *sloc* respectively). The qual, pos and delta sub-contexts are computed from the previous data while the selector, if used, is read directly from the compressed data stream. The selector may be used to switch parameter sets, or simply to group quality strings into arbitrary user-defined sub-sets. The numeric values for each of these components can be passed through lookup tables (*qtab* for quality, *ptab* for positions, *dtab* for running delta and *stab* for turning the selector s into a parameter index x). These all convert the monotonically increasing range 0→M to a (usually smaller) monotonically increasing 0→N. For example if we wish to use the approximate position along a 100 byte string, we may uniformly map 0→127 to 0→15 to utilise 4 bits of our 16-bit combined context.

As some sequencing instruments produce binned qualities, e.g. 0, 10, 25, 35, these values are squashed to

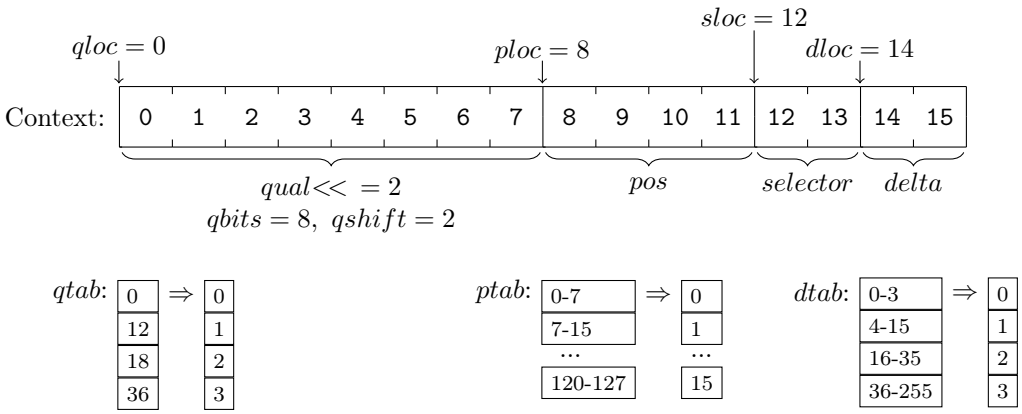

Figure 2: An example FQZComp configuration.

incremental values from 0 to $max_sym - 1$ where max_sym is the maximum number of distinct quality values observed. If this transform is required, the flag $have_qmap$ will be set and a mapping table ($qmap$) will hold the original quality values. The encoded qualities will be the smaller mapped range.

The quality sub-context is constructed by shifting left the previous quality sub-context by $qshift$ bits and adding the current quality after passing through the $qmap$ transform and if defined through the $qtab$ lookup table. The quality context is limited to $qbits$ long and is added to the combined context starting at bit $qloc$. The quality sub-context is reset to zero at the start of each new record. [8]

The position context is simply the number of remaining quality values in this record, so is a value starting at record length (minus 1) and decrementing. As with the quality context it may be passed through a lookup table $ptab$ before shifting left by $ploc$ bits and adding to the combined context.

Delta is a count of the number of times the quality value has changed from one value to a different one. Thus a run of identical values will not increase delta. It gets reset to zero at the start of every record. It may be adjusted by the $dtab$ lookup table and is shifted by $dloc$ before adding to the combined context.

The selector value may also be used as a sub-context, if the do_sel paramter is set. The initial context value (reset per record) is defined within each parameter set, providing a more general purpose alternative to adding the selector value at a defined location ($sloc$) into the context.

Thus the full context can be updated after each decoded quality with the following pseudocode. Note for brevity this is assuming the pos, $delta$, $prevq$, $qctx$ and sel parameters referred are global and updateable.

```
    (Add quality q to produce and return a new context ctx)
 1: function FQZUpdateContext(params, q)
 2:     ctx ← params.context                          ▷ Also the initial value
 3:     qctx ← (qctx << params.qshift) + qtab_q
 4:     ctx ← ctx + ((qctx AND (2^params.qbits − 1)) << params.qloc)
 5:     if params.pflags AND 32 then                  ▷ have_ptab
 6:         p ←Min(pos, 1023)
 7:         ctx ← ctx + (ptab_p << params.ploc)
 8:     end if
 9:     if params.pflags AND 64 then                  ▷ have_dtab
10:         d ←Min(delta, 255)
11:         ctx ← ctx + (dtab_d << params.dloc)
12:         if prevq ≠ q then
13:             delta ← delta + 1
14:         end if
15:         prevq ← q
16:     end if
```

[8] For example if we have 4 quality values in use – 0, 10, 25 and 35 – we will be encoding quality values 0, 1, 2 and 3. We may wish to define $qbits$ to be 6 and $qshift$ to be 2 such that the previous 3 quality values can be used as context, for the prediction of the next quality value. There will likely be little reason to use $qtab$ in this scenario, but an encoder could define $qtab$ to convert {0, 1, 2, 3} to {0, 0, 0, 1} and use $qshift$ of 1 instead, giving us knowledge of which of the previous 6 values were maximum quality.

17: **if** $params.pflags$ AND 8 **then** ▷ do_sel
18: $ctx \leftarrow ctx + (sel << params.sloc)$
19: **end if**
20: **return** ctx AND $(2^{16} - 1)$
21: **end function**

In summary context is produced using the following models:

Model	Max symbol	Context size	Description
$model_qual$	max_sym	2^{16}	Primary model for quality values
$model_len$	256	4	Read length models with the context 0-3 being successive byte numbers (little endian order)
$model_rev$	2	none	Used if $pflags.do_rev$ is defined. Indicating which strings to reverse.
$model_dup$	2	none	Used if $pflags.do_dup$ is defined. Indicates if this whole string is a duplicate of the last one
$model_sel$	max_sel	none	Used if $gflags.multi_param$ or $pflags.do_sel$ are defined.

6.2 FQZComp Data Stream

The start of an FQZComp data stream consists of the parameters used by the decoder. The data layout is as follows.

Bits	Type	Name	Description
8	uint8	*version*	FQZComp format version: must be 5
8	uint8	*gflags*	Global FQZcomp bit-flags. From lowest bit to highest: 1: *multi_param*: indicates more than one parameter block is present. Otherwise set $nparam = 1$ 2: *have_stab*: indicates the parameter selector is mapped through *stab*. Otherwise set $stab_i = i$ 4: *do_rev*: *model_revcomp* will be used (CRAM v3.1)

If multi_param gflag is set:

8	uint8	*nparam*	Number of parameter blocks (defaults to 1)

If have_stab gflag is set:

8	uint8	*max_sel*	Maximum parameter selector value
variable	array	*stab*	Parameter selector table

Parameter block: repeated nparam times: (selected via model_sel and stab)

16	uint16	*context*	Starting context value
8	uint8	*pflags*	Per-parameter block bit-flags. From lowest bit to highest: 1: Reserved 2: *do_dedup*: model_dup will be used 4: *do_len*: model_len will be used for every record 8: *do_sel*: model_sel will be used 16: *have_qmap*: indicates quality map is present 32: *have_ptab*: Load *ptab*, otherwise position contexts are unused 64: *have_dtab*: Load *dtab*, otherwise delta contexts are unused 128: *have_qtab*: Load *qtab*, otherwise set $qtab_i = i$
8	uint8	*max_sym*	Total number of distinct quality values
4	uint4 (high)	*qbits*	Total number of bits for quality context
4	uint4 (low)	*qshift*	Left bit shift per successive quality in quality context
4	uint4 (high)	*qloc*	Bit position of quality context
4	uint4 (low)	*sloc*	Bit position of selector context
4	uint4 (high)	*ploc*	Bit position of position context
4	uint4 (low)	*dloc*	Bit position of delta context

If have_qmap pflag is set:

variable	uint8[*max_sym*]	*qmap*	Map for unbinning quality values.

If have_qtab pflag is set:

variable	array	*qtab*	Quality context lookup table

If have_tab pflag is set:

variable	array	*ptab*	Position context lookup table

If have_tab pflag is set:

variable	array	*dtab*	Delta context lookup table

FQZDecodeParams below describes the pseudocode for reading the parameter block.

```
1:  procedure FQZDecodeParams
2:      vers ←ReadUint8
3:      if vers ≠ 5 then
4:          ERROR
5:      end if
6:      gflags ← ReadUint8
7:      if gflags AND 1 then                                   ▷ multi_param
8:          nparam ← ReadUint8
9:          max_sel ← nparam
10:     else
11:         nparam ← 1
12:         max_sel ← 0
13:     end if
14:     if gflags AND 2 then                                   ▷ have_stab
15:         max_sel ← ReadUint8
16:         stab ← ReadArray(256)
17:     end if
18:     max_sym ← 0
19:     for p ← 0 to nparam − 1 do
20:         param_p ← FQZDecodeSingleParam
21:         if max_sym < param_p.max_sym then
22:             max_sym ← param_p.max_sym                      ▷ Maximum across all param sets
23:         end if
24:     end for
25: end procedure
```

```
1:  function FQZDecodeSingleParam
2:      p.context    ← ReadUint16
3:      p.flags      ← ReadUint8
4:      p.max_sym    ← ReadUint8
5:      p.first_len  ← 1
6:      x            ← ReadUint8
7:      p.qbits      ← x div 16
8:      p.qshift     ← x mod 16
9:      x            ← ReadUint8
10:     p.qloc       ← x div 16
11:     p.sloc       ← x mod 16
12:     x            ← ReadUint8
13:     p.ploc       ← x div 16
14:     p.dloc       ← x mod 16
15:     if p.flags AND 16 then                                 ▷ Have qmap
16:         for i ← 0 to p.max_sym − 1 do
17:             p.qmap_i ← ReadUint8
18:         end for
19:     end if
20:     if p.flags AND 128 then                                ▷ Have qtab
21:         p.qtab ← ReadArray(256)
22:     else
23:         for i ← 0 to 256 do
24:             p.qtab_i ← i
25:         end for
26:     end if
27:     if p.flags AND 32 then                                 ▷ Have ptab
28:         p.ptab ← ReadArray(1024)
29:     end if
30:     if p.flags AND 64 then                                 ▷ Have dtab
```

```
        p.dtab ← ReadArray(256)
    end if
    return p
end function
```

FQZCreateModels creates the decoder models based on the above parameters and the shared range coder.

```
procedure FQZCreateModels
    rc ←RangeDecodeCreate
    for i ← 0 to 3 do
        model_len_i ←ModelCreate(256)
    end for
    for i ← 0 to 2^16 − 1 do
        model_qual_i ←ModelCreate(max_sym + 1)
    end for
    model_dup ←ModelCreate(2)
    model_rev ←ModelCreate(2)
    if max_sel > 0 then
        model_sel ←ModelCreate(max_sel + 1)
    end if
end procedure
```

ReadArray reads an array A of size n which maps values 0 to $n-1$ to a smaller range (0 to $m-1$), both monotonically increasing. For efficiency this is done using a two-level run length encoding.

Assuming $m < n$ there will be runs of the same value. We measure run lengths for all values (even if they are zero). For example an array $A = \{0, 1, 3, 4, 5, 6, 7, 7, 7, 7\}$ may be converted to run lengths $R = \{1, 1, 0, 1, 1, 1, 1, 4\}$. To keep values in this array fitting within one byte, long runs are broken down in a successive series of 255 values, so a run of length 600 becomes 255 255 90.

This array R is no longer monotonically increasing but may still have repeated values, so is run-length encoded by storing the number of additional values whenever the last two lengths match. This converts R to $R2 = \{1, 1, +0, 0, 1, 1, +2, 4\}$ where the '+' symbol is shown purely to indicate the values representing the additional run-length copy numbers. (This also now turns the example run of 600 above into 255 255 0 90.)

The final array $R2$ is the stored data stream. The decoder process is the reverse of the above, starting by converting $R2$ to R and then A. The following pseudocode demonstrates this process.

```
function ReadArray(n)
    i, j, z ← 0
    last ← −1
    while z < n do                                  ▷ Convert R2 to R
        run ← ReadUint8
        R_j ← run
        j ← j + 1
        z ← z + run
        if run = last then
            copy ← ReadUint8
            for x ← 1 to copy do
                R_j ← run
                j ← j + 1
            end for
            z ← z + run × copy
        end if
        last ← run
    end while

    i, j, z ← 0
    while z < n do                                  ▷ Convert R to A
```

```
        run_len ← 0
        repeat
            part ← R_j
            j ← j + 1
            run_len ← run_len + part
        until part ≠ 255
        for x ← 1 to run_len do
            A_z ← i
            z ← z + 1
        end for
        i ← i + 1
    end while

    return A
end function
```

The FQZComp main loop decodes data in the following order per read: read length (if not fixed), the flag for whether this is read 2 (if needed), a bit flag to indicate if the quality is duplicated (if needed), followed by record length number of quality values using various data gathered since the start of this read as context.

The output of this function is an array of quality values in the variable *output*, indexed with the i^{th} value via $output_i$. The output buffer is a concatenation of all quality values for each record. The record lengths are recorded, but note this is the number of qualities encoded in CRAM for this sequence record and this does not necessarily have to match the number of base calls (for example where qualities are explicitly specified for SNP bases but not elsewhere).

```
function FQZNewRecord
    sel ← 0
    x ← 0
    if max_sel > 0 then                                  ▷ Find parameter selector
        sel ← model_sel.ModelDecode(rc)
        if have_stab then
            x ← stab_sel
        end if
    end if
    param ← params_x

    if param.do_len or param.first_len then              ▷ Decode read length
        rec_len ←DecodeLength(rc)
        param.last_len ← rec_len
        param.first_len = 0
    else
        rec_len ← param.last_len
    end if
    pos ← rec_len

    if param.do_rev then                                 ▷ Check if needs reversal
        rev_rec ← model_rev.ModelDecode(rc)
        len_rec ← rec_len
    end if
    rec ← rec + 1

    is_dup ← 0
    if do_dedup then                                     ▷ Duplicate last string if appropriate
        if model_dup.ModelDecode(rc) > 0 then
            is_dup ← 1
        end if
    end if
```

```
    qctx ← 0
    delta ← 0
    prevq ← 0
    return x                                              ▷ Tabulated parameter selector
end function
procedure FQZDecode
    buf_len ← ReadUint7
    FQZDecodeParams
    FQZCreateModels
    i ← 0                                                 ▷ Position in total quality block
    pos ← 0                                               ▷ Remaining base count current quality string
next_record:
    while i < buf_len do
        if pos = 0 then                                   ▷ Reset state at start of each new record
            x ←FQZNewRecord
            if is_dup = 1 then
                for j ← 0 to rec_len − 1 do
                    output_{i+j} ← output_{i+j−rec_len}
                end for
                i ← i + rec_len
                pos ← 0
                go to next_record
            end if

            param ← params_x
            ctx ← param.context
        end if

        q ← model_qual_{ctx}.ModelDecode(rc)              ▷ Decode a single quality value
        if param.have_qmap then
            output_i ← qmap_q
        else
            output_i ← q
        end if

        ctx ←FQZUpdateContext(param, q)                   ▷ Also updates qctx, prevq and delta

        i ← i + 1
        pos ← pos − 1
    end while
    if do_rev then
        ReverseQualities(output, buf_len, rev, len)
    end if
end procedure
```

Read lengths are encoded as 4 8-bit bytes, each having its own model.

```
function DecodeLength(rc)
    rec_len ← model_len_0.ModelDecode(rc)
    rec_len ← rec_len + (model_len_1.ModelDecode(rc)<< 8)
    rec_len ← rec_len + (model_len_2.ModelDecode(rc)<< 16)
    rec_len ← rec_len + (model_len_3.ModelDecode(rc)<< 24)
    return rec_len
end function
```

For CRAM v4.0 quality values are stored in their original FASTQ orientation. For CRAM v3.1 they are stored in their alignment orientation and it may be beneficial for compression purposes to reverse them first. If so *do_rev* will be set and the ReverseQualities procedure called below after decoding.

```
1: procedure ReverseQualities(qual, qual_len, rev, len)
2:     rec ← 0
3:     i ← 0
4:     while i < qual_len do
5:         if rev_rec ≠ 0 then
6:             j ← 0
7:             k ← len_rec − 1
8:             while j < k do
9:                 Swap(qual_{i+j}, qual_{i+k})
10:                j ← j + 1
11:                k ← k − 1
12:            end while
13:            i ← i + len_rec
14:            rec ← rec + 1
15:        end if
16:    end while
17: end procedure
```