### Tag encodings

The tag dictionary (TD) describes the unique combinations of tag id / type that occur on each alignment record. For example if we search the id / types present in each record and find only two combinations – X1:i BC:Z SA:Z: and X1:i: BC:Z – then we have two dictionary entries in the TD map.

Let $L_i = \{T_{i0}, T_{i1}, \ldots, T_{ix}\}$ be a list of all tag ids for a record $R_i$, where $i$ is the sequential record index and $T_{ij}$ denotes $j$-th tag id in the record. The list of unique $L_i$ is stored as the TD value in the preservation map. Maintaining the order is not a requirement for encoders (hence "combinations"), but it is permissible and thus different permutations, each encoded with their own elements in TD, should be supported by the decoder. Each $L_i$ element in TD is assigned a sequential integer number starting with 0. These integer numbers are referred to by the TL data series. Using TD, an integer from the TL data series can be mapped back into a list of tag ids. Thus per alignment record we only need to store tag values and not their ids and types.

The TD is written as a byte array consisting of $L_i$ values separated with \0. Each $L_i$ value is written as a concatenation of 3 byte $T_{ij}$ elements: tag id followed by BAM tag type code (one of A, c, C, s, S, i, I, f, ~~F,~~ Z, H or B, as described in the SAM specification). For example the TD for tag lists X1:i BC:Z SA:Z and X1:i BC:Z may be encoded as X1CBCZSAZ\0X1CBCZ\0, with X1C indicating a 1 byte unsigned value for tag X1.

### Tag values

The encodings used for different tags are stored in a map. The key is 3 bytes formed from the BAM tag id and type code, matching the TD dictionary described above. Unlike the Data Series Encoding Map, the key is stored in the map as an ITF8 encoded integer, constructed using $(char1 << 16) + (char2 << 8) + type$. For example, the 3-byte representation of OQ:Z is {0x4F, 0x51, 0x5A} and these bytes are interpreted as the integer key 0x004F515A, leading to an ITF8 byte stream {0xE0, 0x4F, 0x51, 0x5A}.

| **Key** | **Value data type** | **Name** | **Value** |
|---|---|---|---|
| TAG ID 1:TAG TYPE 1 | encoding<byte[ ]> | read tag 1 | tag values (names and types are available in the data series code) |
| ... | | ... | ... |
| TAG ID N:TAG TYPE N | encoding<byte[ ]> | read tag N | ... |

Note that tag values are encoded as array of bytes. The routines to convert tag values into byte array and back are the same as in BAM with the exception of value type being captured in the tag key rather in the value. Hence consuming 1 byte for types 'C' and 'c', 2 bytes for types 'S' and 's', 4 bytes for types 'I', 'i' and 'f', and a variable number of bytes for types 'H', 'Z' and 'B'.

## 8.5 Slice header block

The slice header block is never compressed (block method=raw). For reference mapped reads the slice header also defines the reference sequence context of the data blocks associated with the slice. Mapped reads can be stored along with **placed unmapped**[3] reads on the same reference within the same slice.

Slices with the Multiple Reference flag (-2) set as the sequence ID in the header may contain reads mapped to multiple external references, including unmapped[3] reads (placed on these references or unplaced), but multiple embedded references cannot be combined in this way. When multiple references are used, the RI data series will be used to determine the reference sequence ID for each record. This data series is not present when only a single reference is used within a slice.

The Unmapped (-1) sequence ID in the header is for slices containing only unplaced unmapped[3] reads.

A slice containing data that does not use the external reference in any sequence may set the reference MD5 sum to zero. This can happen because the data is unmapped or the sequence has been stored verbatim instead of via reference-differencing. This latter scenario is recommended for unsorted or non-coordinate-sorted data.

The slice header block contains the following fields.

[3]Unmapped reads can be *placed* or *unplaced*. By placed unmapped read we mean a read that is unmapped according to bit 0x4 of the BF (BAM bit flags) data series, but has position fields filled in, thus "placing" it on a reference sequence. In contrast, unplaced unmapped reads have have a reference sequence ID of -1 and alignment position of 0.

```
12:         end if
13:         $mate\_ref\_id \leftarrow$ ReadItem(NS, Integer)
14:         $mate\_position \leftarrow$ ReadItem(NP, Integer)
15:         $template\_size \leftarrow$ ReadItem(TS, Integer)
16:     else if $CF$ AND 4 then                                   ▷ Mate is downstream
17:         if $next\_frag.bam\_flags$ AND 0x10 then
18:             $this.bam\_flags \leftarrow this.bam\_flags$ OR 0x20    ▷ next segment reverse complemented
19:         end if
20:         if $next\_frag.bam\_flags$ AND 0x04 then
21:             $this.bam\_flags \leftarrow this.bam\_flags$ OR 0x08    ▷ next segment unmapped
22:         end if
23:         $next\_frag \leftarrow$ ReadItem(NF,Integer)
24:         $next\_record \leftarrow this\_record + next\_frag + 1$
25:         Resolve $mate\_ref\_id$ for $this\_record$ and $next\_record$ once both have been decoded
26:         Resolve $mate\_position$ for $this\_record$ and $next\_record$ once both have been decoded
27:         Find leftmost and rightmost mapped coordinate in records $this\_record$ and $next\_record$.
28:         For leftmost of $this\_record$ and $next\_record$: $template\_size \leftarrow rightmost - leftmost + 1$
29:         For rightmost of $this\_record$ and $next\_record$: $template\_size \leftarrow -(rightmost - leftmost + 1)$
30:     end if
31: end procedure
```

Note as with the SAM specification a template may be permitted to have more than two alignment records. In this case the "mate" for each record is considered to be the next record, with the mate for the last record being the first to form a circular list. The above algorithm is a simplification that does not deal with this scenario. The full method needs to observe when record $this + NF$ is also labelled as having an additional mate downstream. One recommended approach is to resolve the mate information in a second pass, once the entire slice has been decoded. The final segment in the mate chain needs to set *bam_flags* fields 0x20 and 0x08 accordingly based on the first segment. This is also not listed in the above algorithm, for brevity.

## 10.5 Auxiliary tags

Tags are encoded using a tag line (TL data series) integer into the tag dictionary (TD field in the compression header preservation map, see section 8.4). See section 8.4 for a more detailed description of this process.

| **Data series type** | **Data series name** | **Field** | **Description** |
|---|---|---|---|
| int | TL | tag line | an index into the tag dictionary (TD) |
| * | ??? | tag name/type | 3 character key (2 tag identifier and 1 tag type), as specified by the tag dictionary |

```
1: procedure DecodeTagData
2:     $tag\_line \leftarrow$ ReadItem(TL,Integer)
3:     for all $ele \in container\_pmap.tag\_dict(tag\_line)$ do
4:         $name \leftarrow$ first two characters of $ele$
5:         $tag(type) \leftarrow$ last character of $ele$
6:         $tag(name) \leftarrow$ ReadItem($ele$, Byte[])
7:     end for
8: end procedure
```

In the above procedure, *name* is a two letter tag name and *type* is one of the permitted types documented in the SAM/BAM specification. Type is `A` (a single character), `c` (signed 8-bit integer), `C` (unsigned 8-bit integer), `s` (signed 16-bit integer), `S` (unsigned 16-bit integer), `i` (signed 32-bit integer), `I` (unsigned 32-bit integer), `f` (32-bit float), `Z` (nul-terminated string), `H` (nul-terminated string of hex digits) and `B` (binary data in array format with the first byte being one of c,C,s,S,i,I,f using the meaning above, a 32-bit integer for the number of array elements, followed by array data encoded using the specified format). All integers are little endian encoded.

For example a SAM tag `MQ:i` has name `MQ` and type `i` and will be decoded using one of MQc, MQC, MQs, MQS, MQi and MQI data series depending on size and sign of the integer value.