CRAM format specification (version 4.0)

samtools-devel@lists.sourceforge.net

4 Apr 2019

The master version of this document can be found at `https://github.com/samtools/hts-specs`.
This printing is version 924537d-dirty from that repository, last modified on the date shown above.

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 can be 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

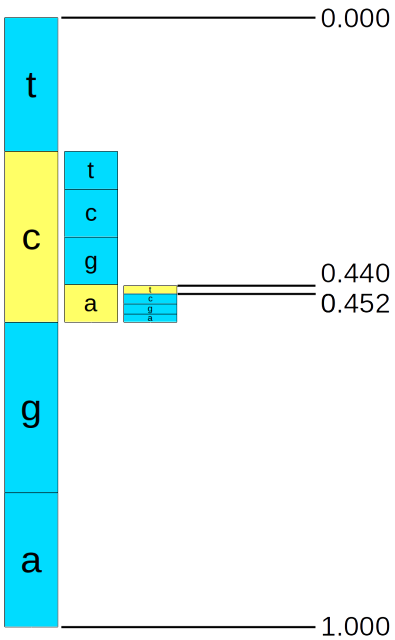
As a *conceptual example* (note: this is not how it is implemented in practice, see below) using arbitrary precision floating point mathematics this could operate as follows.

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	Range high	Symbol	Symbol low	Symbol high
0.000	1.000	c	0.2	0.5
0.200	0.500	a	0.8	1.0
0.440[a]	0.500[a]	t	0.0	0.2
0.440	0.452	<end>		

a. Old range 0.2 to 0.5 plus symbol range 0.8 to 1.0 gives an updated range of 0.44 to 0.5:
$0.2 + 0.8 \times (0.5 - 0.2) = 0.44$
$0.2 + 1.0 \times (0.5 - 0.2) = 0.50$

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.



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 not how the actual implementation works[1]. 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 are are numerically close but stradding 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.

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

```
procedure RangeCreate
    range ← 2^32 − 1                          ▷ Maximum 32-bit unsigned value
    code  ← 0                                 ▷ 32-bit unsigned
    for i ← 0 to 5 do
        code ← (code << 8)+ReadByte
    end for
end procedure
```

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

```
function RangeGetFreq(tot_freq)
    range ← range/tot_freq
```

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

```
    return code/range
end function
```

```
procedure RangeDecode(sym_low, sym_freq, tot_freq)
    code  ← code − sym_low × range
    range ← range × sym_freq
    while range < 2^24 do                                  ▷ Renormalise
        range ← range << 8
        code  ← (code << 8)+ReadByte
    end while
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.

```
procedure RangeEncode(sym_low, sym_freq, tot_freq)
    old_low ← low
    range   ← range/tot_freq
    low     ← low + sym_low × range
    range   ← range × sym_freq

    if low < old_low then
        carry ← 1                                          ▷ overflow
    end if
    while range < 2^24 do                                  ▷ Renormalise
        range ← range << 8
        RangeShiftLow
    end while
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.

```
procedure RangeShiftLow
    if low < 0xff000000 or carry ≠ 0 then
        if carry = 0 then
            WriteByte(cache)                               ▷ top byte cache plus FFs
            while FFnum > 0 do
                WriteByte(0xff)
                FFnum ← FFnum − 1
            end while
        else
            WriteByte(cache + 1)                           ▷ top byte cache + 1 plus 00s
            while FFnum > 0 do
                WriteByte(0)
                FFnum ← FFnum − 1
            end while
        end if
        cache ← low >> 24                                  ▷ Copy of top byte ready for next flush
        carry ← 0
    else
        FFnum ← FFnum + 1
    end if

    low ← low << 8
```

```
end procedure
```

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

```
procedure RangeEncodeStart
    low    ← 0
    range  ← 2^32 − 1
    FFnum  ← 0
    carry  ← 0
    cache  ← 0
end procedure
```

```
procedure RangeEncodeEnd
    for i ← 0 to 5 do                      ▷ Flush any residual state in low
        RangeShiftLow
    end for
end procedure
```

Statistical 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 $2^{16} - 32$. 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.)

```
procedure ModelCreate(num_sym)
    total_freq ← num_sym
    max_sym ← num_sym − 1
    for i ← 0 to max_sym do
        S_i ← i
        F_i ← 1
    end for
end procedure
```

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

```
function ModelDecode(rc)
    freq ← rc.RangeGetFrequency(total_freq)
    x ← 0
    acc ← 0
    while acc + F_x <= freq do
        acc ← acc + F_x
        x ← x + 1
    end while
    rc.RangeDecode(acc, F_x, total_freq)
    F_x ← F_x + 8                          ▷ Update model frequencies
```

```
    total_freq ← total_freq + 8
    if total_freq > 2^16 − 32 then
        ModelRenormalise
    end if
    sym ← S_x
    if x > 0 and F_x > F_{x−1} then
        SwapElement(F, x, x − 1)        ▷ swap F_x with F_{x−1}
        SwapElement(S, x, x − 1)        ▷ swap S_x with S_{x−1}
    end if
    return sym
end function
```

```
procedure SwapElement(A,i,j)
    tmp ← A_i
    A_i ← A_j
    A_j ← A_i
end procedure
```

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

```
procedure ModelRenormalise
    total_freq ← 0
    for i ← 0 to max_sym do
        F_i ← F_i − (F_i div 2)
        total_freq ← total_freq + F_i
    end for
end procedure
```

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 ←ReadByte
    model_lit ←ModelCreate(max_sym)

    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 ←ReadByte
    for i ← 0 to max_sym − 1 do
        model_lit_i ←ModelCreate(max_sym)
    end for

    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
```

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 258).

For example, if we have the string "RRRRUNN" we will decode symbol 'R' run 3, symbol 'U' run 0, symbol 'N' run 1.

```
function DecodeRLE0(len)
    max_sym ← ReadByte
    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

    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 ← ReadByte
    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

    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
    end while
    return out
end function
```

General Purpose Entropy Encoder

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, dictionary lookups for common 4-byte values and 4-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 X4 *flag is set:*			
?	uint7	clen1	Compressed sub-block length 1
?	uint7	clen2	Compressed sub-block length 2
?	uint7	clen3	Compressed sub-block length 3
?	uint7	clen4	Compressed sub-block length 4
?	uint8[]	cdata1	Compressed data sub-block 1 (recurse)
?	uint8[]	cdata2	Compressed data sub-block 2 (recurse)
?	uint8[]	cdata3	Compressed data sub-block 3 (recurse)
?	uint8[]	cdata4	Compressed data sub-block 4 (recurse)
If Cat *flag is set (and* X4 *flag is unset):*			
?	uint8[]	udata	Uncompressed data stream
If Dict *flag is set (and neither* X4 *or* Cat *flags are set):*			
?	uint8[]	dict_meta	Dictionary lookup table
If Pack *flag is set (and neither* X4 *or* Cat *flags are set):*			
?	uint8[]	pack_meta	Pack lookup table
If neither X4 *or* Cat *flags are set:*			
?	uint8[]	cdata	Entropy encoded data stream (see Order / RLE 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	Dict	map 32-bit values to 8-bit
8	X4	4-way interleaving of byte streams.
16	NoSize	original size is not recorded (used by X4)
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.

Of these X4 is the most complex. The uncompressed data must be a multiple of four bytes long. 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 of the last stream. Normally our data format will include the decoded size, but with X4 we can omit this from the internal four compressed streams as we know they will each be a quarter of the outer stream.

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 Dict and Pack formats have meta data. This is decoded first (dict then pack as appropriate), before entropy decoding and finally expanding any specified pack and dict transformations in that order. For example value 193 is indicates a byte stream should be decoded with an RLE aware order-1 entropy encoder and then unpacked.

```
function ArithDecode(len)
    flags ←ReadByte
    if flags AND NoSize ≠ 0 then
        out_len ←ReadInt7
    end if
    if flags AND X4 then
        out ←DecodeX4(len) return out
    end if
    if flags AND Cat then
        data ←DecodeCat
    end if
                                                        ▷ Decode meta-data
    if flags AND Dict then
        D ←DecodeDictMeta
    end if
    if flags AND Pack then                ▷ Swap order of this with Dict for consistency?
        P ←DecodePackMeta
    end if
                                                        ▷ Entropy Decoding
    if flags AND RLE then
        if flags AND Order then
            data ←DecodeRLE1
        else
            data ←DecodeRLE0
        end if
    else
        if flags AND Order then
            data ←DecodeOrder1
        else
            data ←DecodeOrder0
        end if
    end if
                                                        ▷ Apply data transformations
    if flags AND Pack then
        data ←DecodePack(data, P)
    end if
```

if $flags$ AND DICT **then**
  $data \leftarrow$ DECODEDICT($data$, D)
end if
$out \leftarrow data$
end function

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

- **X4**: The byte stream consists of a 7-bit encoded uncompressed length, which must be a multiple of 4, followed by 4 compressed lengths also 7-bit encoded. Each of these compressed byte streams is then itself a valid *cdata* stream and will recurse again, each starting with their own format flag. The total uncompressed byte stream is then an interleaving of one byte in turn from each of the 4 substreams (in order of 1st to 4th). Thus an array of 32-bit unsigned integers could be unpacked using X4 to compress each of the 8-bit components together with their own algorithm.

  function DECODEX4(len)
    $plen \leftarrow out\_len/4$
    for $i \leftarrow 0$ to 3 **do** ▷ Fetch 4 compressed lengths
      $clen_i \leftarrow$ READINT7
    end for
    $X0 \leftarrow$ ARITHDECODE($plen$) ▷ Decode 4 streams
    $X1 \leftarrow$ ARITHDECODE($plen$)
    $X2 \leftarrow$ ARITHDECODE($plen$)
    $X3 \leftarrow$ ARITHDECODE($plen$)

    $i \leftarrow 0$
    for $j \leftarrow 0$ to $plen - 1$ **do** ▷ Interleave
      $out_{i+0} \leftarrow X0_j$
      $out_{i+1} \leftarrow X1_j$
      $out_{i+2} \leftarrow X2_j$
      $out_{i+3} \leftarrow X3_j$
      $i \leftarrow i + 4$
    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 X4 format.

- **CAT**: If present, all other bit flags should be nul, with the possible exception of NOSIZE.

  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.

  function DECODECAT(len)
    for $i \leftarrow 0$ to $len - 1$ **do**
      $out_i \leftarrow$ READBYTE
    end for
    return out
  end function

- **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.

- **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 M converts quantised value x to original symbol M_x where x is the number of distinct possible symbols.

  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[]	*syms*	Symbol map

The first meta-dat byte holds *nsym*, the number of distinct values, followed by *nsym* bytes to construct the M 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 $nsym > 16$ as bit packing is not possible. Bits are unpacked from low to high.

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

```
function DecodePackMeta(M, nsym)
    nsym ←ReadByte
    for i ← 0 to nsym − 1 do
        M_i ← ReadByte
    end for
end function
```

The meta-data is unpacked at before entropy decoding. Once the main data block has been decoded, the byte stream is then expanded with the DecodePack function below.

```
function DecodePack(len)
    if nsym <= 1 then                                  ▷ Constant value
        for i ← 0 to len − 1 do
            out_i ← M_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 ← ReadByte
            end if
            out_i ← M_(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 ← ReadByte
            end if
            out_i ← M_(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 ← ReadByte
            end if
            out_i ← M_(v AND 15)
            v = v >> 4
        end for

    else
        Error
    end if

    return out
end function
```

- **DICT**:

  As with PACK, DICT has disjoint uncompressed meta-data and compressed data. The meta-data describes numeric dictionary.

  If our data consists of a few distinct 32-bit values then compression can be gained by using a lookup table and converting these to 8-bit values. We use the term *stride* to indicate the size of elements in the input data (4-byte for 32-bit values).

  For example, with values $V = \{0x1234 = 0,\ 0x1235 = 1,\ 0x1236 = 2,\ 0xbeef = 3\}$ we can replace this input data stream:

36	12	0	0	35	12	0	0	ef	be	0	0	36	12	0	0	34	12	0	0

  with a copy of map V (see below for format) and a new data byte-stream:

2	1	3	2	0

  Here we describe the reversal of this; the dictionary decoding method. In the example above our elements of V were all 2-bytes long (16-bit). Currently the DICT format is restricted to a *stride* of 4 and element size 2. The map format itself is described in a meta-data block:

Bytes	Type	Name	Description
½	nibble(high)	v	Size in bytes for each value in dictionary V (must be 2)
½	nibble(low)	s	Stride size in bytes for output (must be 4)
1	byte	n	Number of items in dictionary
?	byte[]	V	Dictionary of n v-byte values

  The first byte indicates the multi-byte stride size s to map in the low 4 bits and the size v of the multi-byte values in the top 4 bits, where $s \geq v$. Currently these must be 4 and 2 respectively.

  The next byte holds the number of values n in the map V. Following this is a run-length encoding of the map. The meta-data stream consists of a byte holding the number of literals in a row in the top 4 bits and the subsequent number of runs of consecutive literals in a row in the bottom 4 bits. For each literal, the next v bytes (little endian) represent the value.

  During decode, these literals will be expressed as s-byte little endian values. For each run the value is numerically one higher than before and is not stored.

  For example $V = \{1000, 1100, 1200, 1201, 1202, 1203, 1204, 2200\}$ is stored in the meta-data stream as:

24	08	43	e8	03	4c	04	b0	04	01	98	08
v 2 s 4	n	*run* 4 *lit* 3	1000		1100		1200		*run* 0 *lit* 1	2200	

```
function DecodeDictMeta(len)
    sv ←ReadByte
    stride ← sv mod 16                                  ▷ Currently must be 4
    vsize ← sv div 16                                   ▷ Currently must be 2
    n ←ReadByte
    i ← 0
    while i < n do
        x ←ReadByte
        lit ← x mod 16
        run ← x div 16
        for j ← 0 to lit − 1 do
            D_{i+j} ←ReadInt16                          ▷ For vsize = 2
        end for
        i ← i + lit
        if run > 0 then
            for j ← 0 to run − 1 do
```

$D_{i+j} \leftarrow D_{i+j-1} + 1$
end for
$i \leftarrow i + run$
end if
end while
return D, $stride$
end function

The compressed data stream follows (either immediately after this or following the PACK meta-data), encoded according to the other format bit flags. Once the bytes have been uncompressed a new data stream is generated by replacing each byte value x with the unsigned v-byte value V_x, thus growing the byte stream by v times.

function DECODEDICT($data$, out_len) ▷ Only for $stride = 4$
$i \leftarrow 0$
for $j \leftarrow 0$ to $out\_len - 1$ **do**
$v \leftarrow data_j$
$out_{i+0} \leftarrow (D_v >> 0)$ AND 255
$out_{i+1} \leftarrow (D_v >> 8)$ AND 255
$out_{i+2} \leftarrow (D_v >> 16)$ AND 255
$out_{i+3} \leftarrow (D_v >> 24)$ AND 255
$i \leftarrow i + 4$
end for
return out
end function

Name tokenisation codec

Sequence names (identifiers) typically follow a structured pattern and compression based on columns within those structures usually leads to smaller sizes.

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 is permitted within any single read name.

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

The sequence name (identifier) tokenisation relies heavily on the General Purpose Entropy Encoder described above.

A simplistic pseudocode for decoding the n^{th} individual name is:

function DECODENAME(n)
$type \leftarrow$ GET_TYPE(0, TYPE)
$dist \leftarrow$ GET_INT32(0, TYPE)

if $type$ = `DUP` **then**
    $name_n \leftarrow name_{n-dist}$
    return $name_n$
end if

$pos \leftarrow 0$
repeat
    $pos \leftarrow pos + 1$
    $type \leftarrow$ GET_TYPE(pos, `TYPE`)
    (Append to $name_n$ based on $type$)
until $type$ = `END`
return $name_n$
end function

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 is differs to earlier ones. Used in position 0 only.
1	STRING	String	A string of characters
2	CHAR	Byte	A single character
7	DIGITS	$0 \le \text{Int} < 2^{32}$	A numerical value, not containing a leadng zero
4	DIGITS0	$0 \le \text{Int} < 2^{32}$	A numerical value possibly starting in leading zeros
?	DZLEN	Int length	Length of DIGITS0 token.
11	DDELTA	$0 \le \text{Int} < 256$	A numeric value being stored as the difference to the value of this token on the previous name
12	DDELTA0	$0 \le \text{Int} < 256$	As DDELTA, but for numeric values starting with leading zeros
13	MATCH	(none)	This token is identical type and value to the same position in the previous name
14	END	(none)	Marks end of name

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 required 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 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.

  By convention 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 array, 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 the length.

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

- **DIGITS**: Fetch 4 bytes from the value array 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 bytes value from $T_{pos,DIGITS0}$ and a 1 byte length from $T_{pos,DZLEN}$. 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 primary and secondary lengths must match.

- **DDELTA**: Fetch a 1 byte value and add this to the DIGITS value from the previous name. It is invalid to have a DDELTA token when the previous name does not have a DIGITS token at this position.

  For the purposes of a MATCH type, the DDELTA value is assumed to match meaning we have another increment by the same amount.

- **DDELTA0**: As per DDELTA, 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.

  For the purposes of a MATCH type, the DDELTA0 value is assumed to match meaning we have another increment by the same amount and display to the same length.

- **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.)

  The definition of MATCH is token type specific, as described above. No value is needed for MATCH tokens.

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

Given a complex name and both position and type specific values, this can lead to many separate data streams. These are serialised into a single byte stream.

The packed data stream starts with two unsigned little endiand 32-bit integers holding the total size of uncompressed name buffer and the number of read names. This is followed the array elements themselves.

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.

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
For each token data stream			
1	uint8	*ttype*	Token type code.
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			
?	i7	*clen*	compressed length (7-bit encoding)
clen	*cdata*	stream	compressed data stream

TODO: write simple pseudocode for this stage.

The *cdata* stream itself is as described in the General Purpose Entropy Encoder section above, with the ArithDecode function.

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 quality value, 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`

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 distrinct values. The selector value may be quantised further using *stab* 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*, *rloc* 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→99 to 0→7 to utilise 3 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 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 decoded 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* squashing process 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. [2]

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

[2] 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.

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 various parameters referred are global and updateable.

```
function FQZUpdateContext(params, q)
    ctx ← params.context                                  ▷ Also the initial value
    qctx ← (qctx << params.qshift) + qtab_q
    ctx ← ctx + ((qctx AND (2^params.qbits − 1)) << params.qloc)
    if params.pflags AND 32 then                          ▷ have_ptab
        p ← Min(pos, 1023)
        ctx ← ctx + (ptab_p << params.ploc)
    end if
    if params.pflags AND 64 then                          ▷ have_dtab
        d ← Min(delta, 255)
        ctx ← ctx + (dtab_d << params.dloc)
        if prevq ≠ q then
            delta ← delta + 1
        end if
        prevq ← q
    end if
    if params.pflags AND 8 then                           ▷ do_sel
        ctx ← ctx + (sel << params.sloc)
    end if
    return ctx AND (2^16 − 1)
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 *hflags.multi_param* is defined and $nparam > 1$.

FQZComp Data Stream

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

DecodeFQZParams below describes the pseudocode for reading the parameter block.

```
procedure DecodeFQZParams
    vers ←ReadByte
    if vers ≠ 5 then
        ERROR
    end if
    gflags ← ReadByte
    if gflags AND 1 then                                          ▷ multi_param
        nparam ← ReadByte
        max_sel ← nparam
    else
        nparam ← 1
        max_sel ← 0
    end if
    if gflags AND 2 then                                          ▷ have_stab
        max_sel ← ReadByte
        stab ← ReadArray(256)
    end if
    max_sym ← 0
    for p ← 0 to nparam − 1 do
        param_p ← DecodeFQZSingleParam
        if max_sym < param_p.max_sym then
            max_sym ← param_p.max_sym             ▷ Maximum across all param sets
        end if
    end for
end procedure
```

```
function DecodeFQZSingleParam
    p.context    ← ReadUint16
    p.flags      ← ReadByte
    p.max_sym    ← ReadByte
    p.first_len  ← 1
    x          ← ReadByte
    p.qbits    ← x div 16
    p.qshift   ← x mod 16
    x          ← ReadByte
    p.qloc     ← x div 16
    p.sloc     ← x mod 16
    x          ← ReadByte
    p.ploc     ← x div 16
    p.dloc     ← x mod 16
    if p.flags AND 1 then                                         ▷ Have qmap
        for i ← 0 to p.max_sym − 1 do
            p.qmap_i ← ReadByte
        end for
    end if
    if p.flags AND 128 then                                       ▷ Have qtab
        p.qtab ← ReadArray(256)
    else
        for i ← 0 to 256 do
            p.qtab_i ← i
        end for
    end if
    if p.flags AND 16 then                                        ▷ Have ptab
        p.ptab ← ReadArray(1024)
```

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	table	*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_map pflag is set:

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

If have_qtab pflag is set:

variable	table	*qtab*	Quality context lookup table

If have_tab pflag is set:

variable	table	*ptab*	Position context lookup table

If have_tab pflag is set:

variable	table	*dtab*	Delta context lookup table

```
    end if
    if p.flags AND 64 then                                ▷ Have dtab
        p.qtab ← 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 ← RangeCreate
    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)
    end for
    model_dup ← ModelCreate(2)
    model_rev ← ModelCreate(2)
    if max_sel > 0 then
        model_sel ← ModelCreate(max_sel)
    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 $R = \{1, 1, 0, 1, 1, 1, 1, 4\}$. This array R is no longer monotonically increasing but still has 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.

Finally, for coping with runs of 255 or more any run value of 255 is assumed to be part of a larger run and the next value is read and added to run-length until it is no longer 255. For example run length 600 would be represented as 255 255 90.

The final array $R2$ is the stored data stream. The decoder process is the reverse of the above, starting by creating R and then A, for example using the following pseudocode.

```
function ReadArray(n)
    i, j, z ← 0
    last ← −1
    while z < n do                                        ▷ Convert R2 to R
        run ← ReadByte
        R_j ← run
        j ← j + 1
        z ← z + run
        if run = last then
            copy ← ReadByte
            for x ← 1 to copy do
                R_j ← run
                j ← j + 1
            end for
            z ← z + run × copy
        end if
    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 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 DecodeFQZNewRecord
    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
        if param.do_len = 0 then
            param.first_len = 0
        end if
    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 DecodeFQZ
    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 ← DecodeFQZNewRecord
            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
                rec ← rec + 1
                go to next_record
            end if

            param ← params_x
            last ← 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 qlast, prevq and delta

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

Where $\textsc{Min}(a, b)$ returns the smallest integer value from a and b.

```
function Min(a,b)
    if a < b then
        return a
    else
        return a
    end if
end function
```

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 last_len
end function
```

For CRAMv4 quality values are stored in their original FASTQ orientation. For CRAMv3 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.

```
procedure ReverseQualities(qual, qual_len, rev, len)
    rec ← 0
    for i ← 0 to qual_len − 1 do
        if rev_rec ≠ 0 then
            k ← len_rec − 1
            for j ← 0 to len_rec/2 do
                tmp ← qual_{i+j}
                qual_{i+j} ← qual_{i+k}
                qual_{i+k} ← tmp
            end for
        end if
    end for
end procedure
```