INTERNATIONAL STANDARD

ISO/IEC 23092-2

Second edition
2020-10

Information technology — Genomic information representation —

Part 2: Coding of genomic information

Technologies de l'information — Représentation des informations génomiques —

Partie 2: Codage des informations génomiques

Reference number
ISO/IEC 23092-2:2020(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this specification and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This specification was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this specification may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see http://patents.iec.ch).

Any trade name used in this specification is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This specification was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

This second edition cancels and replaces the first edition (ISO/IEC 23092-1:2019), which has been technically revised.

The main changes compared to the previous edition are as follows:

— The sequence decoding process for mismatches in classes I and HM has been clarified.

— In subclause 10.4 and its subclauses variable *numberOfAlignedRecordSegments* has been renamed to *numberOfMappedRecordSegments*.

— In subclause 10.4.2 the decoding process of pos and rtype descriptors with computed reference has been clarified.

— In subclause 11.3.4 the decoding process of pushin has been revised.

— The decoding of the *reverseComp* values has been revised.

— The determination of the offset of mismatches within spliced segments has been revised.

— The decoding process for signatures has been revised.

— The signalling of computed references has been clarified.

— In Clause 12 some decoding processes and some transformations have been clarified.

A list of all parts in the ISO/IEC 23092 series can be found on the ISO website.

Any feedback or questions on this specification should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

The advent of high-throughput sequencing (HTS) technologies has the potential to boost the adoption of genomic information in everyday practice, ranging from biological research to personalized genomic medicine in clinics. As a consequence, the volume of generated data has increased dramatically during the last few years, and an even more pronounced growth is expected in the near future.

At the moment genomic information is mostly exchanged through a variety of data formats, such as FASTA/FASTQ for unaligned sequencing reads and SAM/BAM/CRAM for aligned reads. With respect to such formats, the ISO/IEC 23092 series provides a new solution for the representation and compression of genome sequencing information by:

— Specifying an abstract representation of the sequencing data rather than a specific format with its direct implementation.

— Being designed at a time point when technologies and use cases are more mature. This permits addressing one limitation of the textual SAM format, for which the incremental ad-hoc addition of features followed along the years, resulting in an overall redundant and suboptimal format which was unnecessarily complicated.

— Separating free-field user-defined information with no clear semantics from the genomic data representation. This allows a fully interoperable and automatic exchange of information between different data producers.

— Allowing multiplexing of relevant metadata information with the data since data and metadata are partitioned at different conceptual levels.

— Following a strict and supervised development process which has proven successful in the last 30 years in the domain of digital media for the transport format, the file format, the compressed representation and the application program interfaces.

The ISO/IEC 23092 series provides the enabling technology that will allow the community to create an ecosystem of novel, interoperable, solutions in the field of genomic information processing. In particular it offers:

— Consistent, general and properly designed format definitions and data structures to store sequencing and alignment information. A robust framework which can be used as a foundation to implement different compression algorithms.

— Speed and flexibility in the selective access to coded data, by means of newly designed data clustering and optimized storage methodologies.

— Low latency in data transmission and consequent fast availability at remote locations, based on transmission protocols inspired by real-time application domains.

— Built-in privacy and protection of sensitive information, thanks to a flexible framework which allows customizable secured access at all layers of the data hierarchy.

— Reliability of the technology and interoperability among tools and systems, owing to the provision of a procedure to assess conformance to this document on an exhaustive dataset.

— Support to the implementation of a complete ecosystem of compliant devices and applications, through the availability of a normative reference implementation covering the totality of the ISO/IEC 23092 series.

The fundamental structure of the ISO/IEC 23092 series data representation is the *genomic record*. The genomic record is a data structure consisting of either a single sequencing read, or a paired sequencing read, and its associated sequencing and alignment information; it may contain detailed mapping and alignment data, a single or paired read identifier (read name) and quality values.

Without breaking traditional approaches, the genomic record introduced in the ISO/IEC 23092 series provides a more compact, simpler and manageable data structure grouping all the information related to a single DNA template, from simple sequencing data to sophisticated alignment information.

The genomic record, although it is an appropriate logic data structure for interaction and manipulation of coded information, is not a suitable atomic data structure for compression. To achieve high compression ratios, it is necessary to group genomic records into clusters and to transform the information of the same type into sets of descriptors structured into homogeneous blocks. Furthermore, when dealing with selective data access, the genomic record unit is too small to allow effective and fast information retrieval.

For these reasons, this document introduces the concept of access unit, which is the fundamental structure for coding and access to information in the compressed domain.

The access unit is the smallest data structure that can be decoded by a decoder compliant with ISO/IEC 23092-2. An access unit is composed of one block for each descriptor used to represent the information of its genomic records; therefore, a block payload is the coded representation of all the data of the same type (i.e. a descriptor) in a cluster.

In addition to clusters of genomic records compressed into access units, reads are further classified in six data classes: five classes are defined according to the result of their alignment against one or more reference sequences; the sixth class contains either reads that could not be mapped or raw sequencing data. The classification of sequencing reads into classes enables the development of powerful selective data access. In fact access units inherit a specific data characterization (e.g. perfect matches in class P, substitutions in class M, indels in class I, half-mapped reads in class HM) from the genomic records composing them, and thus constitute a data structure capable of providing powerful filtering capability for the efficient support of many different use cases.

Access units are the fundamental, finest grain data structure in terms of content protection and in terms of metadata association. In other words each access unit can be protected individually and independently. Figure 1 shows how access units, blocks and genomic records relate to each other in the ISO/IEC 23092 series data structure.

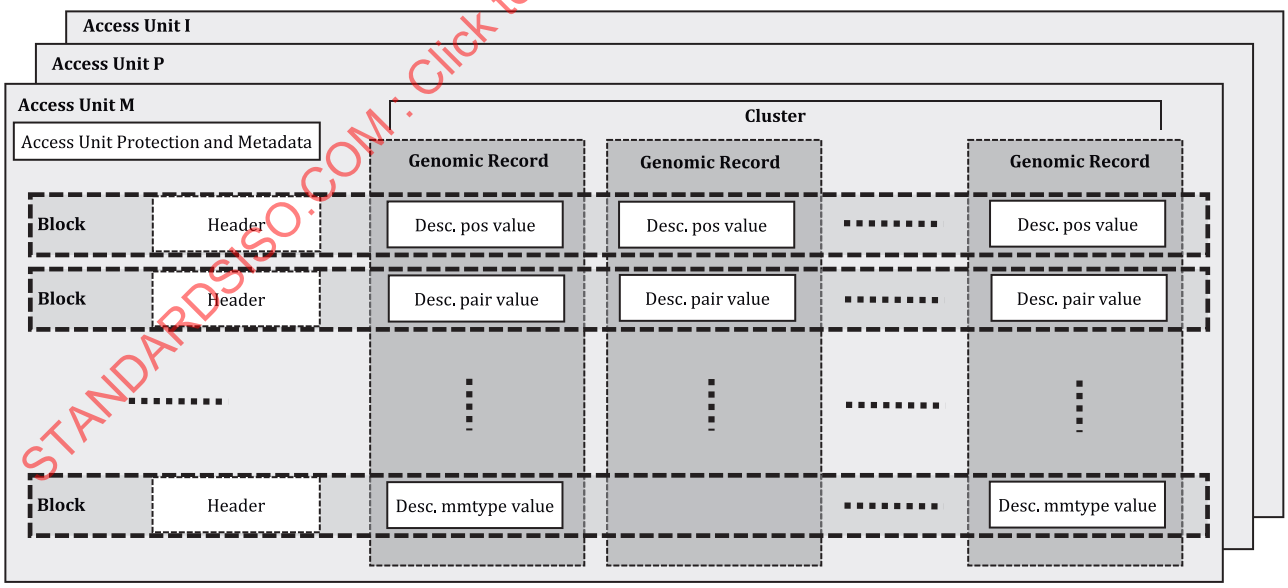

Figure 1 — Access units, blocks and genomic records

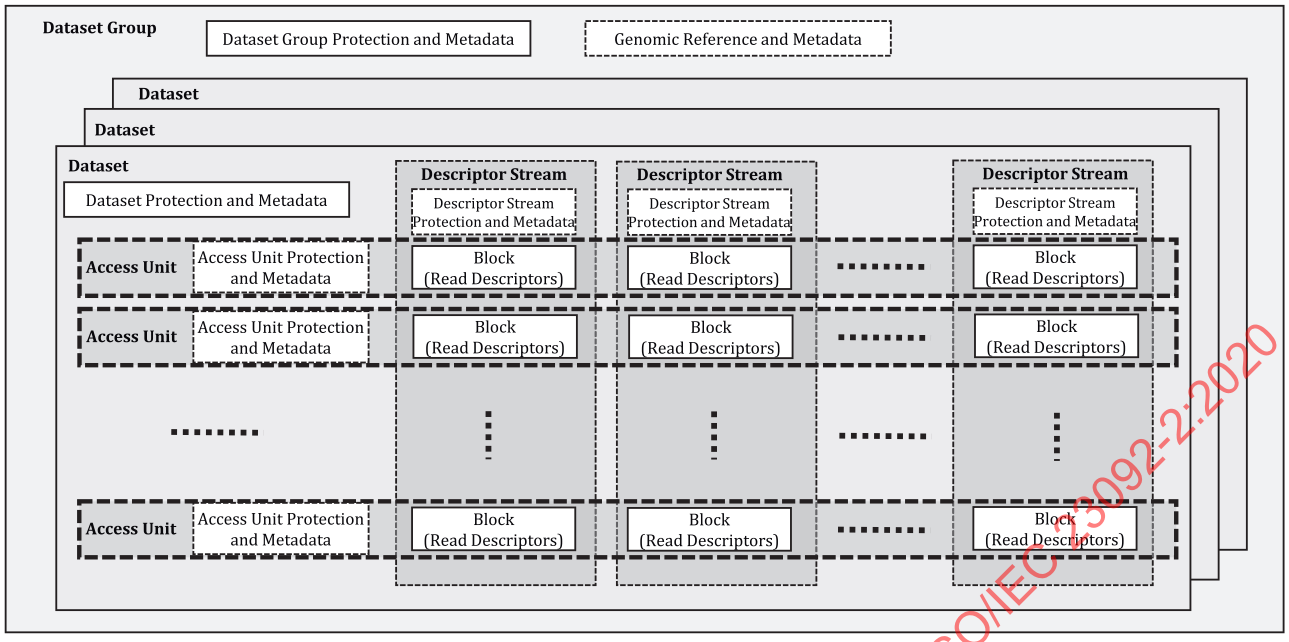

Figure 2 — High-level data structure: datasets and dataset group

A dataset is a coded data structure containing headers and one or more access units. Typical datasets could, for example, contain the complete sequencing of an individual, or a portion of it. Other datasets could contain for example a reference genome or a subset of its chromosomes. Datasets are grouped in dataset groups, as shown in Figure 2.

According to the ISO/IEC 23092 series, the compressed sequencing data can be multiplexed into a bitstream suitable for packetization for real-time transport over typical network protocols. In storage use cases, coded data can be encapsulated into a file format with the possibility to organize blocks per descriptor stream or per access unit, to further optimize the selective access performance to the type of data access required by the different application scenarios. The ISO/IEC 23092 series further provides a reference process to convert a transport stream into a file format and vice versa.

The ISO/IEC 23092 series defines the syntax and semantics of the compressed genome sequencing data representation and the deterministic decoding process that reconstructs the contents of datasets. The decoding process is fully specified such that all decoders that conform to this document will produce identical decoded output. A simplified diagram of the decoding process is shown in Figure 3.

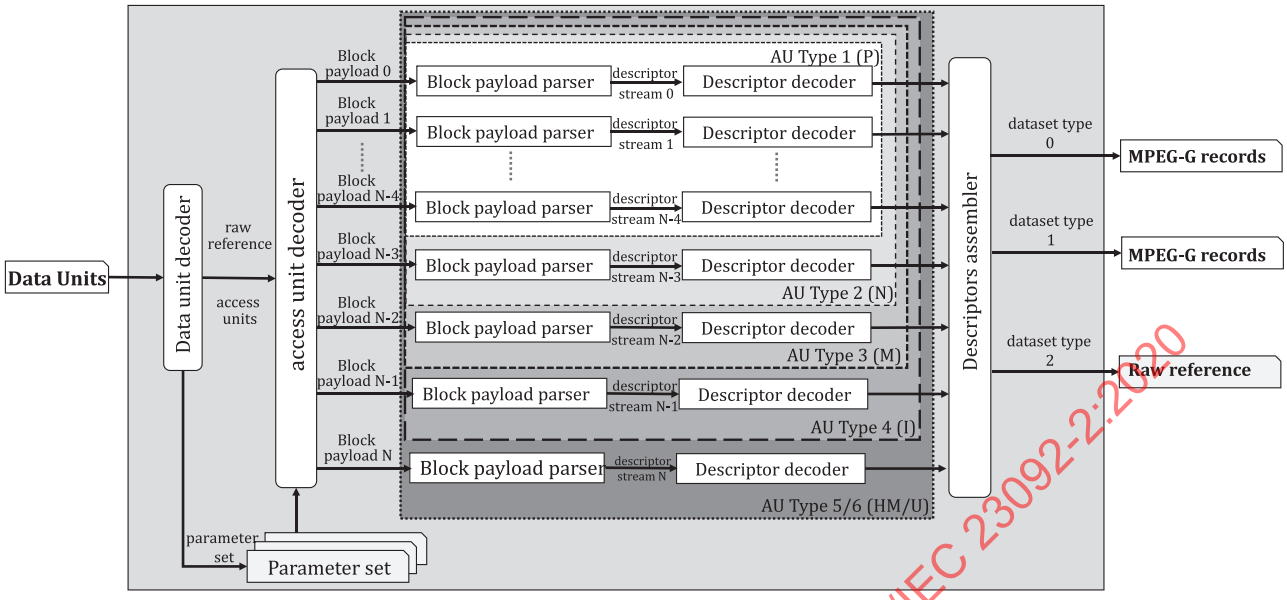

Figure 3 — The decoding process

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured ISO and IEC that he/she is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO and IEC. Information may be obtained from the patent database available at www.iso.org/patents.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those in the patent database. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Information technology — Genomic information representation —

Part 2: Coding of genomic information

1 Scope

This document provides specifications for the representation of the following types of genomic information:

— unaligned sequencing reads including read identifiers and quality values;

— aligned sequencing reads including read identifiers and quality values;

— reference sequences.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 10646, *Information technology — Universal Coded Character Set (UCS)*

ISO/IEC 23092-1:2020, *Information technology — Genomic information representation — Part 1: Transport and storage of genomic information*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 23092-1 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1
alignment
information describing the similarity between a sequence [typically a *sequencing read* (3.28)] and a reference sequence (for instance, a reference genome)

Note 1 to entry: An alignment is described in terms of a position within the reference, the strand of the reference, and a set of edit operations (matches, mismatches, insertions and deletions, clipping of the sequence ends and splicing information) needed to turn the first sequence into the second.

3.2
CIGAR string
CIGAR
textual way of representing an *alignment* (3.1)

Note 1 to entry: Several definitions have been used by different programs; the one referred to here is the one used in the SAM format. It encodes a set of edit operations (matches, mismatches, insertions and deletions, clipping of the sequence ends and splicing information) needed to turn the sequencing read into the reference.

3.3
dataset
compression unit containing one or more of: reference sequences; *sequencing reads* (3.28); and *alignment* (3.1) information

Note 1 to entry: Datasets shall be as specified in ISO/IEC 23092-1.

3.4
deletion
contiguous removal of one or more bases from a genomic sequence

3.5
E-CIGAR
extended CIGAR syntax specified as a superset of the CIGAR syntax

Note 1 to entry: Among other things, E-CIGAR enables the unambiguous representation of substitutions, spliced reads and splice strandedness.

3.6
edit operation
modification of a sequence of *nucleotides* (3.20) by means of a substitution, *deletion* (3.4), *insertion* (3.18) or clip

3.7
FASTA
GIR that includes a name and a *nucleotide* (3.20) sequence for each *sequencing read* (3.28)

Note 1 to entry: Additional information is usually encoded in the read identifier by bioinformatics tools (such as database information, and base calling information).

3.8
FASTQ
GIR that includes *FASTA* (3.7) and *quality values* (3.22)

3.9
first end
end 1
read 1
first segment of a paired-end *template* (3.33)

Note 1 to entry: Illumina platforms usually store first and second ends in two separate files and in the same order — i.e. the n-th read of the first FASTQ file and the n-th read of the second FASTQ file belong to the same template.

3.10
genomic descriptor
descriptor
element of the syntax used to represent a feature of a genomic *sequencing read* (3.28) or associated information such as *alignment* (3.1) information or *quality values* (3.22)

3.11
genomic information representation
way to describe a sequence and some information associated with it

Note 1 to entry: Which information is represented varies depending on the GIR.

3.12
genomic record
record
data structure representing a *tuple* (3.34) optionally associated with *alignment* (3.1) information, *read identifier* (3.24) and *quality values* (3.22)

3.13
genomic record index
position of a genomic record in the sequence of *genomic records* (3.12) encoded in an access unit

3.14
genomic record position
0-based position of the leftmost mapped base on the reference genome of the first *alignment* (3.1) contained in a *genomic record* (3.12)

Note 1 to entry: A base present in the aligned read and not present in the reference sequence (insertion) and bases preserved by the alignment process but not mapped on the reference sequence (soft clips) do not have mapping positions.

3.15
genomic reference
reference
collection of reference sequences

Note 1 to entry: Typical examples are a reference genome or a reference transcriptome.

3.16
hard clip
base or set of bases originally present at either side of a read, and removed from it following *alignment* (3.1)

Note 1 to entry: The bases are no longer present in the sequence of the read.

3.17
indel
contiguous stretch of *nucleotides* (3.20) that, when aligning two sequences, are inserted into one sequence, or alternatively deleted from the other, in order to make the two sequences the same

Note 1 to entry: From “insertion or deletion”.

3.18
insertion
contiguous addition of one or more bases into a genomic sequence

3.19
leftmost read end
leftmost read
sequencing read (3.28) generated by a paired-end sequencing run and mapped at a position on the reference sequence which is smaller than the mapping position of the other read in the pair

3.20
nucleotide
base
base pair
monomer of a nucleic acid polymer such as DNA or RNA

Note 1 to entry: Nucleotides are denoted as letters ('A' for adenine; 'C' for cytosine; 'G' for guanine; 'T' for thymine which only occurs in DNA; and 'U' for uracil which only occurs in RNA). The chemical formula for a specific DNA or RNA molecule is given by the sequence of its nucleotides, which can be represented as a string over the alphabet ('A', 'C', 'G', 'T') in the case of DNA, and a string over the alphabet ('A', 'C', 'G', 'U') in the case of RNA. Bases with unknown molecular composition are denoted with 'N'.

3.21
paired-end read
paired-end template
tuple (3.34) made of two segments

Note 1 to entry: Typically the segments correspond to the beginning and the end of the same nucleic acid molecule.

3.22
quality value
quality score
number assigned to each *nucleotide* (3.20) base call in automated sequencing processes

Note 1 to entry: Quality values express the base-call accuracy, i.e. the probability (or a related measure) for a nucleotide in the sequence to have been incorrectly determined.

3.23
read group
set of reads having some property in common

3.24
read identifier
read header
read name
text string associated with each *sequencing read* (3.28) stored in GIRs such as *FASTA* (3.7), *FASTQ* (3.8) and *SAM* (3.26)

Note 1 to entry: The read identifier is usually unique within its dataset, and may contain additional information as encoded by bioinformatics tools (such as database information, and base calling information).

3.25
rightmost read end
rightmost read
sequencing read (3.28) generated by a paired-end sequencing run and mapped at a position on the reference sequence which is greater than the mapping position of the other read in the pair

3.26
SAM
GIR that is human readable and includes FASTQ plus *alignment* (3.1) and analysis information

Note 1 to entry: From "Sequence Alignment/Map format". SAM originates from the 1000 Genome Sequencing Project. It is represented in plain ASCII, extensible by users and includes sequence, quality, alignment and analysis information.

3.27
second end
read 2
second segment of a paired-end *template* (3.33)

Note 1 to entry: Sequencing platforms usually store first and second ends in two separate files and in the same order — i.e. the n-th read of the first FASTQ file and the n-th read of the second FASTQ file belong to the same template.

3.28
sequencing read
read
readout, by a specific technology more or less prone to errors, of a continuous part of a segment of *nucleotides* (3.20) extracted from an organic sample

3.29
single-end read
tuple (3.34) made of one segment

3.30
soft clip
soft clipped bases
base or set of bases at either side of the read that have been ignored during the *alignment* (3.1) process

Note 1 to entry: The bases are still present in the sequence of the read.

3.31
spliced read
aligned read which, as a consequence of biological splicing, covers non-continuous portions of the reference genome being the result of biological splicing

Note 1 to entry: This means the read must come from RNA-sequencing, and contain at least one junction between two consecutive exons.

3.32
split alignment
aligned *paired-end read* (3.21) whose ends are encoded in two different *genomic records* (3.12)

3.33
template
genomic sequence that is produced by a sequencing machine as a single unit

Note 1 to entry: A template can be made of one or more segments (being called single-end sequencing read when it only has one segment, and paired-end sequencing read when it has two segments — typically they capture both the beginning and the end of a nucleic acid molecule).

3.34
tuple
collection of one or more segments

Note 1 to entry: Each segment can be: unmapped; mapped once; or mapped more than once.

3.35
decoded genomic descriptor
result of multiplexing the *decoded symbols* (3.37) of one or more *descriptor subsequences* (3.36)

3.36
descriptor subsequence
ordered collection of *decoded symbols* (3.37)

3.37
decoded symbol
value needed to reconstruct a *descriptor subsequence* (3.36)

Note 1 to entry: If no inverse subsequence transformation is applied, the transformed symbol shall be equal to the decoded symbol.

3.38
transformed subsequence
ordered collection of *transformed symbols* (3.39)

Note 1 to entry: The transformed symbols of one or more transformed subsequences can be multiplexed to yield decoded symbols.

3.39
transformed symbol
concatenation of one or more *decoded subsymbols* (3.40)

3.40
decoded subsymbol
output of an inverse subsymbol transformation applied on a *transformed subsymbol* (3.41)

Note 1 to entry: See subclause 12.6.2.7. If no inverse subsymbol transformation is applied, the decoded subsymbol shall be equal to the transformed subsymbol.

3.41
transformed subsymbol
decoded cabac subsymbol
atomic value yielded by the cabac decoding process

4 Abbreviated terms

AU	access unit
CRPS	computed reference parameters set
GIR	genomic information representation
LUT	look up table
QVPS	quality values parameters set

5 Conventions

5.1 General

This clause contains the definition of operators, notations, functions, textual conventions and processes used throughout this document.

The mathematical operators used in this document are similar to those used in the C programming language. However, the results of integer division and arithmetic shift operations are specified more precisely, and additional operations are specified, such as exponentiation and real-valued division. Numbering and counting conventions generally begin from 0, e.g., "the first" is equivalent to the 0-th, "the second" is equivalent to the 1-th, etc.

5.2 Arithmetic operators

+	addition
−	subtraction (as a two-argument operator) or negation (as a unary prefix operator)
*	multiplication, including matrix multiplication
x^y	exponentiation Specifies x to the power of y. In other contexts, such notation is used for superscripting not intended for interpretation as exponentiation.
/	integer division with truncation of the result toward zero For example, 7 / 4 and −7 / −4 are truncated to 1 and −7 / 4 and 7 / −4 are truncated to −1.
÷	division in mathematical equations where no truncation or rounding is intended
$\frac{x}{y}$	division in mathematical equations where no truncation or rounding is intended
$\sum_{i=x}^{y} f(i)$	summation of f(i) with i taking all integer values from x up to and including y
x % y	modulus Remainder of x divided by y, defined only for integers x and y with x ≥ 0 and y > 0.

5.3 Logical operators

x && y	Boolean logical AND of x and y
x || y	Boolean logical OR of x and y
!	Boolean logical NOT
x ? y : z	if x is TRUE or not equal to 0, evaluates to the value of y; otherwise, evaluates to the value of z

5.4 Relational operators

>	greater than
≥	greater than or equal to
<	less than
≤	less than or equal to
==	equal to
!=	not equal to

When a relational operator is applied to a syntax element or variable that has been assigned the value "na" (not applicable), the value "na" is treated as a distinct value for the syntax element or variable. The value "na" is considered not to be equal to any other value.

5.5 Bit-wise operators

& AND
When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

| OR
When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

^ exclusive or
When operating on integer arguments, operates on a two's complement representation of the integer value. When operating on a binary argument that contains fewer bits than another argument, the shorter argument is extended by adding more significant bits equal to 0.

x >> y right shift of a two's complement integer representation of x by y binary digits This function is defined only for non-negative integer values of y. Bits shifted into the MSBs as a result of the right shift have a value equal to the MSB of x prior to the shift operation.

x << y left shift of a two's complement integer representation of x by y binary digits
This function is defined only for non-negative integer values of y. Bits shifted into the LSBs as a result of the left shift have a value equal to 0.

! not operator returning 1 if applied to 0 and 0 if applied to 1

5.6 Assignment operators

= assignment operator

++ increment
i.e., *x++* is equivalent to *x = x* + 1; when used in an array index, evaluates to the value of the variable prior to the increment operation.

− − decrement
i.e., x− − is equivalent to x = x − 1; when used in an array index, evaluates to the value of the variable prior to the decrement operation.

+= increment by amount specified
i.e., x += 3 is equivalent to x = x + 3, and x += (−3) is equivalent to x = x + (−3).

−= decrement by amount specified
i.e., x −= 3 is equivalent to x = x − 3, and x −= (−3) is equivalent to x = x − (−3).

5.7 Range notation

x = y..z x takes on integer values starting from y to z, inclusive, with x, y, and z being integer numbers and z being greater than y

array[x, y] sub-array containing the elements of array comprised between position x and y included
If x is greater than y, the resulting sub-array is empty.

5.8 Mathematical functions

Ceil(x) smallest integer greater than or equal to x (1)

Floor(x) largest integer less than or equal to x (2)

Log2(x) base-2 logarithm of x (3)

$$\text{Min}(x, y) = \begin{cases} x & ; \quad x <= y \\ y & ; \quad x > y \end{cases} \tag{4}$$

$$\text{Max}(x, y) = \begin{cases} x & ; \quad x >= y \\ y & ; \quad x < y \end{cases} \tag{5}$$

5.9 Order of operation precedence

When the order of precedence in an expression is not indicated explicitly by use of parentheses, the following rules apply:

— Operations of a higher precedence are evaluated before any operation of a lower precedence.

— Operations of the same precedence are evaluated sequentially from left to right.

Table 1 specifies the precedence of operations from highest to lowest; a higher position in the table indicates a higher precedence.

NOTE For those operators that are also used in the C programming language, the order of precedence used in this document is the same as used in the C programming language.

Table 1 — Operation precedence from highest (at top of table) to lowest (at bottom of table)

operations (with operands x, y, and z)
"x++", "x– –"
"!x", "–x" (as a unary prefix operator)
x^y
"x * y", "x / y", "x ÷ y", "$\frac{x}{y}$", "x % y"
"x + y", "x – y" (as a two-argument operator), "$\sum_{i=x}^{y} f(i)$"
"x << y", "x >> y"
"x < y", "x <= y", "x > y", "x ≥ y"
"x = = y", "x != y"
"x & y"
"x | y"
"x && y"
"x | | y"
"x ? y : z"
"x..y"
"x = y", "x += y", "x –= y"

5.10 Variables, syntax elements and tables

Syntax elements in the bitstream are represented in **bold** type. Each syntax element is described by its name (all lower case letters with underscore characters), and one data type for its method of coded representation. The decoding process behaves according to the value of the syntax element and to the values of previously decoded syntax elements. When a value of a syntax element is used in the syntax tables or the text, it appears in regular (i.e., not bold) type.

In some cases the syntax tables may use the values of other variables derived from syntax elements values. Such variables appear in the syntax tables, or text, named by a mixture of lower case and upper case letter and without any underscore characters (camel case notation). Variables starting with an upper case letter are derived for the decoding of the current syntax structure and all depending syntax structures. Variables starting with an upper case letter may be used in the decoding process for later syntax structures without mentioning the originating syntax structure of the variable. Variables starting with a lower case letter are only used within the clause in which they are derived.

In some cases, "mnemonic" names for syntax element values or variable values are used interchangeably with their numerical values. Sometimes "mnemonic" names are used without any associated numerical values. The association of values and names is specified in the text. The names are constructed from one or more groups of letters separated by an underscore character. Each group starts with an upper case letter and may contain more upper case letters.

NOTE The syntax is described in a manner that closely follows the C-language syntactic constructs.

Functions that specify properties of the current position in the bitstream are referred to as syntax functions. These functions are specified in Clause 6 and assume the existence of a bitstream pointer with an indication of the position of the next bit to be read by the decoding process from the bitstream. Syntax functions are described by their names, which are constructed as syntax element names and end with left and right round parentheses including zero or more variable names (for definition) or values (for usage), separated by commas (if more than one variable).

Functions that are not syntax functions (including mathematical functions specified in subclause 5.2) are described by their names, which start with an upper case letter, contain a mixture of lower and upper case letters without any underscore character, and end with left and right parentheses including zero or more variable names (for definition) or values (for usage) separated by commas (if more than one variable).

A one-dimensional array is referred to as a list. A two-dimensional array is referred to as a matrix. Arrays can either be syntax elements or variables. Subscripts or square parentheses are used for the indexing of arrays. In reference to a visual depiction of a matrix, the first subscript is used as a row (vertical) index and the second subscript is used as a column (horizontal) index. The indexing order is reversed when using square parentheses rather than subscripts for indexing. Thus, an element of a matrix s at horizontal position x and vertical position y may be denoted either as s[x][y] or as s_{yx}. A single column of a matrix may be referred to as a list and denoted by omission of the row index. Thus, the column of a matrix s at horizontal position x may be referred to as the list s[x].

A specification of values of the entries in rows and columns of an array may be denoted by { {...} {...} }, where each inner pair of brackets specifies the values of the elements within a row in increasing column order and the rows are ordered in increasing row order. Thus, setting a matrix s equal to { { 1 6 } { 4 9 } } specifies that s[0][0] is set equal to 1, s[1][0] is set equal to 6, s[0][1] is set equal to 4, and s[1][1] is set equal to 9.

Binary notation is indicated by enclosing the string of bit values by single quote marks. For example, '01000001' represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.

Hexadecimal notation, indicated by prefixing the hexadecimal number by "0x", may be used instead of binary notation when the number of bits is an integer multiple of 4. For example, 0x41 represents an eight-bit string having only its second and its last bits (counted from the most to the least significant bit) equal to 1.

Numerical values not enclosed in single quotes and not prefixed by "0x" are decimal values.

A value equal to 0 represents a FALSE condition in a test statement. The value TRUE is represented by any value different from zero.

5.11 Text description of logical operators

In the text, a statement of logical operations as would be described mathematically in the following form:

```
if( condition 0 )
    statement 0
else if( condition 1 )
    statement 1
...
else /* informative remark on remaining condition */
 statement n
```

may be described in the following manner:

... as follows / ... the following applies:

— If condition 0, statement 0

— Otherwise, if condition 1, statement 1

— ...

— Otherwise (informative remark on remaining condition), statement n

Each "If ... Otherwise, if ... Otherwise, ..." statement in the text is introduced with "... as follows" or "... the following applies" immediately followed by "If ... ". The last condition of the "If ... Otherwise, if ... Otherwise, ..." is always an "Otherwise, ...". Interleaved "If ... Otherwise, if ... Otherwise, ..." statements can be identified by matching "... as follows" or "... the following applies" with the ending "Otherwise, ...".

In the text, a statement of logical operations as would be described mathematically in the following form:

```
if( condition 0a && condition 0b )
    statement 0
else if( condition 1a || condition 1b )
    statement 1
...
else
    statement n
```

... as follows / ... the following applies:

— If all of the following conditions are true, statement 0:

  — condition 0a

  — condition 0b

— Otherwise, if one or more of the following conditions are true, statement 1:

  — condition 1a

  — condition 1b

— ...

— Otherwise, statement n

In the text, a statement of logical operations as would be described mathematically in the following form:

if(condition 0)
    statement 0
if(condition 1)
    statement 1

may be described in the following manner:

— When condition 0, statement 0

— When condition 1, statement 1

5.12 Processes

Processes are used to describe the decoding of syntax elements. A process has a separate specification and invoking. All syntax elements and variables that pertain to the current syntax structure and depending syntax structures are available in the process specification and invoking. A process specification may also have a lower-case variable explicitly specified as input. Each process specification has explicitly specified an output. The output is a variable that can either be an upper-case variable or a lower-case variable.

When invoking a process, the assignment of variables is specified as follows:

— If the variables at the invoking and the process specification do not have the same name, the variables are explicitly assigned to lower-case input or output variables of the process specification.

— Otherwise (the variables at the invoking and the process specification have the same name), assignment is implied.

In the specification of a process, a specific coding block may be referred to by the variable name having a value equal to the address of the specific coding block.

6 Syntax and semantics

6.1 Method of specifying syntax in tabular form

The syntax tables specify a superset of the syntax of all allowed bitstreams. Additional constraints on the syntax may be specified, either directly or indirectly, in other clauses.

Table 2 lists examples of the syntax specification format. When **syntax_element** appears, it specifies that a syntax element is parsed from the bitstream and the bitstream pointer is advanced to the next position beyond the syntax element in the bitstream parsing process.

Table 2 — Examples of the syntax specification format

Syntax	Type
/* A statement can be a syntax element with an associated data type or can be an expression used to specify conditions for the existence, type and quantity of syntax elements, as in the following two examples */	
syntax_element	ue(v)
conditioning statement	
/*A group of statements enclosed in curly brackets is a compound statement and is treated functionally as a single statement. */	

Table 2 *(continued)*

Syntax	Type
{	
Statement	
Statement	
...	
}	
/* A "while" structure specifies a test of whether a condition is true, and if true, specifies evaluation of a statement (or compound statement) repeatedly until the condition is no longer true */	
while(condition)	
statement	
/* A "do ... while" structure specifies evaluation of a statement once, followed by a test of whether a condition is true, and if true, specifies repeated evaluation of the statement until the condition is no longer true */	
do	
statement	
while(condition)	
/* An "if ... else" structure specifies a test of whether a condition is true and, if the condition is true, specifies evaluation of a primary statement, otherwise, specifies evaluation of an alternative statement. The "else" part of the structure and the associated alternative statement is omitted if no alternative statement evaluation is needed */	
if(condition)	
primary statement	
else	
alternative statement	
/* A "for" structure specifies evaluation of an initial statement, followed by a test of a condition, and if the condition is true, specifies repeated evaluation of a primary statement followed by a subsequent statement until the condition is no longer true. */	
for(initial statement; condition; subsequent statement)	
primary statement	

6.2 Bit ordering

For bit-oriented delivery, the bit order of syntax fields in the syntax tables is specified to start with the MSB and proceed to the LSB.

6.3 Specification of syntax functions and data types

The functions presented here are used in the syntactical description. These functions are expressed in terms of the value of a bitstream pointer that indicates the position of the next bit to be read by the decoding process from the bitstream.

byte_aligned( ) is specified as follows:

— If the current position in the bitstream is on a byte boundary, i.e. the next bit in the bitstream is the first bit in a byte, the return value of byte_aligned( ) is equal to TRUE.

— Otherwise, the return value of byte_aligned( ) is equal to FALSE.

read_bits(n) reads the next n bits from the bitstream and advances the bitstream pointer by n bit positions. When n is equal to 0, read_bits(n) is specified to return a value equal to 0 and to not advance the bitstream pointer.

decode_bit() decodes the next bit from the bitstream using either the arithmetic decoding engine (subclause 13.2.4) or read_bits(1), as determined by the decoding configuration.

Size(array_name[]) returns the number of elements contained in the array named array_name.

The following data types specify the parsing process of each syntax element:

— ae(v): context-adaptive arithmetic entropy-coded syntax element. The parsing process for this data type is specified in subclause 12.5.2.2.

— ae(t): context-adaptive arithmetic entropy-coded termination syntax. The parsing process for this data type is specified in subclause 12.5.2.5.

— f(n): fixed-pattern bit string using n bits written (from left to right) with the left bit first. The parsing process for this data type is specified by the return value of the function read_bits(n).

— i(n): signed integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this data type is specified by the return value of the function read_bits(n) interpreted as a two's complement integer representation with most significant bit written first.

— se(v): signed integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this data type is specified in subclause 12.2.4.2.

— st(v): null-terminated string encoded as universal coded character set (UCS) transmission format-8 (UTF-8) characters as specified in ISO/IEC 10646. The parsing process is specified as follows: st(v) reads and returns a series of bytes from the bitstream, beginning at the current position and continuing up to but not including the next byte that is equal to 0x00, and advances the bitstream pointer by (stringLength + 1) * 8 bit positions, where stringLength is equal to the number of bytes returned.

— u(n): unsigned integer using n bits. When n is "v" in the syntax table, the number of bits varies in a manner dependent on the value of other syntax elements. The parsing process for this data type is specified by the return value of the function read_bits(n) interpreted as a binary representation of an unsigned integer with most significant bit written first.

— ue(v): unsigned integer 0-th order Exp-Golomb-coded syntax element with the left bit first. The parsing process for this data type is specified in subclause 12.2.4.

— u7(v): variable sized unsigned integer computed by iteratively reading 8 bits, where the least significant 7 bits are interpreted as a binary representation of an unsigned integer v, with the most significant bit written first, and the 8th bit signaling if the iteration should stop. The parsing process for this data type is specified below:

```
v = 0
do {
  c = read_bits( 8 );
  v = (v << 7) | (c & 0x7f);
} while (c & 0x80)
```

— c(n): sequence of n ASCII characters as specified in ISO/IEC 10646.

6.4 Semantics

Semantics associated with the syntax structures and with the syntax elements within each structure are specified in a clause following the clause containing the syntax structures. When the semantics of a syntax element are specified using a table or a set of tables, any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this document.

7 Data structures

7.1 General

Subclause 7.2 specifies the structure of a data unit. A data unit is a data structure used as container for a raw reference structure, a parameter set structure or an access unit structure.

Subclause 7.3.2 specifies the structure of a raw reference.

Subclause 7.4 specifies the structure of a parameter set. A parameter set consists of a parent parameter set identifier, a parameter set identifier and encoding parameters as specified in subclause 7.4.1.

Subclause 7.5 specifies the structure of an access unit. An access unit consists of an access unit header, followed by one or more blocks. Table 19 in subclause 7.5.1.2 specifies the syntax for an access unit header.

Each block consists of a block header, as specified in subclause 7.5.1.3.2, followed by a block payload as specified in subclause 7.5.1.3.3.

7.2 Data unit

Table 3 — Data unit syntax

Syntax	Type
data_unit() {	
data_unit_type	u(8)
if (data_unit_type == 0) {	
data_unit_size	u(64)
raw_reference()	raw reference
}	
else if (data_unit_type == 1) {	
reserved	u(10)
data_unit_size	u(22)
parameter_set()	parameter set
}	
else if(data_unit_type == 2){	
reserved	u(3)
data_unit_size	u(29)
access_unit()	access unit
}	
else /*(data_unit_type > 2)*/{	
/*skip data unit*/	
}	
}	

data_unit_type specifies the type of data unit. Table 4 lists the values of data_unit_type and the associated data unit types.

Table 4 — Values of data_unit_type and associated data unit types

data_unit_type	Data unit type	Clause
0	raw reference	7.3
1	parameter set	7.4
2	access unit	7.5

data_unit_size is the total size in bytes of the data unit including the bytes used for **data_unit_type** and **data_unit_size**.

raw_reference() is a raw_reference structure as specified in subclause 7.3.

parameter_set() is a parameter_set structure as specified in subclause 7.4.

access_unit() is an access_unit structure as specified in subclause 7.5.

A conformant bitstream containing at least one data unit of type access unit shall contain at least one data unit of type parameter set.

7.3 Raw reference

7.3.1 General

This subclause specifies the data structure used to represent a raw reference. This structure shall be used to:

— deliver reference sequences to the decoder,

— return decoded reference sequences or part thereof from the decoder.

If a raw reference is required to decode access units, this raw reference shall be made available to the decoder prior to any other data unit.

7.3.2 Syntax and semantics

Table 5 — Raw reference syntax

Syntax	Type
raw_reference() {	
seq_count	u(16)
for (i=0; i<seq_count; i++){	
sequence_ID	u(16)
seq_start[sequence_ID]	u(40)
seq_end[sequence_ID]	u(40)
ref_sequence[sequence_ID]	c(seq_end – seq_start + 1)
}	
}	

seq_count is the number of reference sequences in the raw reference.

sequence_ID is reference sequence identifier. Each **sequence_ID** is unique and shall correspond to one **sequence_name** specified in ISO/IEC 23092-1:2020, 6.5.2.3.3.

seq_start[sequence_ID] is the coordinate, on the reference sequence identified by **sequence_ID,** of the first base present in the ref_sequence[] array.

seq_end[sequence_ID] is the coordinate, on the reference sequence identified by **sequence_ID,** of the last base present in the ref_sequence[] array.

ref_sequence[sequence_ID][i] is the i^{th} base in the reference sequence identified by **sequence_ID**.

7.4 Parameter set

7.4.1 Syntax and semantics

This subclause specifies the parameter set syntax and semantics.

Table 6 — Parameter set syntax

Syntax	Type
parameter_set() {	
parameter_set_ID	u(8)
parent_parameter_set_ID	u(8)
encoding_parameters()	
}	

parameter_set_ID is the unique identifier of the parameter set.

parent_parameter_set_ID is the unique identifier of an existing parameter set. Referencing an existing parameter set from another parameter set enables the generation of a hierarchy of parameter sets where the values of the encoding parameters of each element override the corresponding values of the parent node. If equal to parameter_set_ID, the parameter set is at the top level in the hierarchy.

encoding_parameters() are the encoding parameters as specified in subclause 7.4.2 of this document.

7.4.2 Encoding parameters

7.4.2.1 General

The encoding parameters are configuration parameters used during the decoding process.

Table 7 — Encoding parameters syntax

Syntax	Type
encoding_parameters() {	
dataset_type	u(4)
alphabet_ID	u(8)
read_length	u(24)
number_of_template_segments_minus1	u(2)
reserved	u(6)
max_au_data_unit_size	u(29)
pos_40_bits_flag	u(1)
qv_depth	u(3)
as_depth	u(3)
num_classes	u(4)

Table 7 *(continued)*

Syntax	Type
for(j=0; j < num_classes; j++)	This for loop specifies the order of data classes for the entire syntax structure.
class_ID[j]	u(4)
for(i=0; i < NUM_DESCRIPTORS; i++){	
class_specific_dec_cfg_flag[i]	u(1)
if(class_specific_dec_cfg_flag[i] == 0) {	
descriptor_configuration(i)	Descriptor configuration, as specified in subclause 7.4.2.2, applied to all classes.
} else {	
for(j=0; j< num_classes ; j++) {	
descriptor_configuration(i)	Descriptor configuration, as specified in 7.4.2.2, applied to the class identified by class_ID[j].
}	
}	
num_groups	u(16)
for(j=0; j < num_groups; j++)	
rgroup_ID[j]	st(v)
multiple_alignments_flag	u(1)
spliced_reads_flag	u(1)
reserved	u(30)
signature_flag	u(1)
if(signature_flag != 0){	
signature_constant_length_flag	u(1)
if(signature_constant_length_flag != 0){	
signature_length	u(8)
}	
}	
for (c = 0; c < num_classes; c++) {	
qv_coding_mode	u(4)
if(qv_coding_mode == 1){	
qvps_flag	u(1)
if(qvps_flag)	
parameter_set_qvps(class_ID[c])	See subclause 7.4.2.3.
else	
qvps_preset_ID	u(4)
}	
qv_reverse_flag	u(1)
}	
crps_flag	u(1)

Table 7 *(continued)*

Syntax	Type
`if(crps_flag)`	
`parameter_set_crps()`	See subclause 7.4.2.4.
`while( !byte_aligned( ) )`	
`nesting_zero_bit`	f(1)
}	

dataset_type specifies the type of data encoded in the dataset. The possible values are: 0 = non-aligned content; 1 = aligned content; 2 = reference.

alphabet_ID identifies the alphabet of symbols used for data encoded in access units referring to these encoding parameters. shows the alphabets associated to each value of **alphabet_ID**.

read_length specifies the length in bases of sequencing reads. The value 0 indicates the presence of variable read lengths. Variable read lengths are signalled genomic record as specified in subclause 10.4.9).

number_of_template_segments_minus_1 specifies the number of segments in each sequenced template. For single read sequencing it is set to 0, for paired-end sequencing it is set to 1. The variable numberOfTemplateSegments is set to **number_of_template_segments_minus_1** + 1.

max_au_data_unit_size is the maximum value permitted for the field data_unit_size in the data unit, when data_unit_type is equal to 2, as specified in subclause 7.2. A value of 0 indicates an unspecified maximum data unit size.

pos_40_bits_flag is set to 1 when the mapping positions are expressed as 40 bits integers. Otherwise all mapping positions are expressed as 32 bits integers. In the scope of this document the value of the variable posSize is set to 32 when pos_40_bits_flag is equal to 0 and set to 40 otherwise.

qv_depth specifies the number of quality values associated to each nucleotide. A value of 0 means that no quality values are encoded. The maximum value shall be 2.

as_depth specifies the number of alignment scores associated to each alignment. A value of 0 means that no alignment scores are encoded. The maximum value shall be 2.

num_classes specifies the number of data classes encoded in all access units referring to the current Parameters Set.

class_ID is one of the data class identifiers specified in subclause 9.5. For any value of ci greater than 0 it shall always be class_ID[ci] > class_ID[ci - 1].

NUM_DESCRIPTORS is a constant counting the number of genomic descriptors specified in this document and it is set to 18.

class_specific_dec_cfg_flag signals the presence of class-specific decoder configuration for a given desc_ID. If set to 0, only one decoder configuration is signalled for all classes. Otherwise, separate class specific decoder configurations are signalled.

descriptor_configuration(i) signals the descriptor's decoder configuration as specified in subclause 7.4.2.2.

num_groups specifies the number of read groups present in all access units referring to the current Parameters Set. If **num_groups** is set to 0, the **rgroup** descriptor shall not be present in the AUs referring to this parameter set.

rgroup_ID is the null-terminated string identifier of a read group. The maximum allowed length is 64 characters not including the terminating character.

multiple_alignments_flag is a flag signaling the presence of multiple alignments in the access unit. When set to 0 no multiple alignments are present.

spliced_reads_flag signals the presence of spliced reads in the access unit. When set to 0 no spliced reads are present.

reserved is set to 0 and reserved for future use.

signature_flag signals the presence of signatures in the access unit. When set to 0 no signatures are present.

signature_constant_length_flag signals if all signatures in an access unit have the same constant length.

signature_length specifies the length in bases of signatures when the **signature_constant_length_flag** is set to 1.

qv_coding_mode shall be set to 1, all other values are reserved.

qvps_flag signals the presence of a parameter_set_qvps(class_ID[c]) element.

qvps_preset_ID signals the ID of the quality values parameter set preset as specified in subclause 10.4.16.

parameter_set_qvps(class_ID[c]) is the quality values parameter set as specified in subclause 10.4.16. If not present, the parent quality values parameter set identified by **parent_parameter_set_ID** shall be used.

qv_reverse_flag signals if the decoded qv string shall be reversed in the decoding process specified in subclause 10.4.16.2.

crps_flag signals the presence of a parameter_set_crps() element.

parameter_set_crps() is the computed reference parameter set as specified in subclause 11.3. If not present, the computed reference parameters set of the parent parameter set identified by parent_parameter_set_ID shall be used.

nesting_zero_bit is one bit set to 0.

7.4.2.2 Descriptor configuration syntax and semantics

Table 8 — Descriptor configuration syntax

Syntax	Type
descriptor_configuration(desc_ID) {	
dec_cfg_preset	u(8)
if(dec_cfg_preset == 0){	
encoding_mode_ID	u(8)
if(desc_ID != 11 && desc_ID != 15)	
decoder_configuration(encoding_mode_ID)	As specified in 12.3.
else if(desc_ID == 11 || desc_ID == 15){	
decoder_configuration_tokentype(encoding_mode_ID)	As specified in 12.3.5.
}	
}	
else{	
/* reserved for future use */	
}	
}	

dec_cfg_preset shall be set to 0 to signal the presence of a decoder configuration.

encoding_mode_ID when set to 0 it signals the use of CABAC compression. Other values are reserved.

decoder_configuration(encoding_mode_ID) signals the decoder configuration parameters as specified in subclause 12.3.

decoder_configuration_tokentype(encoding_mode_ID) signals the decoder configuration parameters as specified in subclause 12.3.5.

7.4.2.3 Quality values parameter set syntax and semantics

7.4.2.3.1 General

Table 9 — Syntax of the quality values parameter set

Syntax	Type
parameter_set_qvps(class_id) {	
qv_num_codebooks_total	u(4)
for (b = 0; b < qv_num_codebooks_total; b++) {	
qv_num_codebook_entries[b]	u(8)
for (e = 0; e < qv_num_codebook_entries[b]; e++) {	
qv_recon[b][e]	u(8)
}	
}	
}	

qv_num_codebooks_total is the number of quality value codebooks. When qvps_flag is equal to 1, the minimum allowed value is 2 for class_id == Class_I or class_id == Class_HM. Otherwise , the minimum allowed value for all other classes is 1. For class_id == Class_U, this value shall be set to 1.

qv_num_codebook_entries[b] is the number of qv_recon elements in the quality value codebook identified by b. The minimum allowed value is 2 and the maximum allowed value is 94.

qv_recon[b][e] is the quality value reconstructed from a quality value index identified by e, using the quality value codebook identified by b.

qvNumCodebooksAligned is the state variable indicating the number of quality value codebooks used for aligned reads computed as specified in Table 10.

Table 10 — Computation of qvNumCodebooksAligned

```
if( class_id == Class_I || class_id == Class_HM) {
   /* For classes I and HM, the last codebook is reserved for unaligned data */
   qvNumCodebooksAligned = qv_num_codebooks_total - 1
} else if( class_id != Class_U ) { /* Classes P, N, M*/
   qvNumCodebooksAligned = qv_num_codebooks_total
} else { /* Class U */
   qvNumCodebooksAligned = 0
}
```

7.4.2.3.2 Quality values parameter set presets

This specification provides three quality values parameters presets, identified by qvps_preset_ID.

7.4.2.3.2.1 Support of all printable ASCII characters

This set of parameters (see Table 11) supports the representation of all printable ASCII characters. It is identified by qvps_preset_ID equal to 0.

Table 11 — Parameters for the support of all printable ASCII characters

Parameter name	Value
qv_num_codebooks_total	1
qv_num_codebook_entries	94

The reconstructed quality values **qv_recon**[0][i] are derived from quality value indexes i, with i being an integer number in the range 0..93, with the following expression:

qv_recon[0][i] = i + 33

7.4.2.3.2.2 Quantized quality values, offset 33, range 0-41

This set of parameters (see Table 12) supports the representation of quantized quality values in the range 0..41 with an offset equal to 33. It is identified by qvps_preset_ID equal to 1.

Table 12 — Parameters for quantized quality values, offset 33, range 0-41

Parameter name	Value
qv_num_codebooks_total	1
qv_num_codebook_entries	8

Table 13 shows how the reconstructed quality values qv_recon[0][] are derived from the quality value indexes.

Table 13 — Values of qv_recon for each value of entry when qvps_ID is equal to 1

i	qv_recon
0	33
1	41
2	46
3	51
4	56
5	61
6	66
7	74

7.4.2.3.2.3 Quantized quality values, offset 64, range 0-40

This set of parameters supports the representation of quantized quality values in the range 0..40 with an offset equal to 64. It is identified by qvps_preset_ID equal to 2.

Table 14 — Parameters for quantized quality values, offset 64, range 0-40

Parameter name	Value
qv_num_codebooks_total	1
qv_num_codebook_entries	8

Table 14 shows how the reconstructed quality values qv_recon[0][] are derived from the quality value indexes.

Table 15 — Values of qv_recon for each value of i when qvps_preset_ID is equal to 2

i	qv_recon[0][i]
0	64
1	72
2	77
3	82
4	87
5	92
6	97
7	104

7.4.2.4 Computed Reference parameter set

This subclause specifies the data structure used to carry parameters related to the reference computation algorithms specified in subclause 11.3.

Table 16 — Syntax of the computed reference parameter set

Syntax	Type
`parameter_set_crps() {`	
`  cr_alg_ID`	u(8)
`  if(cr_alg_ID == 2 || cr_alg_ID == 3){`	
`  cr_pad_size`	u(8)
`  cr_buf_max_size`	u(24)
`  }`	
`}`	

cr_alg_ID signals the reference computation algorithm as specified in subclause 11.3.4. The possible values for cr_alg_ID are listed in Table 17. The value 0 is reserved.

Table 17 — Values of cr_alg_ID and corresponding reference computation algorithms

cr_alg_ID	algorithm	
0		reserved
1	RefTransform	
2	PushIn	
3	Local Assembly	
4	Global Assembly	
5 ... 255		reserved

cr_pad_size is the number of bases used for padding in the process specified in subclause 11.3.4.

cr_buf_max_size is the maximum size in bytes of the buffer used in the decoding process as specified in subclause 11.3.

7.5 Access unit

An access unit (AU) is a logical data structure containing a coded representation of genomic information. It is the smallest data structure that can be decoded.

7.5.1 Syntax and semantics

7.5.1.1 General

This subclause specifies the access unit syntax (see Table 18) and semantics.

Table 18 — Access unit syntax

Syntax	Type
access_unit() {	
access_unit_header()	access unit header
for (i=0; i<num_blocks; i++) {	
block[i]()	block
}	
access_unit() {	

access_unit_header() is specified in subclause 7.5.1.2.

num_blocks specifies the number of blocks encoded in the access unit and it is encoded in the access_unit_header as specified in subclause 7.5.1.2.

block[i]() is a block as specified in subclause 7.5.1.3.

7.5.1.2 Access unit header

This subclause specifies the access unit header syntax and semantics.

Table 19 — Access unit header syntax

Syntax	Type
access_unit_header() {	
access_unit_ID	u(32)
num_blocks	u(8)
parameter_set_ID	u(8)
AU_type	u(4)
reads_count	u(32)
if(AU_type == N_TYPE_AU || AU_type == M_TYPE_AU){	
mm_threshold	u(16)
mm_count	u(32)
}	
if(dataset_type == 2){	
ref_sequence_ID	u(16)
ref_start_position	u(posSize)
ref_end_position	u(posSize)
}	
if (AU_Type != U_TYPE_AU)	
{	
sequence_ID	u(16)
AU_start_position	u(posSize)
AU_end_position	u(posSize)

Table 19 *(continued)*

Syntax	Type
if (multiple_alignments_flag) {	Specified in subclause 7.4.2.
extended_AU_start_position	u(posSize)
extended_AU_end_position	u(posSize)
}	
}	
else {	
if (signature_flag != 0) {	Specified in subclause 7.4.2.
num_signatures	u(16)
for (i=0; i< num_signatures; i++) {	
if(signature_constant_length_flag == 0){	
signature_length[i]	u(8)
}	
signature[i]	u(signatureSize)
}	
}	
}	
while(!byte_aligned( ))	
nesting_zero_bit	f(1)
}	

access_unit_ID is an unambiguous identifier for each AU_type, zero-based. If AU_type is not equal to U_TYPE_AU, it is encoded with respect to each reference sequence (identified by a specific value of sequence_ID), i.e., it is reset for the first access unit aligned on a specific reference sequence.

num_blocks specifies the number of Blocks in the access unit.

parameter_set_ID is a unique identifier of the parameter set to be used to decode the access unit to which this access unit header belongs. Decoding of an access unit is unspecified if at least one parameter in the hierarchy of parameter sets referred to by the field parameter_set_ID of the access unit and by the fields parent_parameter_set_ID of the parameter sets in the same hierarchy, as specified in subclause 7.4.1, set is not available.

AU_type identifies the type of access unit and the class of data carried therein as specified in subclause 7.5.2.

reads_count signals the number of genomic sequencing reads encoded in the access unit.

mm_threshold specifies the maximum number of substitutions a read (of class N or M) shall contain to be counted by **mm_count**. If set to 0 the feature of counting substitutions in encoded reads is disabled.

mm_count specifies the number of reads encoded in the access unit containing a number of substitutions which is equal to or lower than the threshold specified by **mm_threshold**. **mm_count** shall be set to 0 if the threshold is set to 0.

ref_sequence_ID specifies the identifier of the reference sequence encoded in this access unit.

ref_start_position specifies the position on the reference sequence of the first base encoded in this access unit.

ref_end_position specifies the position on the reference sequence of the last nucleotide encoded in this access unit.

sequence_ID is the identifier of the reference sequence to be used to decode this access unit as specified in clause 10. It corresponds to a **sequence_ID** element in Table 5.

AU_start_position is the position of the leftmost mapped base among the first alignments of all genomic records encoded in the access unit irrespective of the strand.

AU_end_position is the position of the rightmost mapped base among the first alignments of all genomic records encoded in the access unit irrespective of the strand.

extended_ AU_start_position specifies the position of the leftmost mapped base among all alignments of all genomic records contained in the access unit, irrespective of the strand.

extended_AU_end_position specifies the position of the rightmost mapped base among all alignments of all genomic records contained in the access unit, irrespective of the strand.

num_signatures specifies the number of signatures used to index unmapped reads as specified in ISO/IEC 23092-1.

signature_length specifies the signature length in terms of bases of a variable length signature.

signature is the unsigned integer representing the signature of the cluster this access unit belongs to, as specified in ISO/IEC 23092-1. The length in bits of this field, named signatureSize shall be calculated using the **signature_length** specified in Table 19 as follow:

signatureSize = signature_length * bits_per_symbol

with bits_per_symbol corresponding to BitsPerSymbol($S_{alphabet\_ID}$) as specified in Table 34 with alphabet_ID as specified in subclause 7.4.2, and with signature_length corresponding either to signature_length as specified in subclause 7.4.2 when signature_constant_length_flag (as specified in subclause 7.4.2) is equal to 1 or to the signature-specific signature_length[i] specified in Table 19 when signature_constant_length_flag (specified in subclause 7.4.2) is equal to 0.The j-th base in a signature is represented by the u(bits_per_symbol) value computed as follows:

signature_base[i][j] = $S_{alphabet\_ID}$[(signature[i] >> ((signature_length – j – 1) * bits_per_symbol))

& ((1 << bits_per_symbol) - 1)]

with $S_{alphabet\_ID}$ as specified in Table 34 with alphabet_ID as specified in subclause 7.4.2

posSize is specified in subclause 7.4.2.

7.5.1.3 Block

7.5.1.3.1 General

This subclause specifies the block syntax (see Table 20) and semantics.

Table 20 — Block syntax

Syntax	Type
`block() {`	
`  block_header()`	block header
`  block_payload()`	block payload
`}`	

block_header is a block header structure as specified in subclause 7.5.1.3.2.

block_payload is a block payload structure as specified in subclause 7.5.1.3.3.

7.5.1.3.2 Block header

This subclause describes the block header syntax (see Table 21) and semantics.

Table 21 — Block header syntax

Syntax	Type
block_header() {	
reserved	u(1)
descriptor_ID	u(7)
reserved	u(3)
block_payload_size	u(29)
}	

reserved bits used to preserve byte alignment.

descriptor_ID signals the descriptor type as specified in Table 24. Its value shall be unique among all blocks in the access unit.

block_payload_size specifies the size in bytes of the block payload.

7.5.1.3.3 Block payload

This subclause specifies the syntax (see Table 22) and semantics of the block payload structure containing entropy-coded descriptors.

Table 22 — Block payload syntax

Syntax	Type
block_payload(descriptor_ID) {	
if(descriptor_ID == 11 || descriptor_ID == 15){	
encoded_tokentype()	As specified in 10.4.20.2.
}	
else {	
encoded_descriptor_sequences(descriptor_ID)	As specified in 12.6.2.2.
}	
while(!byte_aligned( ))	
nesting_zero_bit	f(1)
}	

encoded_tokentype() is a data structure specified in subclause 10.4.20.2 carrying encoded tokenized strings.

encoded_descriptor_sequences(descriptor_ID) is a data structure specified in subclause 12.6.2.2 carrying the encoded genomic descriptors for sequences and quality values specified in Clause 8.

nesting_zero_bit is one bit set to 0.

7.5.2 Access unit types

AUs can be of different types according to the nature of the coded data. An access unit contains encoded genomic records belonging to a single data class as shown in Table 23.

Table 23 — Class of encoded data per access unit type

Access unit type		Data class
AU type name	Value	
P_TYPE_AU	1	Class P
N_TYPE_AU	2	Class N
M_TYPE_AU	3	Class M
I_TYPE_AU	4	Class I
HM_TYPE_AU	5	Class HM
U_TYPE_AU	6	Class U

The blocks of descriptors encoded in one access unit as specified in subclause 7.5.1.3 are those corresponding to sequencing reads belonging to one class of data as specified in subclause 9.5. Descriptors carried by each access unit type are listed in Table 24.

AUs of any class can be possibly associated with blocks of descriptors representing the read names and/or quality values of the encoded sequencing reads.

8 Descriptors

When dataset_type specified in subclause 7.2 is equal to 0 or 1, the only mandatory descriptors are those required to represent the sequences of nucleotides, whereas read names and quality values are optional.

Descriptors are the output of the decoding process specified in 10.4.

Descriptors required for the representation of sequencing reads, quality values, read names and transformed reference sequences are shown in Table 24. Descriptors are specified in subclause 10.4 and its subclauses.

Table 24 — Genomic descriptors

descriptor_ID	Genomic descriptor name	Number of descriptor subsequences	Decoding process
sequencing reads			
0	**pos**	2	10.4.2
1	**rcomp**	1	10.4.3
2	**flags**	3	10.4.4
3	**mmpos**	2	10.4.5
4	**mmtype**	3	10.4.6
5	**clips**	4	10.4.7
6	**ureads**	1	10.4.8
7	**rlen**	1	10.4.9
8	**pair**	8	10.4.10
9	**mscore**	1	10.4.11
10	**mmap**	5	10.4.12
11	**msar**	2	10.4.13
12	**rtype**	1	10.4.14
13	**rgroup**	1	10.4.15

Table 24 *(continued)*

descriptor_ID	Genomic descriptor name	Number of descriptor subsequences	Decoding process
quality values			
14	**qv**	Variable, as specified in subclause 10.4.16.	10.4.16
read names			
15	**rname**	2	10.4.17
reference sequences			
16	**rftp**	1	10.4.18
17	**rftt**	1	10.4.19

Table 25 — Subsequences for descriptor_ID = 0 (pos descriptor)

subsequence_ID	Semantics	Type
0	Mapping position of the first alignment.	Signed integer.
1	Mapping position of additional alignments.	Signed integer.

Table 26 — Subsequences for descriptor_ID = 2 (flags descriptor)

subsequence_ID	Semantics	Type
0	Read is PCR or optical duplicate.	Unsigned integer with value either 0 or 1.
1	Read fails platform/vendor quality checks.	Unsigned integer with value either 0 or 1.
2	Read mapped in proper pair	Unsigned integer with value either 0 or 1.

Table 27 — Subsequences for descriptor_ID = 3 (mmpos descriptor)

subsequence_ID	Semantics	Type
0	Terminator flag	Unsigned integer with value either 0 or 1.
1	Position value	Unsigned integer.

Table 28 — Subsequences for descriptor_ID = 4 (mmtype descriptor)

subsequence_ID	Semantics	Type
0	Symbol type flag	Unsigned integer with values either 0, 1 or 2.
1	Substitution type	Unsigned integer.
2	Insertions type	Unsigned integer.

Table 29 — Subsequences for descriptor_ID = 5 (clips descriptor)

subsequence_ID	Semantics	Type
0	Record identifier	Unsigned integer.
1	Type/Position flag	Unsigned integer as specified in subclause 10.4.7.
2	Nucleotides indexes with terminators	Unsigned integer as specified in Table 54.
3	Hard clips length	Unsigned integer.

Table 30 — Subsequences for descriptor_ID = 8 (pair descriptor)

subsequence_ID	Semantics	Type
0	Sequence identifying: — the subsequence carrying the next symbol required for the decoding process when values range from 0 to 4. Each value i in the range 0..4 corresponds to subsequence_ID = i + 1 — R1_unpaired decoding case as specified in 10.4.10 when the value is equal to 5. — R2_unpaired decoding case as specified in 10.4.10 when the value is equal to 6.	Unsigned integer.
1	same_rec decoding case as specified in 10.4.9. Sequence of values containing the segment ordering and the distance between the mapping position of read 1 and the mapping position of read 2 on the reference sequence. Encoded as '(delta << 1) | read1_first', where delta is comprised between 0 and 32767 and read1_first is a 1-bit flag.	Unsigned integer.
2	R1_split decoding case as specified in 10.4.10. Sequence of values representing: For classes P, N, M , I the position of read 1 on the reference sequence. The maximum value is $2^{posSize} - 1$ where posSize is specified in subclause 7.4.2. For class U the genomic record index of the genomic record containing read 1 in the current AU.	Unsigned integer.
3	R2_split decoding case as specified in 10.4.10. For classes P, N, M, I the position of read 2 on the reference sequence. The maximum value is $2^{posSize} - 1$ where posSize is specified in subclause 7.4.2. For class U the genomic record index of the genomic record containing read 2 in the current AU.	Unsigned integer.
4	R1_diff_ref_seq decoding case as specified in 10.4.10. Sequence of values representing: for classes P, N, M , I the identifier of the reference sequence to which read 1 is mapped. The maximum value is 2^{16}-1. for class U the identifier of the AU containing the read 1.	Unsigned integer.
5	R2_diff_ref_seq decoding case as specified in 10.4.10. for classes P, N, M , I the identifier of the reference sequence to which read 2 is mapped. The maximum value is 2^{16}-1. for class U the identifier of the AU containing the read 2.	Unsigned integer.
6	R1_diff_ref_seq decoding case as specified in 10.4.10. Sequence of values representing the position of read 1 on the reference sequence. The maximum value is is $2^{posSize} - 1$ where posSize is specified in subclause 7.4.2.	Unsigned integer.
7	R2_diff_ref_seq decoding case as specified in 10.4.10. Sequence of values representing the position of read 2 on the reference sequence. The maximum value is is $2^{posSize} - 1$ where posSize is specified in subclause 7.4.2.	Unsigned integer.

Table 31 — Subsequences for descriptor_ID = 10 (mmap descriptor)

subsequence_ID	Semantics	Type
0	Number of alignments of the leftmost and rightmost reads.	Unsigned integer
1	Index of right alignments.	Unsigned integer
2	Flag signalling the presence of more alignments in other genomic records.	Boolean flag
3	Values representing the identifier of the reference sequence a secondary alignment of the leftmost read is mapped to. The maximum value is 2^{16}-1.	Unsigned integer
4	Values representing a secondary alignment mapping position of the leftmost read on the reference sequence. The maximum value is is $2^{posSize}$ – 1 where posSize is specified in subclause 7.4.2.	Unsigned integer

Table 32 — Subsequences for descriptor_ID = 11 and 15 (msar and rname descriptors)

subsequence_ID	Semantics	Type
0	Output of decode_descriptor_subsequence() for CABAC_METHOD_0 as specified in subclause 10.4.20.4.5.	Unsigned integer
1	Output of decode_descriptor_subsequence()for CABAC_METHOD_1 as specified in subclause 10.4.20.4.6.	Unsigned integer

Table 33 — Subsequences for descriptor_ID = 14 (qv descriptor)

subsequence_ID	Semantics	Type
0	Quality value present flag.	Boolean flag.
1	Quality value codebook identifier.	Unsigned integer.
2 .. (2 + qv_num_codebooks_total – 1)	Quality value index used to look up a reconstructed quality value in the quality value codebook identified by b = (subsequence_ID - 2).	Unsigned integer.

9 Sequencing reads

9.1 General

This clause specifies the semantics of genomic descriptors used to represent nucleotides segments and associated alignment information. Each template produced by a sequencing machine or alignment generated by an aligner is encoded in a genomic record by means of a subset of the genomic descriptors described in this clause. The genomic descriptors are extracted from a compliant bitstream according to the processes described in subclause 12.6 and the genomic templates with the associated alignment information can be reconstructed from the decoded genomic descriptors according to the decoding processes described in subclause 10.4.

9.2 Supported symbols

The supported alphabets are specified in Table 34.

Table 34 — Identifiers of alphabets supported for sequencing reads representation

alphabet_ID	$S_{alphabet\_ID}$	Size($S_{alphabet\_ID}$)	BitsPerSymbol($S_{alphabet\_ID}$)
0	S_0 = [A, C, G, T, N]	5	3
1	S_1 = [A, C, G, T, R, Y, S, W, K, M, B, D, H, V, N, -]	16	5
2 .. 255	reserved		

Each alphabet is identified by an **alphabet_ID** as shown Table 34.

The notation $S_{alphabet\_ID}$[index] specifies a conversion from a numerical index to an ASCII character corresponding to a symbol of the alphabet identified by alphabet_ID, as specified in Table 35.

Table 35 — Conversions from numerical indexes to ASCII characters corresponding to alphabet symbols

$S_{alphabet\_ID}$[index]	S_0[index]	S_1[index]
$S_{alphabet\_ID}$[0]	S_0[0] = “A”	S_1[0] = “A”
$S_{alphabet\_ID}$[1]	S_0[1] = “C”	S_1[1] = “C”
$S_{alphabet\_ID}$[2]	S_0[2] = “G”	S_1[2] = “G”
$S_{alphabet\_ID}$[3]	S_0[3] = “T”	S_1[3] = “T”
$S_{alphabet\_ID}$[4]	S_0[4] = “N”	S_1[4] = “R”
$S_{alphabet\_ID}$[5]	N/A	S_1[5] = “Y”
$S_{alphabet\_ID}$[6]	N/A	S_1[6] = “S”
$S_{alphabet\_ID}$[7]	N/A	S_1[7] = “W”
$S_{alphabet\_ID}$[8]	N/A	S_1[8] = “K”
$S_{alphabet\_ID}$[9]	N/A	S_1[9] = “M”
$S_{alphabet\_ID}$[10]	N/A	S_1[10] = “B”
$S_{alphabet\_ID}$[11]	N/A	S_1[11] = “D”
$S_{alphabet\_ID}$[12]	N/A	S_1[12] = “H”
$S_{alphabet\_ID}$[13]	N/A	S_1[13] = “V”
$S_{alphabet\_ID}$[14]	N/A	S_1[14] = “N”
$S_{alphabet\_ID}$[15]	N/A	S_1[15] = “-”

The notation $Code_{alphabet\_ID}$[symbol] specifies the inversion conversion of $S_{alphabet\_ID}$[index], such that $Code_{alphabet\_ID}$[$S_{alphabet\_ID}$[index]] is always equal to index for any valid value of index as specified in Table 35.

Each alphabet symbol Sym is associated with a complementary symbol Complement(Sym) as specified in Table 36.

Table 36 — Complementary alphabet symbols

S_0[index]	S_0[Complement(index)]	S_1[index]	S_1[Complement(index)]
S_0[0] = “A”	S_0[3] = “T”	S_1[0] = “A”	S_1[3] = “T”
S_0[1] = “C”	S_0[2] = “G”	S_1[1] = “C”	S_1[2] = “G”
S_0[2] = “G”	S_0[1] = “C”	S_1[2] = “G”	S_1[1] = “C”
S_0[3] = “T”	S_0[0] = “A”	S_1[3] = “T”	S_1[0] = “A”
S_0[4] = “N”	S_0[4] = “N”	S_1[4] = “R”	S_1[5] = “Y”
N/A		S_1[5] = “Y”	S_1[4] = “R”
N/A		S_1[6] = “S”	S_1[6] = “S”

Table 36 *(continued)*

S_0[index]	S_0[Complement(index)]	S_1[index]	S_1[Complement(index)]
N/A		$S_1[7]$ = "W"	$S_1[7]$ = "W"
N/A		$S_1[8]$ = "K"	$S_1[9]$ = "M"
N/A		$S_1[9]$ = "M"	$S_1[8]$ = "K"
N/A		$S_1[10]$ = "B"	$S_1[13]$ = "V"
N/A		$S_1[11]$ = "D"	$S_1[12]$ = "H"
N/A		$S_1[12]$ = "H"	$S_1[11]$ = "D"
N/A		$S_1[13]$ = "V"	$S_1[10]$ = "B"
N/A		$S_1[14]$ = "N"	$S_1[14]$ = "N"
N/A		$S_1[15]$ = "-"	$S_1[15]$ = "-"

9.3 Paired-end reads

In case reads are generated in pairs by sequencing devices, each pair can be encoded as a single logical data structure named genomic record where the mapping position of one of the reads is represented using the **pair** descriptor as specified in subclause 10.4.10. The information linking one read to its mate is referred to as "pairing information" in this document.

The two reads are not sequenced from the same strand, but can be aligned to the same strand. The sequencing device determines which read in the pair is marked as read 1, whereas the other one will be read 2. An example is shown in Figure 4.

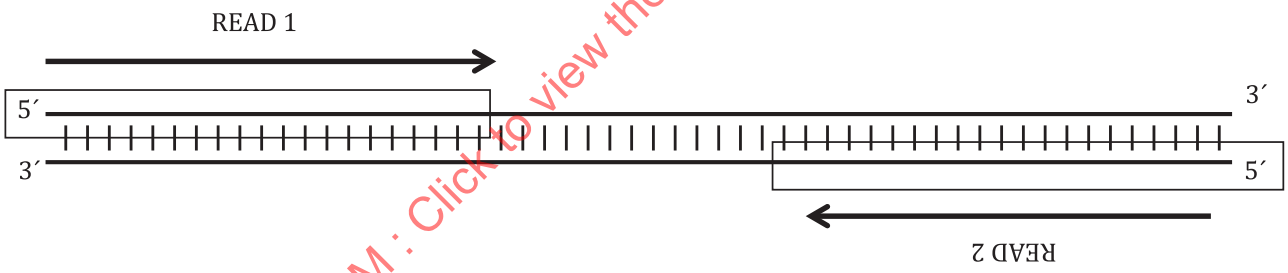

Figure 4 — Read 1 sequenced from the forward strand and read 2 from the reverse strand

Positions of mismatches with respect to the used reference sequence shall be encoded as offset from the leftmost mapped base of the leftmost read. The rightmost read is considered to be contiguous to the leftmost. The calculation of the actual position of mismatches on the rightmost read is described in subclause 10.4.10.

The pair can also be split into two reads that are encoded separately. In this case, the pair shall be reconstructed using both the pairing descriptors and the template name shared by the two reads.

9.4 Reverse-complement reads

The reverse-complement of a read is computed by inverting the order the read bases and replacing each base B with its complementary base Complement(B) as specified in subclause 9.2. If Read[] is the array of bases in a read, the array of bases in the corresponding reverse-complement ReverseComplementRead[] is specified as follows:

ReverseComplementRead[n] = Complement(Read[Size(Read[]) - n - 1]), for n in 0 .. Size(Read[]) - 1.

9.5 Data classes

Six data classes are specified to classify genomic records according to the result of the mapping of the encoded sequencing reads against one or more reference sequences.

If a template contains more than one read, if both reads are mapped, the genomic record belongs to the class of the read with the highest class_ID. In case of multiple aligments the genomic record belongs to the class of the first alignment in the record.

The data classes and their descriptions are specified in Table 37.

Table 37 — Sequence data classes

class_ID	Class Identifier	Genomic record content
1	Class_P	Only reads perfectly matching to the reference sequence.
2	Class_N	Reads perfectly matching to the reference sequence or containing mismatches which are unknown bases only.
3	Class_M	Reads perfectly matching to the reference sequence or containing substitutions or unknown bases, but no insertions, no deletions, no splices and no clipped bases.
4	Class_I	Reads perfectly matching to the reference sequence or containing substitutions, unknown bases, insertions, deletions, splices or clipped bases.
5	Class_HM	Paired-end reads with only one mapped read.
6	Class_U	Unmapped reads only.

When the syntax specified in this document needs to use the maximum number of specified data classes, this is specified by the constant **NUM_CLASSES** = 6.

9.6 Aligned data

In the context of this document, aligned genomic data are genomic segments which require the use of an external or embedded reference genome (as specified in subclause 10.6.2.3) to be decoded.

This subclause specifies the types of descriptors contained in the blocks payload specified in subclause 7.5.1.3.3. Each block contains binary coded descriptors of a single type identified by the descriptor_ID present in the block header as specified in subclause 7.5.1.3.2.

Once decoded, each descriptor shall be used to initialize one or more output record fields as specified in Clause 13. Table 38 lists the descriptors used for aligned reads with a brief description and reference to the corresponding clause.

Table 38 — Descriptors used to represent aligned sequencing reads

descriptor_ID	descriptor	Semantics	subclause
0	**pos**	Read mapping position.	10.4.2
1	**rcomp**	Strand information for reads in a template.	10.4.3
2	**flags**	Additional alignment information usually produced by aligners.	10.4.4
3	**mmpos**	Position of mismatches in reads.	10.4.5
4	**mmtype**	Type of mismatches.	10.4.6
5	**clips**	Information on clipped bases (i.e. Soft clips or hard clips).	10.4.7
6	**ureads**	Unmapped reads encoded verbatim.	10.4.8
7	**rlen**	Read lengths.	10.4.9

Table 38 *(continued)*

descriptor_ID	descriptor	Semantics	subclause
8	**pair**	Represents: 1.a The unsigned distance from one segment to the next. OR 1.b The absolute position on a reference sequence of a segment in a template. AND 2 Information signaling if the leftmost mapped read in the genomic record is read 1.	10.4.10
9	**mscore**	Provides a score per alignment .	10.4.11
10	**mmap**	Used to represent multiple alignments.	10.4.12
11	**msar**	Supports spliced alignments and alternative secondary alignments which do not preserve the same contiguity of mapping of the primary alignment.	10.4.13
13	**rgroup**	Identifier of the read group each genomic record belongs to.	10.4.15

9.7 Unaligned data

Unaligned reads belong to class U only. They are encoded as unmapped reads in aligned datasets. Some of the descriptors specified for reads aligned to an external or internal reference as specified in subclause 9.6 are used to encode unaligned reads (see Table 39). This is motivated by the fact that unaligned reads are encoded using reference sequences built from the data to be encoded. The reference used for mapping is computed according to the procedures described in subclause 11.3.

Table 39 — Descriptors used to represent raw sequencing reads

descriptor_ID	Descriptor	Semantics	Subclause
0	**pos**	Read mapping position.	10.4.2
1	**rcomp**	Strand information for reads in a template.	10.4.3
2	**flags**	Additional alignment information usually produced by aligners.	10.4.4
3	**mmpos**	Mismatch position.	10.4.5
4	**mmtype**	Type of edit operations: — substitutions; — deletions; — insertions.	10.4.6
5	**clips**	String of nucleotides with variable length (e.g. soft clips).	10.4.7
6	**ureads**	Unmapped reads encoded verbatim.	10.4.8
7	**rlen**	Unsigned integer representing the number of bases in the read minus one.	10.4.9

Table 39 *(continued)*

descriptor_ID	Descriptor	Semantics	Subclause
8	**pair**	Represents: 1.a The unsigned distance from one segment to the next. OR 1.b The absolute position on a computed reference sequence of a segment in a template. AND 2 Information signaling if the first read in the genomic record is read 1.	10.4.10
12	**rtype**	This identifies the subset of descriptors needed to decode the read.	10.4.11
13	**rgroup**	Identifier of the read group each genomic record belongs to.	10.4.15

10 Decoding process

10.1 General

This clause describes the decoding process to reconstruct the genomic information encoded in a bitstream compliant with this document.

The input to this process is one data unit. The output of this process can be:

1) a raw reference as specified in subclause 7.3.

2) a list of ISO/IEC 23092 series records as specified in Clause 13.

The decoding process is specified such that all decoders that conform to this document will produce numerically identical decoded output as either ISO/IEC 23092 seriesrecords or raw references. Any decoding process that produces identical decoded output ISO/IEC 23092 series records or raw references to those produced by the process described herein conforms to the decoding process requirements of this document.

10.2 dataset_type = 0 or 1

10.2.1 General

The input to the processes described in the following clauses is decoded genomic descriptors generated as output of the parsing process specified in subclause 11.3.6. The genomic descriptors are contained in the decoded_symbols data structure specified in this subclause.

In the context of the decoding process each decoded symbol is identified by

decoded_symbols[descriptor_ID][descriptor_subsequence_ID][$j_{descriptor\_ID,\ descriptor\_subsequence\_ID}$]

where $j_{descriptor\_ID,\ descriptor\_subsequence\_ID}$ is the index to read the decoded symbols as specified in subclause 12.3. The valid values of descriptor_ID are specified in Table 24. The values of descriptor_subsequence_ID are between 0 and the number of descriptor subsequences minus 1 as specified in Table 24.

At the beginning of the decoding process of one AU all indexes $j_{descriptor\_ID,\ descriptor\_subsequence\_ID}$ are initialized to 0.

The output of this process is a sequence of output records as specified in clause 13. If cr_alg_ID is equal to 3 and the **rftp** and **rftt** descriptors are present, an additional output of this process is a raw_reference$_{output}$ structure as specified in subclause 7.3.2.

The decoding process of each access unit refers to encoding parameters carried by the parameter set identified by the parameter_set_ID specified in subclause 7.5.1.2.

If dataset_type is equal to 0 then only AU of type 6 (CLASS_U) shall be present in the dataset.

10.2.2 References padding

In case of AUs of type P, N, M, I and HM, if the raw reference structure containing the reference sequence to be used during the decoding process specifies a seq_start that is greater than AU_start_pos or a seq_end that is less than AU_end_pos, the decoder shall pad the missing portions of reference sequence with "N". This is shown in Figure 5.

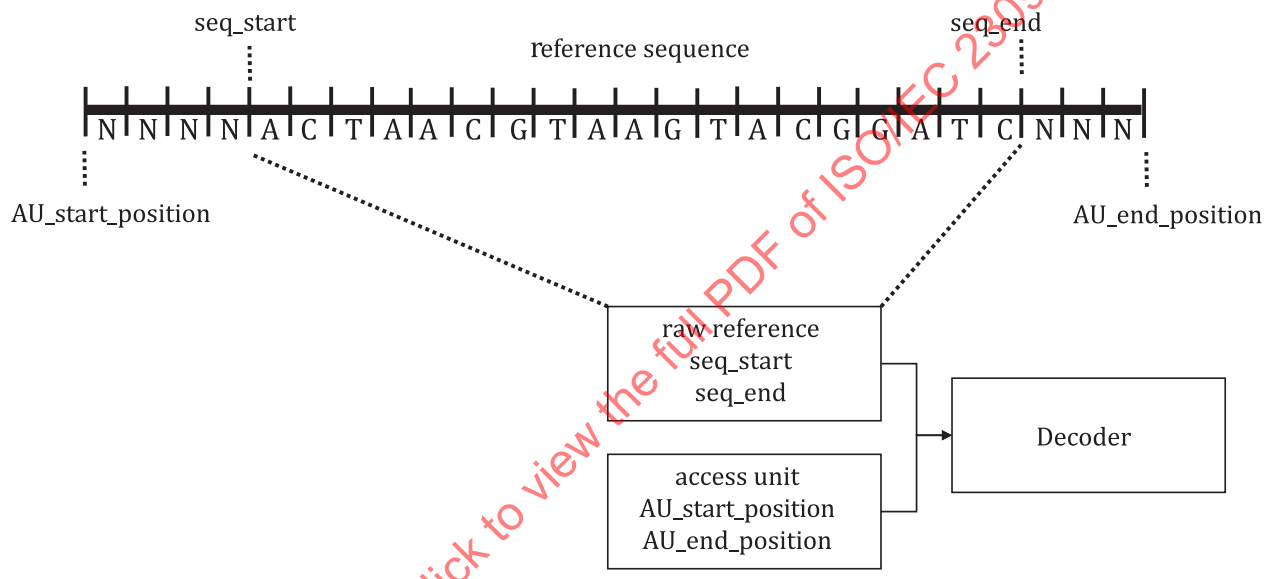

Figure 5 — Padding process for a reference sequence

10.2.3 Type 1 AU (Class P)

Type 1 access units encode aligned sequencing reads which perfectly match to the used reference sequence.

The decoding process of one record within a binary decoded access unit of type 1, which shall be repeated for all the records within the same access unit, is as follows:

1. Set a classId variable equal to the value of **AU_type** as specified in subclause 7.5.1.2.
2. Decode the variables numberOfRecordSegments, numberOfAlignedRecordSegments, numberOfMappedRecordSegments and unpairedRead as specified in subclause 10.4.10.
3. Compute the arrays softClips[][], softClipSizes[][] and hardClips[][] as specified in subclause 10.4.7.
4. Compute the arrays readLength[], numberOfSplicedSeg[], splicedSegLength[][] and splicedSegMappedLength[][] as specified in subclause 10.4.9.
5. Decode the output variables specified in subclause 10.4.12 containing the alignment and mapping information.
6. Decode the **pos** descriptor as specified in subclause 10.4.2.

7. Decode the output variables specified in subclause 10.4.10 containing pairing and/or splicing information.

8. Decode the **rcomp** descriptor as specified in subclause 10.4.3.

9. If **num_groups** specified in subclause 7.4.2 is greater than 0 decode the **rgroup** descriptor as specified in subclause 10.4.15.

10. Decode the readName variable as specified in subclause 10.4.17.

11. If **as_depth** specified in subclause 7.4.2 is greater than 0 decode the **mscore** descriptor as specified in subclause 10.4.11.

12. If **multiple_alignments_flag** specified in subclause 7.4.2 is 1 decode the **msar** descriptor as specified in subclause 10.4.13.

13. If present, decode the following optional descriptors:

    a. decode the **flags** descriptor as specified in subclause 10.4.4.

    b. decode the **qv** descriptor as specified in subclause 10.4.16.

14. If this process is being applied to access units of type 1 (Class P) (i.e., if this process is not being applied to access units of other types as specified in subclauses 10.2.4, 10.2.5 and 10.2.6), or if **crps_flag** specified in Table 7 is equal to 1 and **cr_alg_ID** specified in Table 16 is equal to 2, 3, or 4 and the value of **rtype** descriptor specified in Table 66 is equal to 1, decode the read sequences as specified in subclause 10.5.2.

10.2.4 Type 2 AU (Class N)

Access units of type 2 (Class N) are decoded by following the process described for AUs of type 1 (Class P) in subclause 10.2.3, then applying the information on unknown bases (symbol N) carried by the **mmpos** descriptor as specified in subclause 10.4.5, and finally decoding the read sequences as specified in subclause 10.5.2.

Additional inputs to this process are

— the array splicedSequence[][] specified in subclause 10.5

— the mismatchOffsets[][] and numMismatches[] arrays specified in subclause 10.4.5

The decoded splicedSequence[][] array shall be computed by replacing each base at a position represented by a decoded **mmpos** value in the splicedSequence[][] array obtained as specified in subclause 10.5.2 with the symbol 'N'.

The substitutions are applied as specified in Table 40.

Table 40 — Sequence decoding process for class N

Decoding step	Description
`processSplSegN(segment, splSeg) {`	
`  for(j = 0; j < numMismatches[segment]; j++) {`	
`    splicedSequence[segment][splSeg]` `      [mismatchOffsets[segment][j]] = 'N'`	
`  }`	
`}`	

10.2.5 Type 3 AU (Class M)

Access units of type 3 (Class M) are decoded by following the process described for AUs of type 1 (Class P) in subclause 10.2.3, then applying the information on substitutions obtained by following the decoding process of **mmpos** and **mmtype** descriptors as specified in subclauses 10.4.5 and 10.4.6, and finally decoding the read sequences as specified in subclause 10.5.2.

Additional inputs to this process are

— the mismatchOffsets[][], numMismatches[] arrays specified in subclause 10.4.5;

— the mismatches[][] arrays specified in subclauses 10.4.6.

The substitutions are applied as specified in Table 41.

Table 41 — Sequence decoding process for class M

Decoding step	Description
`processSplSegM(segment, splSeg) {`	
`  for(j = 0; j < numMismatches[segment]; j++) {`	
`    splicedSequence[segment][splSeg]` `      [mismatchOffsets[segment][j]] = mismatches[segment][j]`	
`  }`	
`}`	

10.2.6 Type 4 AU (Class I)

Access units of type 4 (Class I) are decoded by following the process described for AUs of type 1 (Class P) in subclause 10.2.3, then applying the edit operations represented by the decoded **mmpos, mmtype** and **clips** descriptors as specified in subclauses 10.4.5, 10.4.6 and 10.4.7, and finally decoding the read sequences as specified in subclause 10.5.2.

Additional inputs to this process are:

— the mismatchOffsets[][], numMismatches[] arrays specified in subclause 10.4.5;

— the mismatches[][] and mismatchTypes[][] arrays specified in subclause 10.4.6;

— the softClips[][][], softClipsSizes[][] and hardClips[][] arrays specified in subclause 10.4.7;

— the variable seqId set equal to **sequence_ID** as specified in subclause 7.5.1.2;

— the arrays **ref_sequence**[][] and **seq_start**[] specified as in subclause 7.3;

— the mappingPos[0][] array specified in subclause 10.2.3;

The substitutions, insertions and deletions are applied as specified in Table 42.

Table 42 — Sequence decoding process for mismatches in classes I and HM

Decoding step	Description
`processSplSegI(segment, splSeg) {`	
`  rlen = splicedSegLength[segment][splSeg]`	
`  if(splSeg == 0) {`	
`    rlen -= softClipSizes[segment][0]`	
`  }`	
`  if(splSeg == numberOfSplicedSeg[segment] - 1) {`	
`    rlen -= softClipSizes[segment][1]`	
`  }`	

Table 42 *(continued)*

Decoding step	Description
`indelsCount = 0`	
`mmStartIdx = splicedSegMismatchIdx[segment][splSeg]`	
`for(j = 0; j < splicedSegMismatchNumber[segment][ splSeg]; j++) {`	
`if(mismatchTypes[segment][mmStartIdx + j] == 0) {`	Substitution.

`splicedSequence[segment][splSeg]`
`[splicedSegMismatchOffsets[segment][splSeg][j]] =`
`mismatches[segment][mmStartIdx + j]`	

`} else if(mismatchTypes[segment][mmStartIdx + j] == 1) {`	Insertion.
`for(k = rlen - 1;` `k > splicedSegMismatchOffsets[segment][splSeg][j] ; k--) {`	All symbols after the insertion are shifted right by one position. The last element is therefore lost.
`splicedSequence[segment][splSeg][k] =` `splicedSequence[segment][splSeg][k - 1]`	
`}`	

`splicedSequence[segment][splSeg]`
`[splicedSegMismatchOffsets[segment][splSeg][j]] =`
`mismatches[segment][mmStartIdx + j]`	

`indelsCount -= 1`	
`} else if(mismatchTypes[segment][mmStartIdx + j] == 2) {`	Deletion.
`for(k = splicedSegMismatchOffsets[segment][splSeg][j] + 1;` `k < rlen; k++) {`	All symbols after the deletion are shifted left by one position.
`splicedSequence[segment][splSeg][k - 1] =` `splicedSequence[segment][splSeg][k]`	
`}`	

`splicedSequence[segment][splSeg][rlen - 1] =`
`ref_sequence[seqId]`
`[splicedSegMappingPos[segment][splSeq]`
 	
`- seq_start[seqId] + rlen`
`+ indelsCount]`
	A new symbol shall be copied from the reference at the end of segment.

`indelsCount += 1`	
`} else {`	
`/* reserved */`	
`}`	
`}`	
`processClips(segment, splSeg)`	Specified in Table 43.
`}`	

Information on clipped bases is applied as follows:

Soft clips

The contents of softClips[][] array computed as specified in subclause 10.4.7 are applied as specified in Table 43.

Table 43 — Sequence decoding process for soft clips in classes I and HM

Decoding step	Description
`processClips(segment, splSeg) {`	
`  if(splSeg == 0) {`	

`    splicedSequence[segment][splSeg] =`
`      strcat(softClips[segment][0],`
`        splicedSequence[segment][splSeg])`
	strcat returns the concatenation of the two arrays of ASCII characters passed as input.

`  }`	
`  if(splSeg == numberOfSplicedSeg[segment] - 1) {`	

`    splicedSequence[segment][splSeg] =`
`      strcat(splicedSequence[segment][splSeg],`
`        softClips[segment][1])`
	strcat returns the concatenation of the two arrays of ASCII characters passed as input.

`}`	

Hard clips

The hardClips[][] array is used to compute the ecigarString[] and ecigarLength[] arrays specified in subclause 10.6.2.

10.2.7 Type 5 AU (Class HM)

Class HM applies only to paired-end reads. Access units of type 5 are decoded as follows:

1. The mapped read is decoded by following the process specified for class I in subclause 10.2.6 and it is stored as the first record segment in the output record specified in Clause 13.
2. The unmapped read is decoded according to the process specified in subclause 10.5.3.

10.2.8 Type 6 AU (Class U)

10.2.8.1 General

Access units of type 6 (Class U) are decoded as follows:

1. Set a classId variable equal to the value of AU_type as specified in subclause 7.5.1.2.
2. Decode the variables numberOfRecordSegments, numberOfAlignedRecordSegments and numberOfMappedRecordSegments as specified in subclause 10.4.10.
3. Compute the array readLength[], numberOfSplicedSeg[], splicedSegLength[][] and splicedSegMappingPos[][] as specified in subclause 10.4.9.
4. Decode the output variables specified in subclause 10.4.12 containing the alignment and mapping information.
5. Decode the output variables specified in subclause 10.4.10 containing pairing and/or splicing information.
6. Decode the readName variable as specified in subclause 10.4.17.
7. If present, decode the following optional descriptors:
   a. decode the **flags** descriptor as specified in subclause 10.4.4;
   b. decode the **qv** descriptor as specified in subclause 10.4.16.
8. If **num_groups** specified in subclause 7.4.2 is greater than 0, decode the **rgroup** descriptor as specified in subclause 10.4.15.

9. Decode the read sequences as specified in subclause 10.5.3.

10.2.8.2 cr_alg_ID = 2

The "PushIn" computed reference algorithm specified in subclause 11.3.4 is used. In this case the genomic sequencing reads are decoded as for other classes of data by using the **rtype** descriptor as specified in subclause 10.4.14. The rtype descriptor is used to select the class of the next genomic record to be decoded.

10.2.8.3 cr_alg_ID = 4

The "Global Assembly" computed reference algorithm specified in subclause 11.3.6 is used. In this case the genomic sequencing reads are decoded as for other classes of data by using the **rtype** descriptor as specified in subclause 10.4.14. The rtype descriptor is used to select the class of the next genomic record to be decoded.

10.3 dataset_type = 2

10.3.1 General

The input to this process is either

— one AU of type 1, 2, 3 or 4 and a raw_reference data structure already initialized by a previous decoding process;

or

— an AU of type 6.

The output of this process is a $\text{raw\_reference}_{\text{output}}$ structure as specified in subclause 7.3.2. The array $\text{ref\_sequence}_{\text{output}}[]$ identifies the ref_sequence field of $\text{raw\_reference}_{\text{output}}$.

Subclause 7.4.2 specifies that all AUs referring to a parameter set having **dataset_type** set to 2 contain an encoded reference genome or portions thereof. According to the value of **AU_type** specified in subclause 7.5.1.2 the decoding process is as specified in subclauses 10.3.2, 10.3.3, 10.3.4, 10.3.5 and 10.3.6 for classes P, N, M, I and U.

The elements of the $\text{raw\_reference}_{\text{output}}$ syntax specified in subclause 7.3.2 shall be set as follows:

seq_count is set to the number of different values of **ref_sequence_ID,** specified in subclause 7.5.1.2, found in the headers of the AUs with **dataset_type** equal to 2 referring to the same parameter set.

For each value of **ref_sequence_ID** the following applies:

— **sequence_ID** in the raw_reference syntax is set to **ref_sequence_ID.**

— **seq_start** shall be set to the value of ref_start_position specified in subclause 7.5.1.2.

— **seq_end** shall be set to the value of ref_end_position specified in subclause 7.5.1.2.

The decoding process of each access unit refers to encoding parameters carried by the parameter set identified by the parameter_set_ID specified in subclause 7.5.1.2.

The **ref_sequence** element specified in subclause 7.3.2 is initialised with the output $\textbf{ref\_sequence}_{\textbf{output}}$ of the decoding processes specified in subclauses 10.3.2 to 10.3.6.

10.3.2 Type 1 AU

Type 1 access units used to encode a reference sequence carry portions of the reference sequence which perfectly match to the reference sequence identified by **sequence_ID,** specified in subclause 7.5.1.2, used for compression.

The decoding process of a binary decoded access unit of type 1 is as follows:

1. Set an array of ASCII characters outBuf[] equal to the empty array.
2. Decode the value readLength[0] as specified in subclause 10.4.9.
3. Decode one **pos** descriptor as specified in subclause 10.4.2 and set p_n equal to mappingPos[0][0] as specified in subclause 10.4.2.
4. A sequence of nucleotides outSequence is computed as follows:
   a. The position $pRef_0$ in the reference sequence identified by **sequence_ID** as specified in subclause 7.3 is computed as follows:

      $$pRef_0 = p_n - seq\_start[sequence\_ID]$$

   where **seq_start[sequence_ID]** is specified in subclause 7.3;

   b. outSequence = ref_sequence[**sequence_ID**][$pRef_0$, $pRef_0$+ readLength[0])

   where ref_sequence[**sequence_ID**][] is specified as in subclause 7.3.
5. The decoded sequence outSequence is concatenated with all previously decoded sequences in this AU and stored in a buffer outBuf computed as

   outBuf = strcat(outBuf, outSequence)

   where strcat returns the concatenation of the two arrays of ASCII characters passed as input.
6. If more genomic records are present, then go back to step 1 else go to step 7.
7. The buffer outBuf containing the concatenation of all output sequences is stored in the ref_ $sequence_{output}$ array of the raw_$reference_{output}$ structure produced as output of this decoding process:

   $ref\_sequence_{output}$[**ref_sequence_ID**] =

   outBuf[0, seq_end_{output}[**ref_sequence_ID**] - seq_start_{output}[**ref_sequence_ID**]],,

   where

   seq_start_{output} and seq_end_{output} correspond respectively to the seq_start and seq_end fields of the raw_$reference_{output}$ structure, and where the following condition shall always be met:

   Size(outBuf) > seq_end_{output}[**ref_sequence_ID**] - seq_start_{output}[**ref_sequence_ID**].

10.3.3 Type 2 AU

In case of AU of type 2 the sequence obtained at step 3 of subclause 10.3.2 is modified by applying the substitutions of symbol "N" according to the process described in subclause 10.2.4.

The decoding process continues then with step 5 of subclause 10.3.2.

10.3.4 Type 3 AU

In case of AU of type 3 the sequence obtained at step 3 of subclause 10.3.2 is modified by applying the substitutions according to the process described in subclause 10.2.5.

The decoding process continues then with step 5 of subclause 10.3.2.

10.3.5 Type 4 AU

In case of AU of type 4 the sequence obtained at step 3 of subclause 10.3.2 is modified by applying substitutions, insertions, deletions and soft clips according to the process described in subclause 10.2.6.

The decoding process continues then with step 5 of subclause 10.3.2.

10.3.6 Type 6 AU

In an AU of type 6 encoding a reference sequence, only **ureads** descriptors are always present, optionally associated to **rlen** descriptors providing the length of each encoded segment.

The decoding process is as follows:

1. Set an array of ASCII characters outBuf[] equal to the empty array.
2. Decode the value readLength[0] as specified in subclause 10.4.9.
3. Decode readLength[0] bases with decodeUreads(readLength[0]) as specified in subclause 10.4.8 and set outSequence to decodedUreads.
4. The decoded sequence outSequence is concatenated with all previously decoded outSequence in this AU and stored in a buffer outBuf computed as

   outBuf = strcat(outBuf, outSequence)

   where strcat returns the concatenation of the two arrays of ASCII characters passed as input.
5. If more genomic records are present, then go back to step 2 else go to step 6.
6. The buffer outBuf containing the concatenation of all output sequences is stored in the ref_sequence$_{output}$ array of the raw_reference$_{output}$ structure produced as output of this decoding process, according to the process specified at point 7 of subclause 10.3.2.

10.4 Genomic descriptors

10.4.1 General

The inputs to this process are descriptor subsequences generated at output of the parsing process specified in subclause 12.6. Each descriptor subsequence consists of a collection of symbols stored in the decoded_symbols data structure specified in subclause 12.6.2.2.

For a given descriptor_ID, subsequenceN identifies the array decoded_symbols[descriptor_ID][N].

The input to the decoding process of a descriptor sequence identified by descriptor_ID are K descriptor subsequences subsequence0 .. subsequenceK-1, with K equal to the number of descriptor subsequences as specified in Table 24.

The values of subsequenceN are read by means of indexes $j_{M,N}$ where M = descriptor_ID and N = descriptor_subsequence_ID.

Additional inputs are state variables computed during the decoding process described in this clause or other subclauses.

Some state variables listed among the outputs of the decoding processes described in this subclause shall be computed even if the corresponding descriptor is not present in the access unit. The listed inputs of each subclause are not always required; the decoding process described in each subclause specifies which inputs are required and which outputs are generated.

10.4.2 pos

The input to this process (see Table 44) is the array decoded_symbols[descriptor_ID][0] array specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 0 and the current value of $j_{0,0}$, the variable previousMappingPos0 produced by the previous iteration of this same process, and the array numberOfSegmentMappings[] calculated as specified in subclause 10.4.12.

The output of this process is an array mappingPos[][0] and the variable previousMappingPos0.

In this description, subsequenceN is the subsequence identified by descriptor_subsequence_ID = N (i.e. subsequenceN = decoded_symbols[0][N]).

Table 44 — Decoding process of the pos descriptor

Decoding step	Description
if($j_{0,0}$ > 0) {	
mappingPos[0][0] = previousMappingPos0 + subsequence0[$j_{0,0}$]	
}	
else{	
if(**AU_type** == 6) {	Unmapped content using computed reference
mappingPos[0][0] = subsequence0[$j_{0,0}$]	
} else {	
mappingPos[0][0] = **AU_start_position** + subsequence0[$j_{0,0}$]	**AU_start_position** is specified in subclause 7.5.1.2.
}	
}	
previousMappingPos0 = mappingPos[0][0]	
for(i = 1; i < numberOfSegmentMappings[0]; i++) {	numberOfSegmentMappings[0] is specified in subclause 10.4.12.
mappingPos[i][0] = mappingPos[i-1][0]+subsequence1[j0,1]	
j0,1++	
}	
$j_{0,0}$++	

10.4.3 rcomp

The inputs to this process are:

— the array decoded_symbols[descriptor_ID][0] specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 1 and the current value of $j_{1,0}$;

— the array numberOfSegmentMappings[] calculated as specified in subclause 10.4.12;

— the variable numberOfMappedRecordSegments calculated as specified in subclause 10.4.10;

— the array splitMate as specified in subclause 10.4.10;

— the array numberOfSplicedSeg[] specified in subclause 10.4.9.

The output of this process is the array reverseComp[][][].

In this description, subsequenceN is the subsequence identified by descriptor_subsequence_ID = N (i.e. subsequenceN = decoded_symbols[1][N]).

Each decoded **rcomp** descriptor conveys information about the *strandedness* of each segment of an alignment.

When no splices are present in the genomic record, each bit of a decoded **rcomp** descriptor is a flag indicating if the read is on the forward (bit set to 0) or reverse (bit set to 1) strand. Table 45 specifies the computation of reverseComp[][][] values.

Table 45 — Determination of the reverseComp values

Decoding step
`for(i = 0; i < numberOfMappedRecordSegments; i++){`
`  for(j = 0; j < numberOfSegmentMappings[i]; j++) {`
`  if(splitMate[j][i] == 0) {`
`  if(j == 0) {`
`  for(k = 0; k < numberOfSplicedSeg[i]; k++)`
`  reverseComp[k][j][i] = subsequence0[j1,0++]`
`  } else {`
`  reverseComp[0][j][i] = subsequence0[j1,0++]`
`  }`
`  }`
`  }`
`}`

When splices are present each decoded **rcomp** descriptor consists in a flag conveying information about the *strandedness* of each spliced segment of an alignment. It is set to 0 when the spliced segment is on the forward strand and it is set to 1 when the spliced segment is on the reverse strand.

10.4.4 flags

The input to this process is the decoded_symbols[descriptor_ID] array specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 2 and descriptor_subsequence_ID are equal to 0, 1 and 2 as specified in Table 26 and the current values of $j_{2,0}$, $j_{2,1}$ and $j_{2,2}$ as defined in subclause 10.4.

The output of this process is the variable decodedFlags.

In this description, subsequenceN is the subsequence identified by descriptor_subsequence_ID = N (i.e. subsequenceN = decoded_symbols[2][N]).

The flag syntax element carries additional alignment information usually produced by aligners as specified in Table 26.

The flags value shall be calculated according to the process specified in Table 46.

Table 46 — Decoding process of the flags descriptor

Decoding step	Description
`decodedFlags = 0`	
`decodedFlags |= subsequence0[`$j_{2,0}$`] << 0`	
`decodedFlags |= subsequence1[`$j_{2,1}$`] << 1`	
`decodedFlags |= subsequence2[`$j_{2,2}$`] << 2`	
$j_{2,0}$`++, `$j_{2,1}$`++, `$j_{2,2}$`++`	

10.4.5 mmpos

The inputs to this process are:

— two subsequences decoded_symbols[descriptor_ID][descriptor_subsequence_ID] as specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 3 and descriptor_subsequence_ID are equal to 0 and 1 as specified in Table 27;

— the current values of $j_{3,0}$. and $j_{3,1}$ as defined in subclause 10.4;

— the numberOfMappedRecordSegments variable specified in subclause 10.4.10;

— the classId variable specified in subclause 10.2.3;

— the arrays numberOfSplicedSeg[] and splicedSegLength[][] specified in subclause 10.4.9;

— the softClipSizes[][] array specified in subclause 10.4.7.

The output of this process are:

— the array mismatchOffsets[][]containing offsets of the mismatches in the sequencing read or read pair;

— the array numMismatches[] containing the number of elements in the array mismatchOffsets[][];

— the array splicedSegMismatchOffsets[][][] containing the offsets of mismatches within each spliced segment;

— the array splicedSegMismatchIdx[][] containing the positions, within the mismatchOffsets[][], mismatchTypes[][] and mismatches[][] arrays computed as specified in subclause 10.4.6, of the mismatches of each spliced segment;

— the array splicedSegMismatchNumber[][] containing the number of mismatches for each spliced segment.

In this description, subsequenceN is the subsequence identified by descriptor_subsequence_ID = N (i.e. subsequenceN = decoded_symbols[3][N]).

The overall decoding process for the output variables specified in this subclause is specified in Table 47:

Table 47 — Determination of the offset of mismatches

Decoding step	Description
`decodeMmpos()`	As specified in Table 48.
`if(classId == Class_I || classId == Class_HM) {`	
`  mismatchOffsetCorrectionByType()`	As specified in Table 50.
`}`	
`decodeSplicedSegMismatchOffsets()`	As specified in Table 49.

The mismatch offsets for each aligned segment shall be computed as specified in Table 48.

Table 48 — Determination of the offset of mismatches within genomic segments

Decoding step	Description
decodeMmpos() {	
for(i = 0; i < numberOfMappedRecordSegments; i++) {	
previousOffset = 0 j = 0	
for(k = 0; k < numberOfSplicedSeg[i]; k ++) {	
splicedSegMismatchNumber[i][k] = 0	
splicedSegMismatchIdx[i][k] = j	
while(subsequence0[$j_{3,0}$++] == 0){	Loop on subsequence0 until a terminator 1 is found.
mismatchOffsets[i][j] = subsequence1[$j_{3,1}$] + previousOffset	
previousOffset = mismatchOffsets[i][j]	
previousOffset += 1	Adjacent mismatch positions are strictly incremental to prevent overlapping mismatches. Exceptions to this requirement are specified in Table 50.
splicedSegMismatchNumber[i][k]++	
$j_{3,1}$++, j++	Increment read and write pointers.
}	
}	
numMismatches[i] = j	
}	
}	

The mapping from splice mismatch indexes to genomic segment mismatch indexes shall be computed as specified in Table 49.

Table 49 — Determination of the offset of mismatches within spliced segments

Decoding step
decodeSplicedSegMismatchOffsets() {
for(i = 0; i < numberOfAlignedRecordSegments; i++) {
splicedSegStartOffset = 0
splicedSegEndOffset = splicedSegStartOffset + splicedSegLength[i][0] – softClipSizes[i][0]
l = 0
for(k = 0; k < numberOfSplicedSeg[i]; k ++) {
for(j = 0; j < splicedSegMismatchNumber[i][k]; j++) {
splicedSegMismatchOffsets[i][k][j] = mismatchOffsets[i][l] – splicedSegStartOffset
l++
}

Table 49 *(continued)*

Decoding step
`if(k < numberOfSplicedSeg[i] - 1) {`
`splicedSegStartOffset = splicedSegEndOffset`
`splicedSegEndOffset = splicedSegStartOffset + splicedSegLength[i][k + 1]`
`}`
`}`
`}`
`}`

10.4.6 mmtype

The inputs to this process are:

— three subsequences decoded_symbols[descriptor_ID][descriptor_subsequence_ID] as specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 4 and descriptor_subsequence_ID are equal to 0, 1 and 2 as specified in Table 28. The decoding process specified in subclause 12.6.2.3 for decoded_symbols[4][1] shall be performed after the decoding process specified in Table 51;

— the array with the number of mismatches numMismatches[] and the offset array mismatchOffsets[][] calculated for the current genomic record as specified in subclause 10.4.5;

— the arrays splicedSegMismatchNumber[][] and splicedSegMismatchOffsets[][][] as specified in subclause 10.4.5 the current values of $j_{4,0}$, $j_{4,1}$ and $j_{4,2}$ as defined in subclause 10.4;

— the array $S_{alphabet\_ID}$[] as specified in subclause 9.2, for the value of alphabet_ID specified in subclause 9.2;

— the arrays mappingPos[][] and splicedSegMappingPos[][] as specified in subclauses 10.4.2 and 10.4.10;

— the classId variable specified in subclause 10.2.3;

— the numberOfMappedRecordSegments variable specified in subclause 10.4.10;

— the variable seqId set equal to **sequence_ID** as specified in subclause 7.5.1.2. If **crps_flag** specified in Table 7 is equal to 1 and **cr_alg_ID** specified in Table 16 is equal to 2, 3 or 4, seqId is not used;

— the variable seqStart equal to 0 if **crps_flag** specified in Table 7 is equal to 1 and **cr_alg_ID** specified in Table 16 is equal to 2, 3 or 4, else seqStart is set equal to **seq_start**[seqId] with **seq_start**[] as specified in subclause 7.3;

— the array splicedSegMappedLength[][] computed as specified in subclause 10.4.9.

The outputs of this process are arrays containing values identifying the type of edit operations to be performed on the sequencing read or read pair computed as specified in subclause 10.4.20 when classId, specified in subclause 10.2.3, is equal to Class_M, Class_I or Class_HM:

— the modified mismatchOffsets[][] array;

— the array mismatchTypes[][] contains values for the type of mismatch. 0 signals substitutions, 1 signals insertions and 2 signals deletions;

— the array mismatches[][] contains the symbols to be used for substitutions and insertions;

— the array substMappingOffsets[][] containing the offsets of the mismatches within the reference sequence the segment is mapped to;

— the modified splicedSegMappedLength[][] array.

In this description, subsequenceN is the subsequence identified by descriptor_subsequence_ID = N (i.e. subsequenceN = decoded_symbols[4][N]).

If classId is equal either to Class_I or to Class_HM, the output mismatchOffsets[][] array specified in subclause 10.4.5 shall be modified, before any possible use, according to the decoding process specified in Table 50.

Table 50 — Updating mismatchOffsets[][] array based on mismatch types

Decoding step	Description
`mismatchOffsetCorrectionByType() {`	
`  k = j4,0`	
`  for(i = 0; i < numberOfMappedRecordSegments; i++) {`	
`    numOfDeletions = 0`	
`    for(j = 0; j < numMismatches[i]; j++) {`	
`      mismatchOffsets[i][j] -= numOfDeletions`	Deletions can occur at the same position of the next mismatch. Therefore, the extra +1 offset to prevent overlapping mismatches, as specified in Table 90, does not apply to deletions.
`      if(subsequence0[k] == 2) {`	Deletion.
`        numOfDeletions += 1`	
`      }`	
`      k++`	
`    }`	
`  }`	
`}`	

The arrays substMappingOffsets[] and splicedSegMappedLength[][] shall be, respectively, calculated and modified following the process described in Table 51.

Table 51 — Determination of the substMappingOffsets[] arrays.

Decoding step	Description
`k = j4,0`	
`for(i = 0; i < numberOfMappedRecordSegments; i++) {`	
`  l = 0`	
`  substMappingOffsets[i] = {}`	Empty array.
`  if(numberOfSplicedSeg[i] == 1) {`	Case of no splices.
`    mappedMmpos = mappingPos[0][i] - seqStart`	
`    previousOffset = 0`	
`    for(j = 0; j < numMismatches[i]; j++) {`	
`      mappedMmpos +=` `        mismatchOffsets[i][j] - previousOffset`	
`      previousOffset = mismatchOffsets[i][j]`	

Table 51 *(continued)*

Decoding step	Description
`if(subsequence0[k] == 0) {`	Substitution.
`substMappingOffsets[i][l] = mappedMmpos`	
`l++`	
`} else if(subsequence0[k] == 1) {`	Insertion.
`mappedMmpos -= 1`	Insertions increase mmpos descriptor value but, since they do not represent an actual base on the reference sequence, they shall not increase the mapped position, as specified in Table 90.
`} else if(subsequence0[k] == 2) {`	Deletion.
`mappedMmpos += 1`	Deletions do not increase mmpos descriptor value but, since they represent an actual base on the reference sequence, they shall increase the mapped position, as specified in Table 90.
`}`	
`k++`	
`}`	
`} else {`	Case of splices.
`previousOffset = 0`	
`previousSpliceEndOffset = 0`	
`for(s = 0; s < numberOfSplicedSeg[i]; s++) {`	
`mappedMmpos = splicedSegMappingPos[i][s] - seqStart`	
`previousOffset = 0`	
`for(j = 0; j < splicedSegMismatchNumber[i][s]; j++) {`	
`mappedMmpos += splicedSegMismatchOffsets[i][s][j] - previousOffset`	
`previousOffset = splicedSegMismatchOffsets[i][s][j]`	
`if(subsequence0[k] == 0) {`	Substitution.
`substMappingOffsets[i][l] = mappedMmpos`	
`l++`	
`} else if(subsequence0[k] == 1) {`	Insertion.
`mappedMmpos -= 1`	Insertions increase mmpos descriptor value but, since they do not represent an actual base on the reference sequence, they shall not increase the mapped position, as specified in Table 90.
`splicedSegMappedLength[i][s] -= 1`	
`} else if(subsequence0[k] == 2) {`	Deletion.
`mappedMmpos += 1`	Deletions do not increase mmpos descriptor value but, since they represent an actual base on the reference sequence, they shall increase the mapped position, as specified in Table 90.

Table 51 *(continued)*

Decoding step	Description
splicedSegMappedLength[i][s] += 1	
}	
k++	
}	
}	
}	
}	

The remaining output of **mmtype** descriptor decoding process shall be calculated following the process described in Table 52, after having decoded subsequence1 according to the decoding process specified in Table 124 using, if required by the said decoding process specified in Table 124 and by following the decoding process specified in subclause 12.6.2.3, the array substMappingOffsets[] decoded as specified in Table 51.

Table 52 — Determination of the mismatchTypes[] and mismatches[] arrays

Decoding step	Description
for(s = 0; s < numberOfMappedRecordSegments; s++) {	
j = 0	
while(j < numMismatches[s]) {	
if(Size(subsequence0[]) > 0) {	
mismatchTypes[s][j] = subsequence0[j4,0]	
} else {	
mismatchTypes[s][j] = 0	Default to substitution if subsequence0 is empty.
}	
if(mismatchTypes[s][j] == 0)	Substitution.
mismatches[s][j] = $S_{alphabet\ ID}$[subsequence1[$j_{4,1}$]]	
$j_{4,1}$++	
} else if(mismatchTypes[s][j] == 1) {	Insertion.
mismatches[s][j] = $S_{alphabet\ ID}$[subsequence2[$j_{4,2}$]]	
$j_{4,2}$++	
} else if(mismatchTypes[s][j] == 2) {	Deletion.
/* nothing needs to be done */	The value of mismatches[j] is undefined, as it is not relevant for any decoding process.
}	
$j_{4,0}$++, j++	
}	
}	

10.4.7 clips

The inputs to this process are:

— four subsequences decoded_symbols[descriptor_ID][descriptor_subsequence_ID] as specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 5;

— the variable currentRecordCount is the number of processed genomic records in the current AU and it is initialized to 0 at the beginning of current AU decoding process;

— the current values of $j_{5,0}$, $j_{5,1}$, $j_{5,2}$ and $j_{5,3}$ as defined in subclause 10.4;

— the array $S_{alphabet\_ID}$[] as specified in subclause 9.2, for the value of alphabet_ID specified in subclause 7.4.2;

— the value Size($S_{alphabet\_ID}$) as specified in subclause 9.2, for the value of alphabet_ID specified in subclause 7.4.2;

— the variable numberOfMappedRecordSegments calculated as specified in subclause 10.4.10;

— the classId variable specified in subclause 10.2.3.

The four subsequences are identified by subsequences_ID from 0 to 3 as specified in Table 29.

The output of this process is an array softClips[][][], an array softClipSizes[][] and an array hardClips[][] as specified in Table 54.

The decoding process of the clips descriptor is provided in Table 54 where:

— subsequenceN is the subsequence identified by descriptor_subsequence_ID = N;

— subsequence0[$j_{5,0}$] represents the next genomic record containing clipped bases;

— subsequence1[$j_{5,1}$] represent the type and position of clipped bases;

— softClips, softClipSizes, and hardClips are the output of this decoding process:

  — softClips[0][0] and softClips[1][0] contain strings of characters representing soft clips preceding the first mapped base of the leftmost read and rightmost read respectively,

  — softClips[0][1] and softClips[1][1] contain strings of characters representing soft clips following the last mapped base of the leftmost read and rightmost read respectively,

  — softClipSizes[i][j] contain the number of charcters in the strings in softClips[i][j] respectively,

  — hardClips[0][0] and hardClips[1][0] contain the number of hard clips preceding the first mapped base of the leftmost read and rightmost read respectively,

  — hardClips[0][1] and hardClips[1][1] contain the number of hard clips following the last mapped base of the leftmost read and rightmost read respectively;

— the semantics of subsequence1 are as shown in Table 53.

Table 53 — Values and semantics for subsequence1

subsequence1 values	semantics
0	Soft clips before the start of leftmost read. Shall not be used if 4 is present for the same genomic record.
1	Soft clips after the end of leftmost read Shall not be used if 5 is present for the same genomic record.
2	Soft clips before the start of rightmost read. Shall not be used if 6 is present for the same genomic record.
3	Soft clips after the end of rightmost read. Shall not be used if 7 is present for the same genomic record.
4	Hard clips before the start of leftmost read. Shall not be used if 0 is present for the same genomic record.

Table 53 *(continued)*

subsequence1 values	semantics
5	Hard clips after the end of leftmost read. Shall not be used if 1 is present for the same genomic record.
6	Hard clips before start of rightmost read. Shall not be used if 2 is present for the same genomic record.
7	Hard clips after end of rightmost read. Shall not be used if 3 is present for the same genomic record.
8	End-of-clips terminator.

For a decoded genomic record each value of subsequence1 as specified in Table 53 shall not be used more than once.

Table 54 — Decoding process of the clips descriptor

Decoding process	Description
`for(i = 0; i < numberOfMappedRecordSegments; i++) {`	
`  for(j = 0; j < 2; j++) {`	
`  softClips[i][j] = ""`	Empty string.
`  softClipSizes[i][j] = 0`	
`  hardClips[i][j] = 0`	
`  }`	
`}`	
`if(classId == Class_I || classId == Class_HM){`	
`  if(`$j_{5,0}$` < Size(subsequence0) && currentRecordCount == subsequence0[`$j_{5,0}$`]){`	
`  end = 0`	
`  do{`	
`  if(subsequence1[`$j_{5,1}$`] ≤ 3){`	Soft clips.
`  j=0`	
`  segmentIdx = subsequence1[`$j_{5,1}$`] >> 1`	
`  leftRightIdx = subsequence1[`$j_{5,1}$`] & 1`	
`  do{`	
`  softClips[segmentIdx][leftRightIdx][j] =` $S_{alphabet\_ID}$`[subsequence2[`$j_{5,2}$`]]`	
`  `$j_{5,2}$`++`	Increment pointer for subsequence2.
`  j++`	
`  }while(subsequence2[`$j_{5,2}$`] != Size(`$S_{alphabet\_ID}$`))`	Continue reading symbols of clipped bases until the end-of-soft-clips terminator is reached.
`  `$j_{5,2}$`++`	Increment pointer for subsequence2.
`  softClipSizes[segmentIdx][leftRightIdx] = j`	Store soft clips size.
`  }`	
`  else if(subsequence1[`$j_{5,1}$`] ≤ 7){`	Hard clips.
`  segmentIdx = (subsequence1[`$j_{5,1}$`] - 4) >> 1`	
`  leftRightIdx = (subsequence1[`$j_{5,1}$`] - 4) & 1`	
`  hardClips[segmentIdx][leftRightIdx] = subsequence3[`$j_{5,3}$`]`	Store the number of hard clips.

Table 54 *(continued)*

Decoding process	Description
$j_{5,3}$++	Increment pointer for subsequence3.
}	
else if(subsequence1[$j_{5,1}$] == 8){	End-of-clips terminator.
end = 1	
}	
$j_{5,1}$++	Increment pointer for subsequence1.
} while(end == 0)	Continue decoding soft and hard clips until the end of clips terminator is detected.
$j_{5,0}$++	Increment pointer for subsequence0.
}	
currentRecordCount++	
}	

10.4.8 ureads

The inputs to this process (see Table 55) are:

— the array decoded_symbols[descriptor_ID][0] structure as specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 6;

— the current value of $j_{6,0}$;

— the array $S_{alphabet\_ID}$[] as specified in subclause 9.2, for the value of alphabet_ID specified in subclause 7.4.2.

The output of this process is a string decodedUreads.

Table 55 — Decoding process of the ureads descriptor

Decoding process	Description
decodeUreads(length) {	
decodedUreads = ""	Empty string.
for(j = 0; j < length; j++) {	
decodedUreads = strcat(decodedUreads, $S_{alphabet\_ID}$[decoded_symbols[6][0][$j_{6,0}$]]	strcat returns the concatenation of the two arrays of ASCII characters passed as input.
$j_{6,0}$++	
}	
}	

10.4.9 rlen

The **rlen** descriptor is present when read_length is equal to 0 in the parameter set or when there are multiple alignments with splices.

The inputs to this process are:

— the array decoded_symbols[descriptor_ID][0] as specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 7;

— the value read_length as specified in subclause 7.4.2;

— the variable classId computed in subclause 10.2.3;

— the variables numberOfRecordSegments and numberOfAlignedRecordSegments computed as specified in subclause 10.4.10;

— if classId is equal to Class_I or Class_HM, the array hardClips[][] computed as specified in subclause 10.4.7;

— the **spliced_reads_flag** syntax element specified in subclause 7.4.2;

— the softClipSizes[][] array specified in subclause 10.4.7;

— the current value of $j_{7,0}$.

The outputs of this process are:

— the array readLength[];

— the array numberOfSplicedSeg[];

— the array splicedSegLength[][];

— the array splicedSegMappedLength[][].

The decoding process of the rlen descriptor is specified in Table 56. In this description, subsequenceN is the subsequence identified by descriptor_subsequence_ID = N (i.e. subsequenceN = decoded_symbols[7][N]).

Table 56 — Decoding process of the rlen descriptor

Decoding step	Description
`if(read_length == 0){`	
`  for(i = 0; i < numberOfRecordSegments; i++){`	
`  readLength[i] = subsequence0[j7,0++] + 1`	
`  }`	
`}else{`	
`  for(i = 0; i < numberOfRecordSegments; i++){`	
`  if(classId == Class_I){`	
`  readLength[i] = read_length - hardClips[i][0] - hardClips[i][1]`	
`  }`	
`  else if(classId == Class_HM && i == 0){`	
`  readLength[i] = read_length - hardClips[0][0] - hardClips[0][1]`	
`  }`	
`  else {`	
`  readLength[i] = read_length`	
`  }`	
`  }`	
`}`	
`for(i = 0; i < numberOfRecordSegments; i++){`	
`  numberOfSplicedSeg[i] = 1`	
`  splicedSegLength[i][0] = readLength[i]`	
`  splicedSegMappedLength[i][0] = readLength[i]`	

Table 56 *(continued)*

Decoding step	Description
}	
if(spliced_reads_flag && (classId == Class_I || classId == Class_HM)){	
for(i = 0; i < numberOfAlignedRecordSegments; i++){	
remainingLen = readLength[i]	
j = 0	
do{	
spliceLen = subsequence0[$j_{7,0}$++]	
remainingLen -= spliceLen	
splicedSegLength[i][j] = spliceLen	
splicedSegMappedLength[i][j] = spliceLen	
j++	
} while(remainingLen > 0)	
numberOfSplicedSeg[i] = j	
splicedSegMappedLength[i][0] -= softClipSizes[i][0]	
splicedSegMappedLength[i][j-1] -= softClipSizes[i][1]	
}	
}	

10.4.10 pair

Table 57 lists the possible decoding cases for the pair descriptor with the associated description for the first alignment and class U.

Table 57 — Specification of the decoding cases for the pair descriptor for primary alignments and class U

Decoding case	Description		
	Classes P, N, M, I	Class HM	Class U
same_rec	Read 1 and read 2 are encoded in the same genomic record.		
R1_ split	Read 1 in pair is on the same reference sequence but coded separately.	N/A	Read 1 paired with mate in the same AU.
R2_split	Read 2 in pair is on the same reference sequence but coded separately.	N/A	Read 2 paired with mate in the same AU.
R1_diff_ref_seq	Read 1 is on a different reference sequence.	N/A	Read 1 paired with mate in a different AU.
R2_diff_ref_seq	Read 2 is on a different reference sequence.	N/A	Read 2 paired with mate in a different AU.
R1_unpaired	Read 1 is unpaired.	N/A	Read 1 unpaired.
R2_unpaired	Read 2 is unpaired.	N/A	Read 2 unpaired.

Table 58 lists the possible decoding cases for the pair descriptor with the associated description for alignments after the first one.

When the two ends of a paired-end read are coded in two different genomic records, they are part of a split alignment.

Table 58 — Specification of the decoding cases for the pair descriptor for alignments after the first one

Decoding case	Description
	Classes P, N, M, I
same_rec_short	Read 1 and read 2 are encoded in the same genomic record and the absolute pairing distance is smaller than or equal to 32767.
same_rec_long	Read 1 and read 2 are encoded in the same genomic record and the absolute pairing distance is greater than 32767.
R2_diff_ref_seq	Read 2 is on a different reference sequence.

Table 59 lists the possible decoding cases for the pair descriptor with the associated description for spliced reads.

Table 59 — Specification of the decoding cases for the pair descriptor for spliced reads

Decoding case	Description
	Classes I, HM
same_rec_short	The next splice is in the same genomic record as current splice, and the splicing distance is smaller than or equal to 65535.
same_rec_long	The next splice is in the same genomic record as current splice, and the splicing distance is greater than 65535.
splice_diff_ref_seq	The next splice is on a different reference sequence than the current splice.

The inputs to this process are:

— the value of numberOfTemplateSegments as specified in subclause 7.4.2;

— eight subsequences decoded_symbols[descriptor_ID][descriptor_subsequence_ID] as specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 8. The description of each subsequence is provided in Table 30;

— the current values of $j_{8,0}$, $j_{8,1}$, $j_{8,2}$, $j_{8,3}$, $j_{8,4}$, $j_{8,5}$, $j_{8,6}$and $j_{8,7}$;

— the array mappingPos[][0] computed as specified subclause 10.4.2;

— the classId variable specified in subclause 10.2.3;

— a seqId variable set to **sequence_ID** as specified in subclause 7.5.1.2;

— the array alignPtr[][] specified in subclause 10.4.12;

— the variable numberOfAlignments and the array numberOfSegmentAlignments[] specified in subclause 10.4.12;

— the arrays numberOfSplicedSeg[] and splicedSegLength[][] specified in subclause 10.4.9;

— the **crps_flag** value specified in subclause 7.4.2 and the **cr_alg_ID** value specified in subclause 7.4.2.4;

The outputs of this process are:

— a variable numberOfRecordSegments calculated as follows:

  — if numberOfTemplateSegments is equal to 1 then numberOfRecordSegments is set to 1,

  — else if classId is equal to Class_HM as specified in Table 37 then numberOfRecordSegments is set to 2,

  - else if subsequence0[$_{j8,0}$] is equal to 0 then numberOfRecordSegments is set to 2,
  - else numberOfRecordSegments is set to 1;
- a variable numberOfAlignedRecordSegments calculated as follows:
  - if classId is equal to Class_HM as specified in Table 37 then numberOfAlignedRecordSegments is set to 1,
  - else if classId is equal to Class_U as specified in Table 37 then numberOfAlignedRecordSegments is set to 0,
  - else numberOfAlignedRecordSegments is set to the value of numberOfRecordSegments;
- a variable numberOfMappedRecordSegments calculated as follows:
  - if classId is equal to Class_U as specified in Table 37, and **crps_flag** is not equal to 0 and **cr_alg_ID** is equal to 2 or 4 as specified in subclause 7.4.2, then numberOfMappedRecordSegments is set to the value of numberOfRecordSegments,
  - else numberOfMappedRecordSegments is set to the value of numberOfAlignedRecordSegments,
- a variable unpairedRead calculated as follows:
  - if classId is equal to Class_HM as specified in Table 37 then unpairedRead is set to 0,
  - else if numberOfTemplateSegments is equal to 1 or subsequence0[$_{j8,0}$] is equal to 5 or 6 then unpairedRead is set to 1,
  - else unpairedRead is set to 0;
- one flag read1First, whose value follows the same semantics of **read_1_first** output syntax element specified in subclause 13.2.8;
- the arrays splitMate[][i] for i from 1 to numberOfTemplateSegments, where the value of each element follows the same semantics of **split_alignment** output syntax element specified in subclause 13.2.23;
- the arrays splicedSegMappingPos[i][] for i from 0 to numberOfRecordSegments.

When classId is equal to Class_P, Class_N, Class_M or Class_I, additional output of this process is:

- the arrays mappingPos[][i] for i from 1 to numberOfTemplateSegments;
- the arrays mateSeqId[][i] for i from 1 to numberOfTemplateSegments.

When classId is equal to Class_U, additional output of this process is:

- the arrays pairingMate[i] from 1 to numberOfTemplateSegments. A -1 value in an array element is used as reserved value;

In the following descriptions of the decoding process subsequenceN indicates the subsequence identified by descriptor_subsequence_ID equal to N.

The decoding process of the **pair** descriptor is carried out by applying the decoding processes specified in Table 60, Table 61, and Table 62, in this exact order.

The decoding process of the **pair** descriptor for the first alignment and for class U is specified in Table 60.

Table 60 — Decoding process of the pair descriptor subsequences for the first alignment in the record or class U

Decoding step	Description
splitMate[0][0] = 0	
read1First = 1	
if(classId == Class_HM) {	
read1First = (subsequence1[$j_{8,1}$++] & 0x0001) ? 0 : 1	same_rec – in records of class HM, the paired segments are always in the same record.
splitMate[0][1] = 0	
} else {	
for(i = 1; i < numberOfTemplateSegments; i++) {	
if(subsequence0[$j_{8,0}$] == 0){	same_rec
splitMate[0][i] = 0	

if(classId != Class_U
|| (crps_flag != 0 &&
(cr_alg_ID == 2 || cr_alg_ID == 4))) {	

read1First = (subsequence1[$j_{8,1}$] & 0x0001) ? 0 : 1	
delta = subsequence1[$j_{8,1}$] >> 1	0 ≤ delta ≤ 32767
mappingPos[0][i] = mappingPos[0][0] + delta	
if(classId != Class_U) {	
mateSeqId[0][i] = seqId	
}else {	
pairingMate[i] = -1	
}	
$j_{8,1}$++	
} else {	
read1First = 1	
pairingMate[i] = -1	
}	
}	
else if (subsequence0[$j_{8,0}$] == 1){	R1_split
splitMate[0][i] = 1	
read1First = 0	
if(classId != Class_U) {	
mappingPos[0][i] = subsequence2[$j_{8,2}$]	Absolute mapping position of read 1 on the same reference sequence. The maximum value is $2^{posSize} - 1$ where posSize is specified in subclause 7.4.2.
mateSeqId[0][i] = seqId	
} else {	
pairingMate[i] = -1	
}	
$j_{8,2}$++	
}	
else if (subsequence0[$j_{8,0}$] == 2){	R2_split
splitMate[0][i] = 1	
read1First = 1	

Table 60 *(continued)*

Decoding step	Description
if(classId != Class_U) {	
mappingPos[0][i] = subsequence3[$j_{8,3}$]	Absolute mapping position of the read 2 on the same reference sequence. The maximum value is $2^{posSize} - 1$ where posSize is specified in subclause 7.4.2.
mateSeqId[0][i] = seqId	
} else {	
pairingMate[i] = -1	
}	
$j_{8,3}$++	
}	
else if (subsequence0[$j_{8,0}$] == 3){	R1_diff_ref_seq
splitMate[0][i] = 1	
read1First = 0	
if(classId != CLASS_U){	
mateSeqId[0][i] = subsequence4[$j_{8,4}$]	Identifier of the reference sequence to which read 1 is mapped.
mappingPos[0][i] = subsequence6[$j_{8,6}$]	Absolute mapping position of read 1 on the reference sequence identified by mateSeqId[0][i]. The maximum value is $2^{posSize} - 1$ where posSize is specified in subclause 7.4.2.
}else{	
pairingMate[i] = -1	
}	
$j_{8,4}$++, $j_{8,6}$++,	
}	
else if (subsequence0[$j_{8,0}$] == 4){	R2_diff_ref_seq
splitMate[0][i] = 1	
read1First = 1	
if(classId != CLASS_U){	
mateSeqId[0][i] = subsequence5[$j_{8,5}$]	Identifier of the reference sequence to which read 2 is mapped.
mappingPos[0][i] = subsequence7[$j_{8,7}$]	Absolute mapping position of the read 2 on the reference sequence identified by mateSeqId[0][i]. The maximum value is $2^{posSize} - 1$ where posSize is specified in subclause 7.4.2.
}else{	
pairingMate[i] = -1	
}	
$j_{8,5}$++, $j_{8,7}$++,	
}	
else if (subsequence0[$j_{8,0}$] == 5){	R1_unpaired

Table 60 *(continued)*

Decoding step	Description
splitMate[0][i] = 2	
read1First = 1	
if(classId == CLASS_U){	
pairingMate[i] = -1	
}	
}	
else if (subsequence0[$j_{8,0}$] == 6){	R2_unpaired
splitMate[0][i] = 2	
read1First = 0	
if(classId == CLASS_U){	
pairingMate[i] = -1	
}	
}	
$j_{8,0}$++	
}	
}	

The decoding process of the **pair** descriptor for the alignments after the first one is specified in Table 61.

Table 61 — Decoding process of the pair descriptor subsequences for the alignments in the record after the first one

Decoding step	Description
for(i = 1; i < numberOfSegmentAlignments[0]; i++) {	
splitMate[i][0] = 0	
}	

if((classId == Class_P || classId == Class_N
|| class_ID == Class_M || classId == Class_I)
&& !unpairedRead) {	

for(j = 1; j < numberOfTemplateSegments; j++) {	
currAlignIdx = 0	
for(i = 1; i < numberOfAlignments; i++){	
alignIdx = alignPtr[i][j]	
if(alignIdx > currAlignIdx) {	
currAlignIdx = alignIdx	
if(subsequence0[$j_{8,0}$] == 0){	same_rec_short
splitMate[alignIdx][j] = 0	
delta = subsequence1[$j_{8,1}$] >> 1;	0 ≤ delta ≤ 32767
if(subsequence1[$j_{8,1}$] & 0x0001) delta = - delta	read sign bit
mappingPos[alignIdx][j] = mappingPos[alignPtr[i][0]][0] + delta	
mateSeqId[alignIdx][j] = seqId	
$j_{8,1}$++	
}	
else if (subsequence0[$j_{8,0}$] == 2){	same_rec_long
splitMate[alignIdx][j] = 0	

Table 61 *(continued)*

Decoding step	Description
mappingPos[alignIdx][j] = subsequence3[$j_{8,3}$]	For classes P, N, M , I Absolute mapping position of read 2 on the same reference sequence. The maximum value is $2^{posSize} - 1$ where posSize is specified in subclause 7.4.2.
mateSeqId[alignIdx][j] = seqId	
$j_{8,3}$++	
}	
else if (subsequence0[$j_{8,0}$] == 4){	R2_diff_ref_seq
splitMate[alignIdx][j] = 1	
mateSeqId[alignIdx][j] = subsequence5[$j_{8,5}$]	Identifier of the reference sequence to which read 2 is mapped.
mappingPos[alignIdx][j] = subsequence7[$j_{8,7}$]	For classes P, N, M , I Absolute mapping position of read 2 on the reference sequence identified by subsequence5[$j_{8,5}$]. The maximum value is $2^{posSize} - 1$ where posSize is specified in subclause 7.4.2.
$j_{8,5}$++, $j_{8,7}$++,	
}	
else {	
/* other subsequence0[$j_{8,0}$] values */	reserved
}	
$j_{8,0}$++	
}	
}	
}	
}	

The decoding process of the **pair** descriptor for spliced reads is specified in Table 62.

Table 62 — Decoding process of the pair descriptor subsequences for spliced reads

Decoding step	Description
for(i = 0; i < numberOfMappedRecordSegments; i++) {	
splicedSegMappingPos[i][0] = mappingPos[0][i]	
}	
if(classId == Class_I || classId == Class_HM) {	
for(i = 0; i < numberOfAlignedRecordSegments; i++){	
for(j = 1; j < numberOfSplicedSeg[i]; j++) {	
prevSpliceMappingEnd = splicedSegMappingPos[i][j - 1] + splicedSegLength[i][j - 1]	
if(subsequence0[$j_{8,0}$] == 0) {	same_rec_short
delta = subsequence1[$j_{8,1}$] >> 1	0 ≤ delta ≤ 32767

Table 62 *(continued)*

Decoding step	Description
if(subsequence1[$j_{8,1}$] & 0x0001) delta = - delta	read sign bit
splicedSegMappingPos[i][j] = prevSpliceMappingEnd + delta	
$j_{8,1}$++	
}	
else if (subsequence0[$j_{8,0}$] == 2){	same_rec_long
splicedSegMappingPos[i][j] = subsequence3[$j_{8,3}$]	Absolute mapping position of the splice on the same reference sequence as the previous splice. The maximum value is $2^{posSize} - 1$ where posSize is specified in subclause 7.4.2.
$j_{8,3}$++	
}	
else {	
/* other subsequence0[$j_{8,0}$] values */	reserved
}	
$j_{8,0}$++	
}	
}	
}	

10.4.11 mscore

The **mscore** descriptor provides a score per segment in each alignment. Some information on how to use the mscore descriptor to express the mapping quality is provided in Annex B.

The inputs to this process are:

— the decoded_symbols[descriptor_ID] array specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 9;

— the current value of $j_{9,0}$;

— the value of syntax element **as_depth** specified in subclause 7.4.2;

— the array numberOfSegmentAlignments[] calculated as specified in subclause 10.4.12;

— the variable numberOfAlignedRecordSegments calculated as specified in subclause 10.4.10;

— the array splitMate as specified in subclause 10.4.10.

The output of this process is the three-dimensional mappingScores[][][]array.

The decoding process of the **mscore** descriptor is specified in Table 63. In this description, subsequenceN is the subsequence identified by descriptor_subsequence_ID = N (i.e. subsequenceN = decoded_symbols[9][N]).

Table 63 — Decoding process for the mscore descriptor

Decoding step	Description
`for(i = 0; i < as_depth; i++) {`	
`  for(j = 0; j < numberOfAlignedRecordSegments; j++) {`	
`  for(k = 0; k < numberOfSegmentAlignments[j]; k++) {`	
`  if(splitMate[k][j] == 0) {`	
`  mappingScores[k][j][i] = subsequence0[j9,0++];`	
`  }`	
`  }`	
`  }`	
`}`	

10.4.12 mmap

10.4.12.1 General

The **mmap** descriptor is used to signal on how many positions the read or the leftmost read of a pair has been aligned. A genomic record containing multiple alignments is associated with one **mmap** descriptor.

The inputs to this process are:

— the variables unpairedRead, numberOfAlignedRecordSegments and numberOfRecordSegments computed in subclause 10.4.10;

— the subsequences decoded_symbols[descriptor_ID][descriptor_subsequence_ID] as specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 10. The description of each subsequence is provided in Table 31;

— the current values of $j_{10,0}$, $j_{10,1}$, $j_{10,2}$, $j_{10,3}$, $j_{10,4}$;

— the classId variable specified in subclause 10.2.3;

— the value of **multiple_alignments_flag** specified in subclause 7.4.2;

— the **crps_flag** value specified in subclause 7.4.2 and the **cr_alg_ID** value specified in subclause 7.4.2.4.

The outputs of this process are:

— the variable numberOfAlignments containing the total number of alignments;

— the array numberOfSegmentAlignments[] containing the total number of segment-specific alignments;

— the array numberOfAlignmentsPairs[] containing the number of alignments of the rightmost read associated to each alignment of the leftmost read;

— the bi-dimensional array alignPtr[][] containing unsigned integer values representing, for each alignment, the indexes of the corresponding segment-specific alignments;

— the variable moreAlignments;

— the variable moreAlignmentsNextPos;

— the variable moreAlignmentsNextSeqId;

— the variable numberOfSegmentMappings[] calculated as follows:

  — if classId is equal to Class_U as specified in Table 37, and **crps_flag** is not equal to 0 and **cr_alg_ID** is equal to 2, 3 or 4 as specified in subclause 7.4.2, then the elements numberOfSegmentMappings[i] are set 1 for all values of i from 0 to numberOfRecordSegments – 1,

  — else numberOfSegmentMappings[] is set equal to numberOfSegmentAlignments[].

In the following clauses, subsequence0 is the array decoded_symbols[10][0] specified in subclause 12.6.2.2.

The decoding process shown in Table 64 applies.

Table 64 — Decoding process of mmap

Decoding step	Description
`if(classId != Class_U) {`	
`  if(multiple_alignment_flag == 0) {`	
`  numberOfSegmentAlignments[0] = 1`	Total number of alignments of the leftmost read.
`  } else {`	
`  numberOfSegmentAlignments[0] = subsequence0[j10,0++]`	
`  }`	
`} else {`	
`  numberOfSegmentAlignments[0] = 0`	
`}`	
`moreAlignments = 0`	
`if(unpairedRead || classId == Class_HM) {`	
`  numberOfAlignments = numberOfSegmentAlignments[0]`	
`  for(i = 0; i < numberOfAlignments; i++) {`	
`  alignPtr[i][0] = i`	
`  }`	
`} else if(classId == Class_U) {`	
`  if(numberOfRecordSegments > 1)`	
`  numberOfSegmentAlignments[1] = 0`	
`  numberOfAlignments = 0`	
`} else {`	
`  numberOfSegmentAlignments[1] = 0`	
`  k = 0, i = 0`	
`  while(i < numberOfSegmentAlignments[0]) {`	
`  if(multiple_alignments_flag == 0) {`	
`  numberOfAlignmentsPairs[i] = 1`	numberOfAlignmentsPairs[i] is the number of alignments of the rightmost read associated to the i^{th} alignments of the leftmost read.
`  } else {`	
`  numberOfAlignmentsPairs[i] = subsequence0[j10,0++]`	
`  }`	
`  j = 0`	
`  while (j < numberOfAlignmentsPairs[i]){`	
`  if(k != 0){`	Skip this for first alignment.
`  ptr = sequence1[j10,1++]`	
`  } else {`	

Table 64 *(continued)*

Decoding step	Description
ptr = 0	
}	
alignPtr[k][1] = numberOfSegmentAlignments[1] - ptr	
alignPtr[k][0] = i	
if(ptr == 0)	
numberOfSegmentAlignments[1]++	
j++, k++	
}	
i++	
}	
numberOfAlignments = k	
}	
if (multiple_alignments_flag == 1 && classId != Class_U && subsequence2[$j_{10,2}$++]){	More alignments on another reference sequence.
moreAlignments = 1	
moreAlignmentsNextSeqId = subsequence3[$j_{10,3}$++]	Identifier of the reference sequence an additional alignment of read 1 is mapped to in case of multiple alignments.
moreAlignmentsNextPos = subsequence4[$j_{10,4}$++]	Absolute mapping position of an additional alignment of read 1 on the reference sequence identified by moreAlignmentsNextSeqId.
}	

10.4.12.2 Multiple alignments on different sequences

It can happen that the alignment process finds alternative mappings to another reference sequence than the one where the first mapping is positioned.

For read pairs that are uniquely aligned, the **mmap** descriptor shall be used to represent the absolute read positions when there is for example a chimeric alignment with the mate on another chromosome (more alignments on another reference sequence case in Table 64). The **mmap** descriptor shall be used to signal the reference and the position of the next record containing further alignments for the same template. The last record (e.g. the third if alternative mappings are coded in three different access units) shall contain the reference and position of the first record.

10.4.13 msar

The **msar** (multiple segments alignment record) descriptor supports spliced reads and alternative alignments that contain indels or soft clips in case of class I data. It shall be present in a compliant bitstream when **multiple_alignments_flag** specified in subclause 7.4.2 is set to 1.

msar is intended to convey information related to secondary aligments on:

— a mapped segment length;

— a different mapping contiguity (i.e. e-cigar string) for additional alignment and/or spliced reads.

Each **msar** descriptor is an array of ASCII characters following the syntax specified in subclause 10.6.

The syntax, semantics and decoding process for **msar** descriptors are those for the **tokentype** descriptors specified in subclause 10.4.20.

The output of the decoding process of the **msar** descriptor is the array decodedStrings[] specified in subclause 10.4.20.5, when descriptor_ID is equal to 12.

Table 65 shows how the array of strings decodedMsar[][] is computed using the following additional input:

— the array numberOfSegmentAlignments[] calculated as specified in subclause 10.4.12;

— the variable numberOfAlignedRecordSegments calculated as specified in subclause 10.4.10;

— the array splitMate as specified in subclause 10.4.10.

For each genomic record the number encoded msar descriptors is equal to (numberOfAlignments – 1) * numberOfRecordSegments.

Table 65 — Computation of decodedMsar

Decoding step	Description
`k = 0`	
`for(i = 0; i < numberOfAlignedRecordSegments; i++) {`	
`  decodedMsar[][i] = {}`	Empty array.
`  for(j = 0;` `  j < numberOfSegmentAlignments[i]-1; j++){`	
`  if(splitMate[j][i] == 0) {`	
`  decodedMsar[j][i] = decodedStrings[k++]`	
`  }`	
`}`	

10.4.14rtype

10.4.14.1 General

The **rtype** descriptor is used to signal the subset of descriptors used to decode one unmapped read (class HM and class U) or read pair (Class U) in a genomic record as shown in Table 66.

The **rtype** descriptor also enables mixing reference-based and reference-less compression in the same dataset. In this scenario **rtype** = 0 signals reference-based encoded records, while **rtype** > 0 signals the set of descriptors to be used for reference-less compression (in this case descriptors refer to the computed reference, when needed).

The input to this process is the decoded_symbols[descriptor_ID] array specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 12 and the current value of $j_{12,0}$.

The output of this process is the decoded_symbols[descriptor_ID] array itself used by the decoder to select the appropriate descriptors for further decoding the genomic record.

Table 66 — Semantics of the rtype descriptor

rtype	cr_alg_ID	type of encoded reads	description
not used	1	Aligned reads with reference based compression only.	The entire dataset is encoded using reference based compression for mapped reads.
0	3	Aligned reads with both reference-based compression and reference-less compression.	The dataset contains both read (pairs) encoded using reference based compression and reference less compression. The decoding process for this Record uses the external or embedded reference according to the Class of the AU as specified in subclause 10.2.
1 .. 4	2, 4	Unmapped reads only.	1 = the decoding process is obtained by applying the decoding process specified in subclause 10.2.3, but without applying the steps specific to **clips** (subclause 10.4.7), **mscore** (subclause 10.4.11), **msar** (subclause 10.4.13) and **rgroup** (subclause 10.4.15) descriptors. 2 = the decoding process is obtained by applying the decoding process specified in subclause 10.2.4, but without applying the steps specific to **clips** (subclause 10.4.7), **mscore** (subclause 10.4.11), **msar** (subclause 10.4.13) and **rgroup** (subclause 10.4.15) descriptors. 3 = the decoding process is obtained by applying the decoding process specified in subclause 10.2.5, but without applying the steps specific to **mscore** (subclause 10.4.11), **msar** (subclause 10.4.13) and **rgroup** (subclause 10.4.15) descriptors. 4 = the decoding process is obtained by applying the decoding process specified in subclause 10.2.6, but without applying the steps specific to **clips** (subclause 10.4.7), **mscore** (subclause 10.4.11), **msar** (subclause 10.4.13) and **rgroup** (subclause 10.4.15) descriptors.
1, 2, 3, 4, 5, 6	3	Unmapped reads or aligned with reference less compression only.	1 = apply the decoding process specified in subclause 10.2.3. 2 = apply the decoding process specified in subclause 10.2.4. 3 = apply the decoding process specified in subclause 10.2.5. 4 = apply the decoding process specified in subclause 10.2.6. 5 = apply the decoding process specified in subclause 10.2.8. 6 = apply the decoding process specified in subclause 10.2.7.
5	2	Unmapped reads only.	The decoding process is specified in subclause 10.2.8.
5	4	Unmapped reads.	The decoding process is specified in subclause 10.2.8 where the U reads representing the reference sequence are used for compression but do not generate output records as specified in subclause 11.3.6.

In case of class HM, the mapped read is decoded by following the process for the mapped read of class HM specified in subclause 10.2, and the unmapped read is decoded following the decoding process specified in this subclause.

10.4.14.2 PushIn

When class U data are compressed using the "PushIn" computed reference algorithm specified in subclause 11.3.4, the decoding process shall follow the one described for classes P, N, M, I in subclauses 10.2.3 to 10.2.6 (for rtype values 1 to 4 respectively), or by ureads as described in subclause 10.2.8 (rtype equal to 5). The process to be followed is indicated by the descriptor rtype as specified in subclause 10.4.14.

Table 67 provides a description on the use of the **pos** and **pair** descriptors in this decoding process.

Table 67 — Semantics of the pos and pair descriptors for the PushIn algorithm

descriptor	semantics
pos	Matching position of the read on the PushIn computed reference, with coordinate as described in subclause 11.3.4.
pair	Used only for paired end reads. It associates a decoded read with its mate.

10.4.15 rgroup

The **rgroup** descriptor identifies the read group the genomic record belongs to.

The input to this process (see Table 68) is the decoded_symbols[descriptor_ID] array specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 13 and the current value of $j_{13,0}$.

The output of this process is the variable readGroupId.

Table 68 — Determination of the readGroupId value

Decoding step	Description
`readGroupId = subsequence0[j13,0++]`	

10.4.16 qv

10.4.16.1 General

The qv descriptor carries information to reconstruct the quality values.

The process for decoding quality values at a genomic position can be summarized informatively in the following steps:

1. Determine the quality value indexes at the genomic position.
2. Determine the quality value codebook identifier at this genomic position.
3. Use the quality value codebook identifier to select the quality value codebook for the genomic position.
4. Decode the quality value indexes by lookup in the quality value codebook.

10.4.16.2 Decoding process of the quality values of a genomic record

The inputs to this process are:

— the **qv_depth** value specified in subclause 7.4.2;

— the **qv_reverse_flag** value specified in subclause 7.4.2;

— the numberOfRecordSegments value computed in subclause 10.4.10;

— the current value of $j_{14,0}$;

— the decoded_symbols[descriptor_ID] array specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 14;

— the qvCodebookIndexesLoadFlag set to 1 at the beginning of each AU decoding process;

— the reverseComp array computed as specified in subclause 10.4.3.

The outputs of this process are the quality values of each nucleotide for each segment of the current genomic record and the value of qvCodebookIndexesLoadFlag.

In this description, subsequenceN is the subsequence identified by descriptor_subsequence_ID = N (i.e. subsequenceN = decoded_symbols[14][N]).

The decoding process for one genomic record is specified in Table 69:

Table 69 — Decoding process of the quality values of a genomic record

Decoding step	Description
decode_quality_values() {	
if(qvCodebookIndexesLoadFlag == 1){	
decode_qv_codebook_indexes()	As specified in Table 70.
qvCodebookIndexesLoadFlag = 0	
}	
for(tSeg = 0; tSeg < numberOfRecordSegments; tSeg++) {	
for(qs = 0; qs < qv_depth; qs++) {	
if($j_{14,0}$ < Size(subsequence0[])) {	
qvPresentFlag = subsequence0[$j_{14,0}$]	
$j_{14,0}$++	
} else {	
qvPresentFlag = 1	
}	
if(qvPresentFlag == 1) {	
decode_qvs()	As specified in Table 71.
qvString = “”	Empty string.
len = 0	
for(i=0; i < numberOfSplicedSeg[tSeg]; i++){	
revComp = reverseComp[i][0][tSeg]	
qvSplice = qualityValues[tSeq][qs][len,len+splicedSegLength[tSeg][i]-1]	
if(qv_reverse_flag && revComp) {	
qvString = strcat(qvString, reverseStr(qvSplice))	
}	
else{	
qvString = strcat(qvString, qvSplice)	
}	

Table 69 *(continued)*

Decoding step	Description
}	
qualityValues[tSeq][qs] = qvString	
} else {	
qualityValues[tSeg][qs] = ""	Empty string.
}	
}	
}	
}	

reverseStr(str) returns the reverse of the input string str where the n^{th} element of the reversed string reversedStr is computed as

reversedStr[n] = str[Size(str[]) - n - 1], for n in 0 .. Size(str[]) - 1.

10.4.16.3 Decoding processes of quality value codebook indexes and quality values of a segment

The inputs to these processes are:

— the decoded_symbols[descriptor_ID] array specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 14;

— the **qv_num_codebooks_total** and qvNumCodebooksAligned values specified in subclause 7.4.2.3;

— the current values of $j_{14,1}$ for the qvCodebookIds subsequence;

— the current values of $j_{14,N+2}$ with *N* ranging from 0 to qv_num_codebooks_total - 1 for the qv_num_codebooks_total subsequences for quality value indexes;

— the numBases variable equal to number of nucleotide of the segment for which the quality values shall be decoded;

— the basePos array containing the mapping positions relative to the AU_start_position of each nuclotide in the segment for which quality values shall be decoded, as specified in subclause 10.4.2;

— the classId variable specified in subclause 10.2.3;

— the value tSeg identifying the segment within the ISO/IEC 23092 series record for which the quality values shall be decoded;

— the value qs identifying the qs^{th} quality value string for the $tSeg^{th}$ segment within the ISO/IEC 23092 series record for which the quality values shall be decoded.

In this description, subsequenceN is the subsequence identified by descriptor_subsequence_ID = N (i.e. subsequenceN = decoded_symbols[14][N]).

The output of this process is the array of strings qualityValues[][], containing the quality values of each nucleotide in the segment for which the quality values shall be decoded.

In the case that qvNumCodebooksAligned is larger than 1, the value of subsequence1 shall be used to identify the quality value codebook for a genomic position of each aligned base. This quality value codebook is used to reconstruct all quality values at that genomic position. Multiple quality value codebooks can be used in one access unit. The variable qvCodeBookIds contains the indexes of the quality value codebooks associated to a given mapping position relative to AU_start_position as specified in subclause 9.6. The decoding process of qvCodeBookIds variable is specified in Table 70.

Table 70 — Decoding of quality value codebook indexes

Decoding step	Description
decode_qv_codebook_indexes() {	
if(qvNumCodebooksAligned > 1) {	
pos = 0	
for($j_{14,1}$ = 0; $j_{14,1}$ < Size(subsequence1[]); $j_{14,1}$++) {	
qvCodeBookIds[pos] = subsequence1[$j_{14,1}$]	The values qvCodeBookIds[pos] shall be in the range 0 (qvNumCodebooksAligned – 1).
pos++	
}	
}	
}	

The decoding process of the quality values is specified in Table 71.

Table 71 — Decoding process of quality values

Decoding step	Description
decode_qvs() {	
for(baseIdx = 0; baseIdx < numBases; baseIdx++) {	
if((classId == CLASS_I || classId == CLASS_HM) && ! isAligned(baseIdx)) {	Classes I and HM contain bases that are not aligned to the reference sequence, for which the last quality values codebook identifier reserved for unaligned data shall be used, as specified in subclause 7.4.2.3.
qvCodeBookId = qv_num_codebooks_total - 1	
} else if(classId == CLASS_U) {	
qvCodeBookId = 0	For records belonging to Class U, only one codebook shall be used, as specified in subclause 7.4.2.3.
} else if(qvNumCodebooksAligned > 1) {	
qvCodeBookId = qvCodeBookIds[basePos[baseIdx]]	
} else {	
qvCodeBookId = 0	
}	
qvCodeBookSubSeq = qvCodeBookId + 2	See subclause 7.4.2.3.
j = j14,qvCodeBookSubSeq	
j14,qvCodeBookSubSeq++	
qvIndex = decoded_symbols[14][qvCodeBookSubSeq][j]	
qualityValues[tSeq][qs][baseIdx] = qv_recon[qvCodeBookId][qvIndex]	
}	
}	

isAligned(baseIdx) returns 1 if the nucleotide at baseIdx is aligned to the reference sequence, otherwise 0. This means that isAligned(baseIdx) returns 0 for every nucleotide corresponding to a soft clip or to an insertion, or for nucleotides in the second segment of a genomic record in class HM.

Subclause 10.4.2 specifies how to calculate the absolute mapping position of the leftmost mapped base in each read, and thus every quality value, in a read. Figure 6 shows how quality value codebook identifiers relate to sequencing reads, quality values, reconstructed quality values, and genomic positions. The top third of the figure shows how nucleotides of four reads, including quality values, are mapped to genomic positions. The center of the figure shows how each genomic position is associated to a quality value codebook. According to the corresponding quality value index the reconstructed quality value is derived using the associated quality value codebook. The reconstructed quality values are shown in the bottom third of the figure.

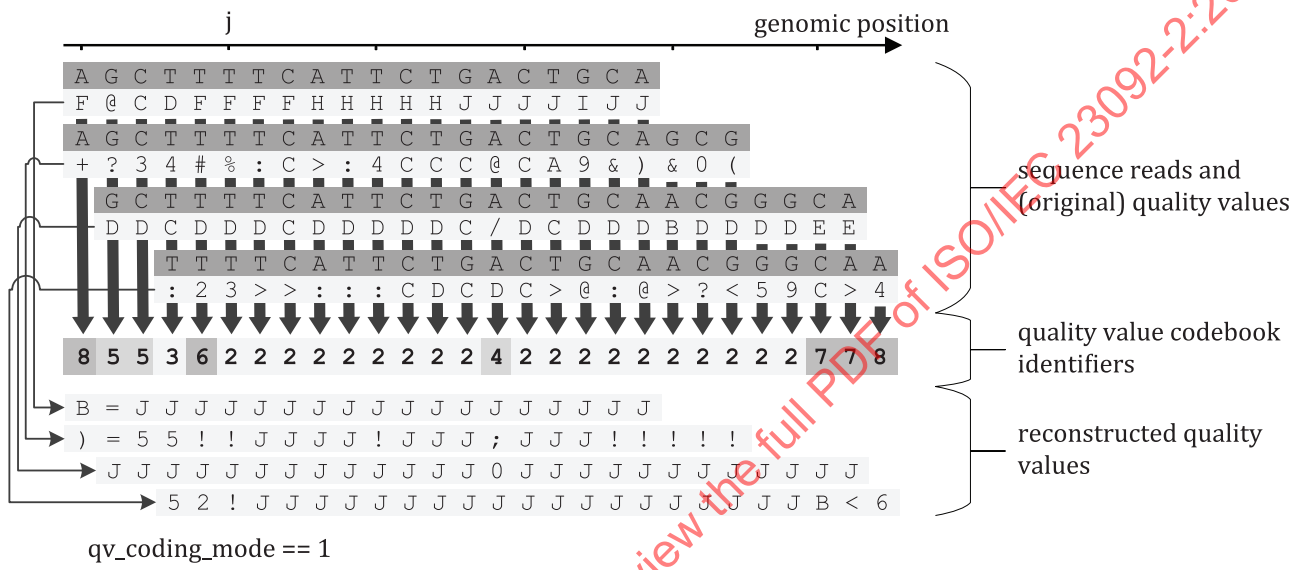

Figure 6 — Relationship between sequencing reads, quality values, reconstructed quality values and genomic positions

10.4.17 rname

Sequencing read identifiers are encoded as a sequence of **rname** descriptors (descriptor_ID equal to 15). Each **rname** descriptor is composed by tokens which have a type and possibly one or more parameters.

The syntax, semantics and decoding process for **rname** descriptors are those for the **tokentype** descriptors specified in subclause 10.4.20. The output of the decoding process of the **rname** descriptor for a i^{th} record in the access unit is the string variable readName equal to decodedStrings[i], using the array decodedStrings[] is specified in subclause 10.4.20.5. If **rname** descriptor is not present, readName is set to the empty string "".

An example of read identifiers tokenization is provided in Annex A.

10.4.18 rftp

The **rftp** descriptor

— shall be present only in access units of type 3 (class M) when cr_alg_ID specified in subclause 7.4.2 is set to 1;

— may be present when cr_alg_ID specified in subclause 7.4.2 is set to 3.

It shall not be present in any other case.

The inputs to this process are:

— the decoded_symbols[descriptor_ID] array specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 16 and the current value of $j_{16,0}$;

— the value **AU_start_position** as specified in subclause 7.5.1.2;

— the value **seq_start** as specified in subclause 7.3.2.

The output of this process is an array refTransfPos[] containing the positions of the transformations to be applied to a decoded raw reference as specified in subclause 11.3.3. The decoding process for **rftp** is specified in Table 72 for an entire access unit.

In this description, subsequenceN is the subsequence identified by descriptor_subsequence_ID = N (i.e. subsequenceN = decoded_symbols[16][N]).

Table 72 — Decoding process of the rftp descriptor

Decoding step	Description
`refTransfPos[0] = subsequence0[`$j_{16,0}$`++] + AU_start_position - seq_start`	Position of the first reference transformation in the current ref_sequence as specified in subclause 7.3.2.
`for(i = 1; i < Size(subsequence0); i++){`	
`refTransfPos[i] = refTransfPos[i - 1] + subsequence0[`$j_{16,0}$`++]`	
`}`	

10.4.19 rftt

The **rftt** descriptor

— shall be present only in access units of type 3 (class M) when cr_alg_ID specified in subclause 7.4.2 is set to 1;

— may be present when cr_alg_ID specified in subclause 7.4.2 is set to 3.

It shall not be present in any other case.

The inputs to this process are:

— the decoded_symbols[descriptor_ID] array specified in subclause 12.6.2.2 when **descriptor_ID** is equal to 17;

— the current value of $j_{17,0}$.

The output of this process is one array refTransfSubs[]containing the type of transformations to be applied to a decoded raw reference as specified in subclause 11.3.3.

In this description, subsequenceN is the subsequence identified by descriptor_subsequence_ID = N (i.e. subsequenceN = decoded_symbols[17][N]).

The output of the **rftt** descriptor decoding process shall be calculated following the process described in Table 73, after having decoded subsequence0 according to the decoding process specified in Table 124 using, if required by the said decoding process specified in Table 124 and by following the decoding process specified in subclause 12.6.2.3, the array refTransfPos[] decoded as specified in Table 72.

Table 73 — Decoding process of the rftt descriptor

Decoding step	Description
`for(i = 0; i < Size(subsequence0); i++){`	
`refTransfSubs[i] = S`$_{\text{alphabet ID}}$`[subsequence0[`$j_{17,0}$`++]]`	
`}`	

10.4.20 tokentype descriptors

10.4.20.1 General

The **msar** and **rname** share the same syntax, semantics and the decoding process specified in this subclause for the generic **tokentype** descriptor. The **tokentype** descriptor is not a genomic descriptor identified by a descriptor_ID, but a simple alias for **rname** and **msar** in the syntax, semantics and decoding process specified in this subclause.

tokentype descriptors can be of three types:

— strings,

— digits,

— single characters.

Both a read identifier and an e-cigar string are represented as set of differences and matches with respect to one of the previously decoded reads identifiers or e-cigar strings, respectively. The first identifier coded in an access unit always starts with a DIFF token followed by the value 0.

A **tokentype** descriptor can take the values listed in the table below. The **tokentype** descriptors can possibly be followed by one or more parameters.

Table 74 — The tokentype values and related semantics.

tokentype value	Token name	Parameters	Semantics
0	DUP	unsigned integer DISTANCE ranging from 0 to 2^{32}-1	Indicates that the current descriptor is an exact duplicate of the descriptor DISTANCE records ago, with "1" being the previously decoded descriptor and counting backwards in the list of previously decoded descriptors. The value of DISTANCE shall always refer to a descriptor coded in the current access unit. If a DUP token is found no further tokens are required to decode the descriptor. DUP can only occur at the first token position.
1	DIFF	unsigned integer DISTANCE ranging from 0 to 2^{32}-1	Indicates which descriptor this token is being compared against, usually "1" to indicate the previous descriptor. DIFF can only occur at the first token position. The first descriptor of a coded access units always starts with "DIFF 0".
2	STRING	st(v)	This is an arbitrary run of ASCII characters (as specified in ISO/IEC 10646) and need not be purely alphabetical. STRING is always null-terminated.
3	CHAR	c(1)	ASCII character as specified in ISO/IEC 10646.
4	DIGITS	unsigned integer ranging from 0 to 2^{32}-1	Numerical value no more than 9 digits long and not starting with a leading zero.
5	DELTA	unsigned integer ranging from 0 to 2^{8}-1	Numerical delta to a previous DIGITS value, between 0 and 255.

Table 74 *(continued)*

tokentype value	Token name	Parameters	Semantics
6	DIGITS0	an 8-bit length and a 32-bit unsigned integer	Fixed-width numerical value no more than 8 digits long, possibly starting with a leading zero.
7	DELTA0	8-bit unsigned integer	Numerical delta to a previous DIGITS0 value. The same fixed length is assumed.
8	MATCH	none	The next token value is identical to the token at the same position in the descriptor the currently decoded descriptor is compared against (regardless of token type).
9	DZLEN	unsigned integer DISTANCE ranging from 0 to 2^8-1	Used internally by DIGITS0 to code length.
10	END	none	Marker indicating the termination of the current tokentype descriptor sequence.

10.4.20.2 Decoding process

The input to this process is the block payload (as specified in subclause 7.5.1.3.3) for descriptor_ID equal to 11 or descriptor_ID equal to 15, which corresponds to the **msar** and **rname** descriptors respectively. The **encoded_tokentype**() structure of this block payload internally contains a list of compressed representation of **tokentype** descriptor sequences.

The output of this process is the list of decompressed representation of these **tokentype** descriptor sequences, which serve as input to the assembly process (specified in subclause 10.4.20.5) to reconstruct the msar descriptors or read identifiers respectively.

10.4.20.3 Syntax and semantics

The syntax of **encoded_tokentype**() is specified in Table 75.

Table 75 — Syntax of encoded_tokentype()

Syntax	Type
encoded_tokentype() {	
num_output_descriptors	u(32)
num_tokentype_sequences	u(16)
for(i = 0; i < num_tokentype_sequences; i++) {	
encoded_tokentype_sequence(i)	
}	
}	

num_output_descriptors specifies the number of descriptors (msar or read identifiers) encoded in the current block payload.

num_tokentype_sequences specifies the number of **tokentype** descriptor sequences in the current block payload.

encoded_tokentype_sequence(i) specifies the data structure containing the byte-aligned compressed representation of the i^{th} **tokentype** descriptor sequence. Its syntax is specified in Table 76.

Table 76 — Syntax of encoded_tokentype_sequence()

Syntax	Type
encoded_tokentype_sequence(i) {	
type_ID	u(4)
method_ID	u(4)
if(method_ID == 0) {	
ref_type_ID	u(16)
COP(i)	
}	
else {	
num_output_symbols	u7(v)
decode_tokentype_sequence(i, method_ID, num_output_symbols)	
}	
}	

type_ID specifies the type of the i^{th} **tokentype** descriptor sequence. This process internally maintains a state variable typeNum, which is initialized with -1 for every block payload of the descriptor and is incremented for every **tokentype** descriptor sequence with **type_ID = 0**. The current values of state variable typeNum and **type_ID** are then used to generate a "mapped" value of **type_ID** as specified in Table 77.

Table 77 — Computation of mappedTypeId

```
if(type_ID == 0)
   typeNum++
mappedTypeId = (typeNum<<4) | (type_ID & 0xf)
```

Every decoded tokentype descriptor for which **ref_type_ID** is equal to a previously calculated mappedTypeId shall be identical to the previously decoded tokentype descriptor.

method_ID specifies the compression method (among those listed in Table 78) used for the i^{th} **tokentype** descriptor sequence.

Table 78 — Description of compression methods for the tokentype descriptor sequence

method_ID	Description	
0	COP	The current tokentype descriptor sequence is an exact duplicate of a previously decoded tokentype descriptor sequence for which mappedTypeId is equal to the current **ref_type_ID** as specified in subclause 10.4.20.4.2.
1	CAT	The null coding, ideal for small data. Its syntax is specified in subclause 10.4.20.4.3.
2	RLE	Run length coding, ideal for long list of repeated symbols. Its syntax is specified in subclause 10.4.20.4.4.
3	CABAC_METHOD_0	The CABAC method 0 as specified in subclause 10.4.20.4.5. The signaling of its configuration parameters are specified in subclause 12.3.5.

Table 78 *(continued)*

method_ID	Description	
4	CABAC_METHOD_1	The CABAC method 0 as specified in subclause 10.4.20.4.5. The signaling of its configuration parameters are specified in subclause 12.3.5.
5	X4	A recursive decorrelation method to split a tokentype_sequence into four equisized interleaved subsequences (whenever size is divisible by 4), each of them being coded with one of the above methods except method_ID 0x0. Its syntax is specified in subclause 10.4.20.4.7.
0x6 .. 0xf	reserved	

ref_type_ID is the mappedTypeId of a previously decoded **tokentype** descriptor sequence of which payload of current **tokentype** descriptor sequence is an exact duplicate.

num_output_symbols signals the number of symbols to be reconstructed from the compressed payload of the i^{th} **tokentype** descriptor sequence.

decode_tokentype_sequence(i, method_ID, numOutputSymbols**)** specifies the syntax for decoding the i^{th} **tokentype** descriptor sequence (of size numOutputSymbols) using the decoding method indicated by method_ID. Its syntax is specified in Table 79.

Table 79 — Syntax of decode_tokentype_sequence()

Syntax
`decode_tokentype_sequence(i, methodID,  numOutputSymbols) {`
`  if(methodID == 1)`
`    CAT(i,  numOutputSymbols)`
`  else if(methodID == 2)`
`    RLE(i,  numOutputSymbols)`
`  else if(methodID == 3)`
`    CABAC_METHOD_0(i,  numOutputSymbols)`
`  else if(methodID == 4)`
`    CABAC_ METHOD_1(i,  numOutputSymbols)`
`  else if(methodID == 5)`
`    X4(i,  numOutputSymbols)`
`  else`
`  /* reserved for future use */`
`}`

10.4.20.4 Decoding process for compressed tokens

10.4.20.4.1 General

The input to this process is the data structure encoded_tokentype_sequence() specifying the byte-aligned compressed representation of the i^{th} **tokentype** descriptor sequence, which is decoded with one of the compression methods listed in Table 78 and specified in this subclause.

The output of this process is the decompressed representation of the i^{th} **tokentype** descriptor sequence.

10.4.20.4.2 COP

The input to this process is **ref_type_ID**, which shall be equal to a previously computed variable mappedTypeId of a previously decoded **tokentype** descriptor sequence as specified in Table 77.

The output of this process is a **tokentype** descriptor sequence, obtained by copying the already decoded reference **tokentype** descriptor sequence uniquely identified by **ref_type_ID**.

10.4.20.4.3 CAT

This subclause specifies the decoding process for the method CAT (see Table 80). The output of this process is a reconstructed **tokentype** descriptor sequence of size numOutputSymbols.

Table 80 — Decoding process for the method CAT

Decoding process	Type
CAT(i, numOutputSymbols) {	
for(j=0; j<numOutputSymbols; j++) {	
decoded_tokens[i][j]	u(8)
}	
}	

decoded_tokens[i][j] specifies the j^{th} token in the i^{th} decompressed **tokentype** descriptor sequence.

10.4.20.4.4 RLE

This subclause specifies the decoding process for the method RLE (see Table 81). The output of this process is a reconstructed **tokentype** descriptor sequence of size numOutputSymbols.

Table 81 — Decoding process for the method RLE

Decoding process	Type
RLE(i, numOutputSymbols) {	
for(j=0; j< numOutputSymbols ;) {	
tmp_value	u(8)
if(tmp_value == rle_guard_tokentype) {	
rle_len	u7(v)
if(rle_len == 0)	
decoded_tokens[i][j++] = rle_guard_tokentype	
else {	
tmp_value	u(8)
for(r=0; r< rle_len ; r++) {	
decoded_tokens[i][j++] = tmp_value	
}	
}	
} else	
decoded_tokens[i][j++] = tmp_value	
}	
}	

rle_guard_tokentype specifies the guard value signalled in decoder configuration for sequences of tokentype descriptors (see 12.3.5).

decoded_tokens[i][j] specifies the j^{th} token in the i^{th} decompressed **tokentype** descriptor sequence.

10.4.20.4.5 CABAC_METHOD_0

This subclause specifies the decoding process for the method CABAC_METHOD_0 used to decompress a **tokentype** descriptor sequence (see Table 82). The output of this process is a reconstructed **tokentype** descriptor sequence.

Table 82 — Decoding process for the method CABAC_METHOD_0

Decoding process	Type
`CABAC_METHOD_0(i, numOutputSymbols) {`	
`decoded_symbols[descriptor_ID][0] = decode_descriptor_subsequence(descriptor_ID, 0, numOutputSymbols, remainingPayloadSize)`	As specified in subclause 12.6.2.2.
`decoded_token[i][] = decoded_symbols[descriptor_ID][0][]`	
`}`	

decode_descriptor_subsequence(descriptor_ID, 0, numOutputSymbols, remainingPayloadSize) specifies the decoding process for the 0^{th} descriptor subsequence (of size numOutputSymbols) of the descriptor identified by descriptor_ID. For the CABAC_METHOD_0, the descriptor_ID is equal to 11 or 15.

decoded_symbols[descriptor_ID][0][] specifies the list of symbols decoded by decode_descriptor_subsequence(descriptor_ID, 0, numOutputSymbols).

remainingPayloadSize is the number of bytes remaining in the current block payload.

decoded_tokens[i] specifies the list of tokens in the i^{th} decompressed tokentype descriptor sequence.

10.4.20.4.6 CABAC_METHOD_1

This subclause specifies the decoding process for the method CABAC_METHOD_1 (see Table 83). The output of this process is a reconstructed **tokentype** descriptor sequence of size numOutputSymbols.

Table 83 — Decoding process for the method CABAC_METHOD_1

Decoding process	Type
`CABAC_METHOD_1(i, numOutputSymbols) {`	
`decoded_symbols[descriptor_ID][1] = decode_descriptor_subsequence(descriptor_ID, 1, numOutputSymbols, remainingPayloadSize)`	As specified in subclause 12.6.2.2.
`decoded_token[i][] = decoded_symbols[descriptor_ID][1][]`	
`}`	

decode_descriptor_subsequence(descriptor_ID, 1, numOutputSymbols, remainingPayloadSize) specifies the decoding process for the 1^{st} descriptor subsequence (of size numOutputSymbols) of the descriptor identified by descriptor_ID. For the CABAC_METHOD_1, the descriptor_ID is equal to 11 or 15.

decoded_symbols[descriptor_ID][1][] specifies the list of symbols decoded by decode_descriptor_subsequence(descriptor_ID, 1, numOutputSymbols).

remainingPayloadSize is the number of bytes remaining in the current block payload.

decoded_tokens[i][] specifies the list of tokens in the i^{th} decompressed **tokentype** descriptor sequence.

10.4.20.4.7 X4

This subclause specifies the decoding process for the method X4, which is be used to decompress a **tokentype** descriptor sequence (see Table 84). The output of this process is a reconstructed **tokentype** descriptor sequence of size numOutputSymbols.

Table 84 — Decoding process for the method X4

Decoding process	Type
X4(i, numOutputSymbols) {	
x4_method_IDs	u(16)
for (s=0; s<4; s++) {	
methodID = (x4_method_IDs >>(12 - (s*4))) & 0xf	
decoded_tokens_x4[s][] = decode_tokentype_sequence(s, methodID, numOutputSymbols/4)	As specified in subclause 10.4.20.3.
}	
/* Multiplexing of interleaved subsequences */	
for(j=0, j< numOutputSymbols ; j += 4) {	
for(s=0, s<4; s++) {	
decoded_tokens[i][j+s] = decoded_tokens_x4[s][j>>2]	
}	
}	
}	

x4_method_IDs specifies the four compression methods (among those listed in Table 78 except method_ID = 0) used to decompress the four interleaved subsequences, where the method_ID for the s^{th} subsequence can be derived as method_ID = (x4_method_IDs >>(12 - (s*4))) & 0xf.

decode_tokentype_sequence(s, method_ID, numOutputSymbols/4) decodes the s^{th} interleaved subsequence (of size numOutputSymbols/4) as a tokentype descriptor sequence using the decoding method indicated by method_ID.

decoded_tokens_x4[s][j] specifies the j^{th} byte token in the s^{th} decompressed interleaved subsequence.

decoded_tokens[i][j] specifies the j^{th} byte token in the i^{th} decompressed tokentype descriptor sequence.

10.4.20.5 Assembly of tokens

The input to this process (see Table 85) is the bi-dimensional array decoded_tokens[][], which is the decompressed representation of encoded_tokentype() specified in subclause 10.4.20.3, containing a list of **num_tokentype_sequences** decompressed **tokentype** descriptor sequences.

The output of this process is the data structure decodedStrings[] containing a list of either msar descriptors (when descriptor_ID is equal to 11) or read identifiers (when descriptor_ID is equal to 15) as strings.

Table 85 — Decoding process of tokentype descriptors into strings representing either msar descriptors or read identifiers

Decoding process
cIdx = 0
refIdx = 0
decodedStrings[ ] = {“”}
do {
t = 0
tokType = get_tok_type(decoded_tokens[t<<4])
distance = get_tok_int(decoded_tokens[t<<4 | tokType])
refIdx = cIdx – distance
if(tokType == 0) /* Token: DUP */
strcpy(decodedStrings [cIdx], decodedStrings [refIdx])
else { /* Token: DIFF */
for (t=1; t< num_tokentype_sequences; t++) {
tokType = get_tok_type(decoded_tokens[t<<4])
if(tokType == 10) /* Token: END */
break
tokStr = extract_tok_value (decoded_tokens, tokType, t, refIdx)
strcat(decodedStrings[cIdx], tokStr)
}
}
} while(cIdx < num_output_descriptors && strlen(decodedStrings[cIdx++]) > 0)

num_output_descriptors specifies the number of descriptors (msar or read identifiers) encoded in the current block payload. It is specified in 10.4.20.3.

get_tok_type(decoded_tokens[ ]) pops and returns one byte from data structure decoded_tokens[ ].

get_tok_int(decoded_tokens[ ]) pops four bytes from data structure decoded_tokens[ ] and decodes them as a 32-bit integer as specified in subclause 6.2.

strcpy(dst, src) specifies the string copying operation from the source string to the destination string.

strcat(dst, src) specifies the string concatenation operation of source string to the destination string.

strlen(str) returns the length of the input string.

extract_tok_value() pops and returns token value based on its type (as listed in Table 74) and the co-located tokens in the reference descriptor (msar or read identifier). The syntax of extract_tok_value() is described in Table 86.

Table 86 — Decoding process associated to a call to extract_tok_value()

Decoding process
`extract_tok_value(decoded_tokens[ ][ ], tokType, t, refIdx) {`
`  tokIdx = (t << 4) | tokType`
`  if(tokType == 2)  /* Token: STRING */`
`    tmp_str = get_tok_string(decoded_tokens[tokIdx])`
`  else if(tokType == 3)  /* Token: CHAR */`
`    tmp_str = get_tok_char(decoded_tokens[tokIdx])`
`  else if(tokType == 4)  /* Token: DIGITS */`
`    tmp_str = get_tok_digits(decoded_tokens[tokIdx])`
`  else if(tokType == 5)  /* Token: DELTA */`
`    tmp_str = get_tok_delta(decoded_tokens[tokIdx], refIdx)`
`  else if(tokType == 6)  /* Token: DIGITS0 */`
`    tmp_str = get_tok_digits0(decoded_tokens[tokIdx])`
`  else if(tokType == 7)  /* Token: DELTA0 */`
`    tmp_str = get_tok_delta0(decoded_tokens[tokIdx], refIdx)`
`  else if(tokType == 8)  /* Token: MATCH */`
`    tmp_str = get_tok_match(refIdx)`
`  return tmp_str`
`}`

get_tok_string(decoded_tokens[ ]) pops and returns a null terminated string from data structure decoded_tokens[ ] as described for token STRING in Table 74.

get_tok_char(decoded_tokens[ ]) pops and returns one ASCII character from data structure decoded_tokens[ ] as described for token CHAR in Table 74.

get_tok_digits(decoded_tokens[ ]) pops four bytes from data structure decoded_tokens[ ], decodes them as a 32-bit integer as specified in subclause 6.2, as described for token DIGITS in Table 74, and returns a string with the big-endian decimal representation of said integer.

get_tok_delta(decoded_tokens[ ], refIdx) pops a one byte delta value from data structure encoded_tokens[ ] as described for token DELTA in Table 74, sums said delta value and the digit value of the co-located DIGITS token in the reference descriptor (msar or read identifier) identified by refIdx, and returns a string with the big-endian decimal representation of the result of said sum.

get_tok_digits0(decoded_tokens[ ]) pops a one byte length value as DZLEN token and a four bytes value, decoded as a 32-bit integer as specified in subclause 6.2, as described for token DIGITS0 in Table 74, and returns a string with the big-endian zero-padded fixed-width decimal representation of said integer.

get_tok_delta0(decoded_tokens[ ], refIdx) pops a one byte delta value from data structure decoded_tokens[ ] as described for token DELTA in Table 74, sums said delta value and the digit value of the co-located DIGITS0 token in the reference descriptor (msar or read identifier) identified by refIdx, and returns. a string with the big-endian zero-padded fixed-width decimal representation of the result of said sum.

get_tok_match(refIdx) returns the token value of the co-located token in the reference descriptor (msar or read identifier) identified by refIdx as described for token MATCH in Table 74.

10.5 sequence

10.5.1 General

This subclause specifies how sequences of nucleotides are computed by a conformant decoder. For class HM, the mapped read is computed as specified in subclause 10.5.2 while the unmapped read as specified in subclause 10.5.3.

The inputs to this process are the variables numberOfRecordSegments and numberOfMappedRecordSegments calculated as specified in subclause 10.4.10.

The output of this process is the array splicedSequence[i][] (with 0 ≤ i < numberOfRecordSegments).

10.5.2 Aligned reads (Classes P, N, M, I, HM)

Additional input to this process are:

— the array mappingPos[0][] is computed as specified in subclause 10.2.3;

— the arrays numberOfSplicedSeg[], and splicedSegLength[][] computed as specified in subclause 10.4.9;

— the array splicedSegMappingPos[][] computed as specified in subclause 10.4.10;

— the array softClipSizes[][] computed as specified in subclause 10.4.7;

— the variable classId is computed as specified in subclause 10.2.3;

— The variable seqId set equal to **sequence_ID** as specified in subclause 7.5.1.2;

— The arrays **ref_sequence**[][] and **seq_start**[] as specified in subclause 7.3.

If **crps_flag** specified in Table 7 is equal to 1 and **cr_alg_ID** specified in Table 16 to is equal to 2, 3 or 4, in the decoding process specified in Table 87, seqId is set equal to 0, ref_sequence[seqId][] is set equal to refBuf[] specified in subclauses 11.3.4, 11.3.5, 11.3.6, respectively, and seq_start[seqId] is set equal to 0.

The decoding process specified in Table 87 shall be applied:

Table 87 — Decoding process of sequence[] array for aligned reads

Decoding step	Description
`for(i = 0; i < numberOfMappedRecordSegments; i++) {`	
`  for(j = 0; j < numberOfSplicedSeg[i]; j++) {`	
`  pRef = splicedSegMappingPos[i][j] - **seq_start**[seqId]`	
`  mappedLength = splicedSegLength[i][j]`	
`  if(classId == Class_I || classId == Class_HM) {`	
`  if(j == 0) {`	
`  mappedLength -= softClipSizes[i][0]`	
`  }` `  if(j == numberOfSplicedSeg[i] - 1) {`	
`  mappedLength -= softClipSizes[i][1]`	
`  }`	
`  }`	
`  splicedSequence[i][j] =` `  **ref_sequence**[seqId][pRef,` `  pRef + mappedLength - 1]`	

Table 87 *(continued)*

Decoding step	Description
`if(classId == Class_N) {`	
`processSplSegN(i, j)`	Specified in subclause 10.2.4.
`} else if(classId == Class_M) {`	
`processSplSegM(i, j)`	Specified in subclause 10.2.5.
`} else if(classId == Class_I || classId == Class_HM) {`	
`processSplSegI(i, j)`	Specified in subclause 10.2.6.
`}`	
`}`	
`}`	

10.5.3 Unmapped reads (Class HM, U)

The decoding process specified in Tables 88 and 89 shall be applied:

Table 88 — Decoding process of sequence[] array for unmapped reads

Decoding step	Description
`for(i = numberOfAlignedRecordSegments; i < numberOfRecordSegments; i++) {`	
`if(crps_flag == 0){`	
`decodeUreads(splicedSegLength[i][0])`	Specified in subclause 10.4.8.
`splicedSequence[i][0] = decodedUreads`	decodedUreads as specified in subclause 10.4.8.
`}else if(crps_flag == 1 && cr_alg_ID == 2){`	
`decode according to the process specified in subclause 11.3.4`	
`}else if(crps_flag == 1 && cr_alg_ID == 4){`	
`decode according to the process specified in subclause 11.3.6`	
`}`	
`}`	

Table 89 — Sequence decoding processes corresponding to crps_flag and cr_alg_ID

crps_flag	cr_alg_ID	sequence decoded as specified in subclause
0	—	10.4.8
1	2	11.3.4
1	4	11.3.6

10.6 e-cigar

10.6.1 Syntax

This subclause specifies an extended CIGAR (E-CIGAR) syntax for strings to be computed from sequences and related mismatches, indels, clipped bases and information on multiple alignments and spliced reads.

Alignments are described as a sequence of consecutive edit operations between the reference sequence and a sequence mapped onto the reference sequence.

Edit operations might involve skipping or replacing part of the sequence of either reference or read; due to this reason one has to keep track of a pointer *R* to the current position within the reference, and a pointer *r* to the current position within the read. They are both set to 0 at the beginning of the alignment process, the 0 of the reference being the position of the match.

Edit operations specified in this document are listed in Table 90.

Table 90 — Syntax of the ISO/IEC 23092 series E-CIGAR string

Operation	Semantics	E-CIGAR representation	Equivalent SAM CIGAR representation
Increment both pointer-to-reference *R* and pointer-to-read *r* by *n* positions (match).	`n` matching bases	`n=`	`nM` in older versions (not equivalent), `=` in recent versions
Replace nucleotide in the read with base `b` from the reference, increment pointer-to-reference *R* and pointer-to-read *r* by 1.	substitution of character `b` (`b` is present in the read and not in the reference) where `b` is one of the symbols of the alphabets defined in subclause 9.2.	`b`	`M` in older versions, `X` in recent versions (not equivalent)
Increment pointer-to-read *r* by *n* positions (insert from the read).	`n` bases are inserted in the read (not present in the reference)	`n+`	`nI`
Increment pointer-to-reference *R* by *n* positions (deletion of sequence `S` in the read).	`n` bases are deleted in the read (but present in the reference).	`n-`	`nD`
Increment pointer-to- read *r* by *n* positions (insertion in the read). Can only occur at beginning or end of read.	`n` soft clips	`(n)`	`nS`
Hard trim. Can only occur at beginning or end of read.	`n` hard clips	`[n]`	`nH`
Increment pointer-to-reference *R* by *n* positions, splice consensus observed (splice in the read).	An undirected splice of `n` bases.	`n*`	`nN`
Increment pointer-to-reference *R* by *n* positions, splice consensus observed on the forward strand (forward splice in the read).	A forward splice of `n` bases.	`n/`	Not existing.
Increment pointer-to-reference *R* by *n* positions, splice consensus observed on the reverse strand (reverse splice in the read).	A reverse splice of `n` bases.	`n%`	Not existing.

The general framework is illustrated in Table 91 shows an example of alignment with soft clips, deletions and substitutions.

Table 91 — Example of e-cigar string

```
00000000001111111111222222222333333333   Position in the reference
01234567890123456789012345678901234567

ACAGATATATCAGAGACCATACAGGAACATAACAGAC   Reference
AAAGATCTAT++++++++++CAGGTACATA          Read

0000000000          1111111111    Position in the read
0123456789          0123456789

E-CIGAR=(2)4=C3=11+4=T5=
```

10.6.2 Decoding process for the first alignment

10.6.2.1 General

The inputs to this process are:

— readLength[] array computed as specified in subclause 10.2.3;

— the classId variable specified in subclause 10.2.3;

— the numberOfAlignedRecordSegments variable specified in subclause 10.4.10.

For classId equal to Class_N, Class_M, Class_I, and Class_HM:

— the mismatchOffsets[][] array computed as specified in subclause 10.4.5;

— the numMismatches[] array computed as specified in subclause 10.4.5.

If **cr_alg_ID** specified in subclause 11.3 is set to 1, for classId equal to Class_M mismatchOffsets[][] and numMismatches[] are pre-processed as per subclause 10.6.4 prior to being decoded as specified in this subclause.

For classId equal to Class_M, Class_I, and Class_HM:

— the mismatches[][] arrays computed as specified in subclause 10.4.6.

If **cr_alg_ID** specified in subclause 11.3 is set to 1, for classId equal to Class_M mismatches[][] is pre-processed as per subclause 10.6.4 prior to being decoded as specified in this subclause.

For classId equal to Class_I and Class_HM:

— the mismatchTypes[] array computed as per subclause 10.4.6;

— the softClips[][][] arrays, the softClipSizes[][] array, and the hardClips[][] array computed as specified in subclause 10.4.7.

The output of this process is the array of strings ecigarString[], and the array of the corresponding string lengths ecigarLength[].

In this subclause, the decoding process uses strings, where strings are sequences of a given length of universal coded character set (UCS) transmission format-8 (UTF-8) characters as specified in ISO/IEC 10646 of a given length.

In this subclause the following strings operators are defined:

arraytostr(a, l)	returns a string of length l created by copying the first l characters from array a, where a is a one-dimensional array of characters
strtoc(s)	returns all characters in string s in a sequence compliant with c(n) data type specified in subclause 6.3, where n corresponds to the length of string s
'...'	returns a string composed by the characters between the quotes
inttostr(i)	returns a string containing the base-10 representation of the integer
strcat(s1, ..., s*N*)	returns the concatenation of the strings from s1 to s*N*. If any of the input strings s1 through s*N* is a single character, it is considered a string of length 1
strlen(s)	returns the length of string s

10.6.2.2 Decoding process without spliced reads

When the **spliced_reads_flag** syntax element specified in subclause 7.4.2 is equal to 0, the decoding process of e-cigar strings is specified in Table 92.

Table 92 — Decoding process for the e-cigar strings of a genomic record without spliced reads

Decoding step	Description
`for(s = 0; s < numberOfAlignedRecordSegments; s++) {`	
`  if(classId == Class_P){`	Class P.
`  mmOffsets = {}`	Empty array.
`  mms = {}`	Empty array.
`  mmTypes = {}`	Empty array.
`  decodeECigarMismatches(classId, readLength[s], 0, mmOffsets, mms, mmTypes)`	As specified in Table 93.
`  ecigar = decodedEcigar`	decodedEcigar computed as specified in Table 93.
`  }`	
`  else if(classId == Class_N){`	Class N.
`  mms = {}`	Empty array.
`  mmTypes = {}`	Empty array.
`  decodeECigarMismatches(classId, readLength[s], numMismatches[s], mismatchOffsets[s], mms, mmTypes)`	As specified in Table 93.
`  ecigar = decodedEcigar`	decodedEcigar computed as specified in Table 93.
`  }`	
`  else if(classId == Class_M){`	Class M.
`  mmTypes = {}`	Empty array.
`  decodeECigarMismatches(classId, readLength[s], numMismatches[s], mismatchOffsets[s], mismatches[s], mmTypes)`	As specified in Table 93.
`  ecigar = decodedEcigar`	decodedEcigar computed as specified in Table 93.

Table 92 *(continued)*

Decoding step	Description
`}`	
`else if(classId == Class_I || classId == Class_HM){`	Classes I or HM.

`leftSoftClips =`
`arraytostr(softClips[s][0][],`
`softClipSizes[s][0])`	

`rightSoftClips =`
`arraytostr(softClips[s][1][],`
`softClipSizes[s][1])`	

`leftHardClips = hardClips[s][0]`	
`rightHardClips = hardClips[s][1]`	
`mappedLength = readLength[s]` `- strlen(leftSoftClips) - strlen(rightSoftClips)`	

`decodeECigarMismatches(classId, mappedLength,`
`numMismatches[s], mismatchOffsets[s],`
`mismatches[s], mismatchTypes[s])`
	As specified in Table 93.

`ecigar = decodedEcigar`	decodedEcigar computed as specified in Table 93.
`if(strlen(leftSoftClips) != 0) {`	

`ecigar = strcat(`
`'(', inttostr(strlen(leftSoftClips)), ')',`
`ecigar)`
	Soft clips are present before the leftmost mapped base.

`}`	
`else if(leftHardClips != 0) {`	

`ecigar = strcat(`
`'[', inttostr(leftHardClips), ']',`
`ecigar)`
	Hard clips are present before the leftmost mapped base.

`}`	
`if(strlen(rightSoftClips) != 0) {`	
`ecigar = strcat(ecigar,` `'(', inttostr(strlen(rightSoftClips)), ')')`	Soft clips are present after the rightmost mapped base.
`}`	
`else if(rightHardClips != 0) {`	
`ecigar = strcat(ecigar,` `'[', inttostr(rightHardClips), ']')`	Hard clips are present after the rightmost mapped base.
`}`	
`}`	
`ecigarString[s] = strtoc(ecigar)`	
`ecigarLength[s] = strlen(ecigar)`	
`}`	

Table 93 — Decoding process for the mismatches within one e-cigar string

Decoding step	Description
decodeECigarMismatches(classId, len, mmNumber, mmOffsets, mms, mmTypes) {	
ecigar = “”	Empty string.
if(classId == Class_P){	Class P.
ecigar = strcat(inttostr(len), ‘=’)	
}	
else if(classId == Class_N){	Class N.
previousOffset =0	
i = 0	
while(i < mmNumber){	
delta = mmOffsets[i] – previousOffset	
previousOffset = mmOffsets[i] + 1	
if(delta == 0){	
ecigar = strcat(ecigar, ‘N’)	
} else {	
ecigar = strcat(ecigar, inttostr(delta), ‘=’)	
ecigar = strcat(ecigar, ‘N’)	
}	
i++	
}	
delta = len – previousOffset	
if(delta > 0) {	
ecigar = strcat(ecigar, inttostr(delta), ‘=’)	
}	
}	
else if(classId == Class_M){	Class M.
previousOffset = 0	
i = 0	
while(i < mmNumber){	
delta = mmOffsets[i] – previousOffset	
previousOffset = mmOffsets[i] + 1	
if(delta == 0){	
ecigar = strcat(ecigar, mms[i]))	
} else {	
ecigar = strcat(ecigar, inttostr(delta), ‘=’)	
ecigar = strcat(ecigar, mms[i])	
}	
i++	
}	
delta = len – previousOffset	
if(delta > 0) {	
ecigar = strcat(ecigar, inttostr(delta), ‘=’)	
}	
}	

Table 93 *(continued)*

Decoding step	Description
else if(classId == Class_I || classId == Class_HM)){	Classes I or HM.
previousOffset = 0	
i = 0	
while(i < mmNumber) {	
count = 0	
delta = mmOffsets[i] - previousOffset	
previousOffset = mmOffsets[i]	
if(delta > 0) {	
ecigar = strcat(ecigar, inttostr(delta), '=')	
delta = 0	
}	
if(mmTypes[i] == 0) {	Substitution.
ecigar = strcat(ecigar, mms[i]))	
previousOffset = mmOffsets[i] + 1	
i++	
}	
else if(mmTypes[i] == 1) {	Insertion.

while(i < mmNumber
&& mmTypes[i] == 1
&& mmOffsets[i] - previousOffset
== 0) {	

previousOffset = mmOffsets[i] + 1	
count++, i++	
}	
ecigar = strcat(ecigar, inttostr(count))	
ecigar = strcat(ecigar, '+')	
}	
else if(mmTypes[i] == 2) {	Deletion.

while(i < mmNumber
&& mmTypes[i] == 2
&& mmOffsets[i] - previousOffset
== 0) {	

previousOffset = mmOffsets[i]	
count++, i++	
}	
ecigar = strcat(ecigar, inttostr(count))	
ecigar = strcat(ecigar, '-')	
}	
}	

Table 93 *(continued)*

Decoding step	Description
delta = len – previousOffset	
if(delta > 0) {	
ecigar = strcat(ecigar, tostr(delta), '=')	
}	
}	
decodedEcigar = ecigar	
}	

10.6.2.3 Decoding process with spliced reads

When the **spliced_reads_flag** syntax element specified in subclause 7.4.2 is equal to 1, the e-cigar strings are decoded as follows.

Additional input to this process are:

For classId equal to Class_N, Class_M, Class_I, and Class_HM:

— the numberOfSplicedSeg[], splicedSegMappedLength[][] and splicedSegLength[][] arrays computed as specified in subclause 10.4.9;

— the splicedSegMismatchOffsets[][][], splicedSegMismatchNumber[][] and splicedSegMismatchIdx[][] arrays computed as specified in subclause 10.4.5;

— the array splicedSegMappingPos[][] computed as specified in subclause 10.4.10;

— the array reverseComp[][][] computed as specified in subclause 10.4.3

The decoding process is specified in Table 94.

Table 94 — Decoding process for the e-cigar strings of a genomic record with spliced reads.

Decoding step	Description
for(s = 0; s < numberOfAlignedRecordSegments; s++) {	
if(classId == Class_P){	Class P.
mmOffsets = {}	Empty array.
mms = {}	Empty array.
mmTypes = {}	Empty array.
decodeECigarMismatches(classId, readLength[s], 0, mmOffsets, mms, mmTypes)	As specified in Table 93.
ecigar = decodedEcigar	decodedEcigar computed as specified in Table 93.
}	
else if(classId == Class_N){	Class N.
mms = {}	Empty array.
mmTypes = {}	Empty array.
decodeECigarMismatches(classId, readLength[s], numMismatches[s], mismatchOffsets[s], mms, mmTypes)	As specified in Table 93.

Table 94 *(continued)*

Decoding step	Description
`ecigar = decodedEcigar`	decodedEcigar computed as specified in Table 93.
`}`	
`else if(classId == Class_M){`	Class M.
`mmTypes = {}`	Empty array.
`decodeECigarMismatches(classId, readLength[s], numMismatches[s], mismatchOffsets[s], mismatches[s], mmTypes)`	As specified in Table 93.
`ecigar = decodedEcigar`	decodedEcigar computed as specified in Table 93.
`}`	
`else if(classId == Class_I || classId == Class_HM){`	Classes I or HM.
`leftSoftClips = arraytostr(softClips[s][0][], softClipSizes[s][0])`	
`rightSoftClips = arraytostr(softClips[s][1][], softClipSizes[s][1])`	
`leftHardClips = hardClips[s][0]`	
`rightHardClips = hardClips[s][1]`	
`ecigar = ""`	Empty string.
`for(i = 0; i < numberOfSplicedSeg[s]; i++) {`	
`length = splicedSegLength[s][i]`	
`if(i == 0) {`	
`length -= softClipSizes[s][0]`	
`}`	
`if(i == (numberOfSplicedSeg[s] - 1)) {`	
`length -= softClipSizes[s][1]`	
`}`	
`if(i > 0) {`	
`spliceOffset = splicedSegMappingPos[s][i] - splicedSegMappingPos[s][i - 1] - splicedSegMappedLength[s][i - 1]`	
`ecigar = strcat(ecigar, inttostr(spliceOffset))`	
`if(reverseComp[i][s][0] == 0) {`	
`ecigar = strcat(ecigar, "/")`	Forward splice.
`} else if(reverseComp[i][s][0] == 1)`	
`ecigar = strcat(ecigar, "%")`	Reverse splice.
`} else if(reverseComp[i][s][0] == 2)`	
`ecigar = strcat(ecigar, "*")`	Undirected splice.
`} else {`	
`/* reserved */`	
`}`	
`}`	

Table 94 *(continued)*

Decoding step	Description
mmStartIdx = splicedSegMismatchIdx[s][i]	
mmEndIdx = mmStartIdx + splicedSegMismatchNumber[s][i] – 1	
decodeECigarMismatches(classId, length, splicedSegMismatchNumber[s][i], splicedSegMismatchOffsets[s][i], mismatches[s][mmStartIdx, mmEndIdx], mismatchTypes[s][mmStartIdx, mmEndIdx])	As specified in Table 93.
ecigar = strcat(ecigar, decodedEcigar)	decodedEcigar computed as specified in Table 93.
}	
if(strlen(leftSoftClips) != 0) {	
ecigar = strcat('(', inttostr(strlen(leftSoftClips)), ')', ecigar)	Soft clips are present before the leftmost mapped base.
}	
else if(leftHardClips != 0) {	
ecigar = strcat('[', inttostr(leftHardClips), ']', ecigar)	Hard clips are present before the leftmost mapped base.
}	
if(strlen(rightSoftClips) != 0) {	
ecigar = strcat(ecigar, '(', inttostr(strlen(rightSoftClips)), ')')	Soft clips are present after the rightmost mapped base.
}	
else if(rightHardClips != 0) {	
ecigar = strcat(ecigar, '[', inttostr(rightHardClips), ']')	Hard clips are present after the rightmost mapped base.
}	
}	
ecigarString[s] = strtoc(ecigar)	
ecigarLength[s] = strlen(ecigar)	
}	

10.6.3 Decoding process for other alignments

For all alignments other than the first one, the e-cigar strings are decoded as specified in subclause 10.4.13.

10.6.4 Reference transformation

When **cr_alg_ID** specified in subclause 11.3 is set to 1, for records belonging to class Class_M, the input arrays mismatchOffsets[][], mismatches[][], and numMismatches[] specified in subclauses 10.4.5 and 10.4.6 shall be pre-processed according to the process described in Table 95 prior to being decoded as specified in subclause 10.6.2.

Additional input to the process is:

— the array mappingPos[][] computed as specified in subclauses 10.4.2 and 10.4.10;

— the readLen[] array computed as specified in subclause 10.4.9;

— the array refSequence equal to **ref_sequence**[i] specified in subclause 7.4.2 where i is equal to **ref_sequence_ID** as specified in subclause 7.5.1;

— the array refTransfOrigSymbols computed in subclause 11.3.3;

— the variables numberOfRecordSegments computed as specified in subclause 10.4.10.

The output of the process are the modified arrays mismatchOffsets[][], mismatches[][], and numMismatches[].

Table 95 — Pre-processing process when cr_alg_ID is equal to 1

Processing step	Description
`for(s = 0; s < numberOfRecordSegments; s++) {`	
`  mPos = mappingPos[0][s] - seq_start`	
`  newMismatchOffsets[] = {}` `  newMismatches[] = {}`	Empty arrays.
`  i = 0, j = 0, k = 0`	
`  while(i < Size(refTransfPos) &&` `  refTransfPos[i] < mPos) i++`	Search for the transformations in the leftmost read range.
`  while(i < Size(refTransfPos) &&` `  refTransfPos[i] < mPos + readLength[s]){`	

`  if(j ≥ numMismatches[s] ||`
`  refTransfPos[i] - mPos <`
`  mismatchOffsets[s][j]){`
	One ref transformation found before the next mismatch position.

`  newMismatchOffsets[k] =` `  refTransfPos[i] - mPos`	
`  newMismatches[k] = refSequence[refTransfPos[i]]`	Read the base in the ref sequence.
`  i++, k++`	
`  }`	
`  else if(refTransfPos[i] - mPos ==` `  mismatchOffsets[s][j]){`	One substitution in the read found at the same place as the reference transformation.
`  if(mismatches[s][j] !=` `  refTransfOrigSymbols[i]){`	Store it only if different from the original reference.
`  newMismatchOffsets[k] = mismatchOffsets[s][j]`	
`  newMismatches[k] = mismatches[s][j]`	
`  k++`	
`  }`	
`  i++, j++`	
`  } else {`	

`  while(j < numMismatches[s] &&`
`  refTransfPos[i] - mPos >`
`  mismatchOffsets[s][j]){`
	Copy all mismatches until the next reference transformation.

Table 95 *(continued)*

Processing step	Description
newMismatchOffsets[k] = mismatchOffsets[s][j]	
newMismatches[k] = mismatches[s][j]	
k++, j++	
}	
}	
}	
while(j < numMismatches[s]){	Copy the remaining mismatches if any.
newMismatchOffsets[k] = mismatchOffsets[s][j]	
newMismatches[k] = mismatches[s][j]	
k++, j++	
}	
mismatchOffsets[s] = newMismatchOffsets	
numMismatches[s] = k	
mismatches[s] = newMismatches	
}	

11 Representation of reference sequences

The reference sequence is usually part of an available reference genome (split into chromosomes and other sequences), but can in principle have any origin. With respect to a bitstream compliant with ISO/IEC 23092-1, the following types of reference sequences are supported:

— **External Reference**: the reference sequence is coded as an independent resource either locally or remotely and shall be retrieved to enable the decoding of the bitstream.

— **Embedded Reference**: the reference sequence is coded within the bitstream as dataset.

— **Computed Reference**: the reference sequence can be computed using the information contained in the sequencing reads coded in the bitstream.

In the scope of this document embedded and computed references are referred to as internal references.

11.1 External reference

The reference used for compression is not included in the bitstream. A mechanism for unique identification is specified in ISO/IEC 23092-1.

11.2 Embedded reference

The reference is stored in the bitstream as dataset as specified in ISO/IEC 23092-1.

11.3 Computed reference

11.3.1 General

A computed reference is used:

— to improve compression efficiency by modifying an available external reference before decoding sequence data, or

— to encode aligned sequencing reads without using the reference sequences used for alignment, or

— to encode raw (unmapped) reads.

In case of aligned reads it can be beneficial to support encoding and decoding without requiring access to the reference sequences used for alignment.

This approach uses the sequencing reads to be encoded to build a local consensus assembly to perform reference-based encoding. In this case all reads shall be encoded using class U descriptors, but the classification in P, N, M, I and HM classes shall be preserved.

When sequencing reads are encoded using a computed reference, the **rtype** descriptor currently specified in subclause 10.4.11 shall be used as specified in Table 96 to:

1. signal the set of descriptors needed to decode the current record,
2. signal the type of reference (embedded reference or computed reference) needed to decode the current record.

11.3.2 Supported Algorithms

Table 96 specifies the supported reference computation algorithms. **cr_alg_ID** is specified in subclause 11.3.

Table 96 — Supported reference computation algorithms

cr_alg_ID	Name	Description
0		reserved
1	RefTransform	To improve compression efficiency, an available external reference is modified before decoding sequence data. This algorithm applies only to aligned data as described in subclause 11.3.3.
2	PushIn	The reference is created by simple concatenation of already decoded reads, with padding. This is described in subclause 11.3.4.
3	Local assembly	The reference is created by performing a local assembly. This algorithm applies only to aligned data as described in subclause 11.3.5.
4	Global assembly	The reference used to perform reference based decoding is encoded in each AU as sequence of ureads descriptors. This is described in subclause 11.3.6.
5 ... 255		reserved

11.3.3 Reference transformation

The input to this process is the **ref_sequence**[seqId] array specified in subclause 7.4.2, with seqId equal to **ref_sequence_ID** as specified in subclause 7.5.1, and the arrays refTransfPos[],and refTransSubs[] computed as specified in subclauses 10.4.18 and 10.4.19 respectively.

The output of this process is the modified **ref_sequence**[seqId] array computed by applying the decoding process shown in Table 97 and a refTransfOrigSymbols[] array containing the substituted symbols in the original reference.

Table 97 — Reference transformation process

Transformation step	Description
`len = Size(refTransfPos[])`	
`refTransfOrigSymbols[] = {}`	Empty array.
`for (i = 0; i < len; i++){`	
`refTransfOrigSymbols[i] = ref_sequence[seqId][refTransfPos[i]]`	Save the symbol in the reference before transformation.
`ref_sequence[seqId][refTransfPos[i]] = refTransSubs[i]`	Substitution.
`}`	

When **cr_alg_ID** is equal to 1 the decoder shall first apply the reference transformation described in Table 97 to the raw reference structure received as input and then use it for reference-based decoding as specified in subclause 10.2.

11.3.4 PushIn

11.3.4.1 General

The reference is created by pushing into a reference buffer refBuf[] of size crBufSize, i.e. concatenating, already decoded reads. In this subclause reads are specified as the sequences computed as output of the process described in Table 66 for **cr_alg_ID** equal to 2. The reference is built from crBufNumReads decoded reads, each composed by a sequence of symbols from one of the alphabets as specified in subclause 9.2.

A decoded read is pushed in front of the computed reference buffer only if it is different from the previous one. The computed reference obtained in this way is padded at its beginning and its end.

11.3.4.2 Process for the construction of the reference

The inputs to this process are:

— the buffer refBuf[] of size crBufSize specified in subclause 11.3.4.3 which contains crBufNumReads;

— cr_buf_max_size as specified in subclause 7.4.2.4;

— cr_pad_size as specified in subclause 7.4.2.4;

— signature_flag, num_signatures, signature_length[] and signature[] fields in the access unit header as specified in subclause 7.5.1.2;

11.3.4.3 Initialization of the reference

At the start of the decoding process of an AU set crBufSize equal to 2*cr_pad_size and crBufNumReads equal to 0.

If signature_flag is equal to 1 and num_signatures is bigger than 0:

1. insert the contents of signature[0] to the refBuf[] (at position cr_pad_size), increment crBufNumReads by 1 and increment crBufSize by signature_length[0];

2. for each remaining signature, if (crBufSize + 2* cr_pad_size + the size of the previous signature) is greater than cr_buf_max_size, oldest signatures are pushed out of the buffer refBuf[] and crBufSize decremented of the length in nucleotides of each pushed out signature until (crBufSize + 2* cr_pad_size + the size of the current signature) is smaller than or equal to cr_buf_max_size. Push the

current signature in front of the previous signature and increment crBufSize with the length in nucleotides of the current signature.

11.3.4.4 Update of the reference

The output of this process is the updated buffer refBuf[] and the updated variable crBufSize.

This process is skipped when the last decoded read perfectly matches the previously pushed read into the refBuf[] in the sense that all the following conditions are all satisfied:

— rtype value of the last decoded read is smaller or equal to 2

— crBufNumReads is greater than 0

— lengths of both reads are equal

This process consists of the following steps:

1. If (crBufSize + the size of the last decoded read) is greater than cr_buf_max_size, oldest reads are pulled out of the buffer refBuf[] and crBufSize decremented of the length in nucleotides of each pushed out read until (crBufSize + the size of the last decoded read) is smaller than or equal to cr_buf_max_size. Decrement crBufNumReads by the number of reads pushed out of the refBuf[].

2. If reads are present in the buffer, the whole buffer, except the leftmost cr_pad_size positions, is pushed back until the leftmost base of the oldest read is at cr_pad_size position.

3. The last decoded read, decoded as described in Table 66 for **cr_alg_ID** equal to 2, is pushed in the refBuf[] after the last decoded read already in the refBuf[], crBufNumReads is incremented by 1 and crBufSize is incremented of the length in nucleotides of the pushed in read.

4. cr_pad_size rightmost remaining positions of refBuf[] are padded with the rightmost base of the newly inserted read.

5. cr_pad_size leftmost positions of refBuf[] are padded with the leftmost base of the oldest read remaining in refBuf[].

The leftmost position in the buffer shall have position 0; by consequence the leftmost base of the oldest read shall have position cr_pad_size.

The output of the computation process described above is a reference sequence contained in the array refBuf[] which shall be used to decode the next genomic records contained in the current AU corresponding to values of **rtype** not equal to 5 as specified in subclause 10.4.14.

The refBuf[] shall be deleted at the end of the decoding process of each AU.

If the reverseComp[][][] flag (as specified in subclause 10.4.3) corresponding o the last decoded read is 1, output the read as reverse-complemented as specified in subclause 9.4 after that this has been pushed to the computed reference.

11.3.5 Local assembly

11.3.5.1 General

The reference is created by computing a local sliding consensus reference sequence. This can be seen as equivalent to performing a local assembly. A local assembly is created by collecting all bases mapping to a unique genomic position and by deriving the consensus base at that position through a majority vote. In this subclause reads are specified as the sequences computed as output of the process described in subclause 10.5.2 This algorithm applies only to aligned data as described in subclause 11.3.5.2.

An array crBuf[][] is built during the decoding process. A number of already decoded reads may be needed and are stored in the array crBuf[][]. The number of decoded reads stored in the array crBuf[]

[] is stored in the variable crBufNumReads. The current size in bytes of the array crBuf[][] is stored in the variable crBufSize.

If the optional **rftp** and **rftt** descriptors are present, an additional output of this decoding process is a raw_reference$_{output}$ structure (specified in subclause 7.3.2) containing the computed Local Assembly reference specific to current Access Unit, as specified in point 6 of subclause 11.3.5.3 and in subclause 11.3.5.4.

11.3.5.2 Process for adding a decoded aligned read to the list crBuf

The inputs to this process is an array crBuf[][] which contains crBufNumReads reads of size in bytes equal to crBufSize.The output of this process is the updated array crBuf[][] and the updated variables crBufNumReads and crBufSize.

This process consists of the following steps:

1. If the variable crBufSize plus the length in bases of the already decoded aligned read is greater than cr_buf_max_size, the oldest reads are removed from the array crBuf[][] until crBufSize plus the size of the already decoded aligned read is smaller than or equal to cr_buf_max_size.
2. The last decoded read is added to the array crBuf[][] as newest read.

11.3.5.3 Process for the construction of the reference

The input to this process is an array crBuf[][] containing at least one aligned read and the position on the reference sequence of each nucleotide.

The output of this process is an array refBuf[] containing a sequence of consensus symbols.

For each position covered by aligned reads in the array crBuf[][], the consensus symbol is derived as follows:

1. Collect all bases mapping to the current position.
2. Count the occurrences of each symbol.
3. If two symbols s_i, s_j (with i < j indexes of one of the alphabets specified in subclause 9.2) have the same maximum number of occurrences, then select s_i as consensus symbol.
4. Otherwise, select the symbol with the maximum number of occurrences as consensus symbol.
5. Append the consensus symbol to the array refBuf[].
6. If the optional **rftp** and **rftt** descriptors are present, copyrefBuf[] into ref_sequence$_{output}$[seqId][] in a raw_reference$_{output}$ structure (specified in subclause 7.3.2) according to the mapping position.

The result of the decoding process described above is a reference sequence contained in the array refBuf[] which shall be used to decode the genomic records contained in the current AU corresponding to values of **rtype** not equal to 0 or 5 as specified in subclause 10.4.14.

11.3.5.4 Decoding process for rftp and rftt

When **cr_alg_ID** is equal to 3, if the optional descriptors **rftp** and **rftt** are present in the bitstream, they shall be used to reconstruct the original reference used for sequence alignment for the records in current Access Unit. The decoder shall apply a transformation to the reference sequence ref_sequence$_{output}$ [seqId][] constructed according to the process described in subclause 11.3.5.3 by replacing the symbols present in the reference sequence ref_sequence$_{output}$[seqId][] at the absolute position represented by each **rftp$_i$** descriptor with the symbols conveyed by each corresponding **rftt$_i$** descriptor.

11.3.6 Global assembly

When **cr_alg_ID** is equal to 4, the the reference sequence and the genomic records are decodedas follows for each AU of type 6 (Class U) or of type 5 (class HM):

1. An array refBuf[] is set equal to the empty array.
2. Decode one **rtype** descriptor as specified in subclause 10.4.14.
3. If the value of the decoded **rtype** descriptor is equal to 5 then go to step 4 else go to step 8.
4. Decode one **rlen** descriptor as specified in subclause 10.4.9.
5. Decode the **ureads** descriptor with decodeUreads(rlen) as specified in subclause 10.4.8, where rlen is the value from **rlen** descriptor decoded at previous step 4.
6. Concatenate the array refBuf[]with the output of step 5.
7. Go to step 2.
8. Decode the next sequence as specified in subclause 10.4.14 according to the value of the rtype descriptor decoded at step 2.
9. For each sequence decoded at the previous step whose reverseComp[][][] flag (as specified in subclause 10.4.3) is 1, replace the sequence with its reverse-complement sequence as specified in subclause 9.4, and set the reverseComp[][][] flag to 0.
10. If more **rtype** descriptors are present go to step 2.

The result of the decoding process specified above is 1) a reference sequence contained in the array refBuf[], and 2) the genomic records contained in the current AU corresponding to values of **rtype** not equal to 5 (as specified in subclause 10.4.14) and decoded using the reference sequence in refBuf[].

12 Block payload parsing process

12.1 General

This clause describes the parsing process of **encoded_descriptor_sequences** and **encoded_tokentype** carried by a block payload as specified in subclause 7.5.1.3.3.

The input to this process is the block payload.

The outputs of this process are decoded symbols of all descriptor subsequences populated into the decoded_symbols[][][] data structure, as specified in subclause 12.6.2.

A graphical representation of the parsing process is show in Figure 7 and Figure 8.

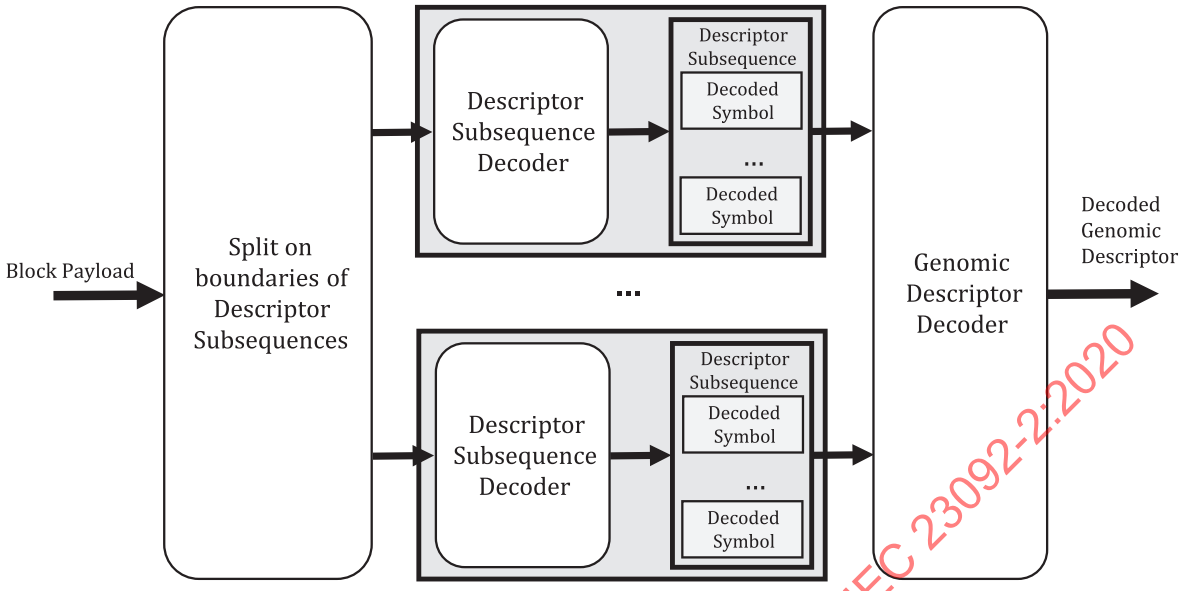

Figure 7 — Block payload parsing process

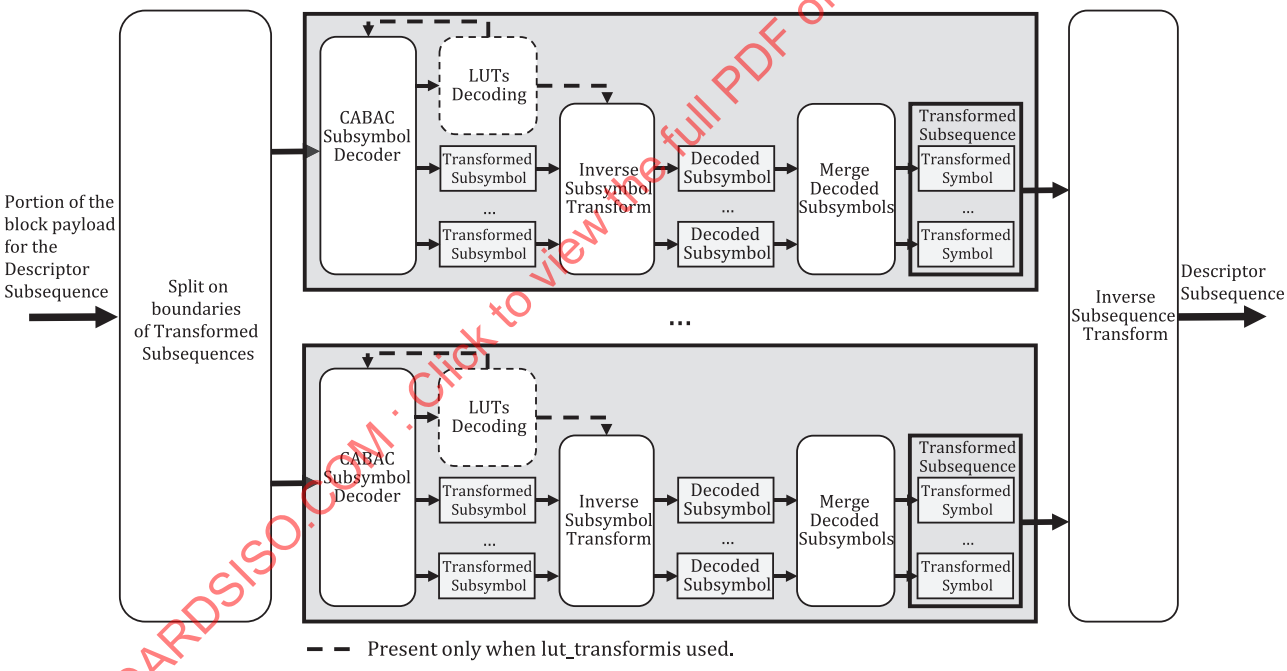

Figure 8 — Decoding process for descriptor subsequences

12.2 Inverse binarizations

12.2.1 General

The process of inverse binarization converts the decoded binary symbols (binVals) into a non-binary-valued symbol (symVal). The following subclauses describe the decoding process for the different binarizations adopted in this document.

The following variables are specified:

— **binVal** is the binary value returned by the decoded_bit().

— **symVal** is the non-binary reconstructed value yielded by the inverse binarization process. In this subclause, it is also referred as decodedCabacSubsym.

— **cmax** is the largest possible binarized value. Larger values are truncated.

Annex C provides examples of inverse binarizations.

12.2.2 Binary (BI)

The inputs to this process are bits from the block payload.

The output of this process is the variable symVal.

The parameter cLength computed in subclause 12.3.6.2 indicates the length in bits of the binarized symVal. The decoding process is described in Table 98.

Table 98 — BI decoding process

```
symVal = 0
for (i=0; i<cLength; i++) {
   symVal = (symVal<<1) | decode_bit()
}
```

12.2.3 Truncated unary (TU)

The inputs to this process are bits from the block payload.

The output of this process is the variable symVal.

The parameter cmax indicates the maximum value of symVal. The decoding process is described in Table 99.

Table 99 — TU decoding process

```
symVal=0;
while(symVal < cmax && decode_bit() == 1) {
   symVal++
}
```

12.2.4 Exponential golomb (EG)

12.2.4.1 General

The inputs to this process are bits from the block payload.

The output of this process is the variable symVal.

The decoding process is described in Table 100.

Table 100 — EG decoding process

```
leadingZeroBits= −1
for( b = 0; !b; leadingZeroBits++ )
   b = decode_bit()
symVal = 0
for( i = 0; i < leadingZeroBits; i++ )
   symVal = (symVal << 1) + decode_bit()
symVal += 2leadingZeroBits − 1
```

12.2.4.2 Signed exponential golomb (SEG) binarization

The input to this process is the output of an exponential golomb binarization as specified in subclause 12.2.4.1.

The output of this process is the variable symVal.

1. Perform the Exponential Golomb decoding process specified in subclause 12.2.4.1.
2. If the output of step 1 is not equal to 0, decode a one-bit sign flag.

12.2.5 If the output of step 2 is 1, symVal= -1*symValTruncated exponential golomb (TEG)

The inputs to this process are bits from the block payload.

The output of this process is the variable symVal.

Truncated exponential golomb is a concatenation of a truncated unary binarization (with cmax equal to cmax_teg signalled in subclause 12.3.3.2) and an exponential golomb binarization. The parsing process for these syntax elements are processed as follows:

1. Perform the truncated unary decoding process with cmax equal to cmax_teg (see 12.2.3).
2. If the output of step 1 is equal to cmax_teg:
   a. Perform the exponential golomb decoding process specified in subclause 12.2.4.

symVal is equal to the sum of step 1 and step 2a.

12.2.6 Signed truncated exponential golomb (STEG)

The inputs to this process are bits from the block payload.

The output of this process is the variable symVal.

Signed truncated exponential golomb is a concatenation of a truncated unary binarization (with cmax equal to cmax_teg signalled in subclause 12.3.3.2), an exponential golomb representation and a 1-bit binary binarization (flag). The decoding process for these syntax elements is as follows:

1. Perform the truncated unary decoding process with cmax equal to cmax_teg (see 12.2.3).
2. If the output of step 1 is equal to cmax_teg:
   a. Perform the exponential golomb decoding process specified in subclause 12.2.4.
3. If the sum of the outputs of step 1 and step 2 is not equal to 0:
   a. Decode a one-bit sign flag.

symVal is equal to the sum of the output values of step 1 and step 2a. If the output of step 3a is 1, symVal= -1*symVal.

12.2.7 Split unit-wise truncated unary (SUTU)

The inputs to this process are bits from the block payload and:

— split_unit_size specified in subclause 12.3.3.2;

— output_symbol_size specified in subclause 12.3.2.

where split_unit_size ≤ output_symbol_size.

The output of this process is the variable symVal.

The SUTU binary string is a concatenation of n TU binarizations (subclause 12.2.3), where n = Ceil(output_symbol_size / split_unit_size).

The decoding process for SUTU binarization is described in Table 101

Table 101 — SUTU decoding process

```
symVal=0
for (i=0; i<output_symbol_size; i+=split_unit_size) {
   unitVal = 0
   cmax = (i == 0 && (output_symbol_size % split_unit_size) != 0) ?
      (1<<(output_symbol_size % split_unit_size))-1 :
      (1<<split_unit_size)-1
   while(unitVal < cmax && decode_bit() == 1)
      unitVal++
   symVal = (symVal<<split_unit_size) | unitVal
}
```

12.2.8 Signed split unit-wise truncated unary (SSUTU)

The inputs to this process are bits from the block payload and:

— split_unit_size specified in subclause 12.3.3.2,

— output_symbol_size specified in subclause 12.3.2,

where split_unit_size ≤ (output_symbol_size-1) and output_symbol_size has one bit reserved for the sign.

The output of this process is the variable symVal.

The SSUTU bin string is extension of the SUTU binarization (subclause 12.2.7) with sign of symVal coded as a separate flag. The decoding process for this binarization is as follows:

1. The SUTU binarization produces the absolute value of symVal (of size output_symbol_size-1).

2. If the output of step 1 is not equal to 0, decode a one-bit sign flag.

If the output of step 2 is 1, symVal= -1*symVal.

12.2.9 Double truncated unary (DTU)

The inputs to this process (see Table 102) are bits from the block payload and:

— cmax_dtu, split_unit_size (specified in 12.3.3.2),

— output_symbol_size (specified in 12.3.2),

where Log2(cmax_dtu) < split_unit_size and split_unit_size ≤ output_symbol_size.

The output of this process is the variable symVal.

The DTU binary string is a concatenation of two binarizations, a TU binarization (subclause 12.2.3) and a SUTU binarization (subclause 12.2.7). The parameter cmax_dtu is used for the TU binarization with cmax equal to cmax_dtu, and the parameters split_unit_size and output_symbol_size are used for the SUTU binarization (where cmax is computed internally).

Table 102 — DTU decoding process

```
symVal = decode_cabac_TU(cmax_dtu)
if(symVal ≥ cmax_dtu) {
   symVal += decode_cabac_SUTU(split_unit_size, output_symbol_size)
}
```

decode_cabac_TU() specifies the decoding process specified in subclause 12.2.3.

decode_cabac_SUTU() specifies the decoding process specified in subclause 12.2.7.

12.2.10 Signed double truncated unary (SDTU)

The inputs to this process are bits from the block payload and:

— cmax_dtu and split_unit_size specified in subclause 12.3.3.2,

— output_symbol_size specified in subclause 12.3.2,

where Log2(cmax_dtu) < split_unit_size, split_unit_size ≤ (output_symbol_size-1) and output_symbol_size has one bit reserved for the sign.

The output of this process is the variable symVal.

The SDTU bin string is an extension of the DTU binarization with sign of symVal coded as a flag. It is obtained as follows:

1. The DTU binarization produces the absolute value of symVal (of size output_symbol_size-1).
2. If the output of step 1 is not equal to 0, decode a one-bit sign flag.

If the output of step 2 is equal to 1 then symVal is set to -1 * symVal.

12.3 Decoder configuration

This subclause provides syntax and semantics to convey information related to the decoder configuration in the parameter set specified in subclause 7.4.

12.3.1 Sequences and quality values

The decoder configuration syntax is specified in Table 103.

Table 103 — Decoder configuration syntax

Syntax	Type
decoder_configuration(encodingModeID){	
if (encodingModeID == 0){ /* CABAC */	
num_descriptor_subsequence_cfgs_minus1	u(8)

for(i = 0;
i ≤ num_descriptor_subsequence_cfgs_minus1;
i++){	

descriptor_subsequence_ID	u(10)
transformSubseqCounter = 1	
transform_subseq_parameters()	As specified in 12.3.4.
for (j = 0; j < transformSubseqCounter ; j++){	
transform_ID_subsym	u(3)
support_values()	As specified in 12.3.2.
cabac_binarization()	As specified in 12.3.3.
}	
}	
} else if(encodingModeID ≥ 1){	
/* reserved for future use */	
}	
}	

num_descriptor_subsequence_cfgs_minus1 plus 1 specifies the number of subsequences the genomic descriptor for which configurations are being signalled in this syntax. The number of descriptor subsequences for each genomic descriptor are specified in Table 24.

descriptor_subsequence_ID identifies the descriptor subsequence to which the current decoder configuration is applied. Its value is comprised between 0 and the number of descriptor subsequences minus 1 as specified in Table 24. Within the same descriptor_configuration(), no value of **descriptor_subsequence_ID** shall be used more than once.

transform_subseq_parameters() signals the parsing of parameters for transformed subsequences. It is specified in subclause 12.3.4.

transform_ID_subsym specifies the subsymbol transform to be applied. Allowed values are specified in in subclause 12.3.4.

support_values() specifies a set of configuration parameters used to parse the transformed subsequence. It is specified in subclause 12.3.2.

cabac_binarization() specifies information about the binarization used for the CABAC decoding of the transformed subsequence. It is specified in subclause 12.3.3.

12.3.2 Support values

Table 104 — Support values data structure

Syntax	Type
support_values(){	
output_symbol_size	u(6)
coding_subsym_size	u(6)
coding_order	u(2)
if(coding_subsym_size < output_symbol_size && coding_order > 0) {	
if(transform_ID_subsym == 1)	
share_subsym_lut_flag	u(1)
share_subsym_prv_flag	u(1)
}	
}	

output_symbol_size signals the size in bits of each transformed symbol of the transformed subsequence to be output by the decoding process. For unsigned binarizations the minium value of **output_symbol_size** is 1, while for signed binarizations the minimum value of **output_symbol_size** is 2. For signed values one bit is used for the sign.

coding_subsym_size signals the size in bits of the transformed subsymbol, which serve as the atomic unit of coding. The value of **coding_subsym_size** shall be a factor (exact divisor) of **output_symbol_size**. It yields X = **output_symbol_size** / **coding_subsym_size** atomic subsymbol slots. These X transformed subsymbols shall be independently decoded with CABAC, go through subsymbol transformations (if any) to yield decoded subsymbols, which shall be combined to output a transformed symbol (of size **output_symbol_size**). If LUTs subsymbol transformation (subclause 12.3.4) is used, the maximum allowed value for **coding_subsym_size** is 8. For signed values, one bit is used for the sign.

coding_order signals the number of previously decoded symbols internally maintained as state variables and is used to decode the next subsymbol. The maximum allowed value is 2.

share_subsym_lut_flag if set to 1 only one look-up-table is signalled (subclause 12.6.2.5) to be shared among all transformed subsymbols to perform inverse LUT subsymbol transformation (subclause 12.6.2.8). Otherwise, for each transformed subsymbol their own look-up-table is signalled and used for inverse LUT subsymbol transformation. The default value is 1.

share_subsym_prv_flag if set to 0 a separate copy of the the previously decoded subsymbols (prvValues in subclause 12.6.2.2) is maintained to decode transformed subsymbol for each subsymbol slot. Otherwise, a single copy of previously decoded subsymbols is circularly shared to decode transformed subsymbols at all subsymbol slots. The default value is 1.

12.3.3 CABAC binarizations

12.3.3.1 General

Table 105 — CABAC binarization data structure

Syntax	Type
cabac_binarization(){	
binarization_ID	u(5)
bypass_flag	u(1)
cabac_binarization_parameters(binarization_ID)	12.3.3.2
if(!bypass_flag){	

Table 105 *(continued)*

Syntax	Type
cabac_context_parameters()	12.3.3.3
}	
}	

binarization_ID indicates the binarization method to be used for CABAC decoding. The list of binarizations is shown in Table 106. The signed binarizations identified by binarization_ID = {3, 5, 7, 9} are only allowed when coding_subsym_size is equal to **output_symbol_size**.

bypass_flag if equal to 1, all bins of the binarization are decoded using the CABAC bypass mode. It can only be set to 1 with **coding_order** equal to 0.

Table 106 — Values of binarization_ID and associated binarizations

binarization_ID	Type of binarization
0	Binary coding as specified in subclause 12.2.2.
1	Truncated unary as specified in subclause 12.2.3.
2	Exponential golomb as specified in subclause 12.2.4.
3	Signed exponential golomb as specified in subclause 12.2.4.2.
4	Truncated exponential golomb as specified in subclause 12.2.5.
5	Signed truncated exponential golomb as specified in subclause 12.2.6.
6	Split unit-wise truncated unary as specified in subclause 12.2.7.
7	Signed split unit-wise truncated unary as specified in subclause 12.2.8.
8	Double truncated unary as specified in subclause in 12.2.9.
9	Signed double truncated unary as specified in subclause in 12.2.10.
10 .. 31	Reserved for future use.

12.3.3.2 CABAC binarizations parameters

The **cabac_binarization_parameters** data structure contains the binarization parameters for the transformed subsequence. **binarization_ID** is specified in subclause 12.3.3.

Table 107 — CABAC binarization parameters

Syntax	Type
cabac_binarization_parameters(binarization_ID) {	
if(binarization_ID == 1) {	
cmax	u(8)
} else if (binarization_ID==4 || binarization_ID==5) {	
cmax_teg	u(8)
} else if (binarization_ID==8 || binarization_ID==9) {	
cmax_dtu	u(8)
}	

Table 107 *(continued)*

Syntax	Type
if (binarization_ID==6 || binarization_ID==7 || binarization_ID==8 || binarization_ID==9) {	
split_unit_size	u(4)
}	
}	

cmax is specified in subclause 12.2.3. The maximum allowed value is 255 and shall always be less than (1<< coding_subsym_size). It shall be greater than zero.

cmax_teg is specified in subclauses 12.2.5 and 12.2.6. The maximum allowed value is 255 and shall always be less than (1<< coding_subsym_size) and greater than 0.

cmax_dtu is specified in clauses 12.2.9 and 12.2.10. The maximum allowed value is 255 and shall always be smaller than (1<<split_unit_size) and greater than 0.

split_unit_size is specified in subclause 12.2.7. The maximum allowed value is 8 and shall always be greater than 0 and smaller than **output_symbol_size** specified in subclause 12.3.2.

The binarizations SUTU (subclause 12.2.7), SSUTU (subclause 12.2.8), DTU (subclause 12.2.9) and SDTU (subclause 12.2.9) shall only be used when coding_order is equal to 0 and **output_symbol_size** is equal to coding_subsym_size, while the internal subsymbol size is signalled by the parameter **split_unit_size**.

12.3.3.3 CABAC context parameters

The **cabac_context_parameters** data structure signals the parameters used for the initialization and adaptation of the ctxTable[] (specified in 12.4) for the transformed subsequence (see Table 108).

Table 108 — Syntax of the cabac_context_parameters data structure

Syntax	Type
cabac_context_parameters(){	
adaptive_mode_flag	u(1)
num_contexts	u(16)
for (i=0; i<num_contexts; i++){	
context_initialization_value[i]	u(7)
}	
if(coding_subsym_size < output_symbol_size) {	
share_subsym_ctx_flag	u(1)
}	
}	

adaptive_mode_flag if set to 1 signals that the arithmetic decoding engine specified in subclause 12.5 uses context adaptation, otherwise contexts adaptation is disabled.

num_contexts signals the size of the table ctxTable[ ] (initialized as defined in 12.4) containing the list of context variables needed for the decoding of the LUTs and the transformed subsequence.

When **num_contexts** is signalled as 0:

— the process defined in 12.3.6.6 shall be used to calculate the state variable numCtxTotal;

— the process defined in 12.4 initializes the contexts in ctxTable[ ] with initState equal to 64 (equiprobability).

Otherwise

— the state variable numCtxTotal is set to the signalled value of **num_contexts**;

— the process defined in 12.4 initializes the contexts in ctxTable[ ] with the values signalled in **context_initialization_values[ ]**.

context_initialization_values[i] specifies the initialization state value for the i^{th} context variable. The state value can range between 0 and 127, with value 64 representing the equiprobable state value.

coding_subsym_size is specified in subclause 12.3.2.

output_symbol_size is specified in subclause 12.3.2.

share_subsym_ctx_flag if set to 1, all transformed subsymbols are decoded on the same set of contexts. Otherwise, separate set of contexts are initialized and used to decode each transformed subsymbol. The default value is 0.

12.3.4 Transformation parameters

Table 109 — Data structure for transformation parameters

Syntax	Type
transform_subseq_parameters(){	
transform_ID_subseq	u(8)
if(transform_ID_subseq == equality_coding){	
transformSubseqCounter += 1	
} else if(transform_ID_subseq == match_coding) {	
match_coding_buffer_size	u(16)
transformSubseqCounter += 2	
} else if(transform_ID_subseq == rle_coding) {	
rle_coding_guard	u(8)
transformSubseqCounter += 1	
} else if (transform_ID_subseq == merge_coding)	
merge_coding_subseq_count	u(4)
transformSubseqCounter = merge_coding_subseq_count	
for(i=0; i<merge_coding_subseq_count; i++)	
merge_coding_shift_size[i]	u(5)
}	
}	

transform_ID_subseq signals the applied subsequence transformation according to Table 110.

Table 110 — Values of transform_ID_subseq and transform_ID_subsym

Sub-sequence transformations		
transform_ID_subseq	**name**	**Remarks**
0	no_transform	No transform is applied.
1	equality_coding	As specified in 12.6.2.10.2.
2	match_coding	As specified in 12.6.2.10.3.
3	rle_coding	As specified in 12.6.2.10.4.
4	merge_coding	As specified in 12.6.2.10.5.
5 .. 255		Reserved for future use.
Subsymbol transformations		
transform_ID_subsym	name	Remarks
0	no_transform	No transformation is applied.
1	lut_transform	It can only be used when coding_order > 0.
2	diff_coding	It can only be used when coding_order is equal to 0.
3 .. 7		Reserved for future use.

transform_ID_subsym specified in subclause 12.3.1 signals the applied subsymbol transformation according to Table 110. The value transform_ID_subsym equal to 1 is not allowed whenever either of the following is true: coding_order is equal to 0, coding_subsym_size is greater than 8, or binarization_ID is equal to one of the values {3, 5, 6, 7, 8, 9}.

transformSubseqCounter is a state variable defined in subclause 12.3.1.

match_coding_buffer_size signals the size of the internal fifo buffer used in match coding transformation (subclause 12.6.2.10.3).

rle_coding_guard is the guard value used in run-length coding transform (subclause 12.6.2.10.4).

merge_coding_subseq_count signals the number of transform subsequences to be merged by the merge subsequence transformation (subclause 12.6.2.10.5). The minimum allowed value is 2.

merge_coding_shift_size[i] signals the number of bits to be shifted in the transformed symbols of each transformed subsequence while applying the merge subsequence transformation (subclause 12.6.2.10.5).

The merge subsequence transformation shall adhere to the following restrictions:

— For each transformed subsequence, coding_subsym_size shall be equal to output symbol_size.

— All transformed subsequences shall have exactly the same number of transformed symbols, which shall also be equal to the number of symbols encoded in the descriptor subsequence.

— The sum of the sizes of transformed symbols (output_symbol_size) for all transformed subsequences shall not be greater than 32.

12.3.5 Msar descriptor and read identifiers

The decoder configuration syntax for the **msar** descriptor and read identifiers (decoded as specified in subclause 10.4.20) is specified in Table 111.

Table 111 — Decoder configuration syntax for msar and read identifiers

Syntax	Type
decoder_configuration_tokentype(encodingModeID){	
if (encodingModeID == 0){	
/* configuration for RLE specified in subclause 10.4.19.3.3 */	
rle_guard_tokentype	u(8)
/* configuration for CABAC_METHOD_0 specified in subclause 10.4.19.3.4 */	
decoder_configuration_tokentype_cabac(0)	
/* configuration for CABAC_METHOD_1 specified in subclause 10.4.19.3.5 */	
decoder_configuration_tokentype_cabac(1)	
} else if(encodingModeID ≥ 1){	
/* reserved for future use */	
}	
}	

rle_guard_tokentype represents the guard value used in the decoding process of RLE method (listed in Table 78 and specified in subclause 10.4.20.4.4) for the decoding of **tokentype** descriptor sequences.

Table 112 — Decoder configuration syntax for CABAC decoding of tokentype descriptors

Syntax	Type
decoder_configuration_tokentype_cabac() {	
transformSubseqCounter = 1	
transform_subseq_parameters()	As specified in 12.3.4.
for (j = 0; j < transformSubseqCounter; j++) {	
transform_ID_subsym	u(3)
support_values()	As specified in 12.3.2.
cabac_binarization()	As specified in 12.3.3.
}	
}	

transform_subseq_parameters() signals the parameters for transformed subsequences. It is specified in subclause 12.3.4.

transform_ID_subsym signals the subsymbol transformion to be applied. Allowed values are as specified in 12.3.4.

support_values() signals a set of configuration parameters used to parse the transformed subsequence. It is specified in subclause 12.3.2.

cabac_binarization() signals information about the binarization used for the CABAC decoding of the transformed subsequence. It is specified in subclause 12.3.3.

12.3.6 State variables

This subclause specifies how to calculate state variables used during the decoding process.

12.3.6.1 Number of alphabet symbols

The number of alphabet symbols for each subsymbol shall be calculated as numAlphaSubsym = 1 << coding_subsym_size. However, for some descriptor subsequences, this calculation produces larger

alphabets than needed. Table 113 lists these special cases and the value of numAlphaSubsym when numAlphaSubsym is not calculated as numAlphaSubsym = 1 << coding_subsym_size.

Table 113 — Special cases for numAlphaSubsym values.

descriptor_ID	subsequence_ID	numAlphaSubsym
4	0	3
4	1	Size($S_{alphabet\_ID}$)
4	2	Size($S_{alphabet\_ID}$)
5	1	9
5	2	Size($S_{alphabet\_ID}$) + 1
6	0	Size($S_{alphabet\_ID}$)
12	0	6
17	0	Size($S_{alphabet\_ID}$)

The number of subsymbols shall be calculated as numSubsyms = output_symbol_size / coding_subsym_size.

12.3.6.2 Number of contexts per subsymbol

When bypass mode is not used (as signalled in subclause 12.3.3), the cabac decoding of the transformed subsymbol uses a number of contexts (as specified in subclause 12.5.2). Table 114 lists the number of contexts needed to decode each transformed subsymbol with all binarizations.

Table 114 — Calculation of numCtxSubsym

binarization_ID	numCtxSubsym
0	`coding_subsym_size`
1	`cmax`
2	`Floor(Log2(numAlphaSubsym + 1)) + 1`
3	`Floor(Log2(numAlphaSubsym + 1)) + 2`
4	`cmax_teg + Floor(Log2(numAlphaSubsym + 1)) + 1`
5	`cmax_teg + Floor(Log2(numAlphaSubsym + 1)) + 2`
6	`(output_symbol_size / split_unit_size) * ((1<< split_unit_size) - 1) + ((1<<(outputSymSize % split_unit_size)) - 1)`
7	`(output_symbol_size / split_unit_size) * ((1<< split_unit_size) - 1) + ((1<<(outputSymSize % split_unit_size)) - 1) + 1`
8	`cmax_dtu + (output_symbol_size / split_unit_size) * ((1<< split_unit_size) - 1) + ((1<<(outputSymSize % split_unit_size)) - 1)`
9	`cmax_dtu + (output_symbol_size / split_unit_size) * ((1<< split_unit_size) - 1) + ((1<<(output_symbol_size % split_unit_size)) - 1) + 1`

coding_subsym_size is specified in subclause 12.3.2.

output_symbol_size is specified in subclause 12.3.2.

cLength is specified as a parameter to BI binarization (subclause 12.2.2) and it is set to coding_subsym_size.

cmax is specified as a parameter to TU (subclause 12.2.3) and signalled in 12.3.3.2.

cmax_teg is specified as a parameter to the TEG (subclause 12.2.5) and STEG (subclause 12.2.5) binarizations, and signalled in 12.3.3.2.

split_unit_size is specified as a parameter to the SUTU (subclause 12.2.7), SSUTU (subclause 12.2.8), DTU (subclause 12.2.9) and SDTU (subclause 12.2.9) binarizations, and signalled in 12.3.3.2.

cmax_dtu is specified as a parameter to the DTU (subclause 12.2.9) and SDTU (subclause 12.2.9) binarizations, and signalled in 12.3.3.2.

12.3.6.3 Coding order context offset

The decoding process of a subymbol can depend on a number of previously decoded subsymbols (at the same bit positions) by signaling coding_order > 0 as specified in subclause 12.3.2.

The process of context selection (subclause 12.6.2.6) requires the context offsets corresponding to the coding order to correctly calculate the starting ctxIdx in the ctxTable[ ], where each subsymbol is to be decoded.

Table 115 specifies how the list codingOrderCtxOffset[ ] containing these offsets for each coding order is calculated. If bypass_flag is equal to 1 (as signalled in subclause 12.3.3), all elements of codingOrderCtxOffset are set to 0.

Table 115 — Calculation of codingOrderCtxOffset[ ]

coding_order	State variable	Value
0	codingOrderCtxOffset[0]	0
1	codingOrderCtxOffset[1]	numCtxSubsym
2	codingOrderCtxOffset[2]	numCtxSubsym * numAlphaSubsym

12.3.6.4 Coding size context offset

The state variable codingSizeCtxOffset specifies the number of contexts needed to decode each transformed subsymbol.

This state variable is used in the contexts selection process (subclause 12.6.2.6) to correctly calculate the starting ctxIdx in the ctxTable[ ] where each transformed subsymbol is to be decoded. It is computed as specified in Table 116. If bypass_flag is equal to 1 (as signalled in subclause 12.3.3), this state variable is set to 0.

Table 116 — Calculation of codingSizeCtxOffset

```
if(share_subsym_ctx_flag)
   codingSizeCtxOffset = 0
else if(coding_order == 0)
   codingSizeCtxOffset = numCtxSubsym
else
   codingSizeCtxOffset = codingOrderCtxOffset[coding_order] * numAlphaSubsym
```

12.3.6.5 Number of contexts for LUTs

The state variable numCtxLuts specifies the number of contexts needed to decode the LUTs using the the decoding process for LUTs (specified in subclause 12.6.2.5), where each LUT symbol shall be decoded using the SUTU binarization (binarization_ID equal to 6) with parameters splitUnitSize equal to 2 and outputSymSize = coding_subsym_size. The value of numCtxLuts is computed as specified in Table 117. If bypass_flag is equal to 1 (as signalled in subclause 12.3.3), this state variable is set to 0.

Table 117 — Calculation of numCtxLuts

```
numCtxLuts = 0
if(transform_ID_subsym == 1)
      /* Compute according to Table 114 for SUTU binarization */
   numCtxLuts = (coding_subsym_size / 2) * ((1<< 2) - 1) +
                  ((1<<(coding_subsym_size % 2)) - 1)
}
```

12.3.6.6 Total number of contexts

The state variable numCtxTotal specifies the total number of contexts needed to decode a transformed subsequence, which includes all the contexts needed for decoding of LUTs (subclause 12.6.2.5) and symbols (subclause 12.6.2.7) and shall be calculated as specified in Table 118. If bypass_flag is equal to 1 (as signalled in subclause 12.3.3), this state variable is set to 0.

Table 118 — Calculation of numCtxTotal

```
if(num_contexts != 0) {
   numCtxTotal = num_contexts
} else {
   numCtxTotal = numCtxLuts
   numCtxTotal +=  ((share_subsym_ctx_flag) ? 1 : numSubsyms) *
                     ((coding_order > 0) ? codingOrderCtxOffset[coding_order] *
                     numAlphaSubsym : numCtxSubsymbol)
}
```

num_contexts is signalled in 12.3.3.3 along with the list of specific context_initialization_values[].

12.4 Initialization process for context variables

ctxTable[ ] is the data structure containing all context variables needed to decode a transformed subsequence. Each element of the ctxTable[ ] represents one context variable and consists of two state variables: pStateIdx and valMps. The variable pStateIdx represents a probability state index and the variable valMps represents the value of the most probable symbol as further described in subclause 12.5.2.

The inputs to this process are:

— ctxTable[] specified in subclause 12.6.2.4;

— the ctxIdx and initState variables specified in12.6.2.4.

The output of this process is an initialized context variable in the **ctxTable** array at index ctxIdx.

The state variables pStateIdx and valMps corresponding to index ctxIdx are initialized based on a 7-bit initState as described in Table 119.

Table 119 — Calculation of ctxTable

Syntax
`context_initialize_state(ctxTable[ ], ctxIdx, initState) {`
`  ctxTable[ctxIdx].valMps = ( initState ≤ 63 ) ? 0 : 1`
`  ctxTable[ctxIdx].pStateIdx = ctxTable[ctxIdx].valMps ? ( initState – 64 ) : ( 63 – initState)`
`}`

where

ctxTable[ctxIdx].valMps represents the variable valMps associated to the element in ctxTable at index ctxIdx

ctxTable[ctxIdx].pStateIdx represents the variable pStateIdx associated to the element in ctxTable at index ctxIdx

12.5 Arithmetic decoding engine

12.5.1 Initialization

The outputs of this process are the initialized decoding engine registers ivlCurrRange and ivlOffset both in 16 bit register precision.

The status of the arithmetic decoding engine is represented by the variables ivlCurrRange and ivlOffset. In the initialization procedure of the arithmetic decoding process, ivlCurrRange is set equal to 510 and ivlOffset is set equal to the value returned from read_bits(9) interpreted as a 9 bit binary representation of an unsigned integer with the most significant bit written first.

The bitstream shall not contain data that result in a value of ivlOffset being equal to 510 or 511.

NOTE The description of the arithmetic decoding engine in this Specification utilizes 16 bit register precision. However, a minimum register precision of 9 bits is required for storing the values of the variables ivlCurrRange and ivlOffset after invocation of the arithmetic decoding process (DecodeBin) as specified in subclause 12.5.2. The arithmetic decoding process for a binary decision (DecodeDecision) as specified in subclause 12.5.2.2 and the decoding process for a binary decision before termination (DecodeTerminate) as specified in subclause 12.5.2.5 require a minimum register precision of 9 bits for the variables ivlCurrRange and ivlOffset. The bypass decoding process for binary decisions (DecodeBypass) as specified in subclause 12.5.2.4 requires a minimum register precision of 10 bits for the variable ivlOffset and a minimum register precision of 9 bits for the variable ivlCurrRange.

12.5.2 Arithmetic decoding process

12.5.2.1 General

The inputs to this process are ctxTable, ctxIdx, and bypass_flag, as specified in subclause 12.6.2.7, and the state variables ivlCurrRange and ivlOffset of the arithmetic decoding engine.

The output of this process is the value of the bin.

Figure 9 illustrates the whole arithmetic decoding process for a single bin. For decoding the value of a bin, the context index table ctxTable and the ctxIdx are passed to the arithmetic decoding process DecodeBin(ctxTable, ctxIdx), which is specified as follows:

— If bypassFlag is equal to 1, DecodeBypass( ) as specified in subclause 12.5.2.4 is invoked.

— Otherwise, if bypassFlag is equal to 0, ctxTable is equal to 0, and ctxIdx is equal to 0, DecodeTerminate( ) as specified in subclause 12.5.2.5 is invoked.

— Otherwise (bypassFlag is equal to 0 and ctxTable is not equal to 0), DecodeDecision( ) as specified in subclause 12.5.2.2 is invoked.

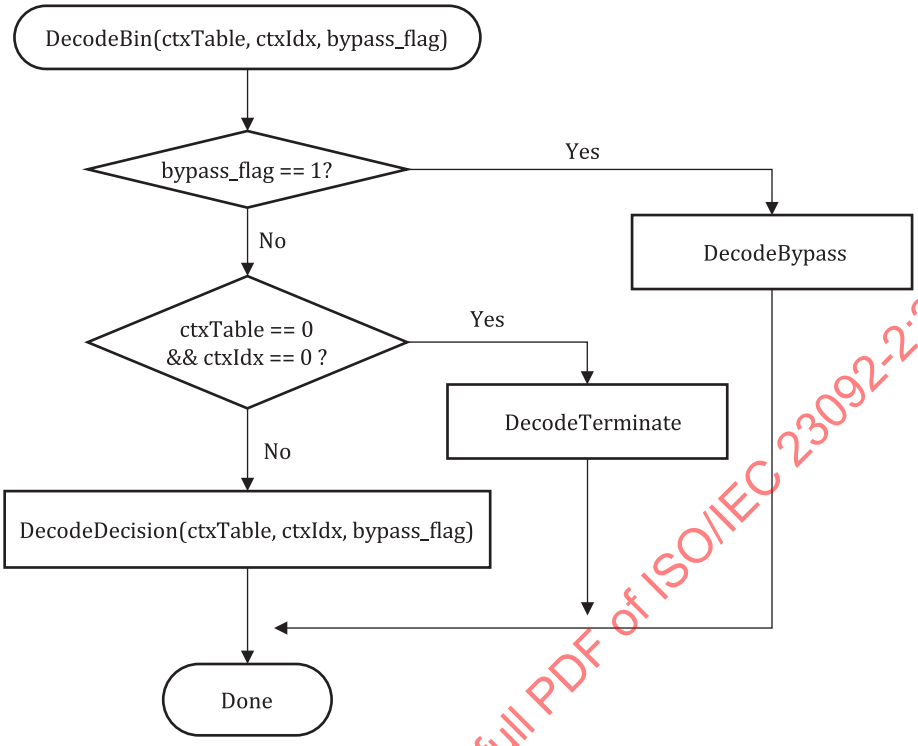

Figure 9 — Overview of the arithmetic decoding process for a single bin

NOTE Arithmetic coding is based on the principle of recursive interval subdivision. Given a probability estimation p(0) and p(1) = 1 – p(0) of a binary decision (0, 1), an initially given code sub-interval with the range ivlCurrRange will be subdivided into two sub-intervals having range p(0) * ivlCurrRange and ivlCurrRange – p(0) * ivlCurrRange, respectively. Depending on the decision, which has been observed, the corresponding sub-interval will be chosen as the new code interval, and a binary code string pointing into that interval will represent the sequence of observed binary decisions. It is useful to distinguish between the most probable symbol (MPS) and the least probable symbol (LPS), so that binary decisions have to be identified as either MPS or LPS, rather than 0 or 1. Given this terminology, each context is specified by the probability p_{LPS} of the LPS and the value of MPS (valMps), which is either 0 or 1. The arithmetic core engine in this document has three distinct properties:

— The probability estimation is performed by means of a finite-state machine with a table-based transition process between 64 different representative probability states { p_{LPS}(pStateIdx) | 0 ≤ pStateIdx < 64 } for the LPS probability p_{LPS}. The numbering of the states is arranged in such a way that the probability state with index pStateIdx = 0 corresponds to an LPS probability value of 0.5, with decreasing LPS probability towards higher state indices.

— The range ivlCurrRange representing the state of the coding engine is quantized to a small set $\{Q_1,...,Q_4\}$ of pre-set quantization values prior to the calculation of the new interval range. Storing a table containing all 64x4 pre-computed product values of $Q_i * p_{LPS}$(pStateIdx) allows a multiplication-free approximation of the product ivlCurrRange * p_{LPS}(pStateIdx).

— For syntax elements or parts thereof for which an approximately uniform probability distribution is assumed to be given a separate simplified encoding and decoding bypass process is used.

12.5.2.2 Arithmetic decoding process for a binary decision

12.5.2.2.1 General

The inputs to this process are the variables ctxTable, ctxIdx, ivlCurrRange, and ivlOffset.

The outputs of this process are the decoded value binVal, and the updated variables ivlCurrRange and ivlOffset.

Figure 10 shows the flowchart for decoding a single decision (DecodeDecision):

1. The value of the variable ivlLpsRange is derived as follows:

   — Given the current value of ivlCurrRange, the variable qRangeIdx is derived as follows:

   qRangeIdx =(ivlCurrRange >> 6) & 3

   — Given qRangeIdx and pStateIdx associated with ctxTable and ctxIdx, the value of the variable rangeTabLps as specified in Table 121 is assigned to ivlLpsRange:

   ivlLpsRange = rangeTabLps[pStateIdx][qRangeIdx]

2. The variable ivlCurrRange is set equal to ivlCurrRange – ivlLpsRange and the following applies:

   — If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to–1 – valMps, ivlOffset is decremented by ivlCurrRange, and ivlCurrRange is set equal to ivlLpsRange.

   — Otherwise, the variable binVal is set equal to valMps.

Given the value of binVal, the state transition is performed as specified in subclause 12.5.2.2.2. Depending on the current value of ivlCurrRange, renormalization is performed as specified in subclause 12.5.2.3.

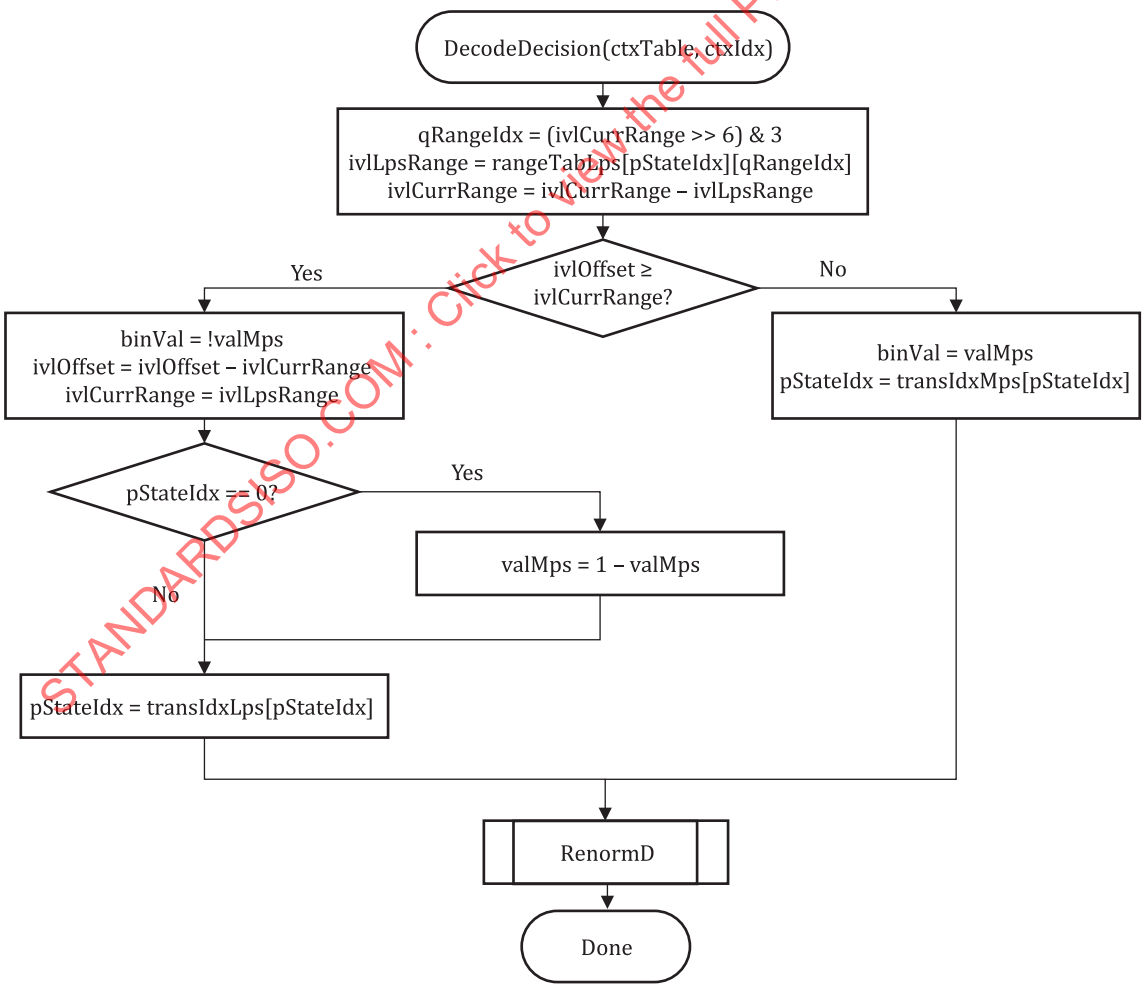

Figure 10 — Flowchart for decoding a decision

12.5.2.2.2 State transition process

The inputs to this process are the current pStateIdx, the decoded value binVal and valMps values of the context variable associated with ctxTable and ctxIdx.

The outputs of this process are the updated pStateIdx and valMps of the context variable associated with ctxIdx.

Depending on the decoded value binVal, the update of the two variables pStateIdx and valMps associated with ctxIdx is derived as specified in Table 120.

Table 120 — Update of the two variables pStateIdx and valMps

```
If (adaptive_mode_flag) {
   if( binVal = = valMps )
      pStateIdx = transIdxMps( pStateIdx )
   else {
      if( pStateIdx = = 0 )
         valMps = 1 – valMps
      pStateIdx = transIdxLps( pStateIdx )
   }
}
```

Table 122 specifies the transition rules transIdxMps( ) and transIdxLps( ) after decoding the value of valMps and 1 – valMps, respectively.

Table 121 — Specification of rangeTabLps depending on the values of pStateIdx and qRangeIdx

pStateIdx	qRangeIdx				pStateIdx	qRangeIdx			
	0	1	2	3		0	1	2	3
0	128	176	208	240	**32**	27	33	39	45
1	128	167	197	227	**33**	26	31	37	43
2	128	158	187	216	**34**	24	30	35	41
3	123	150	178	205	**35**	23	28	33	39
4	116	142	169	195	**36**	22	27	32	37
5	111	135	160	185	**37**	21	26	30	35
6	105	128	152	175	**38**	20	24	29	33
7	100	122	144	166	**39**	19	23	27	31
8	95	116	137	158	**40**	18	22	26	30
9	90	110	130	150	**41**	17	21	25	28
10	85	104	123	142	**42**	16	20	23	27
11	81	99	117	135	**43**	15	19	22	25
12	77	94	111	128	**44**	14	18	21	24
13	73	89	105	122	**45**	14	17	20	23
14	69	85	100	116	**46**	13	16	19	22
15	66	80	95	110	**47**	12	15	18	21
16	62	76	90	104	**48**	12	14	17	20
17	59	72	86	99	**49**	11	14	16	19
18	56	69	81	94	**50**	11	13	15	18
19	53	65	77	89	**51**	10	12	15	17
20	51	62	73	85	**52**	10	12	14	16

Table 121 *(continued)*

pStateIdx	qRangeIdx				pStateIdx	qRangeIdx			
	0	1	2	3		0	1	2	3
21	48	59	69	80	**53**	9	11	13	15
22	46	56	66	76	**54**	9	11	12	14
23	43	53	63	72	**55**	8	10	12	14
24	41	50	59	69	**56**	8	9	11	13
25	39	48	56	65	**57**	7	9	11	12
26	37	45	54	62	**58**	7	9	10	12
27	35	43	51	59	**59**	7	8	10	11
28	33	41	48	56	**60**	6	8	9	11
29	32	39	46	53	**61**	6	7	9	10
30	30	37	43	50	**62**	6	7	8	9
31	29	35	41	48	**63**	2	2	2	2

Table 122 — State transition table

pStateIdx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
transIdxLps	0	0	1	2	2	4	4	5	6	7	8	9	9	11	11	12
transIdxMps	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
pStateIdx	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
transIdxLps	13	13	15	15	16	16	18	18	19	19	21	21	22	22	23	24
transIdxMps	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
pStateIdx	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**
transIdxLps	24	25	26	26	27	27	28	29	29	30	30	30	31	32	32	33
transIdxMps	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
pStateIdx	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**
transIdxLps	33	33	34	34	35	35	35	36	36	36	37	37	37	38	38	63
transIdxMps	49	50	51	52	53	54	55	56	57	58	59	60	61	62	62	63

12.5.2.3 Renormalization process in the arithmetic decoding engine

The inputs to this process are bits from block payload data and the variables ivlCurrRange and ivlOffset.

The outputs of this process are the updated variables ivlCurrRange and ivlOffset.

A flowchart of the renormalization is shown in Figure 11. The current value of ivlCurrRange is first compared to 256 and then the following applies:

— If ivlCurrRange is greater than or equal to 256, no renormalization is needed and the RenormD process is finished;

— Otherwise (ivlCurrRange is less than 256), the renormalization loop is entered. Within this loop, the value of ivlCurrRange is doubled, i.e., left-shifted by 1 and a single bit is shifted into ivlOffset by using read_bits(1).

The bitstream shall not contain data that result in a value of ivlOffset being greater than or equal to ivlCurrRange upon completion of this process.

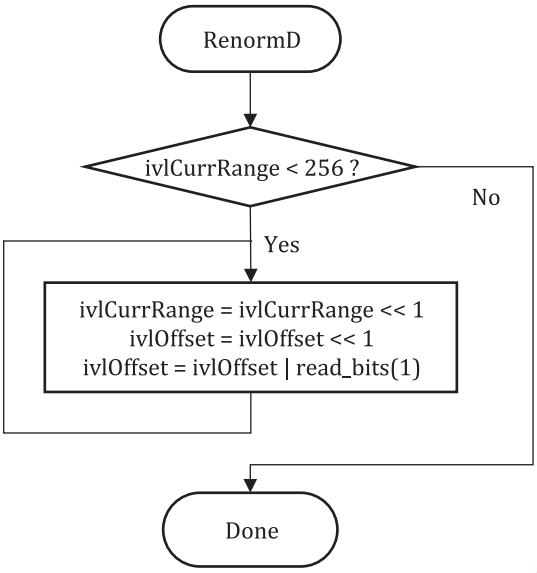

Figure 11 — Flowchart of renormalization

12.5.2.4 Bypass decoding process for binary decisions

The inputs to this process are bits from block payload data and the variables ivlCurrRange and ivlOffset.

The outputs of this process are the updated variable ivlOffset and the decoded value binVal.

The bypass decoding process is invoked when bypassFlag is equal to 1. Figure 12 shows a flowchart of the corresponding process.

First, the value of ivlOffset is doubled, i.e. left-shifted by 1 and a single bit is shifted into ivlOffset by using read_bits(1). Then, the value of ivlOffset is compared to the value of ivlCurrRange and then the following applies:

— If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1 and ivlOffset is decremented by ivlCurrRange.

— Otherwise (ivlOffset is less than ivlCurrRange), the variable binVal is set equal to 0.

The bitstream shall not contain data that result in a value of ivlOffset being greater than or equal to ivlCurrRange upon completion of this process.

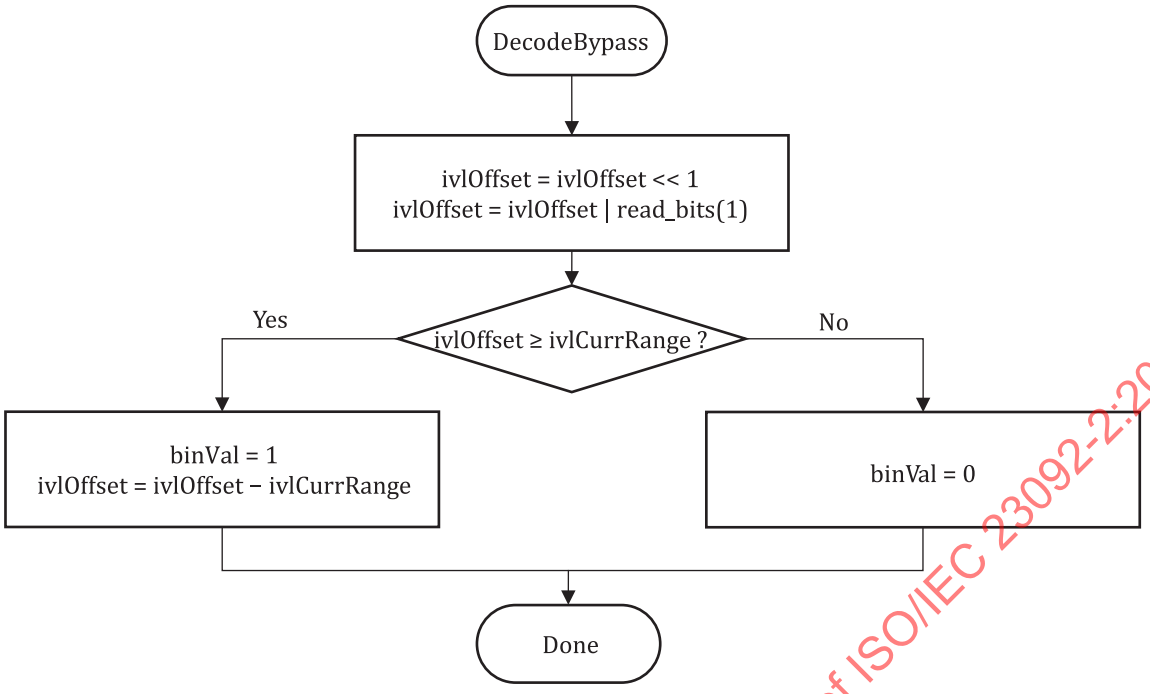

Figure 12 — Flowchart of bypass decoding process

12.5.2.5 Decoding process for binary decisions before termination

The inputs to this process are bits from block payload data and the variables ivlCurrRange and ivlOffset.

The outputs of this process are the updated variables ivlCurrRange and ivlOffset, and the decoded value binVal.

This decoding process applies to decoding of end_of_descriptor_subsequence_terminate corresponding to ctxTable equal to 0 and ctxIdx equal to 0. Figure 13 shows the flowchart of the corresponding decoding process, which is specified as follows:

First, the value of ivlCurrRange is decremented by 2. Then, the value of ivlOffset is compared to the value of ivlCurrRange and then the following applies:

— If ivlOffset is greater than or equal to ivlCurrRange, the variable binVal is set equal to 1, no renormalization is carried out, and CABAC decoding is terminated. The last bit inserted in register ivlOffset is equal to 1. When decoding end_of_descriptor_subsequence_terminate, this last bit inserted in register ivlOffset is interpreted as the stop bit for the decoding of descriptor subsequence.

— Otherwise (ivlOffset is less than ivlCurrRange), the variable binVal is set equal to 0 and renormalization is performed as specified in subclause 12.5.2.3.

This procedure may also be implemented using DecodeDecision(ctxTable, ctxIdx, bypassFlag) with ctxTable = 0, ctxIdx = 0 and bypassFlag = 0. In the case where the decoded value is equal to 1, seven more bits would be read by DecodeDecision(ctxTable, ctxIdx, bypassFlag) and a decoding process would have to adjust its bitstream pointer accordingly to properly decode following syntax elements.